Under consideration for publication in J. Functional Programming

Contract Lenses: Reasoning about Bidirectional Programs via Calculation

HANLIANG ZHANG, WENHAO TANG and RUIFENG XIE
Peking University, China
MENG WANG
University of Bristol, United Kindom
ZHENJIANG HU
Peking University, China

Abstract

Bidirectional transformations (BXs) are a mechanism for maintaining consistency between multiple representations of related data. The need to effectively construct BXs has attracted interest in programming languages research, aiming at correctness by construction through specially designed programming languages. This flourishing language scene provides a fertile ground for this work, which develops a reasoning and optimization framework for BXs.

In this paper, we propose contract lenses, which extends the traditional BX framework (also known as lenses) with a pair of predicates to enable the kind of safe and modular compositions needed for program calculation. We define several contract-lens combinators that capture common computation patterns including *fold*,*filter*,*map*, and *scan*, and develop several bidirectional program calculation laws that can be used to reason about and optimize BX programs. We demonstrate the effectiveness of our new framework 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 & Spyratos, 1981; Bohannon et al., 2006; Tran et al., 2020), software model transformation (Stevens, 2007; He & Hu, 2018; Tsigkanos et al., 2020; Stevens, 2020), graph transformation (Hidaka et al., 2010) etc; in particular there has been seveal 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.[1]

$$\textbf{data}\ S \leftrightarrow V = \mathit{Lens}\ \{\mathit{get} : S \to V, \mathit{put} : S \to V \to S\}$$

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:

$$\mathit{put}\ s\ (\mathit{get}\ s) = s \qquad \text{GetPut}$$
$$\mathit{get}\ (\mathit{put}\ s\ v) = v \qquad \text{PutGet}$$

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: [2]

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

In the forward direction, lens composition is simply a function composition of the two *get* functions. In the backward direction, it will first putback the updated t' to the intermediate v produced by $\mathit{get}_{\ell_1}\ s$ using ℓ_2, and then putback updated v back to s.

Lenses are programmed in special languages that preserve round-tripping 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 & Wang, 2018; Matsuda & Wang, 2015a), effectively creates the paradigm of bidirectional programming.

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

> Is there an algebraic theory of lens combinators that would underpin optimization of lens expressions in the same way that the relational algebra and its algebraic theory are used to optimize relational database queries? ... This

[1] We will use Haskell-like notations in our presentation.

[2] Note that the order of the composition of lenses is left-to-right, while the function composition is right-to-left.

algebraic theory will play a crucial role in a more serious implementation effort. (Foster et al., 2007)

1.1 Program Calculation and the Challenge of Partiality

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

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

However, the more complex setting of BX posts unique challenges to program calculation. First of all, calculating BX cannot be superficially treated as calculating twice, once in each direction, as the round-tripping properties bind *get* and *put* closely together, demanding simultaneous reasoning of both. Moreover, *put* functions are often partially defined, making semantic preservation amid calculation difficult (note that a change in the definedness of a function changes its semantics).

This partiality can be inherent, as some changes to the view may contradict the original source and cannot be accepted by *put* without breaking the round-tripping properties. This happens when *get* is non-surjective; its corresponding *put* is undefined for values outside the range. The partiality can also be of design choices, as forcing a *put* function to be total may introduce unwanted complexity. As an example, consider list mapping as a (higher-order) lens which takes a lens ℓ of type $A \leftrightarrow B$ and return a lens of type $[A] \leftrightarrow [B]$.

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

The backward direction (p) is partial by design: any view update that changes the length of the list is not supported. By dint of hard effort, one can make the definition total (Foster et al., 2007; Pacheco & Cunha, 2011). But it will require an additional parameter of type A as a default list element, which complicates the signature of the combinator.

1.2 Contributions

In this paper, we develop a calculation framework to reason about and optimize bidirectional programs over the list data structure. Specifically, we propose an extension to traditional lenses, which we call contract lenses, to enable the construction and composition of possibly partial transformations. We develop several contract-lens combinators, which are higher-order functions that characterize key bidirectional computation patterns on the list data structure. 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 behavior on propagating changes in a compositional way. Composition of contract lenses is justified by the implication relation between the view predicate of the former lens and the source predicate of the latter lens. We also provide an equivalence relation between contract lenses for calculation. (Section 4)

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

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

Mechanized Proof in Agda We prove all the technical details using Agda. The proof consists of 4k lines of Agda code. [3]

Moreover, we showcase the ability of our framework to construct and calculate lenses by advanced examples that either have intricate partial bidirectional behaviours,

[3] `https://github.com/KomaEc-zhl/ContractLens-artifact/tree/master/proof`

or are well-studied in both bidirectional transformations and program calculation literatue.

2 Background: Program Calculation

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

2.1 Specification with Folds

Fold is a computation pattern that captures structural recursion. In Haskell, there are two versions of fold on list: $foldl : (b \to a \to b) \to b \to [a] \to b$ and $foldr : (a \to b \to b) \to b \to [a] \to b$, which can be used to define a range of functions. We list the following as examples, which are also used in the sequel of the paper.

$$
\begin{aligned}
maximum &= foldr\ max\ (-\infty)\\
sum &= foldr\ (+)\ 0\\
map\ f &= foldr\ (\lambda a\ r \to f\ a : r)\ [\,]\\
filter\ p &= foldr\ (\lambda a\ r \to \textbf{if}\ p\ a\ \textbf{then}\ a : r\ \textbf{else}\ r)\ [\,]\\
scanr\ f\ b_0 &= foldr\ (\lambda a\ bs \to (f\ a\ (head\ bs)) : bs)\ [b_0]\\
inits &= foldr\ (\lambda a\ r \to [\,] : map\ (a:)\ r)\ [[\,]]\\
tails &= foldr\ (\lambda a\ r \to (a : head\ r) : r)\ [[\,]]
\end{aligned}
$$

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 $foldr\ f\ e$ have two arguments, which can be combined into one $foldr'\ alg$ where *alg* can be considered as an algebra of type $Either\ ()\ (a,b) \to b$.

$$
\begin{aligned}
&foldr' : (Either\ ()\ (a,b) \to b) \to [a] \to b\\
&foldr'\ alg\ [\,] = alg\ (Left\ ())\\
&foldr'\ alg\ (x : xs) = alg\ (Right\ (x, foldr'\ alg\ xs))
\end{aligned}
$$

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

$$
\begin{aligned}
&alg\ (Left\ ()) = e\\
&alg\ (Right\ (a,b)) = f\ a\ b
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{unfoldr'} : (b \to \mathit{Either}\ ()\ (a,b)) \to b \to [a] \\
&\mathit{unfoldr'}\ \mathit{coalg}\ b = \mathbf{case}\ \mathit{coalg}\ b\ \mathbf{of} \\
&\qquad\qquad\qquad \mathit{Left}\ () \to [\,] \\
&\qquad\qquad\qquad \mathit{Right}\ (a,b) \to a : \mathit{unfoldr'}\ \mathit{coalg}\ b
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{inits'} = \mathit{tail} \circ \mathit{inits} \\
&\mathit{tails'} = \mathit{init} \circ \mathit{tails} \\
&\mathit{scanl'}\ f\ x = \mathit{tail} \circ \mathit{scanl}\ (\mathit{flip}\ f)\ x \\
&\mathit{scanr'}\ f\ x = \mathit{init} \circ \mathit{scanr}\ f\ x
\end{aligned}
$$

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

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:

$$\frac{h \circ f = g \circ \mathrm{F_L}\ h}{h \circ \mathit{foldr'}\ f = \mathit{foldr'}\ g} \qquad \text{Fold Fusion}$$

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

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

where $+$ and $\times$ on functions are defined by $(f+g)\ (\mathit{Left}\ x) = f\ x$, $(f+g)\ (\mathit{Right}\ y) = g\ y$, and $(f \times g)\ (x,y) = (f\ x, g\ y)$.

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.

$$\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g) \qquad \text{Map Fusion}$$

$$\mathit{foldr'}\ f \circ \mathit{map}\ g = \mathit{foldr'}\ (f \circ \mathrm{F_m}\ g) \qquad \text{Fold-Map Fusion}$$

$$\mathit{map}\ (\mathit{foldl}\ f\ e) \circ \mathit{inits} = \mathit{scanl}\ f\ e \qquad \text{Scan Lemma}$$

Note that $\mathrm{F_m}$ is the so-called map functor, which is defined by

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

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

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

where $x \odot y = (x \otimes y) \oplus a$. □

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. An efficient implementation of it is very difficult to develop, 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.

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

Given a list, we first enumerate all the segments by $map\ tails \circ inits$. Then we calculate the sum of all segments by $map\ (map\ sum)$ and get the maximum of these results of sum by $maximum \circ map\ maximum$. 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, 1989a).

The challenge that this paper aim 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 conditions on source/view changes, as below

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

where ℓ is a traditional lens. 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 $get_\ell\ s$ and updated view v, restricting the permitted values of v 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 $put_\ell\ s\ v$, 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 (list length) of the view, which we specify as the following predicate.

$$eqlength = \lambda xs\ xs' \to (length\ xs = length\ xs')$$

This condition is enough to ensure that *bmap* always executes successfully in the backward direction and consequently the upholding of 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:

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

The source condition can be used to match the view condition of another lens to enable sequential composition. For example,

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

This composition is well formed as the source condition of the latter implies the view of the former. With contract lenses, the partiality issues of lens composition is reduced to locally reasoning of adjacent conditions.

Of course, one alternative is to make the lenses total. But as mentioned in the introduction, such an approach will result in various kinds of complications including having to change the signatures of the lenses and requiring advanced type systems.

3.2 Calculation with Contract Lenses

Once we have established proper compositions, we can start to design a calculation system 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 below.

$$\begin{array}{ll} & filter\ even \circ map\ head \\ = & \{\ \text{expressing } filter \text{ as } foldr\ \} \end{array}$$

$$
\begin{array}{ll}
 & \mathit{foldr}\ (\lambda a\ r \to \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a : r\ \textbf{else}\ r)\ [\,] \circ \mathit{map}\ \mathit{head} \\
= & \quad \{\ \text{Fold-Map Fusion}\ \} \\
 & \mathit{foldr}\ ((\lambda a\ r \to \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a : r\ \textbf{else}\ r) \circ \mathit{head})\ [\,]
\end{array}
$$

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

$$
\begin{array}{l}
\{\mathit{eqlength}\}\ \mathit{bmap}\ \mathit{bhead}\ \{\mathit{eqlength}\}; \{\mathit{eqlength}\}\ \mathit{bfilter}\ \mathit{even}\ \{\mathit{ceven}\} \\
\quad \textbf{where}\ \mathit{bhead} = \mathit{CLens}\ \mathit{head}\ (\lambda \mathit{xs}\ x' \to x' : \mathit{tail}\ \mathit{xs})
\end{array}
$$

The condition $\{\mathit{ceven}\}$ requires that all the elements in the view remain even.

$$
\mathit{ceven}\ \mathit{xs}\ \mathit{xs}' = \mathit{eqlength}\ \mathit{xs}\ \mathit{xs}' \wedge \mathit{all}\ \mathit{even}\ \mathit{xs}'
$$

The combinators *bmap* and *bfilter* are contract lenses for *map* and *filter* respectively (which will be formally defined in Section 5). We have already seen the case of *bmap*. In this example, *bfilter* is also given a view condition including *eqlength*. This is a necessary condition to derive the *eqlength* source condition needed for composing with *bmap*: if the number of even elements is not changed, the total number of elements will not change because the odd elements, which do not appear in the view, cannot be changed by the view update.

Similar to above, *bfilter* is implemented by bfoldr', which is a bidirectional *foldr* but with *eqlength* as its source condition. And we have the following BFoldr'-BMap Fusion law, stating that the composition of a *bmap* and a bfoldr' can be fused into a single bfoldr'. [4]

$$
\begin{array}{l}
\{\mathit{eqlength}\}\ \mathit{bmap}\ \ell_1\ \{\mathit{eqlength}\}; \{\mathit{eqlength}\}\ \mathit{bfoldr}'\ \ell_2\ \{\mathit{cv}\} \\
\quad = \{\mathit{eqlength}\}\ \mathit{bfoldr}'\ (\mathit{bmapF}\ \ell_1; \ell_2)\ \{\mathit{cv}\}
\end{array}
$$

With the ground prepared, the calculation itself (below) is completely unsurprising. Note that when the conditions are obvious from the context, we omit them for brevity.

$$
\begin{array}{ll}
 & \mathit{bmap}\ \mathit{bhead}; \mathit{bfilter}\ \mathit{even} \\
= & \quad \{\ \text{expressing } \mathit{bfilter} \text{ as } \mathit{foldr}'\ \} \\
 & \mathit{bmap}\ \mathit{bhead}; \mathit{bfoldr}'\ (\mathit{bfilterAlg}\ \mathit{even}) \\
= & \quad \{\ \text{BFoldr'-BMap Fusion}\ \} \\
 & \mathit{bfoldr}'\ (\mathit{bmapF}\ \mathit{bhead}; (\mathit{bfilterAlg}\ \mathit{even}))
\end{array}
$$

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] The BFoldr'-BMap Fusion law in Section 6 uses a more powerful bidirectional map, bmap', which takes *bmap* as a special version of it.

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 dynamic constraints, *cs* and *cv*, describing the conditions of changes in the source and the view respectively.

Definition 4.1 (Contract Lenses)
A contract lens[5] between source of type S and view of type V consists of a pair of transformations *get* and *put* together with a pair of predicates: a source condition $cs : S \to S \to Bool$ and a view condition $cv : V \to V \to Bool$.

$$\textbf{data}\ \{cs\}\ S \leftrightarrow V\ \{cv\} = CLens\ \{get : S \to V, put : S \to V \to S\}$$

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

$$cv\ (get\ s)\ v \Rightarrow cs\ s\ (put\ s\ v) \qquad \text{BackwardValidity}$$
$$cv\ (get\ s)\ v \Rightarrow get\ (put\ s\ v) = v \qquad \text{ConditionedPutGet}$$
$$cs\ s\ s \Rightarrow cv\ (get\ s)\ (get\ s) \qquad \text{ForwardValidity}$$
$$cs\ s\ s \Rightarrow put\ s\ (get\ s) = s \qquad \text{ConditionedGetPut}$$

□

For backward transformations, the BackwardValidity law and the ConditionedPutGet says 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 says that if the source *s* satisfies *cs s s*, then the change on the view *get s* should satisfy *cv*, and the get-put law holds. The condition *cs s s* in the ConditionedGetPut law is necessry 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 $cs\ s\ (put\ s\ (get\ s)) = cs\ s\ s$.

We have three remarks to make here. First, for readability, we write $\{cs\}$ and $\{cv\}$ around $S \leftrightarrow V$, but we should realize that a contract lens consists of four components, having $\{cs\}$ and $\{cv\}$ besides *get* and *put*, and these four components can refer to each other if necessary (e.g., *put* may call *get* and $\{cs\}$ may call *get* and *put*). Second, the source/view conditions describe general constraints, which include both

[5] 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.

static and dynamic constraints on the source/view. For example, $cs\ s\ s' = (length\ s = length\ s') \wedge (head\ s' \neq 1)$ requires that the length should be unchanged and the new list s' should not start with 1. Thus, $cs\ s\ s$ is not always true, even though s remains the same. The ForwardValidity law is used to describe the properties propagated by the *get* function. When the source s satisfies the condition $cs\ s\ s$, the result of the forward transformation $v = get\ s$ satisfies the condition $cv\ v\ v$. Third, 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 $get\ s$. [6]

Before we continue, let us discuss some notational conventions.

• We use $\ell : \{cs\}\ S \leftrightarrow V\ \{cv\}$ to refer to a contract lens named ℓ, and use $cs_\ell, cv_\ell, get_\ell, put_\ell$ to refer to its source condition, view condition, forward transformation and backward transformation, respectively.

• We assume a list starts from index 1 and use the notation x_i to refer to the i-th element of x.

• If not explicitly stated, we assume by default that if there is a function with invalid input in the predicate, then the result of the predicate is false. For example, $head\ xs = 1$ is considered to be false when $xs = [\,]$ as the function *head* is not defined on empty list.

Now we give some simple examples of contract lenses. We leave more interesting examples in Section 5.

Example 4.1 (Embedding Traditional Lenses into Contract Lenses)
As our contract lenses is an extension of the traditional lenses, traditional lenses can be embedded into contract lenses by adding dummy conditions *ctrue*, where $ctrue\ \_\ \_ = True$. For any traditional lens $\ell : S \leftrightarrow V$, it can be embedded as a contract lens $\ell : \{ctrue\}\ S \leftrightarrow V\ \{ctrue\}$ with the same implementation of get_ℓ and put_ℓ. □

Example 4.2 (Bidirectional Inits)
An interesting example is a bidirectional version of *inits'* defined in Section 2.1.

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

Thanks to 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. □

[6] One exception is that we do not require the condition $cs\ s\ s$ when we compute $get\ s$ in the contracts. It does not influence the correctness of contract lenses, because in this case, we just take the *get* as a unidirectional function.

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 4.2 (Composition of Contract Lenses)
For two contract lenses $\ell_1 : \{cs_{\ell_1}\}\ S \leftrightarrow V\ \{cv_{\ell_1}\}$ and $\ell_2 : \{cs_{\ell_2}\}\ V \leftrightarrow T\ \{cv_{\ell_2}\}$, 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 $\ell_1;\ell_2 : \{cs_{\ell_1}\}\ S \leftrightarrow T\ \{cv_{\ell_2}\}$ as defined below.

$$
\begin{aligned}
&\ell_1;\ell_2 = \mathit{CLens}\ g\ p \\
&\quad \textbf{where} \\
&\qquad g \quad = get_{\ell_2} \circ\ get_{\ell_1} \\
&\qquad p\ s\ t = put_{\ell_1}\ s\ \ (put_{\ell_2}\ (get_{\ell_1}\ s)\ t)
\end{aligned}
$$

□

Theorem 4.1 (Well-behaved Composition)
The contract lens $\ell_1;\ell_2 : \{cs_{\ell_1}\}\ S \leftrightarrow T\ \{cv_{\ell_2}\}$ defined in Definition 4.2 satisfies the round-tripping properties defined in Definition 4.1. □

We say that two predicates $c_1 : A \to A \to \mathit{Bool}$ and $c_2 : B \to B \to \mathit{Bool}$ are equivalent, written as $c_1 \Leftrightarrow c_2$, if $A = B$ and $\forall\ (a : A)\ (a' : A), c_1\ a\ a' = c_2\ a\ a'$. Note that 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 4.3 (Lens Equivalence)
For lens $\ell_1 : \{cs_{\ell_1}\}\ S \leftrightarrow V\ \{cv_{\ell_1}\}$ and $\ell_2 : \{cs_{\ell_2}\}\ S \leftrightarrow V\ \{cv_{\ell_2}\}$, we say ℓ_1 is equivalent to ℓ_2, written as $\ell_1 = \ell_2$, if

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

□

Theorem 4.2 (Lens Equivalence is an Equivalence Relation)
The equivalence relation between contract lenses is reflexive, symmetric and transitive. □

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

In this section, we define several lens combinators to capture important patterns (higher order functions) for easy construction of complex contract lenses in a compositional manner, as well as to demonstrate power and flexibility of our new contract lens framework. The combinators are a very important component of a program calculation framework, which are the basis for developing calculation laws.

Since bidirectional programs can be considered as unidirectional forward programs with additional putback semantics, our idea is to bidirectionalize 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 higher-order functions, and are useful for different situations. We shall 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 behaviors.

5.1 Bidirectional Fold

We have seen in Section 2 that *folds* are of vital importance in program calculation, so we start with *bfoldr*, a bidirectional version of $foldr'$, without constraints on both source and view:

$$bfoldr : (\{ctrue\}\ Either\ ()\ (S,V) \leftrightarrow V\ \{ctrue\}) \to (\{ctrue\}\ [S] \leftrightarrow V\ \{ctrue\})$$

Given a simple contract lens $\ell : \{ctrue\}\ (Either\ ()\ (S,V)) \leftrightarrow V\ \{ctrue\}$, $bfoldr\ \ell$ returns a contract lens $\{ctrue\}\ [S] \leftrightarrow V\ \{ctrue\}$, synchronizing a list $[S]$ with V. For the *get* direction, we can simply use $foldr'\ get_\ell$ to compute a view from an input list. For the *put* direction, we can recursively construct an updated source list (using $unfoldr'$) from the original source and an updated view step by step through put_ℓ, the putback transformation of ℓ. Formally, we can define *bfoldr* as follows.

$$\begin{aligned}
&bfoldr\ \ell = CLens\ g\ p\\
&\quad \textbf{where}\ g = foldr'\ get_\ell\\
&\qquad\quad p = curry\ \$\ unfoldr'\ coalg\\
&\qquad\quad coalg\ ([\,],v') \quad\ = \textbf{case}\ put_\ell\ (Left\ ())\ v'\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\quad Left\ () \to Left\ ()\\
&\qquad\qquad\qquad\qquad\quad Right\ (a',b') \to Right\ (a',([\,],b'))\\
&\qquad\quad coalg\ (a:as,v') = \textbf{case}\ put_\ell\ (Right\ (a,g\ as))\ v'\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\quad Left\ () \to Left\ ()\\
&\qquad\qquad\qquad\qquad\quad Right\ (a',b') \to Right\ (a',(as,b'))
\end{aligned}$$

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 $g\ as$ in advance using a *scanr*. We will use this definition of *bfoldr* in the following sections.

$$
\begin{aligned}
&bfoldr\ \ell = CLens\ (foldr'\ get_\ell)\ p\\
&\quad \textbf{where}\ p\ as\ b' = \textbf{let}\ bs = tail\ \$\ scanr\ (\lambda a\ b \to get_\ell\ (Right\ (a,b)))\ (get_\ell\ (Left\ ()))\ as\\
&\qquad\qquad \textbf{in}\ go\ as\ bs\ b'\\
&\qquad go\ []\ []\ b' = \textbf{case}\ put_\ell\ (Left\ ())\ b'\ \textbf{of}\\
&\qquad\qquad\qquad Left\ () \to []\\
&\qquad\qquad\qquad Right\ (a', bim') \to a' : go\ []\ []\ bim'\\
&\qquad go\ (a:as)\ (bim:bs)\ b' = \textbf{case}\ put_\ell\ (Right\ (a, bim))\ b'\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad Left\ () \to []\\
&\qquad\qquad\qquad\qquad\qquad Right\ (a', bim') \to a' : go\ as\ bs\ bim'
\end{aligned}
$$

Similarly, we can define *bfoldl*, which is omitted here. To see concretely how *bfold* works, we give an example below.

Example 5.1 (Bidirectional Maximum)
Considering that we want to synchronize a list with its maximum, we can define it in terms of *bfoldr* by

$$
\begin{aligned}
&bmaximum : \{ctrue\}\ [Int] \leftrightarrow Int\ \{ctrue\}\\
&bmaximum = bfoldr\ bmax
\end{aligned}
$$

where *bmax* is a bidirectional version of *max*.

$$
\begin{aligned}
&bmax : \{ctrue\}\ Either\ ()\ (Int, Int) \leftrightarrow Int\ \{ctrue\}\\
&bmax = CLens\ g\ p\\
&\quad \textbf{where}\\
&\qquad g\ (Left\ ()) &&= -\infty\\
&\qquad g\ (Right\ (x,y)) &&= max\ x\ y\\
&\qquad p\ (Left\ ())\ (-\infty) &&= Left\ ()\\
&\qquad p\ (Left\ ())\ v' &&= Right\ (v', -\infty)\\
&\qquad p\ (Right\ (x,y))\ v' &&= \textbf{if}\ x \geq y\ \textbf{then}\ Right\ (v', min\ v'\ y)\ \textbf{else}\ Right\ (min\ v'\ x, v')
\end{aligned}
$$

To see a computation instance of *bmaximum*, let us assume that $get_{bmaximum}\ [9,2,5]$ 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 $[4,2,4]$.

$$
\begin{aligned}
&put_{bmaximum}\ [9,2,5]\ 4\\
=\quad &\{\ \text{since}\ put_{bmax}\ (Right\ (9, get_{bmaximum}\ [2,5]))\ 4 = Right\ (4,4)\ \}\\
&4 : put_{bmaximum}\ [2,5]\ 4\\
=\quad &\{\ \text{since}\ put_{bmax}\ (Right\ (2, get_{bmaximum}\ [5]))\ 4 = Right\ (2,4)\ \}\\
&4 : 2 : put_{bmaximum}\ [5]\ 4\\
=\quad &\{\ \text{since}\ put_{bmax}\ (Right\ (5, get_{bmaximum}\ []))\ 4 = Right\ (4, -\infty)\ \}\\
&4 : 2 : 4 : put_{bmaximum}\ []\ (-\infty)\\
=\quad &\{\ \text{since}\ put_{bmax}\ (Left\ ())\ (-\infty) = Left\ ()\ \}
\end{aligned}
$$

$4:2:4:[\,]$ □

5.1.1 Bidirectional Fold : Preserving Length and Transmitting Constraints

While *bfoldr* is useful when it is total in both *get* and *put* directions, we may wish to keep the length of the source unchanged after *put*. For example, considering the *bmaximum* in Example 5.1, we may wish to keep the length of the source list after $put_{bmaximum}$, and furthermore, we hope that the source and view of *bfoldr* have some extra constraints to consider. All these can be concisely expressed as a contract lens with appropriate source and view conditions:

$$bfoldr' : (\{lift\ cs\ cv\}\ Either\ ()\ (S,V) \leftrightarrow V\ \{cv\}) \to (\{licond\ cs\}\ [S] \leftrightarrow V\ \{cv\})$$
$$bfoldr'\ \ell = bfoldr\ \ell$$

where *lift* and *licond*, two higher-order predicates, express our intentional constraints.

$$lift\ cs\ cv = \lambda s\ s' \to (s = Left\ ()\ \wedge\ s' = Left\ ())\ \vee$$
$$(s = Right\ (x,y)\ \wedge\ s' = Right\ (x',y')\ \wedge\ cs\ x\ x'\ \wedge\ cv\ y\ y')$$
$$licond\ cs\ xs\ xs' = eqlength\ xs\ xs'\ \wedge\ (\forall\ 1 \le i \le |xs|, cs\ xs_i\ xs'_i)$$

bfoldr' is the same as *bfoldr* except that it has different contracts on the lens it takes and the result lens. It is worth noting that for readability, the source and view conditions are written at the type signature, but they are two of the four components of a contract lens (rather than being part of a type).

Example 5.2 (Bidirectional Maximum Preserving Length)
A direct use of *bfoldr'* is to define a bidirectional version of *maximum* that preserves the length of the source list.

$$bmaximum' : \{eqlength\}\ [Int] \leftrightarrow Int\ \{cv\}$$
$$bmaximum' = bfoldr'\ bmax'$$
where
$$cv\ b\ b' = (b = -\infty) \Rightarrow (b' = -\infty)$$
$$bmax' : \{lift\ ctrue\ cv\}\ Either\ ()\ (Int,Int) \leftrightarrow Int\ \{cv\}$$
$$bmax' = bmax$$

Notice that $licond\ ctrue = eqlength$ and the implementation of *bmax'* is the same as *bmax*. As one may doubt about the statement $put\ (Left\ ())\ v' = Right\ (v', -\infty)$ in the code of *bmax*, in fact, it will never be executed because the condition $cv\ (-\infty)\ v'$ is broken. □

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 desired *get* semantics of a filter can be specified by a *foldr* as $filter\ p = foldr\ (\lambda a\ r \to \textbf{if}\ p\ a\ \textbf{then}\ a : r\ \textbf{else}\ r)\ [\,]$, which returns a list of those elements that satisfy a predicate. With the $bfoldr'$ introduced above, we are able to define a bidirectional version of *filter*, where the lengths of the source and view of *filter* remain unchanged.

$$
\begin{array}{l}
bfilter : (pr : a \to Bool) \to (\{eqlength\}\ [a] \leftrightarrow [a]\ \{fcond\ pr\}) \\
bfilter\ pr = bfoldr'\ (bfilterAlg\ pr) \\
\quad \textbf{where} \\
\quad\quad fcond\ pr\ v\ v' = licond\ (\lambda \_\ x' \to pr\ x')\ v\ v' \\
\quad\quad bfilterAlg\ \ :\ \ (pr : a \to Bool) \\
\quad\quad\quad\quad\quad \to \{lift\ ctrue\ (fcond\ pr)\}\ (Either\ ()\ (a, [a])) \leftrightarrow [a]\ \{fcond\ pr\}) \\
\quad\quad bfilterAlg\ pr = CLens\ g\ p \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad g\ (Left\ ()) \quad\quad\quad\quad = [\,] \\
\quad\quad\quad\quad g\ (Right\ (x, xs)) \quad\quad = \textbf{if}\ pr\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs \\
\quad\quad\quad\quad p\ (Left\ ())\ [\,] \quad\quad\quad = Left\ () \\
\quad\quad\quad\quad p\ (Right\ (x, xs))\ xs' = \textbf{if}\ pr\ x\ \textbf{then}\ Right\ (head\ xs', tail\ xs')\ \textbf{else}\ Right\ (x, xs')
\end{array}
$$

The *bfilter* is defined using the $bfoldr'$ with suitable source and view conditions. The $bfilterAlg\ pr$ is essentially a bidirectional version of $\lambda a\ r \to \textbf{if}\ pr\ a\ \textbf{then}\ a : r\ \textbf{else}\ r$. Note that it is allowed to use the function *pr* in the view condition of the result lens of *bfilter*. The following gives an example of *bfilter*.

Example 5.3 (Synchronizing List with its Even Numbers)
Consider the function *filter even* which filters out all the even integers from an integer list. If we only allow even numbers to be changed to even numbers in the view, the bidirectional version of it can be succinctly defined as follows.

$$
\begin{array}{l}
bevens : \ \{eqlength\}\ [Int] \leftrightarrow [Int]\ \{fcond\ even\} \\
bevens = bfilter\ even
\end{array}
$$

□

5.2 Bidirectional Map

Map is another important higher-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 keeps the lengths of both source and view unchanged with the help of source and view conditions.

$$
\begin{aligned}
&bmap : (\{ctrue\}\ S \leftrightarrow V\ \{ctrue\}) \to (\{eqlength\}\ [S] \leftrightarrow [V]\ \{eqlength\})\\
&bmap\ \ell = CLens\ (map\ get_\ell)\ p\\
&\quad \textbf{where}\ p\ as\ bs' = map\ (\lambda(x,y) \to put_\ell\ x\ y)\ (zip\ as\ bs')
\end{aligned}
$$

It is clear to see from this definition that if the change on the view does not change its length, after backward propagation through $put_{bmap\ \ell}$, 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 implement *bmap* using $bfoldr'$ as follows:

$$
\begin{aligned}
&bmap : (\{ctrue\}\ S \leftrightarrow V\ \{ctrue\}) \to (\{eqlength\}\ [S] \leftrightarrow [V]\ \{eqlength\})\\
&bmap\ \ell = bfoldr'\ \ell'\\
&\quad \textbf{where}\\
&\qquad \ell' :: \{lift\ ctrue\ eqlength\}\ Either\ ()\ (S,[V]) \leftrightarrow [V]\ \{eqlength\}\\
&\qquad \ell' = CLens\ g\ p\\
&\qquad g\ (Left\ ()) = [\,]\\
&\qquad g\ (Right\ (a,bs)) = get_\ell\ a : bs\\
&\qquad p\ (Left\ ())\ [\,] = Left\ ()\\
&\qquad p\ (Right\ (a,\_))\ (a' : bs') = Right\ (put_\ell\ a\ a', bs')
\end{aligned}
$$

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. In general, each element in the old source/view may have constraints between it and the corresponding element in the new source/view. Thus, we define another version of bidirectional *map* which takes this kind of constraints into consideration.

$$
\begin{aligned}
&bmap' : (\{\widehat{cs}\}\ S \leftrightarrow V\ \{\widehat{cv}\}) \to (\{licond\ \widehat{cs}\}\ [S] \leftrightarrow [V]\ \{licond\ \widehat{cv}\})\\
&bmap' = bmap
\end{aligned}
$$

As seen above, $bmap'$ is actually a generalized version of *bmap*; by taking $\widehat{cs} = \widehat{cv} = ctrue$ in $bmap'$ we get *bmap*. Also, we can implement $bmap'$ using $bfoldr'$ in the same way with only some modifications on the contracts:

$$
\begin{aligned}
&bmap : (\{\widehat{cs}\}\ S \leftrightarrow V\ \{\widehat{cv}\}) \to (\{licond\ \widehat{cs}\}\ [S] \leftrightarrow [V]\ \{licond\ \widehat{cv}\})\\
&bmap\ \ell = bfoldr'\ \ell'\\
&\quad \textbf{where}\\
&\qquad \ell' :: \{lift\ \widehat{cs}\ (licond\ \widehat{cv})\}\ Either\ ()\ (S,[V]) \leftrightarrow [V]\ \{licond\ \widehat{cv}\}\\
&\qquad \ell' = CLens\ g\ p\\
&\qquad g\ (Left\ ()) = [\,]\\
&\qquad g\ (Right\ (a,bs)) = get_\ell\ a : bs
\end{aligned}
$$

$$p\ (Left\ ())\ [\,] = Left\ ()$$
$$p\ (Right\ (a, \_))\ (a' : bs') = Right\ (put_\ell\ a\ a', bs')$$

Example 5.4 (Bidirectional Doubles)
The following defines a bidirectional version for $map\ (*2) : [Int] \to [Int]$ where the result list only contains even numbers.

$$\begin{aligned}
&bdoubles : \{eqlength\}\ [Int] \leftrightarrow [Int]\ \{licond\ (\lambda_\ b \to mod\ b\ 2 = 0)\}\\
&bdoubles = bmap'\ bdouble\\
&\quad \textbf{where}\ bdouble\ :\ \{ctrue\}\ Int \leftrightarrow Int\ \{\lambda_\ b \to mod\ b\ 2 = 0\}\\
&\qquad\quad bdouble = CLens\ (*2)\ (\lambda_\ v' \to div\ v'\ 2)
\end{aligned}$$

□

5.2.3 Bidirectional Map: Preserving Constraints on Adjacent Elements

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

In bidirectional programming, we have the same situation. Recall that we have a bidirectional program for $inits$ with the following view condition:

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

We cannot write $binits; bmap\ \ell$, because the condition of composition in Definition 4.2 is not satisfied. This motivated us to introduce $bmapl$, another bidirectional version of map that can return a contract lens whose source condition is cv_{binits} or something more general.

$$\begin{aligned}
&bmapl\ :\ as_0 : S\\
&\qquad \to \ell : (a : S \to (\{\lambda_\ a' \to \widetilde{cs}\ a\ a'\ \wedge\ keepsame\ \ell\ a\}\\
&\qquad\qquad\qquad S \leftrightarrow V\\
&\qquad\qquad\qquad \{\lambda_\ b' \to \widetilde{cv}\ (get_{(\ell\ a)})\ a)\ b'\}))\\
&\qquad \to (\{\lambda t\ as \to (\forall\ 1 \le i \le |as|, \widetilde{cs}\ as_{i-1}\ as_i)\ \wedge\ eqlength\ t\ as\}\\
&\qquad\quad [S] \leftrightarrow [V]\\
&\qquad\quad \{\lambda t\ bs \to (\textbf{let}\ bs_0 = get_{(\ell\ as_0)}\ as_0\ \textbf{in}\ \forall\ 1 \le i \le |as|, \widetilde{cv}\ bs_{i-1}\ bs_i)\ \wedge\ eqlength\ t\ bs\})\\
&bmapl\ a_0\ \ell = CLens\ g\ p\\
&\quad \textbf{where}\\
&\qquad g\ as \quad\ = map\ (\lambda(a', a) \to get_{\ell\ a'}\ a)\ (zip\ (a_0 : init\ as)\ as)\\
&\qquad p\ as\ bs' \quad = scanl'\ (\lambda(a, b')\ a' \to put_{\ell\ a'}\ a\ b')\ a_0\ (zip\ as\ bs')\\
&\qquad keepsame\ \ell\ a = \forall\ a'' : S, get_{(\ell\ a)} = get_{(\ell\ a'')}
\end{aligned}$$

Notice that the constraint $keepsame\ \ell\ a$ in the contract $cs_{(\ell\ a)}$ is actually a constraint on ℓ, which requires that for any $a'' : S$, the forward transformation $get_{(\ell\ a'')}$ is the same function. Thus, for $map\ f$, we can take $get_{(\ell\ a)}$ for any $a : S$ to be f and think of $bmapl\ a_0\ \ell$ as a bidirectional version of $map\ f$.

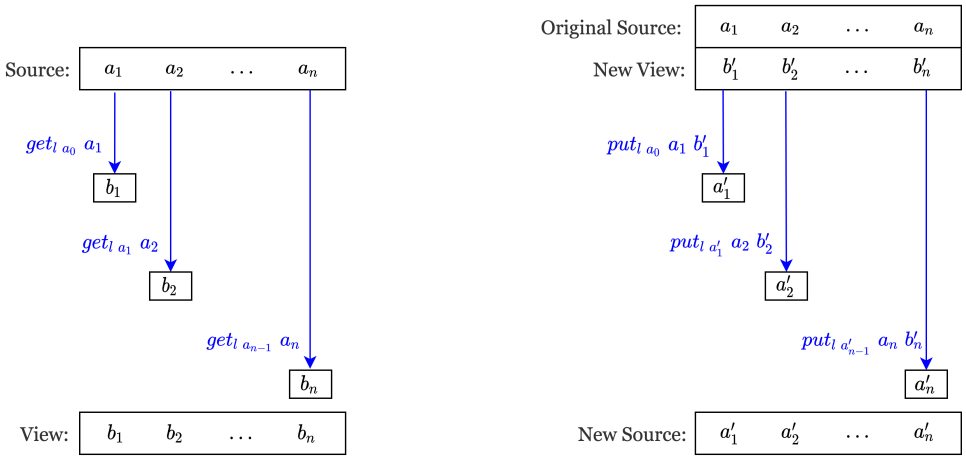

Fig. 1. Implementation of $bmapl\ a_0\ \ell$. The left figure shows the computation of the *get* and the right figure shows the computation of the *put*.

The constraints on adjacent elements of lists are specified by $\widetilde{cs}$ and $\widetilde{cv}$. For example, if we take $\widetilde{cs}$ to be $\lambda x\ y \to (init\ y = x)$ and a_0 to be $[\,]$, then the source condition of the $bmapl\ a_0\ \ell$ is equivalent to cv_{binits}, and thus, the composition $binits; bmapl\ [\,]\ \ell$ is valid.

Let us take a closer look at the implementation of *bmapl*. For any valid a_0 and ℓ, the implementation of $bmapl\ a_0\ \ell$ is depicted in Figure 1. The forward transformation is exactly the same as the unidirectional $map\ f$ except that when computing $get\ a_i$, ℓ additionally takes a_{i-1} which is the left element of a_i. The constraint $\widetilde{cs}\ a_{i-1}\ a_i$ is satisfied because of the ConditionedGetPut law. For the backward transformation, because of the BackwardValidity law of $cv\ (get\ s)\ v \Rightarrow cs\ s\ (put\ s\ v)$, we want the adjacent elements of the updated source to satisfy the constraint $\widetilde{cs}$. Thus, when computing $a'_i = put\ a_i\ b'_i$, ℓ additional takes a'_{i-1} and makes sure the result a'_i satisfies $\widetilde{cs}\ a'_{i-1}\ a'_i$.

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

Example 5.5 (Bidirectional Map on Sorted List)
With the help of *bmapl*, we are able to handle any constraint on adjacent elements of a list, such as a partial order relations. Consider a unidirectional computation $map\ (\lambda x \to mod\ x\ 10) \circ sort : [Int] \to [Int]$, which sorts the list first and then applies the modulo 10 operation on each element. The *sort* can be bidirectionalized as

$$bsort : \{ctrue\}\ [Int] \leftrightarrow [Int]\ \{\lambda t\ as \to (\forall\ 1 < i \le |as|, as_{i-1} \le as_i)\ \wedge\ eqlength\ t\ as\}$$
$$bsort = CLens\ sort\ p$$
$$\textbf{where}\ p\ as\ bs' = \textbf{if}\ sort\ as = bs'\ \textbf{then}\ as\ \textbf{else}\ bs'$$

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

$$
\begin{array}{l}
bmapl\ (-\infty)\ bmod10 : \{\lambda t\ as \to (\forall\ 1 < i \leq |as|, as_{i-1} \leq as_i)\ \wedge\ eqlength\ t\ as\} \\
\qquad\qquad\qquad\qquad\quad [S] \leftrightarrow [V] \\
\qquad\qquad\qquad\qquad\quad \{eqlength\} \\
\quad \textbf{where} \\
\qquad bmod10 : a : S \to (\{\lambda\_\ a' \to (a \leq a')\ \wedge\ keepsame\ bmod10\ a\}\ S \leftrightarrow V\ \{ctrue\}) \\
\qquad bmod10\ a' = CLens\ (\lambda a \to mod\ a\ 10)\ p \\
\qquad\quad \textbf{where}\ p\ a\ b = \textbf{if}\ mod\ a\ 10 = b\ \textbf{then}\ go\ a\ \textbf{else}\ go\ b \\
\qquad\qquad\quad go\ x\ = \textbf{if}\ x > a'\ \textbf{then}\ x\ \textbf{else}\ go\ (x+10)
\end{array}
$$

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

5.2.4 Bidirectional Map using Inner Bidirectional Fold

As we have seen so far, $bmapl\ \ell$ is useful to give a bidirectional version for $map\ f$. What if f is a *fold*? Since ℓ has type $S \to (\{cs\}\ S \leftrightarrow V\ \{cv\})$, we cannot directly pass a *bfold* to *bmapl*. Moreover, since the *bfold* depends on $\widetilde{cs}$ in the source condition of the result of *bmapl*, it is actually difficult to deal with general *bfold*. Good news is that we can define some special *bfold* to cope with some frequently used constraints, such as $\lambda a_{i-1}\ a_i \to (init\ a_i = a_{i-1})$. $bfoldl_{init}$ is such a special *bfold* that can be used inside *bmapl*.

$$
\begin{array}{l}
bfoldl_{init}\ :\ b_0 : V \\
\qquad\qquad \to \ell\ : (b : V\ \ \to (\{\lambda\_\ t' \to (t' = Right\ (\_, b))\ \wedge\ keepsame\ \ell\ b\} \\
\qquad\qquad\qquad\qquad\qquad\quad Either\ ()\ (S, V) \leftrightarrow V \\
\qquad\qquad\qquad\qquad\qquad\quad \{\lambda\_\ b' \to \widetilde{cv}\ b\ b'\})) \\
\qquad\qquad \to \ell'\ : (as : [S] \to (\{\lambda\_\ as' \to (init\ as' = as)\ \wedge\ keepsame\ \ell'\ as\} \\
\qquad\qquad\qquad\qquad\qquad\quad [S] \leftrightarrow V \\
\qquad\qquad\qquad\qquad\qquad\quad \{\lambda\_\ b' \to \widetilde{cv}\ (get_{\ell'\ as}\ as)\ b'\})) \\
bfoldl_{init}\ b_0\ \ell\ as = CLens\ g\ p \\
\quad \textbf{where} \\
\qquad g\ \ \ \ \ \ = foldl\ (\lambda b\ a \to get_{\ell\ b}\ (Right\ (a, b)))\ b_0 \\
\qquad p\ [\,]\ b'\ = \textbf{case}\ put_{\ell\ (g\ as)}\ (Left\ ())\ b'\ \textbf{of} \\
\qquad\qquad\qquad Right\ (a, \_) \to as +\!\!+ [a] \\
\qquad p\ as'\ b' = \textbf{case}\ put_{\ell\ (g\ as)}\ (Right\ (last\ as', g\ (init\ as')))\ b'\ \textbf{of} \\
\qquad\qquad\qquad Right\ (a, \_) \to as +\!\!+ [a]
\end{array}
$$

The constraint $keepsame\ \ell\ b$ in $cs_{\ell\ b}$ guarantees that the forward transformation $get_{\ell\ b''}$ for any $b'' : V$ is the same function $f : Either\ ()\ (S, V) \to V$. Thus, $bfoldl_{init}\ b_0\ \ell$ can be considered as a special bidirectional version of $foldl\ (\lambda b\ a \to f\ (Right\ (a, b)))\ b_0$.

Let us explain more about the definition of $bfoldl_{init}$. For simplicity, assume that $bf = bfoldl_{init}\ b_0\ \ell\ as$. Now considering computing $put_{bf}\ as'\ b' = t$, and the condition $cv_{bf}\ (get_{bf}\ as')\ b'$ is required to be true. By the BackwardValidity property we know $cs_{bf}\ as'\ t$, i.e., $init\ t = as$. Thus, we only need to compute the last element a of t by calling $put_{\ell\ (g\ as)}$, which guarantees the result will be a $Right\ (\_, g\ as)$. Therefore, we have $t = as' +\!\!+ [a]$.

Example 5.6 (Bidirectional Prefix Sums)
An intuitive implementation of the computation of prefix sums is $map\ (foldl\ (+)\ 0) \circ inits$. With the help of $bmapl$ and $bfoldl_{init}$, we can easily bidirectionalize it as

$$binits; bmapl\ [\,]\ (bfoldl_{init}\ 0\ badd)$$

where the $badd$ is defined as

$$\begin{array}{l}
badd : (b : Int) \to (\{\lambda_\ t' \to (t' = Right\ (_, b)) \wedge keepsame\ b\ badd\} \\
\qquad\qquad Either\ ()\ (Int, Int) \leftrightarrow Int\ \{ctrue\}) \\
badd\ b = CLens\ g\ p \\
\quad \textbf{where}\ g\ (Left\ ()) \quad\ = 0 \\
\qquad\quad g\ (Right\ (x, y)) = x + y \\
\qquad\quad p\ _\ s \qquad\quad\ = Right\ (s - b, b)
\end{array}$$

This is a good example showing the 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. □

5.3 Bidirectional Scan

After discussing bidirectional *fold* and *map*, we turn to bidirectional *scan*, which is an efficient computation pattern using an accumulation parameter and is useful for improving efficiency (as will be seen later). The main challenge to bidirectionalize *scan* is that the result of *scan* may have constraints between adjacent elements, which cannot be achieved by traditional lenses without contracts. We shall give a powerful bidirectional version of *scan* with the help of contract lenses.

$$\begin{array}{lll}
bscanl\ : & b_0\ : V & \\
\to & \ell\ : (b : V \to (\{\lambda_\ t' \to (t' = Right\ (_, b)) \wedge keepsame\ \ell\ b\} & \\
& \qquad Either\ ()\ (S, V) \leftrightarrow V & \\
& \qquad \{\lambda_\ b' \to \widetilde{cv}\ b\ b'\})) & \\
\to & (\{eqlength\}\ [S] \leftrightarrow [V] & \\
& \quad \{\lambda t\ bs \to \forall\ 1 \le i \le |bs|, \widetilde{cv}\ bs_{i-1}\ bs_i \wedge eqlength\ t\ bs\}) &
\end{array}$$

$$\begin{array}{l}
bscanl\ b_0\ \ell = CLens\ g\ p \\
\quad \textbf{where} \\
\qquad g \qquad = scanl'\ (\lambda b\ a \to get_{\ell\ b}\ (Right\ (a, b)))\ b_0 \\
\qquad p\ as\ bs' = map\ (\lambda((a, b), (b'', b')) \to fstRight\ (put_{\ell\ b''}\ (Right\ (a, b))\ b'))\ abb
\end{array}$$

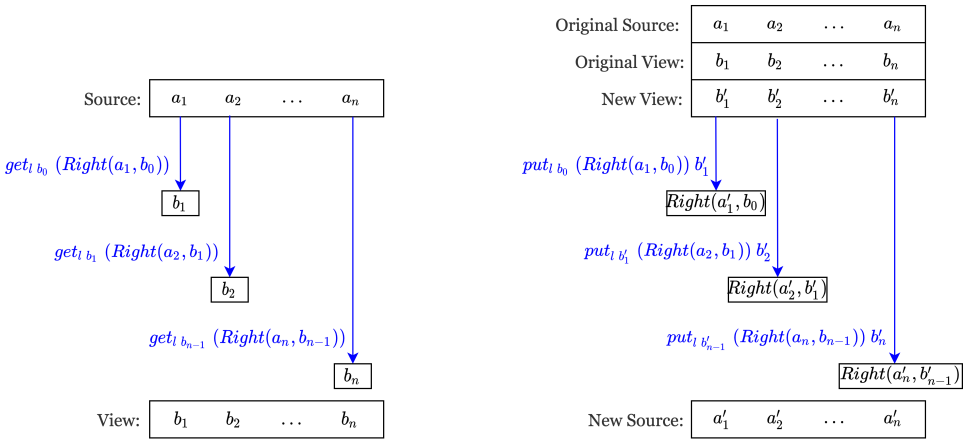

Fig. 2. Implementation of $bscanl\ a_0\ \ell$. The left figure shows the computation of the *get* and the right figure shows the computation of the *put*.

$$\begin{aligned}&\textbf{where}\ bs \quad = g\ as\\ &\qquad abb = zip\ (zip\ as\ (b_0 : init\ bs))\ (zip\ (b_0 : init\ bs')\ bs')\\ &\qquad fstRight\ (Right\ (x, \_)) = x\end{aligned}$$

The constraint $keepsame\ \ell\ b$ in $cs_{\ell\ b}$ guarantees that the forward transformation $get_{\ell\ b''}$ for any $b'' : V$ is the same function $f : Either\ ()\ (S, V) \to V$. Thus, $bscanl\ b_0\ \ell$ can be considered as a special bidirectional version of $scanl'\ (\lambda b\ a \to f\ (Right\ (a, b)))\ b_0$.

Let us explain the idea of the *bscanl*. The implementation of $bscanl\ b_0\ \ell$ is depicted in Figure 2. The forward transformation is the same as the unidirectional $scanl'$. For the backward transformation, we need to take the constraints on the view into consideration. The view condition $cv_{bscanl\ b_0\ \ell}$ describes the constraints on adjacent elements by $\widetilde{cv}$. Suppose that the original source is $[a_1, \ldots, a_n]$, the original view is $[b_1, \ldots, b_n]$ and the updated view is $[b'_1, \ldots, b'_n]$, we want to get the updated source $[a'_1, \ldots, a'_n]$. When computing $t' = put_{\ell\ b'_{i-1}}\ (Right\ (a_i, b_{i-1}))\ b'_i$, ℓ additionally takes b'_{i-1} in order to make sure that the condition $t = Right\ (\_, b'_{i-1})$ is satisfied. Otherwise, the ConditionedGetPut property of $bscanl\ b_0\ \ell$ may be broken.

Example 5.7 (Bidirectional Prefix Products of Integers)
Consider that we want to synchronize a list of integers with its prefix products. The forward transformation is characterized by $prefixProd = scanl'\ (*)\ 1 : [Int] \to [Int]$. 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 *bscanl*.

$$\begin{aligned}&bprefixProd : (\{eqlength\}\ [Int] \leftrightarrow [Int]\\ &\qquad\qquad \{\lambda t\ bs \to (\forall\ 1 < i \le |bs|, mod\ bs_i\ bs_{i-1} = 0)\ \wedge\ eqlength\ t\ bs\})\\ &bprefixProd = bscanl\ 1\ bmul\\ &\quad \textbf{where}\\ &\qquad bmul : (b : Int) \to (\{\lambda \_\ t' \to (t' = Right\ (\_, b))\ \wedge\ keepsame\ bmul\ b\}\\ &\qquad\qquad\qquad Either\ ()\ (Int, Int) \leftrightarrow Int\ \{\lambda \_\ b' \to mod\ b'\ b = 0\})\end{aligned}$$

$$\begin{array}{lll} bmul\ b = CLens\ g\ p\ \textbf{where}\ g\ (Left\ ()) & = 1 \\ \qquad\qquad\qquad\qquad\quad g\ (Right\ (x,y)) & = x * y \\ \qquad\qquad\qquad\qquad\quad p\ \_\ b' & = Right\ (div\ b'\ b, b) \end{array}$$ □

6 Bidirectional Calculation Laws

So far, we have seen that the important higher-order functions such as *fold*, *filter*, *map* and *scan* can be extended naturally from "unidirectional" to "bidirectional", and that these bidirectional versions can be used to describe various bidirectional behaviors through suitable definitions of *get*, *put*, and the source/view conditions. In this section, we shall develop 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 $\mathrm{F_L}$ defined in Section 2.2.

$$\begin{array}{l} blistF\ :\ V \\ \qquad \to (\{ctrue\}\ V \leftrightarrow T\ \{ctrue\}) \\ \qquad \to (\{ctrue\}\ (Either\ ()\ (S,V)) \leftrightarrow (Either\ ()\ (S,T))\ \{ctrue\}) \\ blistF\ b_0\ \ell = CLens\ g\ p \\ \quad \textbf{where} \\ \qquad \begin{array}{ll} g\ (Left\ ()) & = Left\ () \\ g\ (Right\ (a,b)) & = Right\ (a, get_\ell\ b) \\ p\ \_\ (Left\ ()) & = Left\ () \\ p\ (Right\ (a,b))\ (Right\ (a',c')) & = Right\ (a', put_\ell\ b\ c') \\ p\ (Left\ ())\ (Right\ (a',c')) & = Right\ (a', put_\ell\ b_0\ c') \end{array} \end{array}$$

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.

$$\frac{\ell_1;\ell = blistF\ (get_{\ell_1}\ (Left\ ()))\ \ell;\ell_2}{bfoldr\ \ell_1;\ell = bfoldr\ \ell_2}$$ BFoldr Fusion

It reads that the lens composition $bfoldr\ \ell_1;\ell$ can be fused into a single lens $bfoldr\ \ell_2$ if there exists ℓ_2 such that the equation $\ell_1;\ell = blistF\ (get_{\ell_1}\ (Left\ ()))\ \ell;\ell_2$ holds. Note that when the contracts can be inferred from the context, we just omit them for simplicity. Indeed for the BFoldr Fusion law, all omitted source/view conditions are just *ctrue*.

Similarly, we have another fusion law for *bfoldr′*, for which we need a slightly different bidirectional version of the list functor $\mathrm{F_L}$. 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.

$$
\begin{aligned}
&\mathit{blistF'} \;:\; (\{\mathit{cv}\}\ V \leftrightarrow T\ \{\mathit{ct}\}) \\
&\qquad \to (\{\mathit{lift\ cs\ cv}\}\ (\mathit{Either}\ ()\ (S,V)) \leftrightarrow (\mathit{Either}\ ()\ (S,T))\ \{\mathit{lift\ cs\ ct}\}) \\
&\mathit{blistF'}\ \ell = \mathit{CLens}\ g\ p \\
&\quad \textbf{where} \\
&\qquad g\ (\mathit{Left}\ ()) = \mathit{Left}\ () \\
&\qquad g\ (\mathit{Right}\ (a,b)) = \mathit{Right}\ (a, \mathit{get}_\ell\ b) \\
&\qquad p\ (\mathit{Left}\ ())\ (\mathit{Left}\ ()) = \mathit{Left}\ () \\
&\qquad p\ (\mathit{Right}\ (a,b))\ (\mathit{Right}\ (a',c')) = \mathit{Right}\ (a', \mathit{put}_\ell\ b\ c')
\end{aligned}
$$

Then, the fusion law is stated as

$$\frac{\ell_1;\ell = \mathit{blistF}\ \ell;\ell_2}{\mathit{bfoldr}\ \ell_1;\ell = \mathit{bfoldr}\ \ell_2} \qquad \text{BFoldr' Fusion}$$

6.2 Bidirectional Map Fusion

Recall that one special version of the Fold Fusion law is the Map Fusion law. The bidirectional map fusion laws for *bmap* and *bmap′* are quite easy since they just map ℓ 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′*.

$$\mathit{bmap'}\ \ell_1;\mathit{bmap'}\ \ell_2 = \mathit{bmap'}\ (\ell_1;\ell_2) \qquad \text{BMap' Fusion}$$

Note that the equation holds when the compositions on the left-hand side and the right-hand side satisfy the condition of composition defined in 4.2.

Similarly, we can give the bidirectional map fusion law for *bmapl* (and *bmapr*):

$$\mathit{bmapl}\ a_0\ \ell_1;\mathit{bmapl}\ b_0\ \ell_2 = \mathit{bmapl}\ a_0\ (\ell_1;;\ell_2) \qquad \text{BMapl Fusion}$$

where $(;;)$ is the composition of the functions with types $S \to (S \leftrightarrow V)$.

$$
\begin{aligned}
(;;)\ \;:\;\ & \ell_1 : ((a:S) \to (\{\lambda\_\ a' \to \widetilde{cs}\ a\ a'\ \wedge\ \mathit{keepsame}\ \ell_1\ a\} \\
& \qquad\qquad S \leftrightarrow V \\
& \qquad\qquad \{\lambda\_\ b' \to \widetilde{cv}\ (\mathit{get}_{\ell_1\ a}\ a)\ b'\})) \\
& \to \ell_2 : ((b:V) \to (\{\lambda\_\ b' \to \widetilde{cv}\ b\ b'\ \wedge\ \mathit{keepsame}\ \ell_2\ b\} \\
& \qquad\qquad V \leftrightarrow T \\
& \qquad\qquad \{\lambda\_\ c' \to \widetilde{ct}\ (\mathit{get}_{\ell_2\ b}\ b)\ c'\})) \\
& \to ((a:S) \to \quad (\{\lambda\_\ a' \to \widetilde{cs}\ a\ a'\ \wedge\ \mathit{keepsame}\ (\ell_1;;\ell_2)\ a\} \\
& \qquad\qquad S \leftrightarrow T \\
& \qquad\qquad \{\lambda\_\ c' \to \widetilde{ct}\ (\mathit{get}_{\ell_1;;\ell_2\ a}\ a)\ c'\})) \\
\ell_1;;\ell_2 = {} & \lambda a \to \ell_1\ a;\ell_2\ (\mathit{get}_{\ell_1\ a}\ a)
\end{aligned}
$$

Let us briefly explain why the definition of $(;;)$ is correct, i.e., for any $a:S$ the result $(\ell_1;;\ell_2)\ a$ is a valid contract lens satisfying the round-tripping properties

in Definition 4.1 if the composition $\ell_1\ a; \ell_2\ (get_{\ell_1\ a}\ a)$ is valid. This is not obvious because the source condition of $(\ell_1;;\ell_2)\ a$ is $\lambda\_\ a' \to \widetilde{cs}\ a\ a' \wedge keepsame\ (\ell_1;;\ell_2)\ a$ while the source condition of $\ell_1\ a; \ell_2\ (get_{\ell_1\ a}\ a)$ is $\lambda\_\ a' \to \widetilde{cs}\ a\ a' \wedge keepsame\ \ell_1\ a$. However, because of the condition of composition in Definition 4.2, the condition $\forall\ b' : V, (\widetilde{cv}\ (get_{\ell_1\ a}\ a)\ b') \Rightarrow (\widetilde{cv}\ (get_{\ell_1\ a}\ a)\ b' \wedge keepsame\ \ell_2\ (get_{\ell_1\ a}\ a))$ is required to hold. For any $a : S$, the condition $\widetilde{cv}\ (get_{\ell_1\ a}\ a)\ b'$ should hold for at least one $b' : V$, otherwise the lens is meaningless as its round-tripping property is always broken, which we do not need to care about. Thus, we can derive that $keepsame\ \ell_2\ (get_{\ell_1\ a}\ a)$ should always be true. From $keepsame\ \ell_1\ a$ and $keepsame\ \ell_2\ (get_{\ell_1\ a}\ a)$ we can conclude that $keepsame\ (\ell_1;;\ell_2)\ a$ is true, and therefore we can guarantee that the contract lens $(\ell_1;;\ell_2)\ a$ for any $a : S$ satisfies the round-tripping properties.

6.3 Bidirectional Fold-Map Fusion

Recall that a special version of the Fold Fusion law is the Fold-Map Fusion law. We shall give a bidirectional fold-map fusion law for *bfoldr′* and *bmap′*, both of which preserve the length of the source list.

First, we bidirectionalize $\mathrm{F_m}$ defined in Section 2.2 with suitable conditions required by *bfoldr′*.

$$
\begin{aligned}
&bmapF\ :\ (\{cs\}\ S \leftrightarrow V\ \{cv\}) \\
&\qquad \to (\{lift\ cs\ ct\}\ (Either\ ()\ (S,T)) \leftrightarrow (Either\ ()\ (V,T))\ \{lift\ cv\ ct\}) \\
&bmapF\ \ell = CLens\ g\ p \\
&\quad \textbf{where} \\
&\qquad g\ (Left\ ()) &&= Left\ () \\
&\qquad g\ (Right\ (a,c)) &&= Right\ (get_\ell\ a, c) \\
&\qquad p\ (Left\ ())\ (Left\ ()) &&= Left\ () \\
&\qquad p\ (Right\ (a,c))\ (Right\ (b',c')) &&= Right\ (put_\ell\ a\ b', c')
\end{aligned}
$$

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′*.

$$bmap'\ \ell_1; bfoldr'\ \ell_2 = bfoldr'\ (bmapF\ \ell_1; \ell_2) \qquad \text{BFoldr'-BMap Fusion}$$

6.4 Bidirectional Scan Lemma

The Scan Lemma is also 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 the adjacent elements of the view list. Fortunately, the contract-lens combinator *bmapl* can handle the constraints on the adjacent elements. With *bmapl*, $bfoldl_{init}$ and *bscanl*, we can successfully obtain a bidirectional version of scan lemma.

$$binits; bmapl\ [\,]\ (bfoldl_{init}\ b_0\ \ell) = bscanl\ b_0\ \ell \qquad \text{Bidirectional Scan Lemma}$$

It may be surprising to see that the form of bidirectional scan lemma is quite similar to the unidirectional scan lemma.

Example 6.1 (Optimization of Bidirectional Prefix Sums)
Now we give a simple example which makes use of the Bidirectional Scan Lemma to derive an efficient bidirectional program from an inefficient unidirectional program. Recall "$\mathit{binits}; \mathit{bmapl}\,[\,]\,(\mathit{bfoldl}_{\mathit{init}}\,0\,\mathit{badd})$" defined in Example 5.6 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 "$\mathit{bscanl}\,0\,\mathit{badd}$", which has time complexity $O(n)$ in both forward and backward transformations. □

7 Examples

In this section, we will demonstrate further through three examples that with contract lens, 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 on Hyperplane

Let us look at an example to see the expressive power of contract lenses, especially how we can use the source/view condition 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 $xs = [x_1, x_2, \ldots, x_n]$ [7] in the n-dimensional Euclidean space with the projection of it onto the hyperplane $H : \sum_{i=1}^{n} x_i = 0$. Let $m = \frac{1}{n}\sum_{i=1}^{n} x_i$, the projection of X onto H is the point $ys = [x_1 - m, x_2 - m, \ldots, x_n - m]$. 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 direction of the synchronization is on the hyperplane H'. In another word, 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 first write a lens to synchronize the list with a pair of the mean of it and the original list, and then write another lens to synchronize between this pair and the list of differences. Note that we have two constraints saying that the dimension n and the hyperplane H'

[7] Here we use a list of length n to represent a point in a n-dimensional space.

should not be changed, which can be explicitly expressed using contracts. The full implementation looks like this:

$$
\begin{aligned}
&\mathit{bproj} \;:\{\lambda xs\ xs' \to \mathit{mean}\ xs = \mathit{mean}\ xs' \wedge |xs| = |xs'|\}\\
&\qquad [\mathit{Float}] \leftrightarrow [\mathit{Float}]\\
&\qquad \{\lambda xs\ xs' \to \mathit{sum}\ xs' = 0 \wedge |xs| = |xs'|\}\\
&\mathit{bproj} \;= \mathit{bmean}; \mathit{bdiff}\\
&\mathit{bmean}:\{\lambda xs\ xs' \to \mathit{mean}\ xs = \mathit{mean}\ xs' \wedge |xs| = |xs'|\}\\
&\qquad [\mathit{Float}] \leftrightarrow (\mathit{Float}, [\mathit{Float}])\\
&\qquad \{\lambda (m,xs)\ (m',xs') \to m = m' \wedge m = \mathit{mean}\ xs \wedge m' = \mathit{mean}\ xs' \wedge |xs| = |xs'|\}\\
&\mathit{bmean} = \mathit{CLens}\ g\ p\ \textbf{where}\ g\ xs = (\mathit{mean}\ xs, xs)\\
&\qquad\qquad\qquad\qquad p\ \_\ (m,xs') = xs'\\
&\mathit{bdiff} \;:\{\lambda (m,xs)\ (m',xs') \to m = m' \wedge m = \mathit{mean}\ xs \wedge m' = \mathit{mean}\ xs' \wedge |xs| = |xs'|\}\\
&\qquad (\mathit{Float}, [\mathit{Float}]) \leftrightarrow [\mathit{Float}]\\
&\qquad \{\lambda xs\ xs' \to \mathit{sum}\ xs' = 0 \wedge |xs| = |xs'|\}\\
&\mathit{bdiff} \;= \mathit{CLens}\ g\ p\ \textbf{where}\ g\ (m,xs) = \mathit{map}\ (+(-m))\ xs\\
&\qquad\qquad\qquad\qquad p\ (m,\_)\ xs' = (m, \mathit{map}\ (+m)\ xs')\\
&\mathit{mean}:[\mathit{Float}] \to \mathit{Float}\\
&\mathit{mean}\ xs = \mathit{sum}\ xs\ /\ \mathit{fromIntegral}\ (\mathit{length}\ xs)
\end{aligned}
$$

The specifications of synchronization behavior on each lenses are clearly expressed by the conditions, which enables the compositions as we see in the definition.

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 & Wang, 2015b). The novelty 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 digital characters, 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 a string are either numbers or letters):

$$
\begin{aligned}
&\mathit{bformatting} = \mathit{bfilter}\ (\mathit{not} \circ \mathit{isDigit}); \mathit{bmap}'\ \mathit{btoUpper}\\
&\quad \textbf{where}\\
&\qquad \mathit{btoUpper} :: \{\lambda \_\ c \to \mathit{not}\ (\mathit{isDigit}\ c)\}\ \mathit{Char} \leftrightarrow \mathit{Char}\ \{\lambda \_\ c \to \mathit{isUpper}\ c\}\\
&\qquad \mathit{btoUpper} = \mathit{CLens}\ \mathit{toUpper}\ \mathit{putToLower}\\
&\mathit{putToLower}\ x\ y = \textbf{if}\ \mathit{isUpper}\ x\ \textbf{then}\ y\ \textbf{else}\ \mathit{toLower}\ y
\end{aligned}
$$

where the types and contracts of the two lenses are as follows:

$$\begin{aligned}&\mathit{bfilter}\ (\mathit{not} \circ \mathit{isDigit}) :: \{\mathit{eqlength}\}\ \mathit{String} \leftrightarrow \mathit{String}\ \{\mathit{fcond}\ (\mathit{not} \circ \mathit{isDigit})\}\\&\mathit{bmap}'\ \mathit{btoUpper} :: \{\mathit{licond}\ (\lambda \_\ c \to \mathit{not}\ (\mathit{isDigit}\ c))\}\\&\qquad\qquad\qquad \mathit{String} \leftrightarrow \mathit{String}\\&\qquad\qquad\qquad \{\mathit{licond}\ (\lambda \_\ c \to \mathit{isUpper}\ c)\}\end{aligned}$$

The composition is valid, since $\mathit{fcond}\ (\mathit{not} \circ \mathit{isDigit})$ and $\mathit{licond}\ (\lambda \_\ c \to \mathit{not}\ (\mathit{isDigit}\ c))$ are by definition equivalent.

In this naive specification, intermediate structures are created after one lens, and are immediately consumed by another, in both direction. Recall that *bfilter* is an instance of *bfoldr′*, using BFoldr' Fusion, we reason:

$$\begin{aligned}&\quad \mathit{bformatting}\\&= \quad \{\ \text{definition}\ \}\\&\quad \mathit{bfilter}\ (\mathit{not} \circ \mathit{isDigit})\\&\quad ; \mathit{bmap}'\ \mathit{btoUpper}\\&= \quad \{\ \text{BFoldr' Fusion}\ \}\\&\quad \mathit{bfoldr}'\ \mathit{balg}\end{aligned}$$

where

$$\begin{aligned}&\mathit{balg} :: \{\mathit{lift}\ \mathit{ctrue}\ (\mathit{licond}\ (\lambda \_\ c \to \mathit{isUpper}\ c))\}\\&\qquad (\mathit{Either}\ ()\ (\mathit{Char}, [\mathit{Char}])) \leftrightarrow [\mathit{Char}]\\&\qquad \{\mathit{licond}\ (\lambda \_\ c \to \mathit{isUpper}\ c)\}\\&\mathit{balg} = \mathit{CLens}\ g\ p\\&\quad \textbf{where}\\&\qquad g\ (\mathit{Left}\ ()) = [\,]\\&\qquad g\ (\mathit{Right}\ (x, xs)) = \textbf{if}\ \mathit{not}\ (\mathit{isDigit}\ x)\ \textbf{then}\ \mathit{toUpper}\ x : xs\ \textbf{else}\ xs\\&\qquad p\ (\mathit{Left}\ ())\ \_ = \mathit{Left}\ ()\\&\qquad p\ (\mathit{Right}\ (x, xs))\ (x' : xs') = \textbf{if}\ \mathit{not}\ (\mathit{isDigit}\ x)\\&\qquad\quad \textbf{then}\ \mathit{Right}\ (\mathit{putToLower}\ x\ x', xs')\\&\qquad\quad \textbf{else}\ \mathit{Right}\ (x, x' : xs')\\&\qquad p\ (\mathit{Right}\ (x, [\,]))\ [\,] = \textbf{if}\ \mathit{not}\ (\mathit{isDigit}\ x)\ \textbf{then}\ \mathit{Left}\ ()\ \textbf{else}\ \mathit{Right}\ (x, [\,])\end{aligned}$$

The definition of *balg* is not as complicated as it seems: it is essentially the combination of *bfilterAlg* and *btoUpper*.

It is easy to verify the condition of the BFoldr' Fusion law, which is a lens equivalence $\mathit{bfilterAlg}\ (\mathit{not} \circ \mathit{isDigit}); \mathit{bmap}'\ \mathit{btoUpper} = \mathit{blistF}'\ (\mathit{bmap}'\ \mathit{btoUpper}); \mathit{balg}$. The calculated version creates no intermediate structure and hence is more efficient.

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 & Wang, 2020). Let us consider the following simple string encoding algorithm which illustrates the idea of Run Length Encoding.

$$
\begin{array}{l}
\mathit{compression} : [\mathit{String}] \to [\mathit{Int}] \\
\mathit{compression} = \mathit{foldr}'\ \mathit{cat} \circ \mathit{map}\ \mathit{ascii} \circ \mathit{map}\ \mathit{encode} \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{encode} & = (\mathit{head}\ \mathit{ws}, \mathit{length}\ \mathit{ws}) \\
\mathit{ascii}\ (x,y) & = (\mathit{ord}\ x, y) \\
\mathit{cat}\ (\mathit{Left}\ ()) & = [\,] \\
\mathit{cat}\ (\mathit{Right}\ ((x,y),b)) & = x : y : b
\end{array}
\end{array}
$$

For simplicity, the input string are already splitted into a list of strings, which consist 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 $\mathit{foldr}'\ \mathit{cat}$ concatenates the pairs to a single list. For example, [8]

$$\mathit{compression}\ [\texttt{"aaaaa"}, \texttt{"bbbb"}, \texttt{"ccccccccc"}] = [97,5,98,4,99,9]$$

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 $\mathit{cv}_{\mathit{compress}}\ v\ \mathit{as} = (|v| = |\mathit{as}|) \wedge (\forall\, 1 \le i \le |\mathit{as}|, \mathit{odd}\ i \vee (0 \le \mathit{as}_i < 128))$.

$$
\begin{array}{l}
\mathit{bcompression} : \{\lambda\_\ \mathit{as} \to \forall\, 1 \le i \le |\mathit{as}|, \mathit{allsame}\ \mathit{as}_i\}\ [\mathit{String}] \leftrightarrow [\mathit{Int}]\ \{\mathit{cv}_{\mathit{compress}}\} \\
\mathit{bcompression} = \mathit{bmap}'\ \mathit{bencode} \\
\qquad\qquad\quad ;\ \ \mathit{bmap}'\ \mathit{bascii} \\
\qquad\qquad\quad ;\ \ \mathit{bfoldr}'\ \mathit{bcat} \\
\quad \textbf{where} \\
\qquad \mathit{allsame}\ \mathit{xs} = \mathit{and}\ \$\ \mathit{map}\ (= \mathit{head}\ \mathit{xs})\ (\mathit{tail}\ \mathit{xs})
\end{array}
$$

where the following contract lenses are used

$$
\begin{array}{l}
\mathit{bencode} : \{\lambda\_\ \mathit{as} \to \mathit{allsame}\ \mathit{as}\}\ \mathit{String} \leftrightarrow (\mathit{Char}, \mathit{Int})\ \{\mathit{ctrue}\} \\
\mathit{bencode} = \mathit{CLens}\ (\lambda \mathit{ws} \to (\mathit{head}\ \mathit{ws}, \mathit{length}\ \mathit{ws}))\ (\lambda\_\ (a,n) \to \mathit{replicate}\ n\ a) \\
\mathit{bascii} : \{\mathit{ctrue}\}\ (\mathit{Char}, b) \leftrightarrow (\mathit{Int}, b)\ \{\lambda\_\ (x,\_) \to 0 \le x < 128\} \\
\mathit{bascii} = \mathit{CLens}\ (\lambda (x,y) \to (\mathit{ord}\ x, y))\ (\lambda\_\ (x,y) \to (\mathit{chr}\ x, y)) \\
\mathit{bcat} : \{\mathit{lift}\ (\lambda\_\ (x,\_) \to 0 \le x < 128)\ \mathit{cv}_{\mathit{compress}}\}\ \mathit{Either}\ ()\ ((\mathit{Int}, \mathit{Int}), [\mathit{Int}]) \leftrightarrow [\mathit{Int}]\ \{\mathit{cv}_{\mathit{compress}}\} \\
\mathit{bcat} = \mathit{CLens}\ g\ p \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
g\ (\mathit{Left}\ ()) & = [\,] \\
g\ (\mathit{Right}\ ((x,y),b)) & = x : y : b \\
p\ (\mathit{Left}\ ())\ [\,] & = \mathit{Left}\ () \\
p\ (\mathit{Right}\ \_)\ (x : y : b) & = \mathit{Right}\ ((x,y),b)
\end{array}
\end{array}
$$

[8] 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.

With the types and contracts of the components, it is easy to check the contract lens *bcompression* is lawful. 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 traversial simultaneously.

$$
\begin{array}{ll}
 & bcompression \\
= & \quad \{ \text{ definition } \} \\
 & bmap' \; bencode \\
 & ; bmap' \; bascii \\
 & ; bfoldr' \; bcat \\
= & \quad \{ \text{ BMap' Fusion } \} \\
 & bmap' \; (bencode ; bascii) \\
 & ; bfoldr' \; bcat \\
= & \quad \{ \text{ BFoldr'-BMap Fusion } \} \\
 & bfoldr' \; (bmapF \; (bencode ; bascii) ; bcat)
\end{array}
$$

7.3 Bidirectional Maximum Segment Sum

Now let us turn to another example involving more advanced program calculation. 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 are 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 $TailsList\ a = \{as : [[a]] \mid (\forall\ 1 \leq i < n, tail\ as_i = as_{i+1}) \wedge (tail\ as_n = [])\}$. 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

$$
\begin{array}{l}
bmss \ : \ \{eqlength\}\ [Int] \leftrightarrow Int\ \{\lambda v\ v' \rightarrow (v = -\infty) \Rightarrow (v' = -\infty)\} \\
bmss = binits \\
\qquad ; \ bmapl\ []\ btails_{init} \\
\qquad ; \ bmapl\ []\ bmapSum \\
\qquad ; \ bmapl\ []\ bmaximum2 \\
\qquad ; \ bmaximum'
\end{array}
$$

where the definitions of the contract lenses appeared are

$$
\begin{array}{l}
btails_{init} : a : [Int] \rightarrow (\{\lambda \_\ a' \rightarrow (init\ a' = a) \wedge keepsame\ btails_{init}\ a\} \\
\qquad\qquad\qquad\qquad [Int] \leftrightarrow TailsList\ Int \\
\qquad\qquad\qquad\qquad \{\lambda \_\ b' \rightarrow map\ init\ (init\ b') = b\}) \\
btails_{init} = CLens\ tails'\ (\lambda \_\ v \rightarrow head\ v)
\end{array}
$$

$$
\begin{aligned}
&\mathit{bmapSum} : a : \mathit{TailsList}\ \mathit{Int} \to (\{\lambda\_\ a' \to (\mathit{map}\ \mathit{init}\ (\mathit{init}\ a') = a)\ \wedge\ \mathit{keepsame}\ \mathit{bmapSum}\ a\} \\
&\qquad\qquad\qquad\qquad \mathit{TailsList}\ \mathit{Int} \leftrightarrow [\mathit{Int}] \\
&\qquad\qquad\qquad\qquad \{\lambda\_\ b' \to \mathit{map}\ (-\mathit{last}\ b')\ (\mathit{init}\ b') = b\}) \\
&\mathit{bmapSum}\ \mathit{xss} = \mathit{CLens}\ (\mathit{map}\ \mathit{sum})\ p \\
&\quad \textbf{where}\ p\ \_\ \mathit{xs} = \mathit{map}\ (\lambda t \to t +\!\!+ [\mathit{last}\ \mathit{xs}])\ \mathit{xss} +\!\!+ [[\mathit{last}\ \mathit{xs}]]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bmaximum2}\ :\ a : [\mathit{Int}] \\
&\qquad\qquad \to (\{\lambda\_\ a' \to (\mathit{map}\ (-\mathit{last}\ a')\ (\mathit{init}\ a') = a)\ \wedge\ \mathit{keepsame}\ \mathit{bmaximum2}\ a\} \\
&\qquad\qquad\quad [\mathit{Int}] \leftrightarrow \mathit{Int}\ \{\mathit{ctrue}\}) \\
&\mathit{bmaximum2}\ a = \mathit{CLens}\ \mathit{maximum}\ p \\
&\quad \textbf{where}\ p\ \_\ x\ \ = \textbf{let}\ t = a +\!\!+ [0]\ \textbf{in}\ \mathit{map}\ (+(x - \mathit{maximum}\ t))\ t
\end{aligned}
$$

The *binits* and $\mathit{bmaximum}'$ have been already defined in the previous sections. It is easy to check that the condition of composition is satisfied.

Next, we make use of the bidirectional calculation rules we developed in Section 6 to optimize the *bmss*. The calculation goes as follows.

$$
\begin{aligned}
&\quad \mathit{bmss} \\
&= \quad \{\ \text{definition}\ \} \\
&\quad \mathit{binits} \\
&\quad ;\mathit{bmapl}\ [\,]\ \mathit{btails}_{\mathit{init}} \\
&\quad ;\mathit{bmapl}\ [\,]\ \mathit{bmapSum} \\
&\quad ;\mathit{bmapl}\ [\,]\ \mathit{bmaximum2} \\
&\quad ;\mathit{bmaximum}' \\
&= \quad \{\ \text{BMapl Fusion}\ \} \\
&\quad \mathit{binits} \\
&\quad ;\mathit{bmapl}\ [\,]\ (\mathit{btails}_{\mathit{init}} ;; \mathit{bmapSum} ;; \mathit{bmaximum2}) \\
&\quad ;\mathit{bmaximum}' \\
&= \quad \{\ \text{a specific bidirectional Horner's rule (to be discussed below)}\ \} \\
&\quad \mathit{binits} \\
&\quad ;\mathit{bmapl}\ [\,]\ (\mathit{bfoldl}_{\mathit{init}}\ (-\infty)\ \ell) \\
&\quad ;\mathit{bmaximum}' \\
&= \quad \{\ \text{Bidirectional Scan Lemma}\ \} \\
&\quad \mathit{bscanl}\ (-\infty)\ \ell \\
&\quad ;\mathit{bmaximum}'
\end{aligned}
$$

Here, ℓ is defined as follows.

$$
\begin{aligned}
&\ell\ :\ b : \mathit{Int} \\
&\quad \to (\{\lambda\_\ t' \to (t' = \mathit{Right}\ (\_, b))\ \wedge\ \mathit{keepsame}\ \ell\ b\}\ \mathit{Either}\ \mathit{Int}\ (\mathit{Int}, \mathit{Int}) \leftrightarrow \mathit{Int}\ \{\mathit{ctrue}\}) \\
&\ell\ b = \mathit{CLens}\ g\ p \\
&\quad \textbf{where}\ g\ (\mathit{Left}\ ()) \qquad = -\infty \\
&\qquad\quad g\ (\mathit{Right}\ (x, y)) = \mathit{max}\ (x + y)\ x \\
&\qquad\quad p\ \_\ t \qquad\quad = \mathit{Right}\ (t - \mathit{max}\ b\ 0, b)
\end{aligned}
$$

The types and source/view conditions at every step are not shown in the above calculation process for simplicity. One thing to note is that in the third step of calculation we use a specific bidirectional Horner's rule:

$$\mathit{btails}_{\mathit{init}};;\mathit{bmapSum};;\mathit{bmaximum2} = \mathit{bfoldl}_{\mathit{init}}\ (-\infty)\ \ell$$

The get direction of $(\mathit{btails}_{\mathit{init}};;\mathit{bmapSum};;\mathit{bmaximum2})\ a$ for any $a : [\mathit{Int}]$ is similar to the original Horner's rule with $\otimes = +$ and $\oplus = \mathit{max}$. 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.

8 Discussions and Related Works

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 in (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 & Wang, 2015a), HOBiT (Matsuda & 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 Type Systems to Ensure Totality

The totality of lenses can be ensured by advanced type systems. 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 argued in the paper, totality is not always desirable, as the effort in achieving it necessarily complicates program design. As far as we know, there is no existing type systems that can be readily used to flexibly express the bidirectional behaviors we see in this paper. Take *bmap* as an example. To express the "equal length" condition, one would need a new system of dependent lens arrows. Absence of this, one can

try to encode *bmap* indirectly with a notion of dependent/refinement types into something like the following.

$$bmap : (n : \mathbb{N}) \to (A \leftrightarrow B) \to (\{xs : [A] \mid |xs| = n\} \leftrightarrow \{ys : [B] \mid |ys| = n\})$$

This version fixes the length of the lists, which is obviously less general.

8.1.2 Edit Lenses

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

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 & Wang, 2015a; Ko et al., 2016; Xia et al., 2019). The *put* direction is a total function of type $s \to v \to Maybe\ s$ 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. The exact specification of a program must be known statically for the calculation laws to be semantic preserving.

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 behavior of partial lenses.

8.2 Hoare-style Reasoning of Bidirectional Transformation

In (Ko & Hu, 2018), a reasoning framework for BiGUL programs based on Hoare logic is proposed, which are able to precisely characterize the bidirectional behaviors 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 behavior of *put* in a way similar to the Hoare logic: if the original source

and the view satisfy the precondition R, then put_b will successfully produce an updated source satisfying the postcondition R'.

Although the notation of pre- and post-condition is similar to our source and view condition in form, their uses are completely different. They do not indicate under what conditions the computation of *put* is safe, while our source and view conditions do. For contract lenses, when the condition $cv\ (get\ s)\ v$ holds, $put\ s\ v$ can be computed safely and the PutGet law is satisfied.

It is also worth mentioning that in their framework reasoning about lens composition is very difficult. 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 4.2.

8.3 Bidirectionalization

Bidirectionalization is an approach to bidirectional programming that is different from the lens framework. Instead of writing bidirectional programs directly in a specialized language, it aims to mechanically covert existing unidirectional programs into bidirectional ones (Voigtländer, 2009). This work gives a high-order function *bff* that receives a polymorphic *get* function, and returns its *put* counterpart. The technique is extended (Voigtländer et al., 2010) by combining it with syntactic bidirectionalization (Matsuda et al., 2007), which separates view changes in shape and in content.

However, bidirectionalization is done for whole programs which does not provide ways to reason about compositions.

8.4 Calculating with Lenses

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

But very different from ours, their work is based on the traditional lenses without contracts, which means that the problem of partiality seriously limits the composition of lenses. As a result, many crucial calculation laws such as the Scan Lemma are not expressible in their framework.

9 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, 1989a; Bird, 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 automate the verification of round-tripping properties as much as possible using SMT solvers.

References

Abou-Saleh, Faris, Cheney, James, Gibbons, Jeremy, McKinna, James, & Stevens, Perdita. (2018). Introduction to bidirectional transformations. Cham: Springer International Publishing. Pages 1–28.

Bancilhon, F., & Spyratos, N. (1981). Update semantics of relational views. Acm trans. database syst., 6(4), 557â 575.

Barbosa, Davi M.J., Cretin, Julien, Foster, Nate, Greenberg, Michael, & Pierce, Benjamin C. (2010). Matching lenses: Alignment and view update. Page 193â 204 of: Proceedings of the 15th acm sigplan international conference on functional programming. ICFP '10. New York, NY, USA: Association for Computing Machinery.

Bird, Richard S. (1987). An introduction to the theory of lists. Pages 5–42 of: Broy, Manfred (ed), Logic of programming and calculi of discrete design. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bird, Richard S. (1989a). Algebraic identities for program calculation. Comput. j., 32(2), 122â 126.

Bird, Richard S. (1989b). Lectures on constructive functional programming. Pages 151–217 of: Broy, Manfred (ed), Constructive methods in computing science. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bohannon, Aaron, Pierce, Benjamin C., & Vaughan, Jeffrey A. (2006). Relational lenses: A language for updatable views. Page 338â 347 of: Proceedings of the twenty-fifth acm sigmod-sigact-sigart symposium on principles of database systems. PODS '06. New York, NY, USA: Association for Computing Machinery.

Bohannon, Aaron, Foster, J. Nathan, Pierce, Benjamin C., Pilkiewicz, Alexandre, & Schmitt, Alan. (2008). Boomerang: Resourceful lenses for string data. Page 407â 419 of: Proceedings of the 35th annual acm sigplan-sigact symposium on principles of programming languages. POPL '08. New York, NY, USA: Association for Computing Machinery.

Foster, J. Nathan, Greenwald, Michael B., Moore, Jonathan T., Pierce, Benjamin C., & Schmitt, Alan. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. Acm trans. program. lang. syst., 29(3), 17â es.

Foster, J. Nathan, Pilkiewicz, Alexandre, & Pierce, Benjamin C. (2008). Quotient lenses. Page 383â 396 of: Proceedings of the 13th acm sigplan international conference on functional programming. ICFP '08. New York, NY, USA: Association for Computing Machinery.

Gibbons, Jeremy. (2002). Calculating functional programs. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 151–203.

Gibbons, Jeremy. (2011). Maximum segment sum, monadically (distilled tutorial). Electronic proceedings in theoretical computer science, 66(Sep), 181â 194.

Gill, Andrew, Launchbury, John, & Peyton Jones, Simon L. (1993). A short cut to deforestation. Page 223â 232 of: Proceedings of the conference on functional programming languages and computer architecture. FPCA '93. New York, NY, USA: Association for Computing Machinery.

He, Xiao, & Hu, Zhenjiang. (2018). Putback-based bidirectional model transformations. Page 434â 444 of: Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering. ESEC/FSE '18. New York, NY, USA: Association for Computing Machinery.

Hidaka, Soichiro, Hu, Zhenjiang, Inaba, Kazuhiro, Kato, Hiroyuki, Matsuda, Kazutaka, & Nakano, Keisuke. (2010). Bidirectionalizing graph transformations. Page 205â 216 of: Proceedings of the 15th acm sigplan international conference on functional programming. ICFP '10. New York, NY, USA: Association for Computing Machinery.

Hofmann, Martin, Pierce, Benjamin, & Wagner, Daniel. (2011). Symmetric lenses. Page 371â 384 of: Proceedings of the 38th annual acm sigplan-sigact symposium on principles of programming languages. POPL '11. New York, NY, USA: Association for Computing Machinery.

Hofmann, Martin, Pierce, Benjamin, & Wagner, Daniel. (2012). Edit lenses. Page 495â 508 of: Proceedings of the 39th annual acm sigplan-sigact symposium on principles of programming languages. POPL '12. New York, NY, USA: Association for Computing Machinery.

Hu, Zhenjiang, Iwasaki, Hideya, & Takeichi, Masato. (1996). Deriving structural hylomorphisms from recursive definitions. Page 73â 82 of: Proceedings of the first acm sigplan international conference on functional programming. ICFP '96. New York, NY, USA: Association for Computing Machinery.

Ko, Hsiang-Shang, & Hu, Zhenjiang. (2018). An axiomatic basis for bidirectional programming. Proc. acm program. lang., 2(POPL).

Ko, Hsiang-Shang, Zan, Tao, & Hu, Zhenjiang. (2016). Bigul: A formally verified core language for putback-based bidirectional programming. Page 61â 72 of: Proceedings of the 2016 acm sigplan workshop on partial evaluation and program manipulation. PEPM '16. New York, NY, USA: Association for Computing Machinery.

Matsuda, Kazutaka, & Wang, Meng. (2015a). Applicative bidirectional programming with lenses. Page 62â 74 of: Proceedings of the 20th acm sigplan international conference on functional programming. ICFP '15. New York, NY, USA: Association for Computing Machinery.

Matsuda, Kazutaka, & Wang, Meng. (2015b). "bidirectionalization for free" for monomorphic transformations. Sci. comput. program., 111(P1), 79â 109.

Matsuda, Kazutaka, & Wang, Meng. (2018). Hobit: Programming lenses without using lens combinators. Pages 31–59 of: European symposium on programming. Springer.

Matsuda, Kazutaka, & Wang, Meng. (2020). Sparcl: A language for partially-invertible computation. Proc. acm program. lang., 4(ICFP).

Matsuda, Kazutaka, Hu, Zhenjiang, Nakano, Keisuke, Hamana, Makoto, & Takeichi, Masato. (2007). Bidirectionalization transformation based on automatic derivation of view complement functions. Page 47â 58 of: Proceedings of the 12th acm sigplan international conference on functional programming. ICFP '07. New York, NY, USA: Association for Computing Machinery.

Pacheco, Hugo, & Cunha, Alcino. (2010). Generic point-free lenses. Page 331â 352 of: Proceedings of the 10th international conference on mathematics of program construction. MPC '10. Berlin, Heidelberg: Springer-Verlag.

Pacheco, Hugo, & Cunha, Alcino. (2011). Calculating with lenses: Optimising bidirectional transformations. Page 91â 100 of: Proceedings of the 20th acm sigplan workshop on partial evaluation and program manipulation. PEPM '11. New York, NY, USA: Association for Computing Machinery.

Stevens, Perdita. (2007). A landscape of bidirectional model transformations. Berlin, Heidelberg: Springer-Verlag. Page 408â 424.

Stevens, Perdita. (2014). Bidirectionally tolerating inconsistency: Partial transformations. Pages 32–46 of: Gnesi, Stefania, & Rensink, Arend (eds), Fundamental approaches to software engineering. Berlin, Heidelberg: Springer Berlin Heidelberg.

Stevens, Perdita. (2020). Maintaining consistency in networks of models: bidirectional transformations in the large. Software and systems modeling, 19(1), 39–65.

Tran, Van-Dang, Kato, Hiroyuki, & Hu, Zhenjiang. (2020). Birds: Programming view update strategies in datalog. Proc. vldb endow., 13(12), 2897â 2900.

Tsigkanos, Christos, Li, Nianyu, Jin, Zhi, Hu, Zhenjiang, & Ghezzi, Carlo. (2020). Scalable multiple-view analysis of reactive systems via bidirectional model transformations. Page 993â 1003 of: Proceedings of the 35th ieee/acm international conference on automated software engineering. ASE '20. New York, NY, USA: Association for Computing Machinery.

Voigtländer, Janis. (2009). Bidirectionalization for free! (pearl). Page 165â 176 of: Proceedings of the 36th annual acm sigplan-sigact symposium on principles of programming languages. POPL '09. New York, NY, USA: Association for Computing Machinery.

Voigtländer, Janis, Hu, Zhenjiang, Matsuda, Kazutaka, & Wang, Meng. (2010). Combining syntactic and semantic bidirectionalization. Page 181â 192 of: Proceedings of the 15th acm sigplan international conference on functional programming. ICFP '10. New York, NY, USA: Association for Computing Machinery.

Xia, Li-yao, Orchard, Dominic, & Wang, Meng. (2019). Composing bidirectional programs monadically. Pages 147–175 of: European symposium on programming. Springer.

A Agda Proof

We have proved all essential technical details using Agda. The verification consists of 10 files (about 4500 lines of Agda code): Data/IntegerOmega/Base.agda, Data/IntegerOmega/Properties.agda, Lens.agda, Combinators.agda, Theorems.agda, Examples.agda, UniCombinatorsAndProperties.agda, Util.agda, Core.agda and TerminatingFold.agda that can be compiled with Agda version 2.6.1.3 and Agda stdlib version 1.6. All materials can be found in Anonymized Other Supplement. We explain how they support this work.

Data/IntegerOmega folder Maximum segment sum essentially manipulates the Tropical Semiring, which is integer with the negative infinity, and is equipped with maximum and addition operation. The Data/IntegerOmega folder is the formalization of this algebraic structure.

Lens.agda In this file, we give the definition of contract lens (definition 4.1). We define the composition of contract lenses and therefore prove that composition preserves well-behavedness (definition 4.2, theorem 4.1). We give the definition of lens equivalence and prove that it is symmetric, reflexive and transitive (definition 4.3, theorem 4.2). Furthermore, we give several auxiliary functions to help with constructing combinators.

There is a detail not discussed in the paper: the congruence theorems for the equivalence relation. For a programming construct that builds composite lenses by using simple lenses (for instance, composition of lenses, high-order lenses), we generally want a theorem stating that if two simple components are equivalent, then the two resulting composite lenses are equivalent. For example, it is expected that $\ell = \ell'$ implies $\ell; \ell_1 = \ell'; \ell_1$. This congruence theorem of composition is proved in Lens.agda. Such theorems for high order lenses are also proved in other files if necessary.

Combinators.agda, Theorems.agda, Examples.agda and Core.agda In the first two files, we list all important combinators (*bfoldr*, *bfoldr′*, *bmap*, *bmapl*, $\mathit{bfoldl}_{\mathit{init}}$, *bscanl*) and related calculation laws (*bfoldr* fusion law, *bmapl* fusion law, *bfoldr-bmap* fusion law, *bscanl* lemma. The proof of *bmap* fusion law is omitted since it holds trivially). In Examples.agda, we provide the list deforestation example given in the overview and our major example, calculating bidirectional maximum segment sum. All proofs can be found in Core.agda.

A fact worth mentioning is that we verify a slightly different version of *bmapl*, *bscanl* and $\mathit{bfoldl}_{\mathit{init}}$: they consume inner algebras of slightly different type signatures. They are equivalent, but with some type information 'merged' in the source, view contracts. We changed their presentation in the paper for better readability.

TerminatingFold.agda The bidirectional *foldr* defined in this paper may not terminate: when the source list is empty, the *put* computation is equivalent to an *unfold*. As a result, the proof of *bfoldr* and *bfoldr*-fusion law in Core.agda can not pass the termination checker. However, we argue that those theorems hold if we

add a condition stating that given a bidirectional algebra *balg*, the *put* component of *bfoldr balg* terminates on every input. In this case, the 'broken' proof can be recovered by induction on the number of steps it takes to terminate.

The idea on giving an explicit termination condition is also implemented in TerminatingFold.agda. Instead of the standard type signature for *bfoldr*: $\mathit{Either}\ ()\ (S,V) \leftrightarrow V \to [S] \leftrightarrow V$, we index the view type with a natural number. More formally, we first replace the view type $V : \mathit{Set}$ to be an indexed family $V : \mathbb{N} \to \mathit{Set}$, and then replace every occurrence of V in the type signature of *bfold* to be $\Sigma n : \mathbb{N}.\ V\ n$ (dependent Sigma type). Now we impose a termination condition by restricting the *put* behavior of *balg*: when the source is $\mathit{Left}\ ()$ and the modified view is y, the put result must be either $\mathit{Left}\ ()$, which leads to termination, or $\mathit{Right}\ (x, y')$, with the index of y' being strictly smaller than that of y. Using this termination condition, we verify the well-behavedness of *bfoldr* and *bfoldr*-fusion law.

UniCombinatorsAndProperties.agda, Util.agda These two files consist of some useful utility functions and unidirectional combinators to aid the main verification. In Util.agda, we postulate function extensionality.