JFP, **??** pages, 2021. © Cambridge University Press 2021
doi:10.1017/xxxxx

Contract Lenses: Reasoning about Bidirectional Programs via Calculation

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

WENHAO TANG
Peking University, China
(e-mail: tangwh@pku.edu.cn)

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

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

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

Abstract

Bidirectional transformations (BXs) are a mechanism for maintaining consistency between multiple representations of related data. The need to effectively construct BXs has attracted interest in programming languages research, including deriving efficient BXs from specifications via *calculation*. 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 and Spyratos, 1981; Bohannon et al., 2006; Tran et al., 2020), software model transformation (Stevens, 2008; He and Hu, 2018; Tsigkanos et al., 2020; Stevens, 2020), graph transformation (Hidaka et al., 2010) etc; in particular there has been several language-based approaches that allow transformations in both directions to be programmed together (for example Foster et al. (2007); Voigtländer (2009); Matsuda et al. (2007); Ko et al. (2016)).

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

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

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*:

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

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 [1]

$$\begin{aligned}
&\ell_1; \ell_2 = Lens\ g\ p\\
&\quad \textbf{where}\\
&\qquad g \quad\ \ = get_{\ell_2} \circ get_{\ell_1}\\
&\qquad p\ s\ t' = put_{\ell_1}\ s\ \ (put_{\ell_2}\ (get_{\ell_1}\ s)\ t')
\end{aligned}$$

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

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

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

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

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

1.1 Program Calculation and the Challenge of Partiality

Program calculation (Bird, 1989) 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 calculates 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, 2011) to be applied to specific instances without the need of special analysis.

However, the more complex setting of BX as compared to unidirectional programs 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 with both. Moreover, lenses are often *partially* defined, making it hard to reason about the *construction* and *compositions* of combinators like *map*, *fold*, and *scan*. Semantic preservation amid calculation is difficult in this case as well (note that a change in the definedness of a function changes its semantics).

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

[2] In this paper, we assume all functions are total.

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

$$\begin{array}{l} bmap \;\; : (A \leftrightarrow B) \rightarrow ([A] \leftrightarrow [B]) \\ bmap\ \ell = Lens\ (map\ get_\ell)\ p \\ \quad \textbf{where}\ p\ (x:xs)\ (y:ys) = put_\ell\ x\ y : p\ xs\ ys \\ \qquad\quad\ \ p\ \_ \qquad\ \ \_ \qquad\ = [\,] \end{array}$$

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

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

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

1.2 Contributions

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

Contract Lenses The main idea of contract lenses is to utilize a pair of fine-grained predicates, one on source and one on view, to characterize the bidirectional behaviour on 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 dependencies 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 Proofs in Agda We prove the technical details of our calculation framework in Agda, including the correctness of all contract lens combinators and calculation laws, as well as most of the examples. The proof consists of 4k lines of Agda code. (Section 9 and Supplementary Files)

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

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

2 Background: Program Calculation

Program calculation (Bird, 1989; Gibbons, 2002) is a technique for constructing efficient programs that are correct by construction. It is suitable for humans to derive efficient programs by hand (Bird, 1989), 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: $\mathit{foldl} : (b \to a \to b) \to b \to [a] \to b$ and $\mathit{foldr} : (a \to b \to b) \to b \to [a] \to b$, which can be used to define a range of functions. We give some examples as follows, which are also used in the remainder of the paper.

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

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 $\mathit{foldr}\ f\ e$ has two arguments, which can be combined into one $\mathit{foldr}'\ \mathit{alg}$ where *alg* is a function of type $\mathit{Either}\ ()\ (a, b) \to b$.

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

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

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

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 $\mathit{unfoldr}'$ can be easily defined.

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

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:

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

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

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

$$\begin{aligned} \mathit{map}\, f \circ \mathit{map}\, g &= \mathit{map}\ (f \circ g) && \textsc{Map Fusion}\\ \mathit{foldr}'\, f \circ \mathit{map}\, g &= \mathit{foldr}'\ (f \circ \mathrm{F_m}\ g) && \textsc{Fold-Map Fusion}\\ \mathit{map}\ (\mathit{foldl}\, f\ e) \circ \mathit{inits} &= \mathit{scanl}\, f\ e && \textsc{Scan Lemma} \end{aligned}$$

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

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

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

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

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

where $x \odot y = (x \otimes y) \oplus a$, and a is the value passed to $\mathit{foldl}\ (\otimes)$. □

2.3 A Calculational Example

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

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

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

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 (1989).

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

3 Overview

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

3.1 Taming Partiality with Contract Lenses

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

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

where ℓ is a lens with only *get* and *put*. Though we write cs and cv around the lens ℓ for readability in this section, a contract lens is formally defined as a four-tuple consisting of *get*, *put*, cs, and cv. [3]

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 the updated view in relation to the original view. The *source condition cs* has a similar structure. It takes two arguments: the original source s and the updated source $put_\ell\ s\ v$, specifying an invariant that must hold for source changes as a result of valid view changes.

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

$$eqlength = \lambda xs\ xs' \rightarrow length\ xs = length\ xs'$$

[3] In spite of the similarity of syntax, contracts are different from Hoare logic, which we will discuss in details in Section 8.2.

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

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

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

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

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

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

3.2 Calculation with Contract Lenses

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

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

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

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

[4] One might expect a bidirectional version of *map* to have more complicated source and view conditions, e.g., imposing the source and view conditions of the parameter ℓ to all elements in the list. In Section 5.2, we will show alternative definitions of bidirectional map with different contracts.

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

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

The view condition of *bfilter even* is defined as

$$ceven = \lambda xs\ xs' \to eqlength\ xs\ xs' \ \wedge\ all\ even\ xs'$$

which depends on the predicate *even*. [6] The combinator *bfilter* is a bidirectional version of *filter* implemented by *bfoldr′*, which is a bidirectional version of *foldr* with contracts (Section 5). We have already seen *bmap* in Section 1.1. In this example, *bfilter even* is also given source and view conditions including *eqlength*, which is needed to be composed with $\{eqlength\}\ bmap\ bhead\ \{eqlength\}$. The contracts of $\{eqlength\}\ bfilter\ even\ \{ceven\}$ make sense: if the number of even elements is not changed, the total number of elements will neither be changed because the odd elements, which do not appear in the view, remain invariant.

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

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

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

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

4 Contract Lenses

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

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

[7] We omit administrative parameters for contracts taken by higher-order contract lenses *bfoldr′* and *bmapF* for simplicity. They are easy to be reconstructed from the definitions of lens combinators.

4.1 Contract Lenses

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

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

$$
\begin{array}{l}
\textbf{data}\ S \leftrightarrow V = CLens\ \{ \\
\quad get : S \to V, \\
\quad put : S \to V \to S, \\
\quad cs\ \ : S \to S \to Set, \\
\quad cv\ \ : V \to V \to Set \\
\}
\end{array}
$$

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

$$
\begin{array}{rr}
cv\ (get\ s)\ v \Rightarrow cs\ s\ (put\ s\ v) & \textsc{BackwardValidity} \\
cv\ (get\ s)\ v \Rightarrow get\ (put\ s\ v) = v & \textsc{ConditionedPutGet} \\
cs\ s\ s \Rightarrow cv\ (get\ s)\ (get\ s) & \textsc{ForwardValidity} \\
cs\ s\ s \Rightarrow put\ s\ (get\ s) = s & \textsc{ConditionedGetPut}
\end{array}
$$

□

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

We have a few remarks to make here.

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

[8] 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.

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

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

We use the following notational conventions:

• We use $cs_\ell, cv_\ell, get_\ell, put_\ell$ to refer to the source condition, view condition, forward transformation and backward transformation of a contract lens ℓ, respectively.

• Lists start from index 1 and the notation x_i refers to the i-th element of a list x.

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

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

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

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

With the help of the condition on the view change (which keeps the "inits" structure), our putback function becomes very simple, just returning the last element if it is not empty. □

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.

[9] We write $\overline{get}$ when we just want to use *get* as a total function without considering the satisfaction of the source condition.

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

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

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

Notice that we not only require the backward implication $cs_{\ell_2}\ v\ v' \Rightarrow cv_{\ell_1}\ v\ v'$, but also the forward one $cv_{\ell_1}\ v\ v \Rightarrow cs_{\ell_2}\ v\ v$. Intuitively, the latter is used to establish a connection between the FORWARDVALIDITY law of ℓ_1 and ℓ_2. Moreover, we can strengthen the condition of composition to make it easier to use. We say that two predicates $c_1 : A \to A \to Set$ and $c_2 : B \to B \to Set$ are equivalent, written as $c_1 \Leftrightarrow c_2$, if $A = B$ and $\forall\,(a : A)\ (a' : A), c_1\ a\ a' \Leftrightarrow c_2\ a\ a'$. The condition of composition can be strengthened to $cs_{\ell_2} \Leftrightarrow cv_{\ell_1}$, which is sufficient in most cases.

4.3 Equivalence of Contract Lenses

Now we define an equivalence relation over contract lenses.

Definition 3 (Lens Equivalence). *For lens $\ell_1 : S \leftrightarrow V$ and $\ell_2 : S \leftrightarrow V$, we say 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 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

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

Since bidirectional transformations can be considered as unidirectional forward programs with additional *put* semantics, our idea is to bidirectionalise widely used recursion schemes in (forward) functional programming including *fold*, *map*, *filter*, and *scan*. The main challenge is that these functions are usually not bijective, which requires contracts to make them total and suitable for calculation. Different contracts will lead to different bidirectional version of the same high-order functions, and are useful for different situations. We will give both total bidirectional versions of these functions, and their variants which have some additional conditions on the source and the view to make them flexible for composing with each other. It will be interesting to see later that although *map* and *scan* can be implemented by *fold*, it turns out to be more useful to implement bidirectional versions of *map* and *scan* individually to attain better control over their contracts and behaviours.

5.1 Bidirectional Fold

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

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

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

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

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

The type of *bfoldr* can be simplified to

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

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

```
$bfoldr\ \ell = CLens\ (foldr'\ get_\ell)\ (curry\ \$\ unfoldr'\ coalg)\ ctrue\ ctrue$
    **where**
      $coalg\ ([\,], v')$      $=$ **case** $put_\ell\ (Left\ ())\ v'$ **of**
                                $Left\ () \rightarrow Left\ ()$
```

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

Note that the put direction of the above definition is inefficient since it computes "$g\ as$" every time $\mathit{coalg}\ (a : as, v')$ is called. A more efficient implementation is to calculate all $g\ as$ in advance using a *scanr* as shown in Appendix 2.1. We will use the efficient definition of *bfoldr* in the following sections.

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

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

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

where bmax is a bidirectional version of max whose backward transformation uses the modified value to replace the maximum value of the parameter pair. [10]

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

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

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 3, we may wish to keep the length of the source list after $\mathit{put}_{\mathit{bmaximum}}$, and furthermore, we hope that the source and view conditions of *bfoldr* be able to express some extra constraints on the elements. All these can be concisely expressed as the following higher-order contract lens:

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

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

[10] We treat ∞ as a value of type *Int* as well for simplicity.

$$\begin{aligned}
&\mathit{lift}\ p\ q\ a\ a' &&= (a = \mathit{Left}\ ()\ \wedge\ a' = \mathit{Left}\ ())\ \vee \\
& &&\quad (a = \mathit{Right}\ (x,y)\ \wedge\ a' = \mathit{Right}\ (x',y')\ \wedge\ p\ x\ x'\ \wedge\ q\ y\ y') \\
&\mathit{licond}\ p\ xs\ xs' &&= \mathit{eqlength}\ xs\ xs'\ \wedge\ (\forall\, 1 \le i \le |xs|, p\ xs_i\ xs'_i)
\end{aligned}$$

The lens combinator *bfoldr'* takes two predicates $\widehat{cs}$ and $\widehat{cv}$ and have the same definition of *get* and *put* components as *bfoldr*. The $\widehat{cs}$ represents the constraints on the elements of the source list, and the $\widehat{cv}$ represents the view condition. Notice that the predicate parameters $\widehat{cs}$ and $\widehat{cv}$ are kind of administrative; their main role is to guarantee that the source condition of the parameter lens is of shape $\mathit{lift}\ cs'\ cv'$ for some cs' and cv'. Idealy, we can make them existentially bound. We opt to have explicit predicate parameters to make the presentation clear and more consistent with our Agda formalisation.

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

$$\begin{aligned}
&\mathit{bmaximum}'\ :\ [\mathit{Int}] \leftrightarrow \mathit{Int} \\
&\mathit{bmaximum}' = \mathit{bfoldr}'\ \mathit{eqlength}\ \widehat{cv}\ \mathit{bmax}' \\
&\quad \textbf{where} \\
&\qquad \widehat{cv} \quad\ \ = \lambda x\, x' \to x \neq -\infty\ \vee\ x' = -\infty \\
&\qquad \mathit{bmax}' = \mathit{bmax}\ \{cs = \mathit{lift}\ \mathit{ctrue}\ cv, cv = \widehat{cv}\}
\end{aligned}$$

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

5.1.2 Bidirectional Filter

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

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

$$\begin{aligned}
&\mathit{bfilter}\quad\ :\ (pr : a \to \mathit{Bool}) \to ([a] \leftrightarrow [a]) \\
&\mathit{bfilter}\ pr = \mathit{bfoldr}'\ \mathit{ctrue}\ (\mathit{fcond}\ pr)\ (\mathit{bfilterAlg}\ pr) \\
&\quad \textbf{where} \\
&\qquad \mathit{bfilterAlg}\quad\ :\ (pr : a \to \mathit{Bool}) \to (\mathit{Either}\ ()\ (a,[a]) \leftrightarrow [a]) \\
&\qquad \mathit{bfilterAlg}\ pr = \mathit{CLens}\ g\ p\ (\mathit{lift}\ \mathit{ctrue}\ (\mathit{fcond}\ pr))\ (\mathit{fcond}\ pr) \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad g\ (\mathit{Left}\ ()) &&= [\,] \\
&\qquad\qquad g\ (\mathit{Right}\ (x,xs)) &&= \textbf{if}\ pr\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs \\
&\qquad\qquad p\ (\mathit{Left}\ ())\ [\,] &&= \mathit{Left}\ () \\
&\qquad\qquad p\ (\mathit{Right}\ (x,xs))\ xs' &&= \mathit{Right}\ (\textbf{if}\ pr\ x\ \textbf{then}\ (\mathit{head}\ xs', \mathit{tail}\ xs')\ \textbf{else}\ (x, xs'))
\end{aligned}$$

The function *fcond* is defined as $fcond\,pr = licond\,(\lambda\,\_\,x' \to pr\,x')$. The *bfilterAlg pr* is essentially a bidirectional version of the function $\lambda x\,xs \to \textbf{if}\,pr\,x\,\textbf{then}\,x : xs\,\textbf{else}\,xs$. One example of *bfilter* is the *bfilter even* defined in Section 3.2.

5.2 Bidirectional Map

Map is another important high-order function in functional programming and program calculation, which applies a function to each element of a list. In this section, we will give three different definitions of bidirectional *map* with different source and view conditions. The first one is *bmap*, which is just a bidirectional *map* that preserves the length of the source and view list. It has no other constraints on the source and view. The second one is *bmap′*, which takes the constraint on individual elements of the list into consideration. The third one is *bmapl* (and *bmapr*), which goes a step further and takes into account the constraints on adjacent elements of the list as well. These three bidirectional versions of *map* cover a large range of applications. In particular, the most powerful *bmapl* is helpful in our later calculation of bidirectional maximum segment sum.

5.2.1 Bidirectional Map: Preserving Length

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

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

It is clear to see 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 also implement *bmap* using *bfoldr′* as shown in Appendix 2.2. One example of *bmap* is the *bmap bhead* defined in Section 3.2.

5.2.2 Bidirectional Map: Preserving Inner Constraints

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

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

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

In the above definition of $bmap'\ \ell$, we directly use cs_ℓ and cv_ℓ in the contracts of the result lens. The $bmap'\ \ell$ has no requirement on the contracts of ℓ. Another alternative definition of $bmap'$ more similar to the definition of $bfoldr'$ which takes predicate parameters is as follows:

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

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

5.2.3 Bidirectional Map: Preserving Constraints on Adjacent Elements

In practice, it is very common that $map\,f$ 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 need to carefully specify this kind of constraints. Recall the $binits$ in Section 4.1 with the following view condition:

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

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

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

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

Our two syntactic sugars can be used nestedly:

$$\begin{aligned}
&A \Rightarrow (\{cs'\}\ S \leftrightarrow V\ \{cv'\})\ \equiv\\
&\quad \{\ell : A \to \{\ell' : S \leftrightarrow V \mid cs_\ell \Leftrightarrow cs'\ \wedge\ cv_\ell \Leftrightarrow cv'\} \mid \exists f : S \to V. \forall\, a : A.\ get_{\ell\, a} = f\}
\end{aligned}$$

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

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

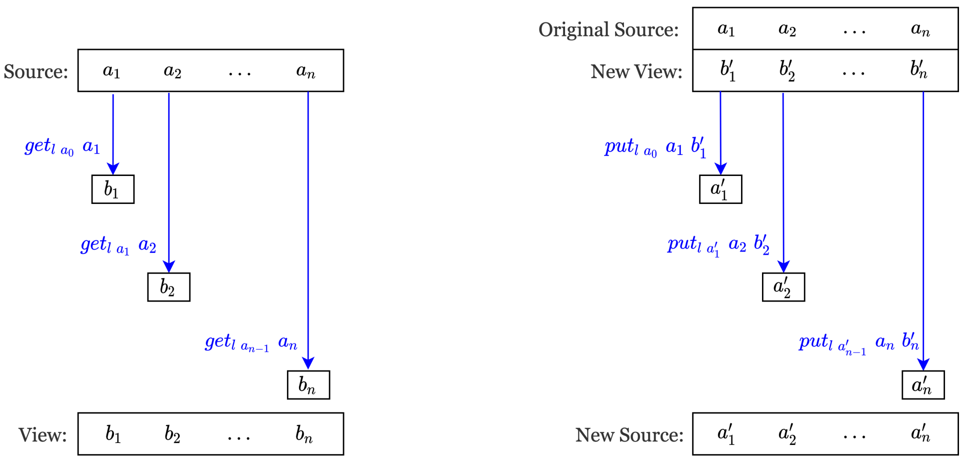
Source:
a_1 a_2 $\dots$ a_n
$get_{l\ a_0}\ a_1$
b_1
$get_{l\ a_1}\ a_2$
b_2
$get_{l\ a_{n-1}}\ a_n$
b_n
View:
b_1 b_2 $\dots$ b_n
Original Source:
a_1 a_2 $\dots$ a_n
New View:
b'_1 b'_2 $\dots$ b'_n
$put_{l\ a_0}\ a_1\ b'_1$
a'_1
$put_{l\ a'_1}\ a_2\ b'_2$
a'_2
$put_{l\ a'_{n-1}}\ a_n\ b'_n$
a'_n
New Source:
a'_1 a'_2 $\dots$ a'_n

Fig. 1. Implementation of *bmapl*. 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 as_0 to be $[\,]$, then the source condition of the $bmapl\, as_0\, \ell$ is equivalent to cv_{binits}, and thus, the composition $binits$; $bmapl\, [\,]\, \ell$ is valid.

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

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 $scanr'$. One example of *bmapl* is shown in Appendix 3.3.

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 $map\, f$ with expressive contraints. What if f is a *fold*? Since *bmapl* takes a parameterised lens of type $S \Rightarrow (S \leftrightarrow V)$, we cannot directly pass either *bfoldr* or $bfoldr'$ to *bmapl*. Moreover, since the bidirectional *fold* we needed depends on the $\widetilde{cs}$ in the source condition of the result of *bmapl*, it is actually difficult to give a general bidirectional *fold*. Fortunately, we can define some special bidirectional versions of *fold* to cope with some frequently used constraints, such as $\lambda a_{i-1}\, a_i \to init\, a_i = a_{i-1}$. The $bfoldl_{init}$ shown below is such a special *bfold* that can be used inside *bmapl*.

$$\begin{aligned} bfoldl_{init}\ :\ & (\widetilde{cv} : V \to V \to Set) \to (b_0 : V) \\ & \to (\ell : (b : V) \Rightarrow (\{\lambda\_\, t' \to t' = Right\, (\_, b)\} \end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad \mathit{Either}\;()\;(S,V) \leftrightarrow V) \\
\qquad\qquad\qquad\qquad\qquad \{\lambda\_\,b' \to \widetilde{cv}\;b\;b'\}) \\
\qquad \to [S] \Rightarrow ([S] \leftrightarrow V) \\
\mathit{bfoldl}_{\mathit{init}}\;\widetilde{cv}\;b_0\;\ell\;\mathit{as} = \mathit{CLens}\;g\;p\;\mathit{cs}'\;\mathit{cv}' \\
\quad \textbf{where}\;g \qquad = \mathit{foldl}\;(\lambda b\;a \to \mathit{get}_{\ell\,b}\;(\mathit{Right}\;(a,b)))\;b_0 \\
\qquad\quad p\;[\,]\;b' \;\; = \textbf{case}\;\mathit{put}_{\ell\,(g\,\mathit{as})}\;(\mathit{Left}\;())\;b'\;\textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Right}\;(a,\_) \to \mathit{as} +\!\!+ [a] \\
\qquad\quad p\;\mathit{as}'\;b' = \textbf{case}\;\mathit{put}_{\ell\,(g\,\mathit{as})}\;(\mathit{Right}\;(\mathit{last}\;\mathit{as}', g\;(\mathit{init}\;\mathit{as}')))\;b'\;\textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Right}\;(a,\_) \to \mathit{as} +\!\!+ [a] \\
\qquad\quad \mathit{cs}' \qquad = \lambda\_\,\mathit{as}' \to (\mathit{init}\;\mathit{as}' = \mathit{as}) \\
\qquad\quad \mathit{cv}' \qquad = \lambda\_\,b' \to (\widetilde{cv}\;(g\;\mathit{as})\;b')
\end{array}
$$

The $\mathit{bfoldl}_{\mathit{init}}$ takes a parameterised contract lens and returns another parameterised contract lens which is suitable to be passed to *bmapl*. Notice that the result parameterised lens $\mathit{bfoldl}_{\mathit{init}}\;\widetilde{cv}\;b_0\;\ell$ of type $[S] \Rightarrow ([S] \leftrightarrow V)$ indeed has the same *get* component for any argument $\mathit{as} : [S]$, because the *get* component does not use *as* at all. The *get* direction is a standard *foldl*, and the *put* direction only computes the last element of the new source list, since other elements are given as the argument indicated by the source condition $\lambda\_\,\mathit{as}' \to (\mathit{init}\;\mathit{as}' = \mathit{as})$.

For an example usage of $\mathit{bfoldl}_{\mathit{init}}$, we refer to Appendix 3.4.

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 optimisation (as will be seen later). The main challenge to bidirectionalize *scan* is that the result of *scan* may have constraints between adjacent elements similar to *bmapl*. In this section, we give a powerful bidirectional version of *scan* with the help of contract lenses.

$$
\begin{array}{l}
\mathit{bscanl}\;\;:\;\;(\widetilde{cv} : V \to V \to \mathit{Set}) \to (b_0 : V) \\
\qquad \to (\ell : (b : V) \Rightarrow (\{\lambda\_\,t' \to t' = \mathit{Right}\;(\_, b)\} \\
\qquad\qquad\qquad\qquad\quad \mathit{Either}\;()\;(S,V) \leftrightarrow V \\
\qquad\qquad\qquad\qquad\quad \{\lambda\_\,b' \to \widetilde{cv}\;b\;b'\})) \\
\qquad \to ([S] \leftrightarrow [V]) \\
\mathit{bscanl}\;\widetilde{cv}\;b_0\;\ell = \mathit{CLens}\;g\;p\;\mathit{eqlength}\;\mathit{cv}' \\
\quad \textbf{where} \\
\qquad g \qquad\;\; = \mathit{scanl}'\;(\lambda b\;a \to \mathit{get}_{\ell\,b}\;(\mathit{Right}\;(a,b)))\;b_0 \\
\qquad p\;\mathit{as}\;\mathit{bs}' = \mathit{map}\;(\lambda((a,b),(b'',b')) \to \mathit{fstRight}\;(\mathit{put}_{\ell\,b''}\;(\mathit{Right}\;(a,b))\;b'))\;\mathit{abb} \\
\qquad\quad \textbf{where}\;\mathit{bs} \;\; = g\;\mathit{as} \\
\qquad\qquad\quad \mathit{abb} = \mathit{zip}\;(\mathit{zip}\;\mathit{as}\;(b_0 : \mathit{init}\;\mathit{bs}))\;(\mathit{zip}\;(b_0 : \mathit{init}\;\mathit{bs}')\;\mathit{bs}') \\
\qquad\qquad\quad \mathit{fstRight}\;(\mathit{Right}\;(x,\_)) = x \\
\qquad \mathit{cv}'\;t\;\mathit{bs} \; = (\forall\, 1 \le i \le |\mathit{bs}|.\; \widetilde{cv}\;\mathit{bs}_{i-1}\;\mathit{bs}_i) \;\wedge\; \mathit{eqlength}\;t\;\mathit{bs}
\end{array}
$$

The implementation of $\mathit{bscanl}\;\widetilde{cv}\;b_0\;\ell$ is visualized in Figure 2. The *get* direction is a standard scanl'. For the *put* direction, when computing a_i' from a_i and b_i', we pass b_{i-1}' to the lens ℓ to restrict the result of *put* is of form $\mathit{Right}\;(\_, b_{i-1}')$.

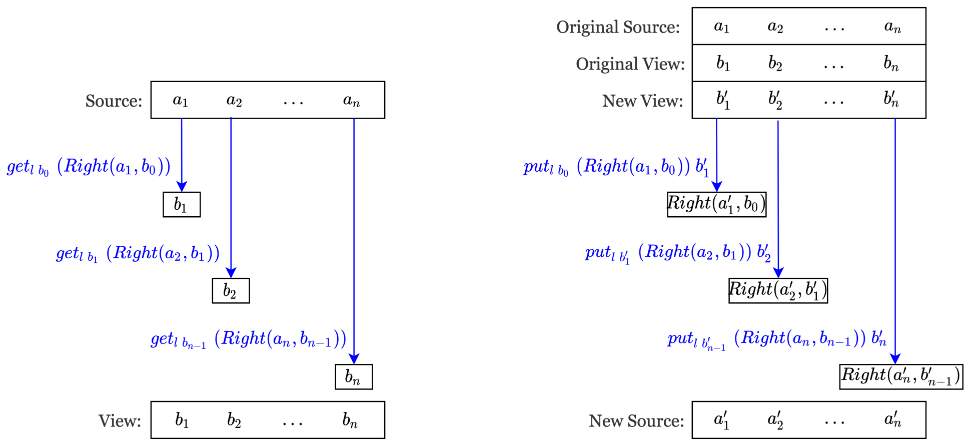
Source: a_1 a_2 $\dots$ a_n
$get_{l\ b_0}\ (Right(a_1, b_0))$
b_1
$get_{l\ b_1}\ (Right(a_2, b_1))$
b_2
$get_{l\ b_{n-1}}\ (Right(a_n, b_{n-1}))$
b_n
View: b_1 b_2 $\dots$ b_n
Original Source: a_1 a_2 $\dots$ a_n
Original View: b_1 b_2 $\dots$ b_n
New View: b'_1 b'_2 $\dots$ b'_n
$put_{l\ b_0}\ (Right(a_1, b_0))\ b'_1$
$Right(a'_1, b_0)$
$put_{l\ b'_1}\ (Right(a_2, b_1))\ b'_2$
$Right(a'_2, b'_1)$
$put_{l\ b'_{n-1}}\ (Right(a_n, b_{n-1}))\ b'_n$
$Right(a'_n, b'_{n-1})$
New Source: a'_1 a'_2 $\dots$ a'_n

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

For an example usage of *bscanl*, we refer to Appendix 3.5.

6 Bidirectional Calculation Laws

So far, we have seen that fundamental high-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 behaviours 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 $\mathrm{F_L}$ in Section 2.2.

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

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.

$$\frac{\ell_1; \ell = \mathit{blistF}\ (\mathit{get}_{\ell_1}\ (\mathit{Left}\ ()))\ \ell; \ell_2}{\mathit{bfoldr}\ \ell_1; \ell = \mathit{bfoldr}\ \ell_2} \qquad \text{BFOLDR FUSION}$$

It reads that the lens composition $\mathit{bfoldr}\ \ell_1; \ell$ can be fused into a single lens $\mathit{bfoldr}\ \ell_2$ if there exists ℓ_2 such that the equation $\ell_1; \ell = \mathit{blistF}\ (\mathit{get}_{\ell_1}\ (\mathit{Left}\ ()))\ \ell; \ell_2$ holds.

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

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

Then, the fusion law is stated as

$$\frac{\ell_1; \ell = \mathit{blistF}'\ \widehat{cs}\ \ell; \ell_2}{\mathit{bfoldr}'\ \widehat{cs}\ \widehat{cv}\ \ell_1; \ell = \mathit{bfoldr}'\ \widehat{cs}\ \widehat{ct}\ \ell_2} \qquad \text{BFOLDR' FUSION}$$

6.2 Bidirectional Map Fusion

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'.

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

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

$$\mathit{bmapl}\ \widetilde{cs}\ \widetilde{cv}\ a_0\ \ell_1; \mathit{bmapl}\ \widetilde{cv}\ \widetilde{ct}\ b_0\ \ell_2 = \mathit{bmapl}\ \widetilde{cs}\ \widetilde{ct}\ a_0\ (\ell_1;;\ell_2) \qquad \text{BMAPL FUSION}$$

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

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

The definition of $\ell_1;;\ell_2$ is quite intuitive. We just pass the parameter a to ℓ_1, and the result of a after the forward transformation of ℓ_1 to ℓ_2. Notice that we still use the syntactic sugar $S \Rightarrow (S \leftrightarrow T)$ for the type of the result parameterised lenses, which means the *get* component is the same for any parameter. This makes natural sense because both ℓ_1 and ℓ_2 have fixed *get* components. It is also easy to check that the composition $\ell_1\, a; \ell_2\, (get_{\ell_1\, a}\, a)$ is well-defined (i.e., satisfies the condition in Definition 2).

6.3 Bidirectional Fold-Map Fusion

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

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

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

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

$$bmap'\;\ell_1; bfoldr'\;\widehat{cv}\;\widehat{ct}\;\ell_2 = bfoldr'\;\widehat{cs}\;\widehat{ct}\;(bmapF\;\widehat{ct}\;\ell_1;\ell_2) \qquad \text{BFOLDR'-BMAP FUSION}$$

6.4 Bidirectional Scan Lemma

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

$$binits; bmapl\;(\lambda a\;a' \to init\;a' = a)\;\widetilde{cv}\;[\,]\;(bfoldl_{init}\;\widetilde{cv}\;b_0\;\ell) = bscanl\;\widetilde{cv}\;b_0\;\ell \qquad \text{BIDIRECTIONAL SCAN LEMMA}$$

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

7 Examples

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

7.1 Projection onto a Hyperplane

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

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

```
bproj : [Float] ↔ [Float]
bproj = bmean; bdiff

bmean : [Float] ↔ (Float, [Float])
bmean = CLens g p cs' cv'
  where g xs                 = (mean xs, xs)
        p _ (m, xs')         = xs'
        cs' xs xs'           = mean xs = mean xs' ∧ eqlength xs xs'
        cv' (m, xs) (m', xs') = m = m' = mean xs = mean xs' ∧ eqlength xs xs'

bdiff : (Float, [Float]) ↔ [Float]
bdiff = CLens g p cs' cv'
  where g (m, xs)            = map (+(−m)) xs
        p (m, _) xs'         = (m, map (+m) xs')
        cs' (m, xs) (m', xs') = m = m' = mean xs = mean xs' ∧ eqlength xs xs'
        cv' xs xs'           = sum xs' = 0 ∧ eqlength xs xs'
```

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

$$\begin{aligned} &mean\ \ : [Float] \to Float \\ &mean\ \ = \lambda xs \to sum\ xs\ /\ fromIntegral\ (length\ xs) \end{aligned}$$

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

7.2 String Formatting and Processing

Specifying programs that manipulate texts/strings bidirectionally is not new, and has been extensively studied in Bohannon et al. (2008); Matsuda and Wang (2015). The novelty of our framework is that it supports a point-free style of specifications and calculational reasonings for such computational tasks.

7.2.1 String Formatting

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

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

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

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

$$\begin{aligned} &\quad bformatting \\ &= \quad \{\ \text{definition}\ \} \\ &\quad bfilter\ (not \circ isDigit); bmap'\ btoUpper \\ &= \quad \{\ \text{expressing } bfilter \text{ as } bfoldr'\ \} \\ &\quad bfoldr'\ ctrue\ (fcond\ (not \circ isDigit))\ (bfilterAlg\ (not \circ isDigit)); bmap'\ btoUpper \\ &= \quad \{\ \text{BFOLDR' FUSION}\ \} \\ &\quad bfoldr'\ ctrue\ (licond\ (\lambda\_\ c \to isUpper\ c))\ balg \end{aligned}$$

where

$$\begin{aligned} &balg :: (Either\ ()\ (Char, [Char])) \leftrightarrow [Char] \\ &balg = CLens\ g\ p\ cs\ cv \\ &\quad \mathbf{where}\ g\ (Left\ ()) = [\,] \\ &\qquad\quad g\ (Right\ (x, xs)) = \mathbf{if}\ not\ (isDigit\ x)\ \mathbf{then}\ toUpper\ x : xs\ \mathbf{else}\ xs \end{aligned}$$

$$
\begin{aligned}
&p\ (Left\ ())\ \_ = Left\ () \\
&p\ (Right\ (x, xs))\ (x' : xs') = \textbf{if}\ not\ (isDigit\ x) \\
&\quad \textbf{then}\ Right\ (putToLower\ x\ x', xs') \\
&\quad \textbf{else}\ Right\ (x, x' : xs') \\
&p\ (Right\ (x, []))\ [\,] = \textbf{if}\ not\ (isDigit\ x)\ \textbf{then}\ Left\ ()\ \textbf{else}\ Right\ (x, [\,]) \\
&cs = lift\ ctrue\ (licond\ (\lambda\_\ c \to isUpper\ c)) \\
&cv = licond\ (\lambda\_\ c \to isUpper\ c)
\end{aligned}
$$

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

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

7.2.2 String Encoding and Decoding

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

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

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

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

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 $cv_{comp}\ v\ as = (|v| = |as|) \wedge (\forall 1 \le i \le |as|, odd\ i \vee (0 \le as_i < 128))$.

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

$$
\begin{aligned}
&bcompression : [String] \leftrightarrow [Int] \\
&bcompression = bmap'\ bencode \\
&\qquad\qquad\quad ;\ bmap'\ bascii \\
&\qquad\qquad\quad ;\ bfoldr'\ (\lambda\_\ (x,\_) \to 0 \le x < 128)\ cv_{comp}\ bcat
\end{aligned}
$$

where the following contract lenses are used

$$
\begin{aligned}
&bencode : String \leftrightarrow (Char, Int) \\
&bencode = CLens\ (\lambda ws \to (head\ ws, length\ ws))\ (\lambda\_\ (a,n) \to replicate\ n\ a) \\
&\qquad\qquad (\lambda\_\ as \to allsame\ as)\ ctrue \\
&\quad \textbf{where}\ allsame\ xs = (xs = \texttt{""}) \lor (and\ \$\ map\ (= head\ xs)\ (tail\ xs)) \\
&bascii : (Char, b) \leftrightarrow (Int, b) \\
&bascii = CLens\ (\lambda (x,y) \to (ord\ x, y))\ (\lambda\_\ (x,y) \to (chr\ x, y)) \\
&\qquad\qquad ctrue\ (\lambda\_\ (x,\_) \to 0 \le x < 128) \\
&bcat : Either\ ()\ ((Int, Int), [Int]) \leftrightarrow [Int] \\
&bcat = CLens\ g\ p\ cs\ cv \\
&\quad \textbf{where}\ g\ (Left\ ()) = [\,] \\
&\qquad\quad g\ (Right\ ((x,y),b)) = x : y : b \\
&\qquad\quad p\ (Left\ ())\ [\,] = Left\ () \\
&\qquad\quad p\ (Right\ \_)\ (x : y : b) = Right\ ((x,y),b) \\
&\qquad\quad cs = lift\ (\lambda\_\ (x,\_) \to 0 \le x < 128)\ cv_{comp} \\
&\qquad\quad cv = cv_{comp}
\end{aligned}
$$

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

$$
\begin{aligned}
&\quad bcompression \\
&= \quad \{\ \text{definition}\ \} \\
&\quad bmap'\ bencode \\
&\quad ;\ bmap'\ bascii \\
&\quad ;\ bfoldr'\ (\lambda\_\ (x,\_) \to 0 \le x < 128)\ cv_{comp}\ bcat \\
&= \quad \{\ \text{BMAP' FUSION}\ \} \\
&\quad bmap'\ (bencode; bascii) \\
&\quad ;\ bfoldr'\ (\lambda\_\ (x,\_) \to 0 \le x < 128)\ cv_{comp}\ bcat \\
&= \quad \{\ \text{BFOLDR'-BMAP FUSION}\ \} \\
&\quad bfoldr'\ (\lambda\_\ as \to allsame\ as)\ cv_{comp}\ (bmapF\ (bencode; bascii); bcat)
\end{aligned}
$$

7.3 *Bidirectional Maximum Segment Sum*

Now let us turn to another example involving more advanced program calculation. The maximum segment sum is a classic problem in the area of program calculation. To demonstrate the ability of our calculation framework, we change the specification of *mss* in

Section 2.3 into a bidirectional version directly using contract-lens combinators, and optimize it to a more efficient version which has time complexity $O(n)$ in both get and put directions, meanwhile the semantics is preserved.

To see this concretely, let us first get a bidirectional version of *mss* without considering efficiency. To achieve this, we introduce a refinement type $\mathit{TailsList}\ a = \{as : [[a]] \mid (\forall\, 1 \le i < n,\ \mathit{tail}\ as_i = as_{i+1}) \wedge (\mathit{tail}\ as_n = [\,])\}$. 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

$$\begin{aligned}
&\mathit{bmss} : [\mathit{Int}] \leftrightarrow \mathit{Int}\\
&\mathit{bmss} = \mathit{binits}\\
&\qquad ;\ \mathit{bmapl}\ \widetilde{cv}_1\ \widetilde{cv}_2\ \ [\,]\ \mathit{btails}_{\mathit{init}}\\
&\qquad ;\ \mathit{bmapl}\ \widetilde{cv}_2\ \widetilde{cv}_3\ \ [\,]\ \mathit{bmapSum}\\
&\qquad ;\ \mathit{bmapl}\ \widetilde{cv}_3\ \mathit{ctrue}\ [\,]\ \mathit{bmaximum2}\\
&\qquad ;\ \mathit{bmaximum}'
\end{aligned}$$

where the definitions of the contracts and contract lenses appeared are

$$\begin{aligned}
&\widetilde{cv}_1 = \lambda a\ a' \to \mathit{init}\ a' = a\\
&\widetilde{cv}_2 = \lambda b\ b' \to \mathit{map}\ \mathit{init}\ (\mathit{init}\ b') = b\\
&\widetilde{cv}_3 = \lambda b\ b' \to \mathit{map}\ (+(-\mathit{last}\ b'))\ (\mathit{init}\ b') = b
\end{aligned}$$

$$\begin{aligned}
&\mathit{btails}_{\mathit{init}} : [\mathit{Int}] \Rightarrow [\mathit{Int}] \leftrightarrow \mathit{TailsList}\ \mathit{Int}\\
&\mathit{btails}_{\mathit{init}}\ a = \mathit{CLens}\ \mathit{tails}'\ (\lambda\, \_\ v \to \mathit{head}\ v)\ \mathit{cs}\ \mathit{cv}\\
&\quad \textbf{where}\ \mathit{cs} = \lambda\, \_\ a' \to \widetilde{cv}_1\ a\ a'\\
&\qquad\qquad\ \mathit{cv} = \lambda\, \_\ b' \to \widetilde{cv}_2\ (\mathit{tails}'\ a)\ b'
\end{aligned}$$

$$\begin{aligned}
&\mathit{bmapSum} : \mathit{TailsList}\ \mathit{Int} \Rightarrow \mathit{TailsList}\ \mathit{Int} \leftrightarrow [\mathit{Int}]\\
&\mathit{bmapSum}\ a = \mathit{CLens}\ (\mathit{map}\ \mathit{sum})\ p\ \mathit{cs}\ \mathit{cv}\\
&\quad \textbf{where}\ p\ \_\ \mathit{xs} = \mathit{map}\ (\lambda t \to t +\!\!+ [\mathit{last}\ \mathit{xs}])\ a +\!\!+ [[\mathit{last}\ \mathit{xs}]]\\
&\qquad\qquad\ \mathit{cs} \quad = \lambda\, \_\ a' \to \widetilde{cv}_2\ a\ a'\\
&\qquad\qquad\ \mathit{cv} \quad = \lambda\, \_\ b' \to \widetilde{cv}_3\ (\mathit{map}\ \mathit{sum}\ a)\ b'
\end{aligned}$$

$$\begin{aligned}
&\mathit{bmaximum2} : [\mathit{Int}] \Rightarrow [\mathit{Int}] \leftrightarrow \mathit{Int}\\
&\mathit{bmaximum2}\ a = \mathit{CLens}\ \mathit{maximum}\ p\ \mathit{cs}\ \mathit{ctrue}\\
&\quad \textbf{where}\ p\ \_\ x = \textbf{let}\ t = a +\!\!+ [0]\ \textbf{in}\ \mathit{map}\ (+(x - \mathit{maximum}\ t))\ t\\
&\qquad\qquad\ \mathit{cs} \quad = \lambda\, \_\ a' \to \widetilde{cv}_3\ a\ a'
\end{aligned}$$

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

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

$$\begin{aligned}
&\quad\ \mathit{bmss}\\
&= \quad \{\ \text{definition}\ \}\\
&\quad\ \mathit{binits}\\
&\quad ;\mathit{bmapl}\ \widetilde{cv}_1\ \widetilde{cv}_2\ \ [\,]\ \mathit{btails}_{\mathit{init}}
\end{aligned}$$

$$
\begin{aligned}
&\quad ;\mathit{bmapl}\ \widetilde{cv}_2\ \widetilde{cv}_3\ \ [\,]\ \mathit{bmapSum}\\
&\quad ;\mathit{bmapl}\ \widetilde{cv}_3\ \mathit{ctrue}\ [\,]\ \mathit{bmaximum2}\\
&\quad ;\mathit{bmaximum}'\\
=&\quad \{\ \text{BMAPL FUSION}\ \}\\
&\quad \mathit{binits}\\
&\quad ;\mathit{bmapl}\ \widetilde{cv}_1\ \mathit{ctrue}\ [\,]\ (\mathit{btails}_{\mathit{init}};\,;\mathit{bmapSum};\,;\mathit{bmaximum2})\\
&\quad ;\mathit{bmaximum}'\\
=&\quad \{\ \text{a specific bidirectional Horner's rule (to be discussed below)}\ \}\\
&\quad \mathit{binits}\\
&\quad ;\mathit{bmapl}\ \widetilde{cv}_1\ \mathit{ctrue}\ [\,]\ (\mathit{bfoldl}_{\mathit{init}}\ \mathit{ctrue}\ (-\infty)\ \ell)\\
&\quad ;\mathit{bmaximum}'\\
=&\quad \{\ \text{BIDIRECTIONAL SCAN LEMMA}\ \}\\
&\quad \mathit{bscanl}\ \mathit{ctrue}\ (-\infty)\ \ell\\
&\quad ;\mathit{bmaximum}'
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{btails}_{\mathit{init}};\,;\mathit{bmapSum};\,;\mathit{bmaximum2} = \mathit{bfoldl}_{\mathit{init}}\ \mathit{ctrue}\ (-\infty)\ \ell\\
&\quad \textbf{where}\\
&\qquad \ell : \mathit{Int} \Rightarrow \mathit{Either}\ \mathit{Int}\ (\mathit{Int},\mathit{Int}) \leftrightarrow \mathit{Int}\\
&\qquad \ell\ b = \mathit{CLens}\ g\ p\ \mathit{cs}\ \mathit{ctrue}\\
&\qquad\quad \textbf{where}\ g\ (\mathit{Left}\ ()) = -\infty\\
&\qquad\qquad\quad g\ (\mathit{Right}\ (x,y)) = \max\ (x+y)\ x\\
&\qquad\qquad\quad p\ \_\ t = \mathit{Right}\ (t - \max\ b\ 0, b)\\
&\qquad\qquad\quad \mathit{cs} = \lambda\,\_\ t' \to t' = \mathit{Right}\ (\_, b)
\end{aligned}
$$

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.

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

$$\begin{tikzcd}
{[3,-1,4,-1,5,-9]} \arrow[d, "get_{bscanl\ ctrue\ (-\infty)\ \ell}"] \arrow[dr, gray, bend left=10] & {[3,-1,4,-1,1,-5]} \\
{[3,2,6,5,10,1]} \arrow[d, "get_{bmaximum'}"] \arrow[dr, gray, bend left=10] & {[3,2,6,5,6,1]} \arrow[u, "put_{bscanl\ ctrue\ (-\infty)\ \ell}"'] \\
10 \arrow[r] & 6 \arrow[u, "put_{bmaximum'}"']
\end{tikzcd}$$

Fig. 3. Visualization of an example calculation of *bmss*.

8 Related Work

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

8.1 Lens Family and Partiality of *put*

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

8.1.1 Formulation of Contracts and Relation to Type Systems

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

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

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

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

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

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

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

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

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

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

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 because 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 dynamic checks are needed to ensure that the computation of edit lenses will not fail. For contract lenses, the *get* and *put* will not fail as long as the source conditions and view conditions are satisfied.

8.1.3 Totality with Maybe Monad

Another approach is to wrap the return type of *get* and *put* in the *Maybe* monad to remove partiality (Matsuda and Wang, 2015; Ko et al., 2016; Xia et al., 2019). The *put* direction is a total function of type $s \to v \to Maybe\ s$ 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. We want to know the static specification of a program and get meaning results instead of just getting a *Nothing* when the program fails. Moreover, the specification can guide the design of program calculation laws.

8.1.4 Other Discussions

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

8.2 Hoare-style Reasoning of Bidirectional Transformation

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

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

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

8.3 Bidirectionalization

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 convert existing unidirectional programs into bidirectional ones. Voigtländer (2009) 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 lacks modular reasoning of compositions, and therefore is not suitable for program calculation.

8.4 Calculating with Lenses

The goal of *generic point-free* lenses (Pacheco and Cunha, 2010) is the most similar to ours. In that work, lens combinators are designed for many traditional high-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 and 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 Formalisation with Agda

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

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

```
record Lens (S : Set) (V : Set) where
  field
  -- four-tuple
  get : S → V
  put : S → V → S
  cs  : S → S → Set
  cv  : V → V → Set
  -- laws
  BackwardValidity : ∀ (a : S) (b : V) → cv (get a) b → cs a (put a b)
  ForwardValidity  : ∀ (a : S) → cs a a → cv (get a) (get a)
  PutGet           : ∀ (a : S) (b : V) → cv (get a) b → get (put a b) = b
  GetPut           : ∀ (a : S) → cs a a → cv put a (get a) = a
```

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

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

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

is formalised in Agda using existential types as

$$bmap : \forall\, \{S\; V : Set\} \to (\exists\, (S \leftrightarrow V) \lambda \ell \to cv_\ell \Leftrightarrow ctrue \;\wedge\; cs_\ell \Leftrightarrow ctrue) \to ([S] \leftrightarrow [V])$$

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

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

10 Conclusion

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

This work focuses on the calculation for bidirectional transformations on lists, which mirrors the classic work on the *theory of list* (Bird, 1989, 1987) in the literature of program calculation. Generalizing this bidirectional program calculation framework to algebraic datatypes generated by polynomial functors is a natural next step. Another possible future work is to design practical systems based on contract lenses to reason about and optimize BXs, automating the verification of round-tripping properties and lens composition using SMT solvers.

Conflict of Interest

None.

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J. & Stevens, P. (2018) Introduction to bidirectional transformations. In *Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*. Springer International Publishing. Cham. chapter 1, pp. 1–28. Available at: https://doi.org/10.1007/978-3-319-79108-1_1.

Bancilhon, F. & Spyratos, N. (1981) Update semantics of relational views. *ACM Trans. Database Syst.* **6**(4), 557–575.

Barbosa, D. M., Cretin, J., Foster, N., Greenberg, M. & Pierce, B. C. (2010) Matching lenses: Alignment and view update. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 193–204.

Bird, R. S. (1987) An introduction to the theory of lists. Logic of Programming and Calculi of Discrete Design. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 5–42.

Bird, R. S. (1989) Algebraic identities for program calculation. *Comput. J.* **32**(2), 122–126.

Bird, R. S. (1989) Lectures on constructive functional programming. Constructive Methods in Computing Science. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 151–217.

Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A. & Schmitt, A. (2008) Boomerang: Resourceful lenses for string data. Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 407–419.

Bohannon, A., Pierce, B. C. & Vaughan, J. A. (2006) Relational lenses: A language for updatable views. Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. New York, NY, USA. Association for Computing Machinery. p. 338–347.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* **29**(3), 17–es.

Foster, J. N., Pilkiewicz, A. & Pierce, B. C. (2008) Quotient lenses. Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 383–396.

Gibbons, J. (2002) Calculating functional programs. Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. Springer-Verlag. pp. 148–203.

Gibbons, J. (2011) Maximum segment sum, monadically (distilled tutorial). *Electronic Proceedings in Theoretical Computer Science*. **66**, 181–194.

Gill, A., Launchbury, J. & Peyton Jones, S. L. (1993) A short cut to deforestation. Proceedings of the Conference on Functional Programming Languages and Computer Architecture. New York, NY, USA. Association for Computing Machinery. p. 223–232.

He, X. & Hu, Z. (2018) Putback-based bidirectional model transformations. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA. Association for Computing Machinery. p. 434–444.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K. & Nakano, K. (2010) Bidirectionalizing graph transformations. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 205–216.

Hofmann, M., Pierce, B. & Wagner, D. (2011) Symmetric lenses. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 371–384.

Hofmann, M., Pierce, B. & Wagner, D. (2012) Edit lenses. Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 495–508.

Hu, Z., Iwasaki, H. & Takeichi, M. (1996) Deriving structural hylomorphisms from recursive definitions. Proceedings of the First ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 73–82.

Ko, H.-S. & Hu, Z. (2018) An axiomatic basis for bidirectional programming. Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery.

Ko, H.-S., Zan, T. & Hu, Z. (2016) Bigul: A formally verified core language for putback-based bidirectional programming. Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. New York, NY, USA. Association for Computing Machinery. p. 61–72.

Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2007) Bidirectionalization transformation based on automatic derivation of view complement functions. Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 47–58.

Matsuda, K. & Wang, M. (2015) Applicative bidirectional programming with lenses. Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. Association for Computing Machinery. p. 62–74.

Matsuda, K. & Wang, M. (2015) ”bidirectionalization for free” for monomorphic transformations. *Sci. Comput. Program.* **111**(P1), 79–109.

Matsuda, K. & Wang, M. (2018) Hobit: Programming lenses without using lens combinators. European Symposium on Programming. Springer. pp. 31–59.

Matsuda, K. & Wang, M. (2020) *Sparcl: A language for partially-invertible computation*. **4**(ICFP).

Pacheco, H. & Cunha, A. (2010) Generic point-free lenses. Proceedings of the 10th International Conference on Mathematics of Program Construction. Berlin, Heidelberg. Springer-Verlag. p. 331–352.

Pacheco, H. & Cunha, A. (2011) Calculating with lenses: Optimising bidirectional transformations. Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. New York, NY, USA. Association for Computing Machinery. p. 91–100.

Stevens, P. (2008) A landscape of bidirectional model transformations. Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 408–424.

Stevens, P. (2014) Bidirectionally tolerating inconsistency: Partial transformations. Fundamental Approaches to Software Engineering. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 32–46.

Stevens, P. (2020) Maintaining consistency in networks of models: bidirectional transformations in the large. *Software and Systems Modeling*. **19**(1), 39–65.

Tran, V.-D., Kato, H. & Hu, Z. (2020) Birds: Programming view update strategies in datalog. 46th International Conference on Very Large Data Bases. VLDB Endowment. p. 2897–2900.

Tsigkanos, C., Li, N., Jin, Z., Hu, Z. & Ghezzi, C. (2020) Scalable multiple-view analysis of reactive systems via bidirectional model transformations. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. New York, NY, USA. Association for Computing Machinery. p. 993–1003.

Voigtländer, J. (2009) Bidirectionalization for free! (pearl). Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 165–176.

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

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

1 Calculating with Total Lenses

It is possible to make *bmap* total:

```
$bmap_{total} \quad : A \to (A \leftrightarrow B) \to ([A] \leftrightarrow [B])$
$bmap_{total}\ a_0\ \ell = Lens\ (map\ get_\ell)\ p$
  where $p\ \_ \quad [\,] \qquad = [\,]$
        $p\ (x : xs)\ (y : ys) = put_\ell\ x\ y : p\ xs\ ys$
        $p\ [\,] \qquad (y : ys) = put_\ell\ a_0\ y : p\ [\,]\ ys$
```

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

One can develop an associated map fusion law for it:

$$\frac{get_{\ell_1}\ a_0 = b_0}{bmap_{total}\ a_0\ \ell_1 ; bmap_{total}\ b_0\ \ell_2 = bmap_{total}\ a_0\ (\ell_1 ; \ell_2)} \qquad \textsc{BMapTotal Fusion}$$

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

2 Equivalent Implementation of Combinators

This appendix shows the code for equivalent implementations of some contract-lens combinators in Section 5.

2.1 *Efficient bfoldr*

This section shows an efficient implementation of *bfoldr*.

```
$bfoldr\ \ell = CLens\ (foldr'\ get_\ell)\ p\ ctrue\ ctrue$
  where $p\ as\ b' = $ let $bs = tail\ (scanr\ (\lambda a\ b \to get_\ell\ (Right\ (a, b)))\ (get_\ell\ (Left\ ()))\ as)$
                 in $go\ as\ bs\ b'$
        $go\ [\,]\ [\,]\ b' = $ case $put_\ell\ (Left\ ())\ b'$ of
                          $Left\ () \to [\,]$
                          $Right\ (a', bim') \to a' : go\ [\,]\ [\,]\ bim'$
        $go\ (a : as)\ (bim : bs)\ b' = $ case $put_\ell\ (Right\ (a, bim))\ b'$ of
                                         $Left\ () \to [\,]$
                                         $Right\ (a', bim') \to a' : go\ as\ bs\ bim'$
```

2.2 *Implementation of bmap and bmap′ with bfoldr′*

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

```
$bmap : (\{ctrue\}\ S \leftrightarrow V\ \{ctrue\}) \to ([S] \leftrightarrow [V])$
$bmap\ \ell = bfoldr'\ ctrue\ eqlength\ \ell'$
  where
    $\ell' :: Either\ ()\ (S, [V]) \leftrightarrow [V]$
    $\ell' = CLens\ g\ p\ (lift\ ctrue\ eqlength)\ eqlength$
    $g\ (Left\ ()) = [\,]$
```

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

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

where

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

3 Examples for Combinators

This appendix shows examples for some contract-lens combinators and calculation laws in Section 5 and Section 6.

3.1 Computation Instances of *bmaximum*

This section shows two calculation instances of the *bmaximum* example in Section 5.1. Let us assume that $get_{bmaximum}\ [9, 2, 5]$ 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 $[4, 2, 4]$.

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

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

$$\begin{aligned}
& put_{bmaximum}\ [9,2,5]\ 10 \\
= & \quad \{ \text{ since } put_{bmax}\ (Right\ (9, get_{bmaximum}\ [2,5]))\ 10 = Right\ (10,5) \ \} \\
& 10 : put_{bmaximum}\ [2,5]\ 5 \\
= & \quad \{ \text{ since } put_{bmax}\ (Right\ (2, get_{bmaximum}\ [5]))\ 5 = Right\ (2,5) \ \}
\end{aligned}$$

$$\begin{array}{ll} & 10:2:put_{bmaximum}\,[5]\,5 \\ = & \{\text{ since } put_{bmax}\,(Right\,(5, get_{bmaximum}\,[\,]))\,5 = Right\,(5, -\infty)\,\} \\ & 10:2:5:put_{bmaximum}\,[\,]\,(-\infty) \\ = & \{\text{ since } put_{bmax}\,(Left\,())\,(-\infty) = Left\,()\,\} \\ & 10:2:5:[\,] \end{array}$$

3.2 Example of *bmap′*

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

$$\begin{array}{l} bdoubles : [Int] \leftrightarrow [Int] \\ bdoubles = bmap'\; bdouble \\ \quad \textbf{where}\; bdouble \;:\; Int \leftrightarrow Int \\ \qquad\qquad bdouble = CLens\,(*2)\,(\lambda\_\,v' \to div\,v'\,2)\; ctrue\,(\lambda\_\,b \to mod\,b\,2 = 0) \end{array}$$

3.3 Example of *bmapl*

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

$$\begin{array}{l} bsort : [Int] \leftrightarrow [Int] \\ bsort = CLens\; sort\; p\; ctrue\; (\lambda t\; as \to (\forall\, 1 < i \leq |as|.\; as_{i-1} \leq as_i)\; \wedge\; eqlength\; t\; as) \\ \quad \textbf{where} \\ \qquad p\; s\; v = \textbf{let}\; positions = map\, fst\; \$\; sortOn\; snd\; (zip\; [0\,..]\; s)\; \textbf{in} \\ \qquad\quad map\; snd\; \$\; sortOn\; fst\; (zip\; positions\; v) \end{array}$$

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.

$$\begin{array}{l} bmapl\;(\leq)\; ctrue\;(-\infty)\; bmod10 : [S] \leftrightarrow [V] \\ \quad \textbf{where} \\ \qquad bmod10 : (a : S) \Rightarrow (S \leftrightarrow V) \\ \qquad bmod10\; a = CLens\;(\lambda x \to mod\;x\;10)\; p\;(\lambda\_\,a' \to a \leq a')\; ctrue \\ \qquad\quad \textbf{where}\; p\;x\;y = \textbf{if}\; mod\;x\;10 = y\; \textbf{then}\; go\;x\; \textbf{else}\; go\;y \\ \qquad\qquad\qquad go\;x\; = \textbf{if}\; x > a\; \textbf{then}\; x\; \textbf{else}\; go\;(x + 10) \end{array}$$

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

3.4 Example of $bfoldl_{init}$

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

$$\begin{aligned}&bprefixSum = \\ &\quad binits; bmapl\ (\lambda s\ s' \to init\ s' = s)\ ctrue\ [\,]\ (bfoldl_{init}\ ctrue\ 0\ badd) : [Int] \leftrightarrow [Int]\end{aligned}$$

where the $badd$ is defined as

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

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

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

3.5 Example of $bscanl$

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 $bscanl$.

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

3.6 Example of Bidirectional Scan Lemma

We give a simple example which makes use of the BIDIRECTIONAL SCAN LEMMA to derive an efficient bidirectional program from an inefficient one. Recall the *bprefixSum* defined in Appendix 3.4 for calculating the prefix sums of a list. It has time complexity $O(n^2)$. Applying the BIDIRECTIONAL SCAN LEMMA to it, we can derive $bscanl\ ctrue\ 0\ badd$, which has time complexity $O(n)$ in both forward and backward transformations.