Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Generic recursive lens combinators and their calculation laws

Ruifeng Xie [a,b], Zhenjiang Hu [a,b,*]

[a] Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[b] School of Computer Science, Peking University, Beijing, China

A R T I C L E I N F O

Article history:
Received 21 July 2021
Received in revised form 6 February 2022
Accepted 14 February 2022
Available online xxxx

Keywords:
Bidirectional transformations
Program calculation
Programming language theory

A B S T R A C T

Bidirectional transformation is a generic method for synchronizing two related data structures, with applications in databases, software model transformation, graph transformation, etc. Since the data to synchronize often have complicated and disparate structures, bidirectional transformations are hard to develop, correctly comprehend, reason about, or maintain. Programming language researchers have invested much efforts to invent new frameworks and paradigms, to aid the development of complex bidirectional transformations. However, these frameworks are not always applicable due to certain subtleties in the problem specification; some bidirectional programs become too complicated and unstructured to verify their correctness, not to mention their poor efficiency. In this paper, we propose a collection of generic recursion patterns for bidirectional transformations, and develop corresponding calculation laws. With these generic recursion patterns, we can develop structured bidirectional transformations, which are easy to understand and maintain. With these calculation laws, we can effectively optimize bidirectional programs, while preserving its very semantics. We demonstrate, through calculation of bidirectional transformations on trees, the expressiveness of the combinators and the effectiveness of the calculation laws.

1. Introduction

John Backus first proposed the notion of *functional programming* back in 1977 in his Turing Award lecture [1]. Traditional von-Neumann-style programs are word-at-a-time by repetition and modification. The use of an internal state makes imperative programs hard to understand. In contrast, functional programs are compositional, built from small functions and functional forms.

Apart from the use of immutable state and the lack of side effects, functional programming also discourages explicit recursions. Recursions are like loops with goto in imperative languages, so the abuse of explicit recursion with free structure would lead to *unstructured* functional programs which is difficult to understand. Ideally, we should first build generic *combinators* (such as *map* and *reduce*), capturing primitive recursive structures, and then construct functional programs with these combinators.

Based on functional programming, Richard Bird [2] proposed the idea of *program calculation*, making *calculational* proofs based on algebraic reasoning. Not only explicit recursion is discouraged, explicit induction in proofs should also be discouraged. By Curry-Howard isomorphism, induction in logic and recursion in programming is isomorphic, so using induction in

* Corresponding author.
E-mail addresses: xieruifeng@pku.edu.cn (R. Xie), huzj@pku.edu.cn (Z. Hu).

https://doi.org/10.1016/j.tcs.2022.02.019
0304-3975/© 2022 Elsevier B.V. All rights reserved.

proofs, in one way or another, always requires reducing those functional combinators into the raw recursions they are built upon. The method of program calculation suggests first building calculation laws for primitive functional combinators, and then perform calculations directly on these combinators.

All mentioned above are *unidirectional programs*. The need of bidirectional programs arose as the "view-update" problem in database: the program first fetches the original data from the database (called *source*), and performs some calculations on that source to get an *abstract view* (the forward transformation). After that, the view gets modified, and the program needs to synchronize the modification back to the source (the backward transformation). Since the forward transformation $get : S \to V$ usually does not preserve all the necessary information in the abstract view V to recover the source S, the backward transformation requires an additional original source besides the updated view: $put : V \times S \to S$. Bidirectional transformations must satisfy some roundtrip properties to ensure well-behavedness. Besides, it is usually the case that more than one well-behaved *put* exists for a specific *get*, among which we must have an approach to decide which one is intended. Nathan Foster et al. proposed the *lens* framework [3] to solve this problem: by providing a set of *lens combinators*, in which every expression simultaneously specifies both a *get* function and the corresponding *put*, bidirectional transformations can be built up with these combinators.

Lens combinators make development of bidirectional transformations modular, but the current lens framework still lacks combinators that manipulate recursive data structures and the ability to reason directly about those combinators (program calculation for lenses). Hugo Pacheco presented some recursive lens combinators [4] and the corresponding calculation laws [5], however, he used a different flavor of lens, requiring an extra $create : V \to S$ component, and yet many useful combinators (such as those on tree structures as proposed by Jeremy Gibbons [6]) are still missing.

It turns out that the existing lens framework is not powerful enough to define those missing combinators, because it has no flexible control over changes for both forward and backward transformations. To solve this problem, Zhang et al. proposed an idea of contract-lenses [7]: lenses are equipped with a pair of contracts (predicates that are expected/ensured to be true) *csource* and *cview*, to guard the changes on sources and views, which provides us a more accurate description of the backward semantics. The good thing is that the existing lens framework can be seamlessly embedded into contract lenses, and the missing combinators can be defined with their respective calculation laws. However, the original description of contract lenses works only on lists, and it is a challenge to extend it to arbitrary algebraic data types:

- List is a *concrete* data structure, i.e., the constructors (and therefore the overall structure) of lists are known in advance. Combinators on lists can take this advantage and make full use of various properties of lists. Also, whenever we need a default value, the empty list is probably a sensible candidate. However, to generalize over arbitrary data structures, we have to be careful on what properties we can assume, and we have no default values as a fallback.
- List is *linear* in its nature: every node (except for the first and the last) has exactly one predecessor and one successor. This simplifies the relationship between different nodes in a list: usually nodes only depend on its two neighbors, and we can describe such dependencies with simple binary relations.
- List, no matter seen left-to-right or right-to-left, has essentially the same structure. Therefore bottom-up and top-down recursions on lists are isomorphic, but generally these two recursive patterns are vastly different, especially when considering the backward semantics for lenses.
- Every node in a list has exactly one piece of data associated with it. Accumulations (the *scan* combinator) also produce a single result on each node. Due to this coincidence, *scanl* and *scanr* for lists have the list type itself as their return type. But in the general case, a new type is necessary.

In this paper, we extend the theory of contract lenses from lists to generic recursive data structures, together with a set of calculation laws for reasoning about generic bidirectional transformations. Our main contributions can be summarized as follows.

- *Lifting Contracts.* For every polynomial (strictly-positive) functor F, which is used to define generic data structures, we provide a generic method to describe contracts (binary relations) on a pair of F-structures $F\,A \times F\,B$ by *lifting* contracts on $A \times B$. This enables description of a wide class of recursive contract-lenses combinators on generic data structures. The results have been verified using Agda [8].[1]
- *Expressive Recursive Lens Combinators.* We provide various recursive combinators for contract-elaborated lenses, classified into two groups: bottom-up and top-down. For bottom-up ones, different from previous works [4,5] which require an additional *create* component on all lenses involved, ours are more flexible, allowing an explicit and optional *create* parameter. Our top-down recursive combinators are new for the bidirectional world, and both *fold* and *scan* are provided, while we expect the *scan* scheme to be arguably more natural and useful in practice.
- *Generic Calculation Laws.* We develop a set of calculation laws for lens combinators mentioned-above, following the idea of program calculation by Richard Bird. This will be useful for systematic development and reasoning about bidirectional

[1] The Agda implementation of this paper can be found in the following GitHub repository: https://github.com/ruifengx/generic-recursive-lens-combinators.git.

Notation	Definition	Description
$f \circ g$	$\lambda x. f\,(g\,x)$	The *composition* of two functions, `(.)` in Haskell Reused for *lens composition*, ambiguity resolved by context
id	$\lambda x. x$	The *identity* function, `id` in Haskell
$\underline{a}$	$\lambda\_. a$	The *constant* function, `const` in Haskell
$f \vartriangle g$	$\lambda x. (f\,x, g\,x)$	The *fork* for f and g, `Control.Arrow.&&&` in Haskell
$f \triangledown g$	$\lambda \begin{cases} (i_1\ x).\ f\ x \\ (i_2\ y).\ g\ y \end{cases}$	The *join* for f and g, `either` in Haskell
π_1	$\lambda(x, \_). x$	Projection 1, `fst` in Haskell
π_2	$\lambda(\_, y). y$	Projection 2, `snd` in Haskell
$f \times g$	$(f \circ \pi_1) \vartriangle (g \circ \pi_2)$	The *product* for f and g, `Data.Bifunctor.bimap` in Haskell
i_1		Injection 1, `Left` in Haskell
i_2		Injection 2, `Right` in Haskell
$f + g$	$(i_1 \circ f) \triangledown (i_2 \circ g)$	The *sum* for f and g, `Data.Bifunctor.bimap` in Haskell
$F\,f$	$fmap_F\,f$	Functor mapping of functor F on function f, `fmap` in Haskell
$F\,create\,f$		Functor mapping of functor F on lens f, with creation function *create*
$A \times B$	(A, B)	The *product type* of A and B, known in Haskell as a 2-tuple (pair)
$\mathcal{P}(X)$	$X \to Bool$	Predicates on X
$\mathcal{P}(A \times B)$	$A \times B \to Bool$	Contracts on $A \times B$
$\{x : A \mid P(x)\}$		Refined type: works the same way as set comprehensions Note that $\{x : A \mid \cdots\} \to B$ makes x available in B
$\Sigma\,(x : A)\,(B\,x)$		Dependent sums: works the same ways as set unions $\bigcup_{x \in A} B(x)$ Note that $B : A \to Set$ gives a set $B(x)$ for any $x \in A$

Fig. 1. Notations used throughout this paper.

transformations. We take the approach [9,10] of first establishing some *universal properties*, and then automatically deriving other fusion laws.

The organization of the rest of the paper is as follows. In Section 2, we briefly summarize the idea of elaborating lenses with a pair of source and view contracts. Then, in Section 3, we provide a generic method to describe contracts (binary relations, or predicates) on pairs of functor structures, and then use it to define functor mappings for such contract-elaborated lenses. Finally, in Sections 4 and 5, we discuss two types of recursive combinators: bottom-up and top-down, with special cares for the *scan* combinator for the inherent data dependency involved.

In Fig. 1, we list the important notations that will be used throughout this paper. Most of these notations are derived from [4]. In addition, we will use Agda/Haskell-like notations as default in the rest of this paper. Regarding the associativity of the operators, function arrow ($\to$) has the lowest precedence, and associates to the right; all the other binary operators are associative, precedence given explicitly using parentheses. Regarding the naming convention, for single-letter names, we will use A, B for types, C for containers, F, G for functors, P, Q for predicates, l, f, g, h for lenses,[2] n, k for natural numbers, s, p for container-related stuffs (details in Section 3.3), and other lowercase letters for parameters and local variables.

2. Review: lenses and contract-lenses

In this section, we briefly summarize the lens framework and contract lenses.

2.1. Lenses

A lens [3] consists of two functions:

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

We write $f : A \leftrightarrow B$ for f being a lens from A to B, and $get\,f$, $put\,f$ for its two respective components. A lens $f : A \leftrightarrow B$ is well-behaved, iff the following two laws are satisfied:

$$put\,f \circ (get\,f \vartriangle id) = id \tag{GetPut}$$

$$get\,f \circ put\,f = \pi_1 \tag{PutGet}$$

Or in pointful notation: for all $x \in A$ and $y \in B$, the following hold.

[2] f and g are reused for normal functions in Sections 3.1, 3.2, 3.5, and 4.1, because f and g are symbols for functions by convention, and no lenses ever appear in those sections, so we should be safe from ambiguities.

$$\mathit{put}\, f\ (\mathit{get}\, f\ x, x) = x \qquad (\text{GetPut})$$

$$\mathit{get}\, f\ (\mathit{put}\, f\ (y, x)) = y \qquad (\text{PutGet})$$

The GetPut law tells us that if the view is not changed at all, then so is the source; the PutGet law tells us that a source synchronized with some view yields that exact view upon subsequent *get*s.

2.2. Problem of partiality

We usually expect the *get* and *put* components of any lens are total functions, i.e., they are well-defined on every point of their respective domain. However, this is not a trivial requirement, and some lenses do fail to fulfill this requirement. Consider the following definition of $\mathit{double} : \mathbb{N} \leftrightarrow \mathbb{N}$ for synchronizing a natural number with one that doubles itself:

$$\begin{aligned}&\mathit{double} : \mathbb{N} \leftrightarrow \mathbb{N}\\ &\mathit{get\ double}\ n = 2 \times n\\ &\mathit{put\ double}\ (n, \_) = \left\lfloor \tfrac{n}{2} \right\rfloor\end{aligned}$$

Notice how a flooring division is used in *put double*. Thanks to the use of natural numbers, it is the closest thing we can get. But then for every odd number y and an arbitrary x, the PutGet law is not satisfied: $\mathit{get\ double}\ (\mathit{put\ double}\ (y, x)) = 2 \times \left\lfloor \frac{y}{2} \right\rfloor = y - 1 \neq y$. The essence of this problem is that the modified source y lands outside the range of *get*, and thus a proper *put* does not exist: there is no such x that $\mathit{get\ double}\ x = y$.

This specific problem can be solved by shrinking the view type to $2\mathbb{N}$: a lens $\mathit{double}' : \mathbb{N} \leftrightarrow 2\mathbb{N}$ (where $2\mathbb{N} = \{2\ n \mid n \in \mathbb{N}\}$) with the same definition satisfies both roundtrip laws. Now $\mathit{get\ double}'$ is a surjection onto the view type $2\mathbb{N}$, so theoretically a law-preserving lens always exists.

However, even if *get* is a surjection, it is still possible that we have no meaningful way to define the *put* component. The following example is completely unwarranted, but should serve well as an illustration:

$$\begin{aligned}&\mathit{foo} : \mathit{Maybe}\ \{s : \mathit{String} \mid \mathit{length}\ s \geqslant 3\} \leftrightarrow \mathit{Maybe\ String}\\ &\mathit{get\ foo\ Nothing} = \mathit{Nothing}\\ &\mathit{get\ foo}\ (\mathit{Just}\ s) \ \ = \mathit{Just}\ (\mathit{drop}\ 3\ s)\\ &\mathit{put\ foo}\ (\mathit{Nothing}, \_) \qquad = \mathit{Nothing}\\ &\mathit{put\ foo}\ (\mathit{Just}\ s, \mathit{Just}\ s_0) \ \ = \mathit{Just}\ (\mathit{take}\ 3\ s_0 +\!\!+ s)\\ &\mathit{put\ foo}\ (\mathit{Just}\ s, \mathit{Nothing}) = \mathit{Just}\ (? +\!\!+ s)\end{aligned}$$

This *foo* manipulates on an optional string. If the string is presented, *get foo* drops the first 3 characters. The backward transformation tries to recover the original string by concatenating fragment from the original source ($\mathit{take}\ 3\ s_0$) and the modified view (s). Note how we have already refined the source and view types, and still it leaves out one possibility: what if the original source was *Nothing*, but modified view is some *Just s*? In that case, we should fill in some string of length 3 at the place of the question mark, but we have no obvious choice. In fact, there is no *real* obstacle to prevent us from filling in a random string there, but that is *far* from ideal.

What can we do in this case to solve the problem? Note that *put* is actually defined on $V \times S$, so shrinking V alone is not enough. Generally, we can enforce some constraint on $V \times S$ pairs. In the previous example, to prevent *put*ting a *Just s* into a *Nothing*, we can write:

$$\begin{aligned}&\mathit{put\ foo} : \{(s', s) : \mathit{Maybe\ String} \times \mathit{Maybe\ String}' \mid \mathit{SameShape}\ s'\ s\} \rightarrow \mathit{Maybe\ String}'\\ &\quad \textbf{where}\ \mathit{String}' = \{s : \mathit{String} \mid \mathit{length}\ s \geqslant 3\}\\ &\qquad\quad \mathit{SameShape\ Nothing\ Nothing} = \mathit{True}\\ &\qquad\quad \mathit{SameShape}\ (\mathit{Just}\ \_)\ \ (\mathit{Just}\ \_) \ = \mathit{True}\\ &\qquad\quad \mathit{SameShape}\ \_ \qquad\quad \_ \qquad\ \ = \mathit{False}\end{aligned}$$

However, after experimenting with the new type of *put* for a while, one will eventually notice that this type does not compose well. The type of *put* is asymmetric, and the *put* component of the composition of $f : B \leftrightarrow C$ and $g : A \leftrightarrow B$ involves *get g*:

$$\mathit{put}\ (f \circ g)\ (z', x) = \mathit{put}\ g\ (\mathit{put}\ f\ (z', \mathit{get}\ g\ x), x)$$

In a long chain of composition $f_n \circ \cdots \circ f_2 \circ f_1$ (where $f_k : A_k \leftrightarrow A_{k+1}$), all of $\mathit{get}\, f_1, \cdots, \mathit{get}\ f_{n-1}$ are involved into the *put* component of the final composition. If f_n has a non-trivial constraint P on $A_{n+1} \times A_n$, the composition will in turn have a constraint $P'(x_{n+1}, x_1) = P(x_{n+1}, f_{n-1}\ (\cdots\ (f_1\ \ x_1)))$. In this way, every f_k that has such a non-trivial constraint contributes one term of size $O(k)$ to the constraint of the composition, which results in a $O(n^2)$-sized constraint in total. It is horrible to keep track of (not to mention to reason with) such constraints.

The inconvenience mentioned above arises from the entanglement of *get* and *put* in lens composition. Fortunately, there is another representation (or encoding) for lenses with a composition rule with more symmetry: we can represent a lens (equivalently) as *get* and *modify*.

$$\mathbf{data}\ S \leftrightarrow V = \mathit{Lens}\ \{\mathit{get} : S \to V, \mathit{modify} : (V \to V) \to (S \to S)\}$$

This encoding is at somewhere between the *get*-*put* encoding and the van Laarhoven encoding [11]. One can easily prove that the *get*-*modify* encoding is indeed equivalent to the *get*-*put* encoding: for every lens $l : S \leftrightarrow V$, $\mathit{put}\ l\ (y, x) = \mathit{modify}\ l\ \underline{y}\ x$ and $\mathit{modify}\ l\ t\ x = \mathit{put}\ l\ (t\ (\mathit{get}\ l\ x), x)$; also, the roundtrip laws are equivalent to ModifyId ($\mathit{modify}\ l\ \mathit{id} = \mathit{id}$) and Modify$\overline{\text{G}}$et (for every t, $\mathit{get}\ l \circ \mathit{modify}\ l\ t = t \circ \mathit{get}\ l$).

However, the real significance of this encoding is reflected in its composition rule:

$$\begin{aligned} \mathit{get}\ (f \circ g) &= \mathit{get}\ f \circ \mathit{get}\ g \\ \mathit{modify}\ (f \circ g) &= \mathit{modify}\ g \circ \mathit{modify}\ f \end{aligned}$$

Note how *get* and *modify* are disentangled this time. Instead of enforcing a constraint on the input of *put* ($V \times S$), since *modify* is effectively propagating a change on its view V to that on its source S, we can enforce constraints on these changes. In other words, these constraints are specialized Hoare conditions for *modify*: where generally we could use predicates of type $\mathcal{P}(X \to X)$ (with X respectively being S and V) for the pre- and post- conditions, we use a $\mathcal{P}(X \times X)$ instead.

2.3. Contract lenses

With the *get*-*modify* encoding, lenses are elaborated with contracts as the following:

$$\begin{array}{l} \mathbf{record}\ \{\mathit{csource}\}\ S \leftrightarrow V\ \{\mathit{cview}\}\ \mathbf{where} \\ \quad \mathit{get} \quad\ \ : \ S \to V \\ \quad \mathit{modify} \ : \ ((v : V) \to \{v' : V \mid \mathit{cview}\ v'\ v\}) \\ \qquad\qquad \to ((s : S) \to \{s' : S \mid \mathit{csource}\ s'\ s\}) \end{array}$$

Or with the better-known (and often more convenient) *get*-*put* encoding [7]:

$$\begin{array}{l} \mathbf{record}\ \{\mathit{csource}\}\ S \leftrightarrow V\ \{\mathit{cview}\}\ \mathbf{where} \\ \quad \mathit{get} : S \to V \\ \quad \mathit{put} : \{(v', s) : V \times S \mid \mathit{cview}\ v'\ (\mathit{get}\ s)\} \to \{s' : S \mid \mathit{csource}\ s'\ s\} \end{array}$$

We call the two contracts (or predicates, or binary relations indeed) $\mathit{csource} : \mathcal{P}(S \times S)$ and $\mathit{cview} : \mathcal{P}(V \times V)$ the source and view contracts respectively. New source and new view are chosen to be the left parameter of *csource* and *cview*, to match the position of new view in *put*. Also, in combinators introduced later (namely *clift* and *fzip*; for details, refer to Section 3), this choice makes the first parameter the "preferred" one, whose shape is always respected by *clift* and preserved by *fzip*.

Contract lenses can be *composed* in the same way as normal lenses except for an additional requirement on their contracts: for two contract lenses $f : \{P_1\}\ B \leftrightarrow C\ \{Q_1\}$ and $g : \{P_2\}\ A \leftrightarrow B\ \{Q_2\}$, $f \circ g$ is well-defined iff for every pair of $(x', x) \in B \times B$, $P_1(x', x) \Rightarrow Q_2(x', x)$.

Besides, contract lenses have their roundtrip properties a little differently:

$$\mathit{cview}\ f\ (\mathit{get}\ f\ x, \mathit{get}\ f\ x) \Rightarrow \mathit{put}\ f\ (\mathit{get}\ f\ x, x) = x \tag{GetPut}$$

$$\mathit{cview}\ f\ (y, \mathit{get}\ f\ x) \Rightarrow \mathit{get}\ f\ (\mathit{put}\ f\ (y, x)) = y \tag{PutGet}$$

The changes here involve adding appropriate premises for both laws.

Remark 2.1. It might be surprising at first glance that both of the laws have $\mathit{cview}\ f\ (-, -)$ as premises. Note that the predicate *csource* only ever appears in the result type of *put*, and the predicate *cview* only ever appears in the parameter type of *put*. Both of the laws involve applying *put*, therefore require *cview* as premises. In theory, *csource* would appear in the conclusion of both laws. They are omitted here because (a) they are implied by the type signature of *put* anyway, and (b) they are not essential to both of the laws.

Now that we have all the things prepared, we are going to define some combinators and develop the calculation laws for these combinators. Recursive combinators have always been defined on fixpoints of strictly positive functors. Therefore we first specify how contracts work with functors by defining a *clift* combinator on contracts.

3. Preparations: functor for contract lenses

Recursion patterns (or recursive combinators) are said to be *generic* if they are universally applicable to any recursive data types. We see recursive data types as fixpoints of strictly positive functors, and we *abstract over* these functors in the definition of recursive combinators.

Recursive lens combinators manipulate lenses instead of normal unidirectional functions, so for lenses, we need a functor mapping of F to lift a lens $f : A \leftrightarrow B$ to $F\,f : F\,A \leftrightarrow F\,B$. As noted in [5], such functor mappings can be constructed from the *fzip* combinator:

$$\begin{aligned}&\mathit{fzip} : (A \to B) \to F\,A \times F\,B \to F\,(A \times B)\\ &\mathit{get}\ (F\ \mathit{create}\ f) \ = F\ (\mathit{get}\ f)\\ &\mathit{put}\ (F\ \mathit{create}\ f) = F\ (\mathit{put}\ f) \circ \mathit{fzip}\ \mathit{create}\end{aligned}$$

Note that the above *fzip* requires a *create* function (with type $A \to B$). The reason can be roughly explained as follows: *fzip* works by *pairing up* As and Bs in the input, but values of $F\,A$ and $F\,B$ might have different *shapes* (e.g., for $F = \mathit{Maybe}$, *Nothing* and *Just* x have different shapes), and therefore a *create* is required to fill in the missing parts. We will further explain about *create* in Section 3.2, and provide formalized descriptions for *shapes* etc. in Section 3.3.

To extend the above idea from lenses to contract-lenses, we should take into account contracts on the source and view of lenses. To do so, we need to come up with (1) a way to describe contracts over pairs of F-structures (for simplicity of notations, we refer to values of type $F\,A$ for any A as an *F-structure*), and (2) a variant of *fzip* that can react to these contracts.

Before we start, we would like to state (in advance) that the theory discussed in the following subsections applies to functors with arbitrary arities. In Section 3.3, we will construct concretely relevant combinators for n-ary functors in general. However, in the first two subsections, we restrict all the combinators and laws to monofunctors, since we believe that it conveys the idea better, and the full version (for n-ary functors) can be recovered without difficulty. At the end of this section, we provide an example.

3.1. *Lifting contracts with clift*

If we allow arbitrary contracts on $F\,A$ and $F\,B$, we can hardly use these opaque contracts. Instead, the inherent structure of functor F should be reflected in the relational contract on a pair of F-structures. A contract P is essentially a binary relation on two sets A and B, or predicates on $A \times B$ pairs, encoded as $P : \mathcal{P}(A \times B)$. We abstract over F by *lifting* contracts on $A \times B$ to those on $F\,A \times F\,B$:

$$\mathit{clift} : \mathcal{P}(A \times B) \to \mathcal{P}(F\,A \times F\,B)$$

This *clift* can be considered as an additional primitive combinator for functors on contract lenses, just as *fzip* is to functors on lenses without contracts, or as functor mapping $F-$ (i.e., *fmap* in Haskell) is to a normal unidirectional functor F.

We will state some critical properties of *clift*; some of these properties might not immediately appear useful, but all of them can be satisfied by the concrete definition we shall give, and will come in handy for proving the properties of *clift*, *fzip*, and the functor mapping for contract lenses. Intuitively, the above *clift* should be consistently defined on all pairs of A and B (in Haskell terms, we expect *clift* to be parametric polymorphic in A and B).

As shown in Fig. 2, we have four important properties. Transformations inside *clift* can be moved out (here $P : \mathcal{P}(A, B)$ is a contract, $f : A \to A'$ and $g : B \to B'$ are two transformations):

$$\mathit{clift}(P \circ (f \times g)) = \mathit{clift}(P) \circ (F\,f \times F\,g) \qquad (\textsc{CLiftCoherence})$$

and reflexivity, implication and conjunctions of predicates are preserved by *clift*[3]:

$$\frac{\forall x : A,\ P(x,x)}{\forall fx : F\ A,\ \mathit{clift}(P)(fx,fx)} \qquad (\textsc{CLiftRefl})$$

$$\frac{P \subseteq Q}{\mathit{clift}(P) \subseteq \mathit{clift}(Q)} \qquad (\textsc{CLiftImply})$$

$$\frac{\mathit{clift}(P)(fx,fy) \wedge \mathit{clift}(Q)(fx,fy)}{\mathit{clift}(P \cap Q)(fx,fy)} \qquad (\textsc{CLiftConj})$$

[3] If we consider any contract $P : \mathcal{P}(A, B)$ as a set of pairs $\{(x, y) : A \times B \mid P(x, y)\}$, implication $(\forall x, y, P(x, y) \Rightarrow Q(x, y))$ becomes the subset relation $(P \subseteq Q)$, and conjunction $((x, y) \mapsto P(x, y) \wedge Q(x, y))$ becomes the set intersection $(P \cap Q)$.

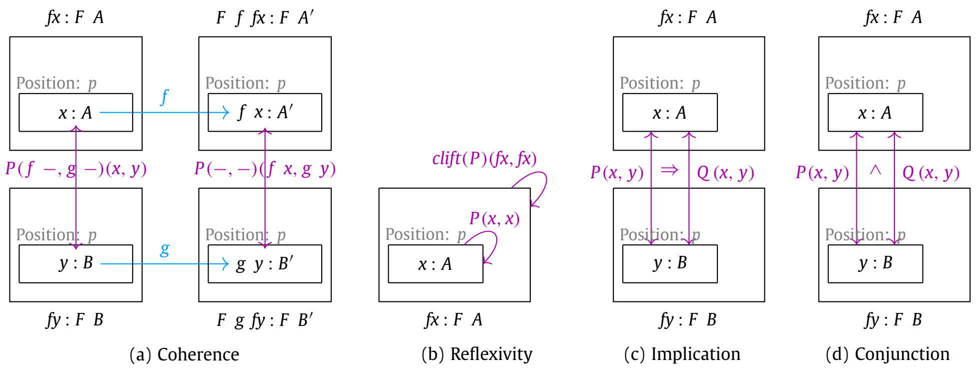


Fig. 2. Properties of the *clift* combinator.

3.2. Interaction between clift and fzip

The combinator *fzip* is used to construct the functor mapping for lenses. To ensure that the constructed lens satisfies the roundtrip properties, *fzip* has the following two laws to satisfy [4]:

$$F\ \pi_1 \circ \mathit{fzip\ create} = \pi_1 \tag{FZIPPROJ1}$$

$$\mathit{fzip\ create} \circ (F\,f \mathbin{\triangle} F\,g) = F\ (f \mathbin{\triangle} g) \tag{FZIPFORK}$$

Here we have $f : A \to B$ and $g : A \to C$; intuitively, $\mathit{fzip\ create} : F\,A \times F\,B \to F\ (A \times B)$ pairs up two F-structures, FZIPPROJ1 states that *fzip* preserves the shape of its first argument, and FZIPFORK states that two F-structures of the same shape will be paired up trivially. Additionally, if every lens $f : A \leftrightarrow B$ has an additional component $\mathit{create}\,f : B \to A$ (which is not the case for us, but a separate *create* also works), the functor mapping can be defined as:

$$\begin{aligned}
&F\ - : A \leftrightarrow B \to F\,A \leftrightarrow F\,B\\
&\mathit{get}\ (F\,f) = F\ (\mathit{get}\,f)\\
&\mathit{put}\ (F\,f) = F\ (\mathit{put}\,f) \circ \mathit{fzip}\ (\mathit{create}\,f)\\
&\mathit{create}\ (F\,f) = F\ (\mathit{create}\,f)
\end{aligned}$$

For contract lenses, the *put* component requires considering additional contracts, and it is the responsibility of *fzip* to take care of these contracts. Also, our lenses have no *create* component, and *fzip* should require one if necessary. Functor mappings shall lift the contracts of its parameter with *clift*:

$$\frac{f : \{P\}\,A \leftrightarrow B\,\{Q\}}{F\,f : \{\mathit{clift}(P)\}\,F\,A \leftrightarrow F\,B\,\{\mathit{clift}(Q)\}}$$

Therefore, *fzip* should have type:

$$\begin{aligned}
\mathit{fzip}\ \ :\ \ &((x : A) \to \{y : B \mid P(x, y)\})\\
&\to \{(fx, fy) : F\,A \times F\,B \mid \mathit{clift}(P)(fx, fy)\}\\
&\to F\,\{(x, y) : A \times B \mid P(x, y)\}
\end{aligned}$$

The $\mathit{create} : (x : A) \to \{y : B \mid P(x, y)\}$ promises to fill in any hole y of type B with a hint x of type A, and also that x and y satisfy the contract $P(x, y)$. And *fzip* does more than pairing the As and Bs: it also promises that all pairs inherit contract P from $\mathit{clift}(P)$.

Now, what is the inverse of *fzip*? Instead of pairing two F-structures, splitting an already-paired-up F-structure into two separate ones requires no extra primitives: we can just use projections. We only need the following property (Fig. 3):

$$\frac{fxy : F\ \{(x, y) : A \times B \mid P(x, y)\}}{\mathit{clift}(P)(F\ \pi_1\ fxy,\ F\ \pi_2\ fxy)} \tag{CONTRACTSPLIT}$$

This property allows us to construct a *clift*ed contract.

We have already seen that *fzip* preserves the shape of its first parameter (FZIPPROJ1), and now we would like to add another useful property following the same intuition:

$$\mathit{fzip}_F\ \mathit{create} \circ (F\ \ f \times \mathit{id}) = F\ (f \times \mathit{id}) \circ \mathit{fzip}_F\ (\mathit{create} \circ f) \tag{FZIPLIFT1}$$

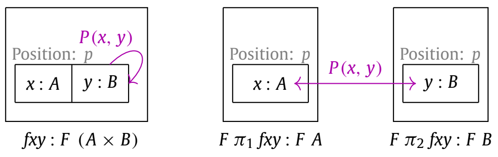


Fig. 3. Inverse of *fzip*: producing a *clift*ed contract.

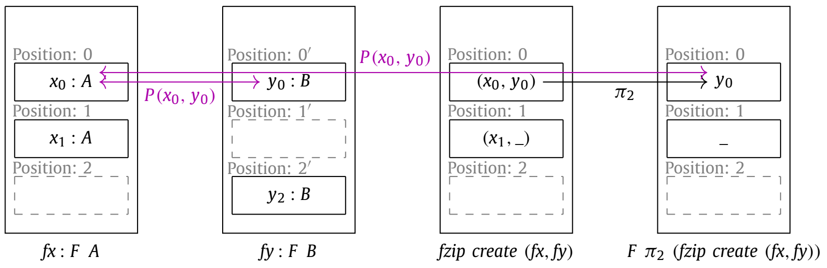


Fig. 4. Substitution with projection of *fzip* for *clift*ed contract.

This new property can also be equivalently expressed as the following FZipProj2:

$$\begin{array}{ll}
& fzip_F\ create \circ (F\,f \times id) = F\,(f \times id) \circ fzip_F\ (create \circ f) \\
\Leftrightarrow & \{\ \text{FZipFork: } fzip_F\ create \circ (F\,\pi_1 \vartriangle F\,\pi_2) = F\,(\pi_1 \vartriangle \pi_2) = F\,id = id\ \} \\
& (F\,\pi_1 \vartriangle F\,\pi_2) \circ fzip_F\ create \circ (F\,f \times id) = (F\,\pi_1 \vartriangle F\,\pi_2) \circ F\,(f \times id) \circ fzip_F\ (create \circ f) \\
\Leftrightarrow & \{\ \text{projections eliminate fork: } h = f \vartriangle g \Leftrightarrow \pi_1 \circ h = f \wedge \pi_2 \circ h = g\ \} \\
& F\,\pi_1 \circ fzip_F\ create \circ (F\,f \times id) = F\,\pi_1 \circ F\,(f \times id) \circ fzip_F\ (create \circ f) \\
& F\,\pi_2 \circ fzip_F\ create \circ (F\,f \times id) = F\,\pi_2 \circ F\,(f \times id) \circ fzip_F\ (create \circ f) \\
\Leftrightarrow & \{\ \text{composition of } F\ \} \\
& F\,\pi_1 \circ fzip_F\ create \circ (F\,f \times id) = F\,f \circ F\,\pi_1 \circ fzip_F\ (create \circ f) \\
& F\,\pi_2 \circ fzip_F\ create \circ (F\,f \times id) = F\,\pi_2 \circ fzip_F\ (create \circ f) \\
\Leftrightarrow & \{\ \text{FZipLift1, projections eliminate fork}\ \} \\
& F\,\pi_2 \circ fzip_F\ create \circ (F\,f \times id) = F\,\pi_2 \circ fzip_F\ (create \circ f) \qquad \text{(FZipProj2)}
\end{array}$$

Unfortunately, the shape of the second parameter of *fzip* is usually not preserved. However, considering the relationship between *clift* and *fzip*, the following property must hold (Fig. 4):

$$\frac{\dfrac{fx : F\ A \quad fy : F\ B \quad clift(P)(fx, fy)}{fy' = F\,\pi_2\ (fzip\ create\ (fx, fy))} \qquad clift(Q)(fx, fy')}{clift(Q)(fx, fy)} \qquad \text{(ContractSubst2)}$$

Although the exact shape of the second parameter might not be preserved after $F\,\pi_2 \circ fzip\ create$, it remains unaltered as far as *clift*ed contracts would care.

3.3. Define clift and fzip for containers

Container, as proposed by Abbott et al. [12], is a formalization of strictly positive functors (i.e., candidates for generating recursive data structures). The combinators we are going to discuss involve bifunctors, therefore we shall give a short review of *Set*-based *n*-ary containers. The following definition is essentially the same as the original definition for containers, but with the base category $\mathcal{C}$ instantiated with $\mathcal{S}et^n$.

Definition 3.1 *(n-ary container).* An *n*-ary container consists of a set of *shapes* S and a family of *positions* $P_{s,k}$ ($s \in S$, $0 \leqslant k < n$). The *extension* of a container $C = (S, P)$ (written as $[\![C]\!]$) is a *n*-ary functor from $\mathcal{S}et^n$ to $\mathcal{S}et$: given two objects X and X' in $\mathcal{S}et^n$ (in other words, they are two families of set: $X = \{X_k \mid 0 \leqslant k < n\}$ and $X' = \{X'_k \mid 0 \leqslant k < n\}$) and a morphism f in $\mathcal{S}et^n$ from X to X' (in other words, f is a family of functions $f = \{f_k : X_k \to X'_k \mid 0 \leqslant k < n\}$), we have

$$\begin{array}{l}
[\![C]\!]\,X = \Sigma\,(s : S)\,(p_k : P_{s,k} \to X_k) \\
[\![C]\!]\,f = \lambda(s, \{p_k\}).\,(s, \{f_k \circ p_k\})
\end{array}$$

We can verify that $[\![C]\!]$ is indeed a functor. In fact, every *n*-ary strictly positive functor (i.e., every *n*-ary polynomial functor) can be written as the extension of some *n*-ary container.

The notion of containers allows us to discuss the detailed internal structure of functors on $\mathcal{S}et^n$ without losing generality. We can formally discuss the *shape* of functors. Also, we can refer to any specific occurrence of type A in values of $F\,A$ thanks to the introduction of *positions*. Despite the fine-grained control over the detailed structure of functors, we do not sacrifice the expressivity at all, since a functor is considered a valid candidate for generating recursive data structures (i.e., it is *strictly positive* or *polynomial*) iff it is expressible as the extension of some container C [12]. That said, the use of containers is strictly limited to providing universal definitions of the primitive combinators (namely *clift* and *fzip*); it is possible to define these primitive combinators without containers, although it would be hard to generalize such definitions, so as to make them applicable to *any* strictly positive functor.

If two F-structures have the same shape, *fzip* should pair up elements that appear on the same positions; but if they have different shapes, we are out of luck. To allow *fzip*ing arbitrary pairs of F-structures, we require a new primitive combinator $mapPos_C$ for container $C = (S, P)$:

$$\frac{s, s' : S \quad p : P_{s,k} \quad 0 \leqslant k < n}{mapPos_C\ s\ s'\ p : Maybe\ P_{s',k}} \qquad (\textsc{MapPos})$$

Let $F = [\![C]\!]$. Given the shape of two F-structures *fx* and *fy*, for each position in *fx*, we use *mapPos* to point to a (possible) corresponding position in *fy*. This may fail, when there is no proper position in *fy*, therefore this *mapPos* is of type *Maybe*. However, when the two F-structures are of the same shape, *mapPos* should simply succeed in returning the source position as the result:

$$\frac{s : S \quad p : P\ s\ k \quad 0 \leqslant k < n}{mapPos_C\ s\ s\ p = Just\ p} \qquad (\textsc{MapPosSameShape})$$

We can finally have a concrete definition of *fzip* and *clift* with the help of this new primitive *mapPos*:

$$clift^K(P)((s_1, u_1), (s_2, u_2)) = \forall p : P_{s_1,k}, \begin{cases} P(u_1\ p, u_2\ p'), & mapPos_C\ s_1\ s_2\ p = Just\ p' \\ K, & mapPos_C\ s_1\ s_2\ p = Nothing \end{cases}$$

$$fzip\ create\ ((s_1, u_1), (s_2, u_2)) = \left(s_1,\ \lambda(p : P_{s_1,k}).\begin{cases} (u_1\ p, u_2\ p'), & mapPos_C\ s_1\ s_2\ p = Just\ p' \\ (u_1\ p, create\ (u_1\ p)), & mapPos_C\ s_1\ s_2\ p = Nothing \end{cases}\right)$$

Note the K and *create* in the definition above: it is an approach we take to make the *create* component optional. If we pick $K = \bot$, the case where *mapPos* returns *Nothing* is eliminated, and thus *clift* requires both of its arguments have (effectively) the same shape, the *create* function is not callable, and *fzip* relies completely on *mapPos* instead of an additional *create*. In contrast, if we pick $K = \top$, *clift* lets different shapes pass by, and *fzip* relies on *create* to fill in the blanks. We can easily verify that the above definition of *fzip* and *clift* satisfy all the properties we have specified.

3.4. Functor mappings

The functor mapping is necessary both for the construction of our recursive lens combinators (to be defined later), and for discussing their properties. It can be defined as follows.

$$\begin{aligned}
&F\ create\ - : \{P\}\ A \leftrightarrow B\ \{Q\} \to \{clift(P)\}\ F\ A \leftrightarrow F\ B\ \{clift(Q)\} \\
&get\ (F\ create\ f) = F\ (get\ f) \\
&put\ (F\ create\ f) = F\ (put\ f) \circ fzip\ create
\end{aligned}$$

Instead of relying on a *create* component in f (as is the case in [4]), this definition requires a standalone *create* function. In this way, we can switch seamlessly between $K = \bot$ and $K = \top$, and therefore the *create* component is in fact optional.

We now state the identity and composition laws of this functor mapping.

Theorem 3.1 *(Identity). $F\ create\ bId = bId$. Here $bId : \{P\}\ A \leftrightarrow A\ \{P\}$ is the lens version of the identity function $id : A \to A$. In this specific case, we have $create : (x : A) \to \{x_0 : A \mid P(x, x_0)\}$. If the predicate P is reflexive, the identity function id will also be a good candidate for this create.*

Theorem 3.2 *(Composition). $F\ create_f\ f \circ F\ create_g\ g = F\ create_{f\circ g}\ (f \circ g)$. Note that here we have $create_{f\circ g} = create_g \circ put_f \circ (id \vartriangle create_f)$ instead of a direct composition of $create_f$ and $create_g$. However, if $create_f$ further satisfies that $get_f \circ create_f = id$, we have*

$$put_f \circ (id \vartriangle create_f) = put_f \circ (get_f \circ create_f \vartriangle create_f) = put_f \circ (get_f \vartriangle id) \circ create_f = create_f$$

and thereby $create_{f\circ g}$ is indeed a simple composition of $create_f$ and $create_g$ (as illustrated in [4]).

Corollary 3.3 *(Exchangeable functor mappings). For every bifunctor F, the two functor mappings F create$_f$ id f bId and F id create$_g$ bId g are exchangeable, i.e.,*

$$F\ create_f\ id\ f\ bId \circ F\ id\ create_g\ bId\ g = F\ id\ create_g\ bId\ g \circ F\ create_f\ id\ f\ bId$$

This is because id and bId are both identities for function and lens composition respectively.

Besides, we prove that this functor mapping is a well-behaved lens, i.e., it satisfies the roundtrip property:

Theorem 3.4 *(Well-behavedness of functor mappings). The contracts associated with this functor mapping are indeed enforced, and the resulting lens is well-behaved.*

Proof. For simplicity of proof, we define an auxillary function as follows:

$$\begin{aligned} &put' : \{(y,x) : B \times A \mid cview(y, get\ f\ x)\} \to \{(x,x') : A \times A \mid csource(x', x)\} \\ &put'\ (y,x) = (x, put\ f\ (y,x)) \end{aligned} \qquad (\textsc{Put'})$$

This function is similar to *put*, but it keeps a copy of the old source in its result. Recall that the view contract mentions the original source, so having a copy around makes our reasoning easier. It is true that $\pi_2\ (put'\ (y,x)) = put\ (y,x)$ and $\pi_1\ (put'\ (y,x)) = x$.

We first prove the contracts are enforced in *put* by the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{fy' = F\ \pi_2\, fm \qquad \dfrac{\dfrac{\dfrac{fx : F\ A \quad fy : F\ B \quad clift(cview(-, get\ -))(fy, fx)}{fzip\ create\ (fy,fx) : F\ \{(y,x) : A \times B \mid cview(y, get\ f\ x)\}}\ \textsc{FZip}}{fm = F\ put'\ (fzip\ create\ (fy,fx))}\ \text{Apply } put}{clift(csource(-, get\ -))(F\ \pi_2\, fm, F\ \pi_1\, fm)}\ \textsc{ContractSplit}}{clift(csource(-, get\ -))(fy', F\ \pi_1\, fm)}\ \text{Factor out } fy'}{clift(csource(-, get\ -))(fy', F\ \pi_1\ (F\ put'\ (fzip\ create\ (fy,fx))))}\ \text{Inline } fm}{clift(csource(-, get\ -))(fy', F\ \pi_2\ (fzip\ create\ (fy,fx)))}\ \text{Functorial in } F}{clift(csource(-, get\ -))(fy', F\ \pi_2\ (fzip\ create\ (F\ (get\ f)\ fy', fx)))}\ \textsc{PutGet}$$

$$\dfrac{\dfrac{clift(csource(-, get\ -))(fy', F\ \pi_2\ (fzip\ create\ (F\ (get\ f)\ fy', fx)))}{clift(csource(-, get\ -))(fy', F\ \pi_2\ (fzip\ create\ (fy', fx)))}\ \textsc{FzipLift1}}{clift(csource(-, get\ -))(fy', fx)}\ \textsc{ContractSubst2}$$

Here we see the usefulness of ContractSubst2: after performing a series of transformations on *fzip create* (fy, fx), we eventually need to establish a relationship between the final result fy' and the original source fx.

Now it is time for the roundtrip property. Firstly, for GetPut:

$$\begin{aligned} & F\ (put\ f)\ (fzip\ create\ (F\ (get\ f)\ x)\ x) \\ = & \ \{\ \text{Functor Identity}\ \} \\ & F\ (put\ f)\ (fzip\ create\ (F\ (get\ f)\ x)\ (F\ id\ x)) \\ = & \ \{\ \textsc{FZipProj1}\ \} \\ & F\ (put\ f)\ (F\ (get\ f \times id)\ x) \\ = & \ \{\ \text{Functor Composition}\ \} \\ & F\ (put\ f \circ (get\ f \times id))\ x \\ = & \ \{\ \textsc{GetPut}\ \} \\ & x \end{aligned}$$

PutGet is similar:

$$\begin{aligned} & F\ (get\ f)\ (F\ (put\ f)\ (fzip\ create\ y\ x)) \\ = & \ \{\ \text{Functor Composition}\ \} \\ & F\ (get\ f \circ put\ f)\ (fzip\ create\ y\ x) \\ = & \ \{\ \textsc{PutGet}\ \} \\ & F\ \pi_1\ (fzip\ create\ y\ x) \\ = & \ \{\ \textsc{FZipProj1}\ \} \\ & y \end{aligned}$$

These proofs are similar to those of lenses without contracts, except that here we take special care to propagate and generate the contracts wherever they are needed. □

3.5. Example: the TreeF functor

Consider the following bifunctor for binary trees:

$$\textbf{data}\ \mathit{TreeF}\ a\ r = \mathit{Empty} \mid \mathit{Branch}\ r\ r \mid \mathit{BranchVal}\ a\ r\ r$$

TreeF is indeed a polynomial functor ($\mathit{TreeF}\ a\ r = 1 + r \times r + a \times r \times r$), therefore it can be written as the extension of the following container $\mathit{TreeC} = (S, P)$ (unique up to isomorphism):

$$\begin{array}{ll}
S = \{\mathit{empty}, \mathit{branch}, \mathit{branch\_val}\} & \\
P(\mathit{empty}, \_) & = \varnothing \\
P(\mathit{branch}, 0) & = \varnothing \\
P(\mathit{branch}, 1) & = \{L_1, R_1\} \\
P(\mathit{branch\_val}, 0) & = \{V\} \\
P(\mathit{branch\_val}, 1) & = \{L_2, R_2\}
\end{array}$$

For the second argument of P, we use index 0 for fields of type a, and index 1 for those of type r.

Note that for nodes shaped *branch* or *branch_val*, there are always the left and right branches of type r, therefore L and R positions (for index 1) are shared between the two shapes. Shape *empty* has no position, which corresponds to the fact that constructor *Empty* has no fields. Shape *branch_val*, compared to shape *branch*, has an additional position V for index 0, that corresponds to the extra field of type a in constructor *BranchVal*. The primitive *mapPos* can be defined following the above intuition:

$$\begin{array}{lllll}
\mathit{mapPos}\ \{k=1\} & \mathit{branch} & \mathit{branch} & p & = \mathit{Just}\ p \\
\mathit{mapPos}\ \{k=1\} & \mathit{branch\_val} & \mathit{branch\_val} & p & = \mathit{Just}\ p \\
\mathit{mapPos}\ \{k=1\} & \mathit{branch} & \mathit{branch\_val} & L_1 & = \mathit{Just}\ L_2 \\
\mathit{mapPos}\ \{k=1\} & \mathit{branch} & \mathit{branch\_val} & R_1 & = \mathit{Just}\ R_2 \\
\mathit{mapPos}\ \{k=1\} & \mathit{branch\_val} & \mathit{branch} & L_2 & = \mathit{Just}\ L_1 \\
\mathit{mapPos}\ \{k=1\} & \mathit{branch\_val} & \mathit{branch} & R_2 & = \mathit{Just}\ R_1 \\
\mathit{mapPos}\ \{k=\_\} & \_ & \_ & \_ & = \mathit{Nothing}
\end{array}$$

If we expand the definition of *clift* and *fzip* just for functor $\mathit{TreeF} = [\![\mathit{TreeC}]\!]$, we get the following specific definitions (application of *create* functions are $\boxed{\text{enclosed with boxes}}$; for $K = \bot$, the lifted contract $\mathit{clift}\ k\ p\ q$ ensures all patterns that require calling these *create* functions are unreachable):

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{clift} : \mathit{Set} \to \mathcal{P}(a \times a') \to \mathcal{P}(r \times r') \to \mathcal{P}(\mathit{TreeF}\ a\ r \times \mathit{TreeF}\ a'\ r')} \\
\mathit{clift}\ \_\ \_\ \_\ \mathit{Empty} & \_ & = \top \\
\mathit{clift}\ k\ \_\ \_\ (\mathit{Branch}\ \_\ \_) & \mathit{Empty} & = k \\
\mathit{clift}\ \_\ \_\ q\ (\mathit{Branch}\ l\ r) & (\mathit{Branch}\ l'\ r') & = q(l, l') \wedge q(r, r') \\
\mathit{clift}\ \_\ \_\ q\ (\mathit{Branch}\ l\ r) & (\mathit{BranchVal}\ \_\ l'\ r') & = q(l, l') \wedge q(r, r') \\
\mathit{clift}\ k\ \_\ \_\ (\mathit{BranchVal}\ x\ l\ r) & \mathit{Empty} & = k \\
\mathit{clift}\ k\ \_\ q\ (\mathit{BranchVal}\ x\ l\ r) & (\mathit{Branch}\ l'\ r') & = k \wedge q(l, l') \wedge q(r, r') \\
\mathit{clift}\ \_\ p\ q\ (\mathit{BranchVal}\ x\ l\ r) & (\mathit{BranchVal}\ x'\ l'\ r') & = p(x, x') \wedge q(l, l') \wedge q(r, r')
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{fzip} : (k \to a \to a') \to (k \to r \to r')} \\
\multicolumn{3}{l}{\quad \to \{\mathit{TreeF}\ a\ r \times \mathit{TreeF}\ a'\ r' \mid \mathit{clift}\ k\ p\ q\}} \\
\multicolumn{3}{l}{\quad \to \mathit{TreeF}\ \{a \times a' \mid p\}\ \{r \times r' \mid q\}} \\
\mathit{fzip}\ \_\ \_\ (\mathit{Empty}, & \_) & = \mathit{Empty} \\
\mathit{fzip}\ \_\ g\ (\mathit{Branch}\ l\ r, & \mathit{Empty}) & = \mathit{Branch}\ (l, \boxed{g\ l})\ (r, \boxed{g\ r}) \\
\mathit{fzip}\ \_\ \_\ (\mathit{Branch}\ l\ r, & \mathit{Branch}\ l'\ r') & = \mathit{Branch}\ (l, l')\ (r, r') \\
\mathit{fzip}\ \_\ \_\ (\mathit{Branch}\ l\ r, & \mathit{BranchVal}\ \_\ l'\ r') & = \mathit{Branch}\ (l, l')\ (r, r') \\
\mathit{fzip}\ f\ g\ (\mathit{BranchVal}\ x\ l\ r, & \mathit{Empty}) & = \mathit{BranchVal}\ (x, \boxed{f\ x})\ (l, \boxed{g\ l})\ (r, \boxed{g\ r}) \\
\mathit{fzip}\ f\ \_\ (\mathit{BranchVal}\ x\ l\ r, & \mathit{Branch}\ l'\ r') & = \mathit{BranchVal}\ (x, \boxed{f\ x})\ (l, l')\ (r, r') \\
\mathit{fzip}\ \_\ \_\ (\mathit{BranchVal}\ x\ l\ r, & \mathit{BranchVal}\ x'\ l'\ r') & = \mathit{BranchVal}\ (x, x')\ (l, l')\ (r, r')
\end{array}$$

Basically, every *field* (with a one-to-one correspondence to every *position*) in the left parameter contributes one atomic proposition (i.e., an application of the predicates) to the contract and one field to the output.

4. Bottom-up recursive lens combinators

Recursive data structures are tree-like in nature. Generally speaking, when it comes to traversing a tree, we have two directions: bottom-up and top-down. Structural recursion is naturally bottom-up: calculation is first performed on substruc-

$$\begin{tikzcd}
F\ A \arrow[r, "f"] \arrow[d, "F\ h"'] & A \arrow[d, "h"] \\
F\ B \arrow[r, "g"'] & B
\end{tikzcd}$$

(a) Homomorphism

$$\begin{tikzcd}
F\ (\mu F) \arrow[r, "in"] \arrow[d, "F\ (fold\ f)"'] & \mu F \arrow[d, "fold\ f"] \\
F\ A \arrow[r, "f"'] & A
\end{tikzcd}$$

(b) Initial algebra and *fold*

Fig. 5. Algebra, homomorphism, initial algebra, and catamorphism.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & F\ A \arrow[d, "F\ h"] \\
B \arrow[r, "g"'] & F\ B
\end{tikzcd}$$

(a) Homomorphism

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "unfold\ f"'] & F\ A \arrow[d, "F\ (unfold\ f)"] \\
\nu F \arrow[r, "out"'] & F\ (\nu F)
\end{tikzcd}$$

(b) Terminal coalgebra and *unfold*

Fig. 6. Coalgebra, homomorphism, terminal algebra, and anamorphism.

tures, and after that the "parent node" is processed. This type of recursion has already been discussed by Pacheco using lenses without contracts in [4] and [5], and is arguably easier to analyze, so we shall start here.

4.1. Fixpoints and unidirectional recursive combinators

In this subsection we provide a brief overview of the categorical theory of fixpoints and their unidirectional recursive combinators [9]. Later discussions on their bidirectional counterpart will use similar notations, with the names of the combinators capitalized and prefixed with a lowercase *b* for *bidirectional* (e.g., *fold* becomes *bFold*, and *in* becomes *bIn*).

A recursive data structure generated by a (strictly positive) functor F is the least fixpoint of functor F, or in other words, the least solution to the equation $x \cong F\,x$ (the $\cong$ sign denotes isomorphism), with $in : F\,\mu F \to \mu F$ and $out : \mu F \to F\,\mu F$ being the invert of each other. The greatest fixpoint νF can be defined similarly as the greatest solution to the same equation $x \cong F\,x$.

Catamorphism is the "most basic" recursive combinator on recursive data structures. It walks up the tree structure from the leaves, *fold*ing it into a final result. To properly define catamorphisms we need some other notions. An F-algebra (A, f) for some functor F is an object A together with a morphism $f : F\,A \to A$. A *homomorphism* h from (A, f) to (B, g) is a morphism $h : A \to B$, with a special property that $h \circ f = g \circ F\,h$ (see Fig. 5a). The special F-algebra $(\mu F, in)$ is an *initial algebra* (see Fig. 5b): there is a unique homomorphism from $(\mu F, in)$ to any F-algebra (A, f); that unique homomorphism is the catamorphism for f, denoted *fold* f.

Anamorphism is dual to catamorphism. It starts with a "seed" and *unfold*s the seed into a tree structure. An F-coalgebra (A, f) is an object A together with a morphism $f : A \to F\,A$; homomorphism is defined alike, only with the arrows in the transmuting square inverted (see Fig. 6a). The special F-coalgebra $(\nu F, out)$ is an *terminal coalgebra* (see Fig. 6b): there is a unique homomorphism from any F-coalgebra (A, f) to $(\nu F, out)$; that unique homomorphism is the anamorphism for f, denoted *unfold* f. In this paper we only consider functions that are guaranteed to terminate, so we can reuse the least fixpoint μF for anamorphisms.

A catamorphism followed by an anamorphism is called a hylomorphism, also known as *refold*. It is a common pattern for recursive programs. Also, *id* can be written as both catamorphism and anamorphism, therefore catamorphism and anamorphism are both special cases of hylomorphism.

There is another interesting combinator $hoist : (F \Rightarrow G) \to \mu F \to \mu G$ for traversing and editing the whole tree. The parameter type $F \Rightarrow G$ is a *natural transformation* from functor F to functor G: in Haskell, $f : F \Rightarrow G$ is a parametrically polymorphic function $f : \forall A. F\,A \to G\,A$ (we will explain the details about natural transformations later in Section 4.5). The special property about *hoist* is that it can be written as an instance of both *fold* and *unfold*:

$$\begin{aligned}
&hoist : (F \Rightarrow G) \to \mu F \to \mu G \\
&hoist\,f = fold_F\ (in_G \circ f) \\
&hoist\,f = unfold_G\ (f \circ out_F)
\end{aligned}$$

For bifunctors, *map* is a useful special case of *hoist*:

$$\begin{aligned}
&map : (A \to B) \to \mu(F\,A) \to \mu(F\,B) \\
&map\,f = hoist\ (F\,f\ id)
\end{aligned}$$

Accumulation is another type of interesting recursive combinators. They can be classified by the two different orders: bottom-up and top-down. We will discuss the details in Section 4.7 and 5.

Instantiating the functor with the list functor ($L\,a\,r = 1 + a \times r$) gives definitions of these combinators on lists. The following definitions are equivalent to those defined in the Haskell Prelude:

$$
\begin{array}{l}
\mathit{fold} : (\mathit{Maybe}\ (a, b) \to b) \to [a] \to b \\
\mathit{fold\ alg}\ [\,] \qquad\ = \mathit{alg\ Nothing} \\
\mathit{fold\ alg}\ (x :: xs)\ = \mathit{alg}\ (\mathit{Just}\ (x, \mathit{fold\ alg\ xs})) \\[4pt]
\mathit{unfold} : (b \to \mathit{Maybe}\ (a, b)) \to b \to [a] \\
\mathit{unfold\ coalg}\ b_0 = \mathbf{case}\ \mathit{coalg}\ b_0\ \mathbf{of} \\
\quad \mathit{Nothing} \quad \to [\,] \\
\quad \mathit{Just}\ (x, b) \to x :: \mathit{unfold\ coalg}\ b \\[4pt]
\mathit{map} : (a \to b) \to [a] \to [b] \\
\mathit{map}\ f\ [\,] \qquad = [\,] \\
\mathit{map}\ f\ (x :: xs) = f\ x :: \mathit{map}\ f\ xs \\[4pt]
\mathit{scan} : (\mathit{Maybe}\ (a, b) \to b) \to [a] \to [b] \\
\mathit{scan\ alg}\ [\,] \qquad = [\mathit{alg\ Nothing}] \\
\mathit{scan\ alg}\ (x :: xs) = \mathit{alg}\ (\mathit{Just}\ (x, y)) :: ys \\
\quad \mathbf{where}\ ys@(y :: \_) = \mathit{scan\ alg\ xs}
\end{array}
$$

Here $\mathit{Maybe}\ (a, b)$ is the representation for $L\,a\,b$: *Nothing* corresponds to $[\,]$, and *Just* corresponds to $(::)$.

4.2. Bidirectional catamorphism, or bFold

Since contracts are now part of the signature of lenses, before we can say anything meaningful to the definition of *bFold*, we have to first decide the contracts. In analogy to the unidirectional *fold*, we expect *bFold* to have a similar *universal property*:

$$\mathit{bFold\ create}\ f = h \Leftrightarrow h \circ \mathit{bIn} = f \circ F\ \mathit{create}\ h$$

Let us first have a look at *bIn*, the bidirectionalized version of the unidirectional data constructor *in*:

$$\mathit{bIn} : \{\mathit{clift}(?)\}\ F\ \mu F \leftrightarrow \mu F\ \{?\}$$

This *bIn* is in fact an isomorphism, it merely packs (and unpacks, as the backward transformation) one layer of "F" into (and respectively, out of) μF. Therefore the contract with which we would like to fill in the blank should be P satisfying the following (logical) equation:

$$P(\mathit{in}\,fx, \mathit{in}\,fy) = \mathit{clift}(P)(fx, fy)$$

We name this fixpoint $\mu\mathit{clift}$ from now on.

Remark 4.1. The definition of *clift* in the previous sections works on every n-ary functor F, so the generic *clift* accepts n predicates in total, each for one type parameter of F. The above equation is really a simplified case, for the full version, $\mu\mathit{clift}$ should be parametrized by $n-1$ predicates $P_2, \cdots, P_n$ as follows:

$$\mu\mathit{clift}(P_2, \cdots, P_n)(\mathit{in}\,fx, \mathit{in}\,fy) = \mathit{clift}(\mu\mathit{clift}(P_2, \cdots, P_n), P_2, \cdots, P_n)(fx, fy)$$

We will make use of this for bifunctors (and thus $\mu\mathit{clift}$ is parametrized by one predicate). ■

Now suppose $f : \{P\}\ F\ A \leftrightarrow A\ \{Q\}$, and $h = \mathit{bFold\ create}\ f : \{P'\}\ \mu F \leftrightarrow A\ \{Q'\}$. Taking into account the requirements of contract lens composition (i.e., in $f \circ g$, $\mathit{csource}\,f$ should imply $\mathit{cview}\,g$), we consider the LHS and RHS of the equation in the universal property. The type of the LHS can be derived as such:

$$
\dfrac{P \Rightarrow \mathit{clift}(Q') \qquad f : \{P\}\ F\ A \leftrightarrow A\ \{Q\} \qquad \dfrac{h : \{P'\}\ \mu F \leftrightarrow A\ \{Q'\}}{F\ \mathit{create}\ h : \{\mathit{clift}(P')\}\ F\ \mu F \leftrightarrow F\ A\ \{\mathit{clift}(Q')\}}}{f \circ F\ \mathit{create}\ h : \{\mathit{clift}(P')\}\ F\ \mu F \leftrightarrow A\ \{Q\}}
$$

and similarly we derive the type of the RHS:

$$\begin{tikzcd}
\mu F & & & & F\ \mu F \arrow[llll, "in_F"'] \\
A \times \mu F \arrow[u, "unfold\ putAlg"] \arrow[r, "id \times out_F"'] & A \times F\ \mu F \arrow[r, "id \times F\ get"'] \arrow[rr, bend left, "\pi_2"] \arrow[rr, phantom, bend left=15, "\triangle"] & A \times F\ A \arrow[r, "put\ f"'] & F\ A \times F\ \mu F \arrow[r, "fzip_F\ create"'] & F\ (A \times \mu F) \arrow[u]
\end{tikzcd}$$

Fig. 7. The *put* component of *bFold* is an anamorphism.

$$\frac{P' \Rightarrow \mu clift \qquad h : \{P'\}\ \mu F \leftrightarrow A\ \{Q'\} \qquad bIn : \{clift(\mu clift)\}\ F\ \mu F \leftrightarrow \mu F\ \{\mu clift\}}{h \circ bIn : \{clift(\mu clift)\}\ F\ \mu F \leftrightarrow A\ \{Q'\}}$$

These derivations are based on composition rules of contract lenses.

According to the universal property, the LHS and RHS should be equal, thus they must have the same type. Together with the two implications required by lens composition (the leftmost hypotheses in the proof trees), we have the following constraints concerning the yet-to-be-solved contracts P, Q, P', and Q':

$$\begin{cases} P \subseteq clift(Q') \\ P' \subseteq \mu clift \\ clift(P') = clift(\mu clift) \\ Q' = Q \end{cases}$$

Thus, we get a proper solution as follows:

$$\begin{gathered}
\frac{Q : A \times A \to Bool \qquad f : \{clift(Q)\}\ F\ A \leftrightarrow A\ \{Q\} \\ create : (a : A) \to \{x : \mu F \mid Q\ (a, fold\ (get\ f)\ x)\}}{bFold\ create\ f : \{\mu clift\}\ \mu F \leftrightarrow A\ \{Q\}} \\
get\ (bFold\ create\ f) = fold\ (get\ f) \\
put\ (bFold\ create\ f) = unfold\ putAlg \\
putAlg : A \times \mu F \to F\ (A \times \mu F) \\
putAlg\ (a, in\ fx) = fzip\ create\ (put\ f\ (a, F\ (get\ (bFold\ create\ f))\ fx), fx)
\end{gathered} \qquad (\text{BFold})$$

This definition, with contracts stripped, is essentially the same as the one Pacheco provides in [5]. It is worth noting that the *put* component is an anamorphism (Fig. 7), which serves as an essential part in the proof for the universal property.

This *bFold* satisfies the roundtrip property and the universal property. To prove the roundtrip property, we can elaborate proper predicates to the type of *putAlg*. For PutGet, we use the following type:

$$putAlg : ((a, x) : A \times \mu F) \to \{fr : F\ (A \times \mu F) \mid get\ f\ (F\ \pi_1\ fr) = a\}$$

And for GetPut, we can use the following type:

$$putAlg : ((a, x) : A \times \mu F) \to \{fr : F\ (A \times \mu F) \mid fold\ (get\ f)\ x = a \Rightarrow in\ (F\ (unfold\ putAlg)\ fr) = x\}$$

Remark 4.2. The predicates above are self-referential, i.e., the type of *putAlg* mentions itself. We solve this problem by restructuring *unfold*; we push the recursion part to its parameter f as follows:

$$\begin{aligned}
unfold' \ :\ & (f\ :\ (x : A) \\
& \quad \to (rec : (y : A) \to \{t : \mu F \mid P\ t\ y\}) \\
& \quad \to \{t : \mu F \mid P\ t\ x\}) \\
& \to (x : A) \to \{t : \mu F \mid P\ t\ x\}
\end{aligned}$$

Now it is f's responsibility to call $in \circ F\ rec$ on what would have been its result, and thus *putAlg* is free from self-reference in its type:

$$\begin{aligned}
putAlg' \ :\ & ((a, t) : A \times \mu F) \\
& \to (f\ :\ ((a, t) : A \times \mu F) \\
& \quad \to (rec : ((a, t) : A \times \mu F) \to \{t' : \mu F \mid fold\ (get\ f)\ t = a \Rightarrow t' = t\}) \\
& \quad \to \{t' : \mu F \mid fold\ (get\ f)\ t = a \Rightarrow t' = t\}) \\
& \to \{t' : \mu F \mid fold\ (get\ f)\ t = a \Rightarrow t' = t\}
\end{aligned}$$

Anyways, this is a proof detail rather than a key insight. ■

$$\begin{tikzcd}
\mu F \times A \arrow[r, "out_F \times get\ f"] \arrow[rr, bend right, "\triangle", "\pi_2"'] \arrow[d, "unfold\ unfoldAlg"'] & F\ \mu F \times F\ A \arrow[r, "fzip_F\ create"] & F\ (\mu F \times A) \times A \arrow[d] \\
\mu(F - \times A) \arrow[d, "fold\ (put\ f)"'] & & F\ (\mu(F - \times A)) \times A \arrow[ll, "in_{F - \times A}"'] \arrow[d] \\
A & & F\ A \times A \arrow[ll, "put\ f"]
\end{tikzcd}$$

Fig. 8. The *put* component for *bUnFold* is a hylomorphism.

For the universal property, the original proof by Pacheco can be applied here [5]. The only thing to note is that the *create* is now explicitly supplied as an extra parameter, rather than being constructed as a component of *bFold*. This does not affect the proof, since no special properties of *create* were used in the proof; instead, the proof relies mainly on properties of *put* and *get*.

Here is another important thing to discuss: the terminating condition of the *put* component of *bFold*. Since it is an anamorphism (unfold), its termination requires extra conditions. A simple and general approach is to have a well-founded recursion, where we have the conditions of (1) there is an order on $A \times \mu F$, (2) for every input $x : A \times \mu F$ and every appearance x' of $A \times \mu F$ in $putAlg\, x : F\ (A \times \mu F)$ meet the condition $x' \leqslant x$, and (3) every descending chain in $A \times \mu F$ has an least element, the termination is guaranteed.

There is a natural order on μF, i.e., $x \leqslant y$ iff x is a substructure of y: suppose $y = in\ (s, u)$ (encoded as an n-ary container), there exists some position $p : P_{s,k}$ such that $x = u\, p$. If there is also a suitable order on A, we can use the lexicographical order on $A \times \mu F$ to establish a well-founded recursion. Specifically, the flat order on A is usually enough for this purpose, where all elements of A is equal. In this special case, the resulting order on $A \times \mu F$ is the order on its μF component; in other words, descending on $A \times \mu F$ is descending on μF alone.

Remark 4.3. Recall that there is an extra parameter K for functor mappings, and it controls the exact meaning of *clift*, and whether *create* is required. It gets inherited into *bFold*. If we let $K = \bot$ in *bFold*, we get a specialized version where the shape of source is guaranteed unchanged. ■

4.3. Bidirectional anamorphism, or bUnFold

Similar to *bFold*, *bUnFold* can be defined as another lens combinator, and it also resembles the one for normal lenses in [4]. We can determine the contracts for *bUnFold* in the same way as before, the resulting definition is essentially the same as that by Pacheco, but with special care for contracts.

$$\begin{array}{l}
\dfrac{\begin{array}{c} P : A \times A \to Bool \qquad f : \{P\}\ A \leftrightarrow F\ A\ \{clift(P)\} \\ create : (x : \mu F) \to \{a : A \mid P(x, unfold\ (get\ f)\ a)\}\end{array}}{bUnFold\ create\ f : \{P\}\ A \leftrightarrow \mu F\ \{\mu clift\}} \\
get\ (bUnFold\ f) = unfold\ (get\ f) \\
put\ (bUnFold\ f) = fold\ (put\ f) \circ unfold\ unfoldAlg \\
unfoldAlg : \mu F \times A \to F\ (\mu F \times A) \times A \\
unfoldAlg\ (in_F\ fx, a) = (fzip_F\ create\ (fx, get\ f\ a), a)
\end{array} \tag{BUnFold}$$

The *put* component above is a hylomorphism (Fig. 8). The roundtrip property, universal property, and terminating conditions can be defined just like *bFold*, and are therefore omitted here.

4.4. Bidirectional hylomorphism, or bReFold

Although some recursive functions cannot be written directly as a catamorphism or an anamorphism, they can be written as a hylomorphism: a composition of an anamorphism and a catamorphism. We first use *unfold* to *produce* a tree-like intermediate structure, and then *consume* it with *fold* to get a final result. We may also apply the technique called "deforestation", to completely eliminate the intermediate data structure, fusing the producer and the consumer into a single function.

For contract lenses, we can do the same thing. We can compose *bFold* and *bUnFold* to form a bidirectional hylomorphism. It is usually desirable to use the versions with $K = \bot$, since the intermediate structure is only used in internal processes, and never exposed to the outside world, so the constraint of intermediate structure having the same shape is (very likely)

$$\begin{tikzcd}
B \times A \arrow[r, "id \times get_g"] \arrow[rrrr, bend right, "\pi_2"', "\triangle"] \arrow[d, "unfold\ uAlg'"'] & B \times F\ A \arrow[r, "id \times F\ get"] \arrow[rr, bend right, "\pi_2"', "\triangle"] & B \times F\ B \arrow[r, "put\ f"] & F\ B \times F\ A \arrow[r, "fzip_F\ create"] & F(B \times A) \times A \arrow[d] \\
\mu(F - \times A) \arrow[d, "fold\ (put\ f)"'] & & & & F\ (\mu(F - \times A)) \times A \arrow[llll, "in_{F - \times A}"'] \arrow[d] \\
A & & & & F\ A \times A \arrow[llll, "put\ f"]
\end{tikzcd}$$

Fig. 9. The *put* component of *bReFold* is a hylomorphism.

easier to enforce, which saves us the effort to write a proper *create* function. We can fuse the two combinators into *bReFold*, eliminating the intermediate structure:

$$\frac{\begin{array}{cc} P : A \times A \to Bool & f : \{P\}\ A \leftrightarrow F\ A\ \{clift(P)\} \\ Q : B \times B \to Bool & g : \{clift(Q)\}\ F\ B \leftrightarrow B\ \{Q\} \\ \multicolumn{2}{c}{create_f : (a : A) \to \{x : \mu F \mid Q\,(a, fold\ (get\ f)\ x)\}} \\ \multicolumn{2}{c}{create_g : (x : \mu F) \to \{b : B \mid P(x, unfold\ (get\ g)\ b)\}} \end{array}}{bReFold\ create_f\ f\ create_g\ g : \{P\}\ A \leftrightarrow B\ \{Q\}} \qquad (\text{BReFold})$$

$$get\ (bReFold\ create_f\ create_g\ f) = refold\ (get\ f)\ (get\ g)$$
$$put\ (bReFold\ create_f\ create_g\ f) = refold\ uAlg'\ (put\ f)$$
$$uAlg' : B \times A \to F(B \times A) \times A$$
$$uAlg'\ (b, a) = (fzip_F\ create\ (put\ f\ (b, F\ get\ (get\ g\ a)), get\ g\ a), a)$$

The *put* component is again a hylomorphism (Fig. 9).

The roundtrip property and the terminating conditions can be derived from those of *bFold* and *bUnFold*. Pacheco et al. [4] did not provide a fused version of *bReFold*, and used directly the composition of *bFold* and *bUnFold*. Here the fused version should arguably run faster, giving the exact same result as the raw composition in any case.

4.5. Bidirectional natural transformations, bHoist and bMap

A natural transformation α from functor F to G is a family of morphisms $\alpha_a : \forall a.F\ a \to G\ a$, such that for every $f : a \to b$, we have $\alpha_b \circ F\, f = G\, f \circ \alpha_a$ (this equation is called the *naturality condition*). We write $\alpha : F \Leftrightarrow G$ for α being a natural transformation for lenses. Note that functors for lenses require an additional (optional) *create*, so the naturality condition is in fact written as $\alpha_b \circ F\, create\, f = G\, create\, f \circ \alpha_a$. Since $\alpha_a : \{clift\ (P)\}\ F\ a \leftrightarrow G\ a\ \{clift\ (Q)\}$ should be parametrically polymorphic in a, P and Q is destined to be the trivial contract $P(\_, \_) = True$ and $Q(\_, \_) = True$.

Given a natural transformation $\alpha : F \Leftrightarrow G$, the combinator *bHoist* synchronize between two fixpoints μF and μG:

$$\frac{f : F \Leftrightarrow G}{bHoist\ f : \mu F \leftrightarrow \mu G} \qquad (\text{Hoist})$$

It can be simultaneously written as *bFold* and *bUnFold*:

$$bHoist : F \Leftrightarrow G \to \{\mu clift\}\ \mu F \leftrightarrow \mu G\ \{\mu clift\}$$
$$bHoist\ create\ \alpha = bFold_F\ create\ (bIn_G \circ \alpha_{\mu G})$$
$$bHoist\ create\ \alpha = bUnFold_G\ create\ (\alpha_{\mu F} \circ bOut_F)$$

Thanks to the naturality condition of α, termination of *bHoist* is always guaranteed. We can also show that the two definitions are indeed equivalent:

Proof. The equivalence of the two definitions is justified by reducing it to the naturality condition of α.

$$\begin{array}{ll} & bFold_F\ crt\ (bIn_G \circ \alpha_{\mu G}) = bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F) \\ \Leftrightarrow & \{\ \text{universal property of } bFold_F\ \} \\ & bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F) \circ bIn_F = bIn_G \circ \alpha_{\mu G} \circ F\ crt\ (bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F)) \\ \Leftrightarrow & \{\ bOut_G \text{ is the inverse of } bIn_G\ \} \\ & bOut_G \circ bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F) \circ bIn_F = \alpha_{\mu G} \circ F\ crt\ (bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F)) \\ \Leftrightarrow & \{\ \text{universal property of } bUnFold_G, \text{ and } bOut_F \circ bIn_F = bId\ \} \\ & G\ crt\ (bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F)) \circ \alpha_{\mu F} = \alpha_{\mu G} \circ F\ crt\ (bUnFold_G\ crt\ (\alpha_{\mu F} \circ bOut_F)) \end{array}$$

A special case for *bHoist* is *bMap*. Given a family of lenses $f_k : \{P_k\}\, A_k \leftrightarrow B_k\, \{Q_k\}$ (as the actual transformation) and a family of functions $create_k : (y : B_k) \to \{x : A_k \mid P_k\ (get\, f_k\ x, y)\}$ (as required by functor mappings), we have

$$\begin{aligned}&bMap\ create\ f : \{\mu clift(P_1, \cdots, P_n)\}\ \mu(F\ A_1 \cdots A_n) \leftrightarrow \mu(F\ B_1 \cdots B_n)\ \{\mu clift(Q_1, \cdots, Q_n)\}\\&bMap\ create\ f = bHoist\ (map\ create_1 \cdots create_n)\ (F\ create_1 \cdots create_n\ id\ f_1 \cdots f_n\ bId)\end{aligned}$$

Now fixpoints of $(n+1)$-ary functors are also functors with an arity of n, with *bMap* serving as the functor mapping. (It is thereby possible to have fixpoints of fixpoints, although usually not very useful.)

Remark 4.4. The above-mentioned *map* is the plain old *map* combinator for unidirectional programs; it accepts n parameters because F is an $(n+1)$-ary functor. The notation $F\ create_1 \cdots create_n\ id\ f_1 \cdots f_n\ bId$ might look a bit complex, and we would like to explain as follows: $create_1, \cdots, create_n$, and id are the $(n+1)$ *create* parameters; $f_1, \cdots, f_n$, and *bId* are the $(n+1)$ lens parameters.

4.6. Calculation laws for bottom-up lens combinators

We have already discussed the universal property of *bFold* and *bUnFold*, and provided definition of *bMap* as specialized forms of both *bFold* and *bUnFold*. Now it is possible to derive various calculation laws the same way as in [9].

4.6.1. Bidirectional fold fusion

Fold fusion states that, under some specific conditions, the composition of some transformation h and a catamorphism can be fused to form one monolithic catamorphism:

$$\begin{aligned}&g \circ F\ create_h\ h = h \circ f \Rightarrow h \circ bFold\ create_f\ f = bFold\ create_g\ g\\&\quad \textbf{where}\ create_g = create_f \circ put_h \circ (id \vartriangle create_h)\end{aligned}$$

Proof. Derivation of this fusion law is essentially the same as for unidirectional programs:

$$\begin{aligned}&\quad h \circ bFold\ create_f\ f = bFold\ create_g\ g\\&\Leftrightarrow \ \{\ \text{universal property for } bFold\ create_g\ g\ \}\\&\quad g \circ F\ create_g\ (h \circ bFold\ create_f\ f) = h \circ bFold\ create_f\ f \circ bIn\\&\Leftrightarrow \ \{\ \text{universal property for } bFold\ create_f\ f\ \}\\&\quad g \circ F\ create_g\ (h \circ bFold\ create_f\ f) = h \circ f \circ F\ create_f\ (bFold\ create_f\ f)\\&\Leftrightarrow \ \{\ \text{functor composition (Theorem 3.2)}\ \}\\&\quad g \circ F\ create_h\ h \circ F\ create_f\ (bFold\ create_f\ f) = h \circ f \circ F\ create_f\ (bFold\ create_f\ f)\\&\Leftarrow \ \{\ \text{Leibniz}\ \}\\&\quad g \circ F\ create_h\ h = h \circ f\end{aligned}$$

4.6.2. Bidirectional fold-map fusion

Bidirectional fold-map fusion states that a *bMap* followed by a *bFold* can be (unconditionally) fused into a single *bFold*, thus saving one traversal:

$$\begin{aligned}&bFold\ create_f\ f \circ bMap\ create_g\ g = bFold\ create_h\ h\\&\quad \textbf{where}\ h = f \circ F\ create_g\ id\ g\ bId\\&\qquad\qquad create_h = map\ create_g \circ put\ (bFold\ create_f\ f) \circ (id \vartriangle create_f)\end{aligned}$$

Usually we want to require that $get_f \circ create_f = id$, and we have $create_h = map\ create_g \circ create_f$.

Proof. This fusion law is usually treated as a special case for Fold Fusion:

$$\begin{aligned}&\quad bFold\ create_f\ f \circ bMap\ create_g\ g = bFold\ create_h\ h\\&\Leftrightarrow \ \{\ \text{definition of } bMap \text{ using } bFold\ \}\\&\quad bFold\ create_f\ f \circ bFold\ (map\ create_g)\ (bIn_G \circ F\ create_g\ id\ g\ bId) = bFold\ create_h\ h\\&\Leftarrow \ \{\ \text{Fold Fusion}\ \}\\&\quad h \circ F\ id\ create_f\ bId\ (bFold\ create_f\ f) = bFold\ create_f\ f \circ bIn_G \circ F\ create_g\ id\ g\ bId\\&\Leftrightarrow \ \{\ \text{universal property for } bFold\ create_f\ f\ \}\\&\quad h \circ F\ id\ create_f\ bId\ (bFold\ create_f\ f) = f \circ F\ id\ create_f\ bId\ (bFold\ create_f\ f) \circ F\ create_g\ id\ g\ bId\\&\Leftrightarrow \ \{\ \text{exchange functor mappings (Corollary 3.3)}\ \}\\&\quad h \circ F\ id\ create_f\ bId\ (bFold\ create_f\ f) = f \circ F\ create_g\ id\ g\ bId \circ F\ id\ create_f\ bId\ (bFold\ create_f\ f)\\&\Leftarrow \ \{\ \text{Leibniz}\ \}\\&\quad h = f \circ F\ create_g\ id\ g\ bId\end{aligned}$$

4.6.3. *Unfold fusion and unfold-map fusion*

Similar to fold fusion, under certain circumstances, composition of an anamorphism and some transformation h can also be fused into a monolithic anamorphism:

$$F\ create_h\ h \circ g = f \circ h \Rightarrow bUnFold\ create_f\ f \circ h = bUnFold\ create_g\ g$$
$$\textbf{where}\ create_g = create_h \circ put_f \circ (id \vartriangle create_f)$$

bUnFold and *bMap* can also be unconditionally fused together:

$$bMap\ create_g\ g \circ bUnFold\ create_f\ f = bUnFold\ create_h\ h$$
$$\textbf{where}\ h = F\ create_g\ id\ g\ bId \circ f$$
$$create_h = create_f \circ put\ (bMap\ create_g\ g) \circ (id \vartriangle map\ create_g)$$

Again, if $get_g \circ create_g = id$, we have $create_h = create_f \circ map\ create_g$.

The proofs are dual to respectively fold fusion and fold-map fusion, except for one peculiarity: our composition law for functor mappings (Theorem 3.2). Under duality, the direction of composition is flipped, and therefore the form of *create* changes a little bit. But since the general idea is essentially the same, we choose not to repeat ourselves here.

4.7. *Bidirectional bottom-up scan: handling dependencies*

We have discussed the definitions and properties of *bFold*, *bUnFold*, *bReFold*, and *bMap*, but we intentionally avoided *bScan* (accumulation), the bidirectional counterpart for *scan*. We found no discussions about this combinator in previous literatures, and it turns out that it is for good reasons.

To discuss the accumulation combinator *scan*, we have to use bifunctors (or functors with arity greater than 2). It is best known (and perhaps most widely used) on lists. Lists have linear structures, and linear structures are in fact degenerated forms of trees: the head of a list is the root, and every internal node has exactly one child. In this point of view, we are going to look closely at the right-to-left *scanr*, since we are currently interested in bottom-up accumulations for contract lenses.

$$scanr : (a \to b \to b) \to b \to [a] \to [b]$$

As we can see, it returns another list (i.e., the *same* recursive data structure as its input). However, as is pointed out by Jeremy Gibbons [6], for tree structures, the return type of *scan* on $\mu(F\,A)$ should be $\mu(B \times F\,1\,-)$. They just happen to be seemingly[4] isomorphic for lists.

Just as for unidirectional programs, we first consider the *subtrees* combinator: given a tree structure, it returns on each node the subtree of that node. We call this bidirectionalized version *bSubTrees*, which is defined by

$$bSubTrees : \{\mu clift\}\ \mu(F\ a) \leftrightarrow SubTrees\ F\ a\ \{\mu clift\}$$
$$get\ bSubTrees = fold\ (\lambda fr.\ in\ (in\ (F\ id\ (\pi_1 \circ out)\ fr), F\ \underline{1}\ id\ fr))$$
$$put\ bSubTrees = \pi_1 \circ out \circ \pi_1$$

where

$$SubTrees\ F\ a = \{x : \mu(\mu(F\ a) \times F-) \mid isValidSubTrees\ x\}$$
$$SubTreesDep(P) : \mu(\mu(F\ a) \times F-) \to Bool$$
$$SubTreesDep(P)\ (in\ (t, fx)) = P(t, fx) \wedge SubTreesDep(P)\ fx$$
$$isValidSubTrees : \mu(\mu(F\ a) \times F-) \to Bool$$
$$isValidSubTrees = SubTreesDep(\lambda(t, fx).\ F\ \underline{1}\ \pi_1\ (out\ t) = F\ id\ (\pi_1 \circ out)\ fx)$$

As we have discussed in Section 2, the view type of *bSubTrees* must be refined so that its *put* component can be defined in a way that satisfies the roundtrip property. This view type $SubTrees\ F\ A$ is refined from type $\mu(\mu(F\,A) \times F\,1\,-)$, with the extra constraint that the tree labeled on every node is a subtree of that of its parent node.

In this section, all the combinators we are going to discuss have such refined types involved, and thus they require extra handling of these inherent dependency constraints. Also, we need a specialized version for every combinator we wish to compose with them, or we will have a type mismatch.

Recall the following known *scan lemma* of unidirectional programs:

[4] It is only "seemingly" isomorphic because *scanr* only returns *non-empty* lists: consider for instance $scanr\ (+)\ 0\ [\,] = [0]$.

$$\mathit{scan}\,f\,e = \mathit{map}\ (\mathit{fold}\,f\,e) \circ \mathit{subtrees}$$

We cannot (yet) write a corresponding version of this lemma for contract lenses, since we have a type mismatch when composing *bMap* and *bSubTrees*. Therefore, we need a specialized version of *bMap* here.

4.7.1. *Definition of bMapU*

We have defined *bMap* whose parameter is required to be a natural transformation. It can be written as either a *bFold* or a *bUnFold*, which gives us a hint that the order of traversal does not matter, and that the mapping of each element on the tree is completely irrelevant to one another. However, for *bMapU* on a *SubTrees* structure, the inherent dependency contributes to the relevance of a node and its subtrees. If we perform transformations on each element separately, it is highly unlikely that the result will meet the requirement of that inherent dependency.

We need to provide some *hint* to the transformation f, the parameter of *bMapU*. The *SubTreesDep*(P) relation imposes $P\,(t, fx)$ on each node *in* (t, fx). A copy of t is already both passed as an argument to f and returned as the result from f, as it is the source data. The hint we need is the other half of the contract $P\,(t, fx)$: a collection of subtrees of the current node; we call it the *context* of said node.

What can we say about the view type of *bMapU*? Can we refine it also to a *SubTrees* type? The answer is yes. We require f to provide a promise to justify this refinement of the view type:

$$f : (\mathit{ctxt} : F\,\mu(A \times F-)) \to \{\lambda\_\,\_.\,\mathit{True}\}\,\{x : A \mid P(x, \mathit{ctxt})\} \leftrightarrow B\,\{\lambda y'\,y.\,Q\,(y', \mathit{map}\ (\mathit{get}\,f)\ \mathit{ctxt})\}$$

$$\frac{\mathit{ctxt} : F\,\mu(A \times F-) \qquad x : A \qquad P(x, \mathit{ctxt})}{Q(\mathit{get}\,f\,x, \mathit{map}\ (\mathit{get}\,f)\ \mathit{ctxt})}$$

Here *map* is the regular unidirectional combinator *map* for $\mu(A \times F-)$ (because the semantics of forward transformations are generally unaffected by inherent dependencies).

Then *bMapU* has the following type:

$$\begin{aligned}\mathit{bMapU}\,f\ \ :\ \ & \{x : \mu(A \times F-) \mid \mathit{SubTreesDep}(P)(x)\}\\ \to\ & \{y : \mu(B \times F-) \mid \mathit{SubTreesDep}(Q)(y)\}\end{aligned}$$

Some f might not be able to produce any non-trivial relation Q on its result y and the mapped context *map* (*get f*) *ctxt* due to the simplicity of type B, or such relation is hard to formalize due to the complexity of f. In this case, we may simply pick $Q(\_, \_) = \mathit{True}$.

Here comes the final definition of *bMapU*:

$$\begin{aligned}&\mathit{bMapU}\,f = \mathit{bUnFold}\ \mathit{coAlg}\\ &\quad \textbf{where}\ \mathit{coAlg} : \mu(A \times F-) \leftrightarrow B \times F\,\mu(A \times F-)\\ &\qquad\quad \mathit{get}\ \mathit{coAlg}\ (\mathit{in}\ (a, \mathit{fa})) = (\mathit{get}\ (f\,\mathit{fa})\ a, \mathit{fa})\\ &\qquad\quad \mathit{put}\ \mathit{coAlg}\ ((b, \mathit{fa}'), \mathit{in}\ (a, \_)) = \mathit{in}\ (\mathit{put}\ (f\,\mathit{fa}')\ (b, a), \mathit{fa}')\end{aligned}$$

4.7.2. *Definition of bScan*

Now, we have a specialized version of *bMap* (*bMapU*), but the problem is not solved yet: *bMapU f* delegates the burden of maintaining the inherent dependency to lens f. Recall that f is a *fold* in the unidirectional scan lemma. Thus, we have to define a specialized version of *bFold*, which should be parametrized by a *ctxt*:

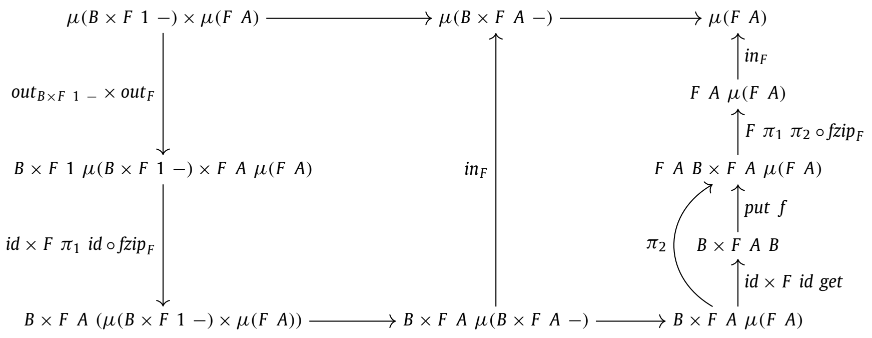
$$\begin{gathered}\frac{\begin{array}{l}P(x, \mathit{ctxt}) = F\ \underline{1}\ \pi_1\ (\mathit{out}_F\ t) \equiv F\ \mathit{id}\ (\pi_2 \circ \mathit{out})\ \mathit{ctxt}\\ \mathit{ctxt} : F\ 1\ \mu(\mu(F\ C) \times F\ 1\ -) \qquad f : \{\mathit{clift}(Q) \wedge \mathit{SubTreeEq}\}\ F\ C\ B \leftrightarrow B\ \{Q\}\end{array}}{\mathit{bFoldDep}\ f\ \mathit{ctxt} : \{\mu\mathit{clift}\}\ \{x : \mu(F\ C) \mid P(x, \mathit{ctxt})\} \leftrightarrow B\ \{Q\}}\\ \begin{aligned}&\mathit{get}\ (\mathit{bFoldDep}\ f\ \mathit{ctxt}) = \mathit{fold}\ (\mathit{get}\ f)\\ &\mathit{put}\ (\mathit{bFoldDep}\ f\ \mathit{ctxt})\ (b, \mathit{fb}) = F\ \pi_1\ \pi_2\ (\mathit{fzip}\ (\mathit{put}\ f\ (b, F\ \mathit{id}\ \mathit{get}\ \mathit{ctxt}'), \mathit{ctxt}'))\\ &\quad \textbf{where}\ \mathit{ctxt}' = F\ \mathit{id}\ (\pi_1 \circ \mathit{out})\ \mathit{ctxt}\end{aligned}\end{gathered} \tag{BFoldDep}$$

Here, the contract *SubTreeEq* describes that two $F\,A\,B$ has their B parts equal respectively:

$$\mathit{SubTreeEq}(\mathit{fx}, \mathit{fy}) = F\ \underline{1}\ \mathit{id}\ \mathit{fx} \equiv F\ \underline{1}\ \mathit{id}\ \mathit{fy} \tag{SubTreeEq}$$

If we look carefully at the *put* component of this *bFoldDep*, we see that instead of performing a *put* recursively (as one might expect), it performs a single *put* only on the root node, and fills the subtrees with that *ctxt*′. This behavior is justified by the existence of contract *SubTreeEq* on f: we know in advance the result of a recursive *put* would be just the same as *ctxt*′; there are unique known values for these subtrees.

Now we can state the scan lemma for contract lenses as follows.



$$\begin{tikzcd}
\mu(B\times F\ 1\ -)\times\mu(F\ A) \arrow[r] \arrow[dd, "out_{B\times F\ 1\ -}\times out_F"'] & \mu(B\times F\ A\ -) \arrow[r] & \mu(F\ A) \\
 & & F\ A\ \mu(F\ A) \arrow[u, "in_F"'] \\
B\times F\ 1\ \mu(B\times F\ 1\ -)\times F\ A\ \mu(F\ A) \arrow[dd, "id\times F\ \pi_1\ id\circ fzip_F"'] & & F\ A\ B\times F\ A\ \mu(F\ A) \arrow[u, "F\ \pi_1\ \pi_2\circ fzip_F"'] \\
 & & B\times F\ A\ B \arrow[u, "put\ f"'] \\
B\times F\ A\ (\mu(B\times F\ 1\ -)\times\mu(F\ A)) \arrow[r] & B\times F\ A\ \mu(B\times F\ A\ -) \arrow[uuuu, "in_F"] \arrow[r] & B\times F\ A\ \mu(F\ A) \arrow[u, "id\times F\ id\ get"'] \arrow[uu, bend left=60, "\pi_2"]
\end{tikzcd}$$

Fig. 10. The *put* component of *bScan* is a hylomorphism.

$$bScan\,f = bMapU\ (bFoldDep\,f)\circ bSubTrees$$

We may go one step further to fuse the three combinators on the RHS to get:

$$\begin{array}{l}
\dfrac{f : \{clift(Q)\wedge SubtreeEq\}\ F\ C\ B\leftrightarrow B\ \{Q\}}{bScan\ f : \{\mu clift\}\ \mu(F\ A)\leftrightarrow\mu(B\times F\ 1\ -)\ \{\mu clift(Q)\}}\\
get\ (bScan\ f) = fold\ scanAlg\\
put\ (bScan\ f) = refold\ pairAlg\ putAlg\\
scanAlg\,fr = in\ (get\ f\ (F\ id\ (\pi_1\circ out)\ fr),\ F\ \underline{1}\ id\,fr)\\
pairAlg\ (in\ (b, fb), in\,fa) = (b,\ F\ \pi_1\ id(fzip_F\ (fb, fa)))\\
putAlg\ (b, fa) = in_F\ (F\ \pi_1\ \pi_2\ (fzip_F\ (put\ f\ (b,\ F\ id\ get\,fa), fa)))
\end{array}\qquad(\text{BScan})$$

The *put* component is (again) a hylomorphism (Fig. 10). It can be verified (by substitution and induction) that this fused definition satisfies the scan lemma.

5. Top-down recursive lens combinators

So far we have covered the bottom-up recursive lens combinators. They are relatively easy to define, and have plenty of use cases. Top-down recursions, on the other hand, are much more complicated, but have their own use cases, too. The best known top-down recursive combinators would be *foldl*, *scanl*, etc. on lists. Nevertheless, as we have mentioned before, list is a very special data structure, and investigating only lists will hide from our view the real difficulty of the general case.

The problem is that list is linear, and the top-down recursions on lists can be seen as the bottom-up recursions on *SnocList*s, which happens to be isomorphic to lists:

$$\begin{array}{l}
\textbf{data}\ SnocList\ a = Nil\mid Snoc\ (SnocList\ a)\ a\\
foldl\qquad\quad : (b\to a\to b)\to b\to[a]\to b\\
foldSnocList : (b\to a\to b)\to b\to SnocList\ a\to b
\end{array}$$

From another point of view, *foldl* is just a *foldr* after a *reverse*, and *scanl* a *scanr* sandwiched by *reverse*:

$$\begin{array}{l}
foldl\,f\ e = foldr\ (flip\,f)\ e\circ reverse\\
scanl\,f\ e = reverse\circ scanr\ (flip\,f)\ e\circ reverse
\end{array}$$

But this not the case for tree-like data structures. Luckily, Gibbons [6,13] has already discussed unidirectional top-down recursions in detail. So here we only need to extend the theory to contract lenses. This time, however, we shall start directly from *scan* instead of *fold*: as we will see, a top-down *fold* is extremely unnatural, and we hardly see any use case that cannot be replaced by a bottom-up *fold*.

5.1. Definition of bPaths

Top-down recursions traverse the tree structure downwards, from the root to its leaves. Similar to what we have done for top-down recursions, we first consider the combinator *bPaths*, a bidirectionalization of *paths*, which replaces every element of a tree with a chain of its ancestors.

To capture ancestors, Gibbons makes use of derivatives of functors:

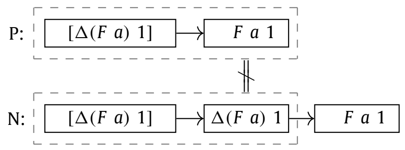


Fig. 11. The path for *P* is not a substructure of the path for *N*.

Definition 5.1 *(Derivative).* The derivative of functor F is another functor ΔF such that $\Delta F\,a$ is like $F\,a$ but with precisely one a missing: $\Delta F\,a$ is a "one-hole context" for $F\,a$ with respect to a. There are two functions to produce and consume holes:

$$\begin{aligned}&posns : F\,a \to F\,(a \times \Delta F\,a)\\&plug : (a \times \Delta F\,a) \to F\,a\end{aligned}$$

such that

$$\begin{aligned}&F\,\pi_1\,(posns\,x) = x\\&F\,plug\,(posns\,x) = F\,\underline{x}\,x\end{aligned}$$

In this way, $\Delta F\,a$ contains exactly the following pieces of information: (1) all the information in $F\,a$ which is irrelevant to a (i.e., the shape of this F-structure, and all label data of type b where $b \neq a$), and (2) the position p_0 of exactly one missing child. Given one $\Delta F\,1$ for each ancestor of a node, we can recover the complete ancestor chain. For some bifunctor F, we may represent a path with $[\Delta(F\,a)\,1] \times F\,a\,1$.

However, there is an efficiency concern with this representation. For some node N and its parent node P, the ancestor chain of P is not a substructure of that of N (Fig. 11). To extend from the ancestor chain of P to that of N, we have to first remove the $F\,a\,1$ for P, and add back a $\Delta(F\,a)\,1$ for P and a $F\,a\,1$ for N.

Therefore, we must switch to the following representation:

$$\begin{aligned}&\textbf{data}\ PathF\ F\ a\ r = StartF\ (F\ a\ 1)\ |\ StepF\ r\ \Delta(F\ a)\ 1\ (F\ a\ 1)\\&\textbf{type}\ Path\ F\ a = \mu(PathF\ F\ a)\end{aligned}$$

And we get a natural (bottom-up) *bFold* for this *Path* type:

$$\frac{f : \{clift(Q)\}\ PathF\ F\ a\ b \leftrightarrow b\ \{Q\}}{bFoldPath\ f : \{clift(\mu clift)\}\ Path\ F\ a \leftrightarrow b\ \{Q\}} \qquad (\textsc{BFoldPath})$$

Note that according to the definition of *Path*, in an ancestor chain of some node, the root node gets wrapped by a *StartF* constructor (as the leaf node of *Path*), and the parent node appears in a outermost *StepF* constructor (as the root node of *Path*). Therefore a bottom-up traversal of *Path* visits all the ancestors in descending order, just as we expected.

Now we are ready to define the *bPaths* combinator:

$$\begin{aligned}&bPaths : \{\mu clift\}\ \mu(F\ a) \leftrightarrow Paths\ F\ a\ \{\mu clift\}\\&get\ bPaths\ (in_F\ fr) = in\ (p,\ F\ \underline{1}\ paths'\ p\ (posns\ fr))\\&\quad \textbf{where}\ p = Start\ (F\ id\ \underline{1}\ fr))\\&put\ bPaths\ (fp, \_) = hoist\ invPathAlg\ fp\\&\quad \textbf{where}\ invPathAlg\ (StartF\ p, fr) = F\ \pi_1\ \pi_2\ (fzip_F\ (p, fr))\\&\qquad\qquad invPathAlg\ (StepF\ \_\ \_\ p, fr) = F\ \pi_1\ \pi_2\ (fzip_F\ (p, fr))\\&paths' : Path\ F\ a \to \mu(F\ a) \times \Delta(F\ a)\ a \to \mu(Path\ F\ a \times F\ 1\ -)\\&paths'\ ts\ (in_F\ fr, t) = in\ (p,\ F\ \underline{1}\ (paths'\ p)\ (posns\ fr))\\&\quad \textbf{where}\ p = StepF\ ts\ (\Delta(F\ \underline{1})\ id\ t)\ (F\ id\ \underline{1}\ fr)\end{aligned}$$

The view type of *bPaths* is also a refined type:

$$\begin{aligned}&Paths\ F\ a = \{x : \mu(Path\ F\ a \times F\ 1\ -)\ |\ isValidPaths\ x\}\\&PathsDep(P) : \mu(Path\ F\ a \times F\ 1\ -) \to Bool\\&PathsDep(P)\ (in\ (t, fx)) = all_F\ (P(t, -))\ (F\ id\ (\pi_1 \circ out)\ fx) \wedge PathsDep(P)\ fx\\&isValidPaths : \mu(Path\ F\ a \times F\ 1\ -) \to Bool\\&isValidPaths = PathsDep(\lambda\ t\ (StepF\ t'\ \_\ \_).\ t = t')\end{aligned}$$

5.2. Definition of bMapD and bScanD

Just like *bMapU*, *bMapD* is a specialized version of *bMap* to handle the inherent dependency described by *PathsDep*. We handle the dependencies again by passing a context to f:

$$\frac{P : A \times A \to Bool \qquad \dfrac{ctxt : Maybe\ A}{f\ ctxt : \{x : A \mid ctxt = Just\ x' \Rightarrow P(x, x')\} \leftrightarrow B}}{bMapD\ f : \{x : \mu(A \times F-) \mid PathsDep(P)(x)\} \leftrightarrow \mu(B \times F-)}$$

$$\begin{aligned}
&get\ (bMapD\ f) = unfold\ getAlg \circ (-, Nothing)\\
&put\ (bMapD\ f) = unfold\ putAlg \circ (-, Nothing)\\
&getAlg\ (in\ (a, fa), ctxt) = (get\ (f\ ctxt)\ a,\ F\ (-, Just\ a)\ fa)\\
&putAlg\ ((in\ (b, fb), in\ (a, fa)), ctxt) = (a',\ F\ (-, Just\ a')\ (fzip_F\ (fb, fa)))\\
&\quad \textbf{where}\ a' = put\ (f\ ctxt)\ (b, a)
\end{aligned} \tag{BMapD}$$

The type of the context values is *Maybe A*, because the dependency relation here is a node depending on its parent. Since the root node does not have any parent, we use *Nothing* to capture this case, and otherwise a *Just a* is passed to f, where a is the data from the parent node.

After the introduction of *bMapD*, we want *bFoldDDep*, a specialized version of *bFold* to be used as the parameter of *bMapD*, similar to *bFoldDep*:

$$P(t, ctxt) = ctxt \equiv Just\ t' \Rightarrow t \equiv in\ (StepF\ t'\ \_\ \_)$$

$$\frac{ctxt : Maybe\ (Path\ F\ A) \qquad f : \{clift(Q) \wedge SubPathEq\}\ PathF\ F\ A\ B \leftrightarrow B\ \{Q\}}{bFoldDDep\ f\ ctxt : \{\mu clift\}\ \{x : Path\ F\ A \mid P(x, ctxt)\} \leftrightarrow B\ \{Q\}}$$

$$\begin{aligned}
&get\ (bFoldDDep\ f\ \_) = fold\ (get\ f)\\
&put\ (bFoldDDep\ f\ Nothing)\ (b, in\ (StartF\ fa)) = in\ (put\ f\ (b, StartF\ fa))\\
&put\ (bFoldDDep\ f\ (Just\ t))\ (b, in\ (StepF\ \_\ d\ fa)) = in\ (put\ f\ (b, StepF\ t\ d\ fa))
\end{aligned} \tag{BFoldDDep}$$

The contract *SubPathEq* describes that two values *fx*, *fy* of type *PathF F A B* has the same shape, and every pair of occurrences of type B in x and y are equal correspondingly. We formalize this idea by discarding (map with $\underline{1}$) all the As and asserting the result being equal:

$$SubPathEq(fx, fy) = PathF\ F\ \underline{1}\ id\ fx \equiv PathF\ F\ \underline{1}\ id\ fy \tag{SubPathEq}$$

And finally, putting these together, we have reached the following top-down definition:

$$\frac{f : \{clift(Q) \wedge SubPathEq\}\ PathF\ F\ A\ B \leftrightarrow b\ \{Q\}}{bScanD\ f : \{\mu clift\}\ \mu(F\ A) \leftrightarrow \mu(B \times F\ 1\ -)\ \{\mu clift\}}$$
$$bScanD\ f = bMapD\ (bFoldDDep\ f) \circ bpaths \tag{BScanD Lemma}$$

One can also fuse this definition, just like what we have done for *bScan*. But the resulting definition of *bScanD* involves complex manipulations on *Path*, which does not provide much insight; also, fusing the definition does not really affect the time complexity.

5.3. Definition of bFoldD

Let us move to define a top down *fold*. Go down from the root of a tree, and we stop at every leaf of that tree; but a *fold* has only one result, so a top-down *fold* cannot be defined naturally with *bPaths*. We have to specify a linear order of traversal for each tree. Here we provide one possible semantics of *bFoldD*, it (in its forward transformation) first flattens the tree into a list, and then performs a left-fold on that list.

First, we define an auxillary combinator *bFlatten*:

$$\begin{aligned}
&bFlatten : \{\mu clift\}\ \mu F \leftrightarrow [F\ 1]\ \{sameShapeDeep\} \\
&get\ bflatten = fold\ alg \\
&put\ bflatten = unfold\ alg' \\
&alg : F\ [F\ 1] \to [F\ 1] \\
&alg\ fl = F\ \underline{1}\ fl : toList\ fl \\
&alg' : [F\ 1] \times \mu F \to F\ ([F\ 1] \times \mu F) \\
&alg'\ (t : ts, in_F\ fx) = fzip_F\ (fromList\ fn\ ts, fx) \\
&\quad \textbf{where}\ fn = F\ \pi_2\ (fzip_F\ (t,\ F\ (length \circ toList \circ out_F)))
\end{aligned} \tag{BFLATTEN}$$

where we require the functor F to support two following functions:

$$\begin{aligned}
toList_F &: F\ a \to [a] \\
fromList_F &: F\ \mathbb{N} \to [a] \to F\ [a]
\end{aligned}$$

such that the two following properties hold:

$$\begin{aligned}
&concat \circ toList_F \circ fromList_F\ fn = id \\
&clift_F (\lambda\ n\ xs. n = length\ xs)(fn, fromList_F\ fn\ xs)
\end{aligned}$$

Intuitively, $fromList_F$ divides the list *fn* into segments, and when we collect and concatenate these segments, we get back the original list. In this process, *fn* is used to determine the shape of the result, and each $n : \mathbb{N}$ gets replaced by a segment of length n.

Now we define a bidirectional version of *foldl* on lists. We take an easy path by first transforming the list into a *SnocList* and then performing a bottom-up fold on it:

$$\begin{aligned}
&\textbf{data}\ SnocListF\ a\ b = NilF \mid SnocF\ b\ a \\
&isoSnocList : \{\mu clift\}\ [a] \leftrightarrow \mu(SnocListF\ a)\ \{\mu clift\} \\
&get\ isoSnocList = fromList_{SnocList} \\
&put\ isoSnocList = toList_{SnocList} \circ \pi_1 \\
&\frac{f\ : \{clift(Q)\}\ SnocListF\ a\ b \leftrightarrow b\ \{Q\}}{bFoldL\ f\ : \{\mu clift\}\ \mu(SnocListF\ a) \to b\ \{Q\}} \\
&bFoldL\ f = bfold_{SnocListF}\ f \circ isoSnocList
\end{aligned} \tag{BFOLDLLIST}$$

With all these well prepared, we obtain the following definition of *bFoldD*:

$$\begin{aligned}
&\frac{f\ : \{clift(Q)\}\ SnocListF\ (F\ 1)\ B \leftrightarrow B\ \{Q\}}{bFoldD\ f\ : \{\mu clift\}\ \mu F \leftrightarrow B\{Q\}} \\
&bFoldD\ f = bFoldL\ f \circ bflatten
\end{aligned} \tag{BFOLDD}$$

As is already mentioned, this definition of *bFoldD* is far from natural. There are still plenty of other possible definitions, and we found no good way to pick out the "best" one. In practice, *bFoldD* may have rare use cases.

6. Related works

In this section, we discuss three most important related works: lens frameworks for writing bidirectional transformations, automatic bidirectionalization of unidirectional transformations, and the generic theory of program calculation.

6.1. Lens frameworks

The original *get*-and-*put*-based lens framework [3] established the fundamental idea that both directions of a bidirectional transformation should be developed simultaneously, and that lenses should be manipulated as a whole with lens combinators.

Pacheco et al. [4,5] proposed many lens combinators including *fold*, *unfold*, *map*, etc. Their lenses consist of three components *get*, *put*, and *create*, and they make use of the *create* component to handle the partiality problem for *put*. The requirement of *create* is unconditional, even if it is never called at runtime, for there is no static information to determine whether a *create* is mandatory. We address this issue by elaborating contract lenses and using the parameter K to enable (disable) the use of *create*. Also, accumulations and top-down recursions are not mentioned.

Edit lenses [14] model changes to the view type as monoids. In one way, possible edits are restricted by the selected monoid structure; but in another way, applying edits is still allowed to fail, and thus the static information of an edit lens (e.g. its type) does not accurately reflect the semantics of its backward transformation. Symmetric lenses [15] associate a complement type C with every lens $l : X \leftrightarrow Y$, making both directions of transformations symmetric in their types: $putr : X \times C \rightarrow Y \times C$, $putl : Y \times C \rightarrow X \times C$. Sufficient information is provided in the complement C, therefore the combinators presented in the paper are all total in both directions. However, the combinators are built upon a fundamentally different basis (symmetric lenses), and accumulations and top-down recursions are not mentioned.

This paper is a direct followup work of contract-lenses [7], which focuses on lists instead of generic tree-like data structures. Contract lenses provide useful combinators and the respective calculation laws. Most contracts, combinators, and calculation laws on lists there can be seen as specialized version of the ones presented here.

6.2. Automatic bidirectionalization

We have mentioned that the calculation of *get* is often accompanied by loss of information, and that is why we require an original source in *put*. Mu et al. developed a framework [16] where $get : S \rightarrow V$ is injective, $put : V \rightarrow S$ requires no original source, and $put \circ get = id$. Along this road, there is a further study on automatic derivation of backward transformation based on the notion of *constant complement* [17]. Instead of requiring injective *get* functions, they calculate a complement $get^c : S \rightarrow C$ for any $get : S \rightarrow V$, such that $get \vartriangle get^c : S \rightarrow V \times C$ is injective. Now a *put* can be derived as $put = (get \vartriangle get^c)^{-1} \circ (id \times get^c)$. Voightländer makes another bidirectionalization framework accepting normal Haskell functions as input [18].

All these frameworks are *passive* on lens contracts: they provide no approach to control what contracts the automatically derived *put* will result in; program may only *verify* a contract (by crashing on error, due to partiality of backward transformations), but they may never *require* a contract (by specifying contracts for its callee). We have already shown in Section 4.7 that it is sometimes necessary to be active in selecting proper contracts for the callee (recall *SubTreeEq* in *bScan*), and it is only possible if we write out the contracts explicitly.

6.3. Program calculation

Program calculation [2] is well established for algebraic reasoning and program optimization. In the original approach, calculation laws are discovered individually and proven by explicit induction and case analysis. Backhouse [10] first observed that *universal properties* are convenient for calculating programs. The categorical approach to recursive data types were discovered and developed by the ADJ group [19,20], Hagino [21,22], and Malcolm [23]. Gibbons has some detailed discussions on tree algebras in [24,6,13]. We follow his method to reproduce respective accumulation combinators for lenses (*bScan* and *bScanD*).

7. Conclusion

In this paper, we have shown how existing lens combinators can be extended to support lenses elaborated with source and view contracts, and how such lenses help provide proper definitions for the lens combinators that were left out. Besides, we developed important calculation laws, which enables various optimizations for bidirectional programs. The approach taken in this paper follows the tradition of functional programming and program calculation: we provide modular building blocks for bidirectional programming, which would be helpful for developing and reasoning about bidirectional transformations.

Some possible future works are summarized as follows. First, inherent dependencies are correctly handled with special design for relevant combinators. However, such dependencies cannot be nested easily: using our method, the lens counterpart for the unidirectional program $map\ (map\ f) \circ map\ paths \circ subtrees$ still requires a special version of *bMap* to properly encode both dependencies from *bPaths* and *bSubTrees*. We are interested in a generic and composable way of handling such data dependencies. Second, currently our composition of contract lenses is all-or-nothing: for $f \circ g$, either $csource\ f \Rightarrow cview\ g$ and the composition is valid, or the composition is invalid. However, it is highly probable that one lens can be associated with several different pairs of source and view contracts. In that case, there might be some combinations which makes the composition valid. We partly solved this problem by making lens combinators polymorphic in its contracts, but further investigation is still desirable. Finally, all of our lens combinators resemble their unidirectional counterpart in their pure form. This fact suggests that it is possible to bidirectionalize a unidirectional program built with these generic recursive combinators by simply bidirectionalizing its building blocks. The feasibility of this approach is yet to be verified.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Commun. ACM 21 (8) (1978) 613–641, https://doi.org/10.1145/359576.359579.
[2] R.S. Bird, Lectures on constructive functional programming, in: M. Broy (Ed.), Constructive Methods in Computing Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 151–217.
[3] J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem, ACM Trans. Program. Lang. Syst. 29 (3) (2007) 17–es, https://doi.org/10.1145/1232420.1232424.
[4] H. Pacheco, A. Cunha, Generic point-free lenses, in: Proceedings of the 10th International Conference on Mathematics of Program Construction, in: MPC '10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 331–352.
[5] H. Pacheco, A. Cunha, Calculating with lenses: optimising bidirectional transformations, in: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 91–100.
[6] J. Gibbons, Upwards and downwards accumulations on trees, in: R.S. Bird, C.C. Morgan, J.C.P. Woodcock (Eds.), Mathematics of Program Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 122–138.
[7] H. Zhang, W. Tang, R. Xie, M. Wang, Z. Hu, Contract lenses: reasoning about bidirectional programs via calculation, J. Funct. Program. (2021), submitted for publication.
[8] P. Wadler, W. Kokke, J.G. Siek, Programming language foundations in Agda, 2020, http://plfa.inf.ed.ac.uk/20.07/.
[9] J. Gibbons, Calculating functional programs, in: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Springer, 2002, pp. 151–203.
[10] R.C. Backhouse, An exploration of the Bird-Meertens formalism, University of Groningen, Department of Mathematics and Computing Science, 1988.
[11] T. van Laarhoven, CPS based functional references, https://www.twanvl.nl/blog/haskell/cps-functional-references, 2009.
[12] M. Abbott, T. Altenkirch, N. Ghani, Containers: constructing strictly positive types, Theor. Comput. Sci. 342 (1) (2005) 3–27, https://doi.org/10.1016/j.tcs.2005.06.002.
[13] J. Gibbons, Accumulating attributes (for Doaitse Swierstra, on his retirement), in: J. Hage, A. Dijkstra (Eds.), Een Lawine van Ontwortelde Bomen: Liber Amicorum voor Doaitse Swierstra, 2013, pp. 87–102, http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/accatt.pdf.
[14] M. Hofmann, B. Pierce, D. Wagner, Edit lenses, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 495–508.
[15] M. Hofmann, B. Pierce, D. Wagner, Symmetric lenses, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 371–384.
[16] S.-C. Mu, Z. Hu, M. Takeichi, An algebraic approach to bi-directional updating, in: W.-N. Chin (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 2–20.
[17] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, M. Takeichi, Bidirectionalization transformation based on automatic derivation of view complement functions, in: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 47–58.
[18] J. Voigtländer, Bidirectionalization for free! (pearl), SIGPLAN Not. 44 (1) (2009) 165–176, https://doi.org/10.1145/1594834.1480904.
[19] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, An introduction to categories, algebraic theories and algebras, IBM Thomas J. Watson Research Division, 1975.
[20] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, J. ACM 24 (1) (1977) 68–95, https://doi.org/10.1145/321992.321997.
[21] T. Hagino, Categorical programming language, 1987.
[22] T. Hagino, A typed lambda calculus with categorical type constructors, in: D.H. Pitt, A. Poigné, D.E. Rydeheard (Eds.), Category Theory and Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 140–157.
[23] G. Malcolm, Data structures and program transformation, Sci. Comput. Program. 14 (2) (1990) 255–279, https://doi.org/10.1016/0167-6423(90)90023-7.
[24] J. Gibbons, Algebras for tree algorithms, Ph.D. thesis, Oxford University, UK, 1991.