Delta Lenses over Inductive Types

Hugo Pacheco, Alcino Cunha, Zhenjiang Hu

hpacheco@di.uminho.pt, alcino@di.uminho.pt, hu@nii.ac.jp

Techn. Report TR-HASLab:02:2012

Feb. 2012

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:02:2012

Delta Lenses over Inductive Types

by Hugo Pacheco, Alcino Cunha, Zhenjiang Hu

Abstract

Existing bidirectional languages are either state-based or operation-based, depending on whether they represent updates as mere states or as sequences of edit operations. In-between both worlds are delta-based frameworks, where updates are represented using alignment relationships between states. In this paper, we formalize delta lenses over inductive types using dependent type theory and develop a point-free delta lens language with an explicit separation of shape and data. In contrast with the already known issue of data alignment, we identify the new problem of shape alignment and solve it by lifting standard recursion patterns such as folds and unfolds to delta lenses that use alignment to infer meaningful shape updates.

Keywords: bidirectional lenses, model alignment, point-free programming

1 Introduction

One of the most successful approaches to bidirectional transformation (BX) are the so-called lenses [FGM+07], an instantiation of the well-known view-update problem. A lens $S \trianglerighteq V$ encompasses a forward transformation $get : S \to V$ that abstracts sources of type S into abstract views of type V, together with a backward transformation $put : V \times S \to S$ that synchronizes a modified view with the original source to produce a new modified source model. Naturally, such synchronization in general is non-deterministic, since there may be many possible modified sources that reflect a certain view-update.

The above state-based formulation of the view-update problem, where the backward transformation receives only the updated view, underpins many BX languages that have been proposed to various application domains. Although very flexible, this formulation implies that the put function must somehow *align* models and recover a high-level description of the update (a *delta* describing the relation between elements of the updated and original view), to be then propagated to the source model. A large part of the non-determinism in the design space of a state-based BX language concerns precisely the choice of a suitable alignment strategy.

Some state-based languages [FGM+07, MHN+07, PC10] do not even explicitly consider this alignment step, and end up aligning values positionally, i.e., elements of the view are always matched with elements of the source at the same position, even when they are rearranged by an update. This suffices for in-place updates that only modify data locally without affecting their order, but produces unsatisfactory results for many other examples. Other state-based languages [XLH+07, BFP+08] go slightly further and align values by keys rather than by positions. Nevertheless, this specific alignment strategy is likewise fixed in the language and might not be suitable for values without natural keys (or for translating updates that modify keys themselves).

On the other hand, operation-based BX languages [MHT04, HHI+10, HPW12] avoid this potential alignment mismatch by relying on an alternative formulation, where the backward transformation receives a description of the update as a low-level sequence of edit operations. The drawback of this approach is that put only considers a fixed update language (typically allowing just add, delete, and move operations), defined over very specific types, making it harder to integrate such languages in a legacy application that does not record such edits.

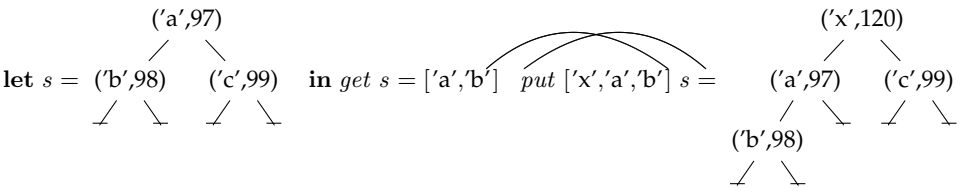
To unify both worlds and benefit from both the loose coupling of state-based approaches and the more refined updatability of operation-based approaches, Diskin *et al* [DXC11] formulated an abstract *delta lens* framework that encompasses an explicit alignment operation (that computes view deltas), and where put is an update-based transformation that propagates view deltas into source deltas. *Matching lenses* [BCF+10] are the first bidirectional language that we are aware of promoting this separation principle. They generalized *dictionary lenses* [BFP+08] over strings, by decomposing values into a rigid structure or shape, a container with "holes" denoting element placeholders, and a list of data elements that populate such shape. This enables elements to be freely rearranged according to delta information. Users can then specify an alignment strategy that computes the view update delta as a correspondence between element positions.

The main limitation of their matching lens combinators is that they are shape preserving: when recast in the context of general user defined data types, their expressivity amounts to a mapping transformation $map\ l : T\ A \trianglerighteq T\ B$ over a polymorphic data type, being $l : A \trianglerighteq B$ a normal state-based lens operating on its elements. In this setting, lenses are sensible to data modifications (on the A and B components of values) but not

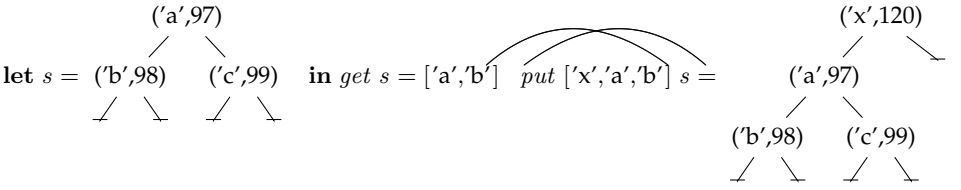
to shape modifications (on the T component of values) and the behavior of the backward transformation is rather simple: it just copies the shape of the view, overlapping the original source shape, and realigns elements using the explicitly computed delta rather than by position. Consider, as an example, the following transformation that computes the left elements of the left spine of a binary tree of pairs:

$$\textbf{data}\ \mathit{Tree}\ A = \mathit{Empty} \mid \mathit{Node}\ A\ (\mathit{Tree}\ A)\ (\mathit{Tree}\ A)$$
$$\mathit{map}\ {\pi_1}^{\mathit{ord}} \circ \mathit{lspine} : \mathit{Tree}\ (\mathit{Char}\ \times\ \mathit{Int}) \trianglerighteq [\mathit{Char}]$$

This transformation is expressed in a point-free language similar to the one introduced in [PC10]. The forward transformation first computes the left spine, followed by mapping the lens ${\pi_1}^{\mathit{ord}}$ that projects the left element of a pair (using ord to create default right values). By porting the matching lens approach to this domain, we could easily define $\mathit{map} : (A \trianglerighteq B) \rightarrow ([A] \trianglerighteq [B])$. Unfortunately, the same does not apply to $\mathit{lspine} : \mathit{Tree}\ a \rightarrow [a]$, since it changes the shape of the source. Leaving lspine as a standard state-based lens would produce less-then-optimal results. For example, if we insert a new element at the head of the view and use a best match alignment strategy for inferring the deltas, put would behave as follows (with deltas represented graphically):



Although the order of the elements in the view list changes, put successfully retrieves their associations with the original pairs due to the improved behavior of map modulo deltas. Unfortunately, the propagated deltas are ignored by lspine since it does not fit the mapping corset imposed by the matching lens framework. With the extra information at hand we could have done better though: using delta information we could recognize 'x' as a new insertion to the head of the view list, and propagate it back as an insertion to the head of the source tree, as depicted below.



It is easy to justify that this behavior induces a smaller change and is thus more predictable. As another example, consider the transformation $\mathit{filter\_l} : [\mathit{Either}\ a\ b] \trianglerighteq [a]$ that filters all the left alternatives of a list. Again, this lens is not a mapping and thus not expressible as a matching lens. If we consider that it behaves positionally, inserting a new element at the head of the view and deleting the rear element would produce the following result:

$$\textbf{let } s = [L\ 1, R\text{'a'}, L\ 2, R\text{'b'}] \quad \textbf{in } get_{filter\_l}\ s = [1,2] \overbrace{\quad put_{filter\_l}\ [0,1]\ s = [L\ 0, R\text{'a'}, L\ 1, R\text{'b'}]}$$

A better solution would be to use the deltas to recognize insertions and deletions in the view, and propagate them to the same relative positions, as follows:

$$\textbf{let } s = [L\ 1, R\text{'a'}, L\ 2, R\text{'b'}] \quad \textbf{in } get_{filter\_l}\ s = [1,2] \overbrace{\quad put_{filter\_l}\ [0,1]\ s = [L\ 0, L\ 1, R\text{'a'}, R\text{'b'}]}$$

The lesson to learn is that likewise a positional *data alignment* (the matching of data elements) is only reasonable for in-place updates, a positional behavior on shapes (that ignores the shapes of the original source and overrides it with the shape of the updated view) is innate for mapping scenarios but again ineffective for shape-changing transformations that restructure source shapes into different target shapes and for which simple overriding for put is not possible. In this paper, we focus on the treatment and propagation of generic deltas (independently of the more particular heuristic techniques that can be used to infer this information for specific application scenarios), identify the new problem of *shape alignment* (the matching of new and old shapes) and propose to answer it with the development of a delta lens language, whose inhabitants are lenses with an explicit notion of shape and data that can perform both data and shape alignment. Our language is designed in such a way that many lens programs written in our previous state-based lens language from [PC10] can be lifted to delta lens programs without significant effort by users.

In the next section (Section 2), we introduce the theoretical concepts required for our development, formalize the notion of deltas and present our particular application domain of inductive types. Section 3 reviews the abstract delta lens framework [DXC11] and proposes a lower-level variant that is more suitable for the implementation of our bidirectional delta-based language. In Section 4, we provide a set of primitive delta lens combinators and redefine the point-free lens combinators from [PC10] as delta lenses over shapes. Section 5 studies the construction of recursive delta lenses and lifts standard recursion patterns such as folds and unfolds to lenses that propagate shape updates as inferred from the deltas between data elements. Section 6 compares related work and Section 7 synthesizes the main contributions and directions for future work.

2 Deltas over Polymorphic Inductive Types

Higher-order functors The central requirement for this paper is the existence of types that have an explicit notion of shape and data. In functional programming, these are known as polymorphic data types, i.e., types parameterized by type variables like the trees and lists in our introduction. A non-polymorphic inductive data type T can be defined as the fixed point $T = \mu F$ of a regular (endo) functor $F : * \to *$ in the semantic domain of choice. The isomorphism $out_F : T \to F\ T$ can then be used to expose its top-level structure, and its converse $in_F : F\ T \to T$ to determine how values of that type can be constructed. Likewise, a polymorphic inductive data type $T\ A$ can be defined as the fixed point $T\ A = \mu(B\ A)$ of a partially applied regular bi-functor

$B : * \to * \to *$ [JJ97], with $out_B : T\ A \to B\ A\ (T\ A)$. An alternative formulation, that we will use in this paper, is to characterize the polymorphic type constructor T as the fixed point $T = \mu\mathcal{F}$ of a higher-order functor (ho-functor) $\mathcal{F} : (* \to *) \to (* \to *)$, such that $T\ A = (\mu\mathcal{F})\ A$ and $out_{\mathcal{F}} : T\ A \to \mathcal{F}\ T\ A$. This formulation is more expressive, namely enabling the definition of non-regular nested data types [JG07], but we restrict the syntax of ho-functors to the following regular family equivalent to the formulation as bi-functors (though necessary to model shapes):

$$\mathcal{F} = \underline{A} \mid \mathit{Par} \mid \mathit{Id} \mid \mathcal{F} \boxplus \mathcal{F} \mid \mathcal{F} \boxtimes \mathcal{F} \mid F \boxdot \mathcal{F}$$

In this language, $\underline{A}$ returns the constant type A, Par denotes the type parameter, Id recursive invocation and $\boxplus$ and $\boxtimes$ higher-order sums and products. Composition $\boxdot$ applies an unary functor to an higher order functor. The application of a regular ho-functor $\mathcal{F}$ to a regular functor F yields a regular functor, that can be defined by addition ($F \oplus G$), multiplication ($F \otimes G$) and composition ($F \odot G$) of regular functors, plus identity (Id) and constants ($\underline{A}$), for a given non-polymorphic type A. In particular, for the primitives we have $\underline{A}\ F = \underline{A}$, $\mathit{Par}\ F = Id$, and $\mathit{Id}\ F = F$, and for the combinators $(\mathcal{F} \boxplus \mathcal{G})\ F = \mathcal{F}\ F \oplus \mathcal{G}\ F$, $(\mathcal{F} \boxtimes \mathcal{G})\ F = \mathcal{F}\ F \otimes \mathcal{G}\ F$, and $(F \boxdot \mathcal{F})\ G = F \odot \mathcal{F}\ G$. For example, the type constructors of lists and trees can be represented as follows:

$$[\,] = \mu\mathcal{L} \qquad \mathcal{L} = \underline{1} \boxplus \mathit{Par} \boxtimes \mathit{Id} \qquad \mathit{Tree} = \mu\mathcal{T} \qquad \mathcal{T} = \underline{1} \boxplus \mathit{Par} \boxtimes (\mathit{Id} \boxtimes \mathit{Id})$$

A transformation f between functors F and G applied to data elements of type A and B is denoted by $f : F\ A \to G\ B$, or to emphasize the shape, $f : F_A \to G_B$. Whenever f is polymorphic, i.e., independent from the type of data, it is called a natural transformation and denoted by $f : F \dot{\to} G$.

Positions Polymorphic inductive data types can be seen as instances of *container types* [AAG05]. A container type $S \triangleright P$ consists of a type of shapes S together with a family P of position types indexed by values of type S. The extension of a container is a functor $[\![S \triangleright P]\!]$ that when applied to a type A (the type of the content) yields the dependent product[1] $\Sigma s : S.\ (P\ s \to A)$. A value of type $[\![S \triangleright P]\!]\ A$ is thus a pair (s, f) where $s : S$ is a shape and $f : P\ s \to A$ is a total function from positions to data elements. A polymorphic data type $T\ A$ is isomorphic to the extension $[\![T\ 1 \triangleright P]\!]\ A$, where the dependent type of positions can be inductively defined over functor representations [AAG05] and polymorphically for the type A of data elements, for each value $v : T\ A$[2]:

$$\begin{array}{lll}
P\ : \forall\{\, T : * \to *\}, v : T\ A. * & & \\
P\ \{Id\}\ \ a & = 1 & \text{-- unit type} \\
P\ \{\underline{C}\}\ \ a & = 0 & \text{-- empty type} \\
P\ \{F \oplus G\}\ (i_1\ a) & = P\ \{F\}\ a & \text{-- left branches} \\
P\ \{F \oplus G\}\ (i_2\ b) & = P\ \{G\}\ b & \text{-- right branches} \\
P\ \{F \otimes G\}\ (a, b) & = P\ \{F\}\ a + P\ \{G\}\ b & \text{-- left or right components} \\
P\ \{\mu\mathcal{F}\}\ \ a & = P\ \{\mathcal{F}\ (\mu\mathcal{F})\}\ (out\ a) & \text{-- recursive unfolding}
\end{array}$$

[1] A dependent type may depend on values. The dependent function space $\forall a : A.\ B\ a$ characterizes functions that, given a value $a : A$ emits values of the dependent type $B\ a$. When B does not depend on a, this degenerates into the normal function space $A \to B$. The dependent cartesian product $\Sigma a : A.\ B\ a$ models pairs where the type of the second component depends on the first. Again, when B does not depend on a, it models the cartesian product $A \times B$. To simplify the presentation, we will often mark some arguments of a dependent function space as implicit using curly braces. In principle, these parameters can be omitted and their value inferred from the context.

[2] Note that the type of positions for a value of type $T\ A$ is the same as the type of positions for its shape of type $T\ 1$.

$$
\begin{array}{ll}
(\circ) & : (B \sim C) \to (A \sim B) \to (A \sim C) \\
(\cup) & : (A \sim B) \to (A \sim B) \to (A \sim B) \\
(\cap) & : (A \sim B) \to (A \sim B) \to (A \sim B) \\
(-) & : (A \sim B) \to (A \sim B) \to (A \sim B) \\
\langle \cdot, \cdot \rangle & : (A \sim B) \to (A \sim C) \to (A \sim B \times C) \\
(\times) & : (A \sim B) \to (C \sim D) \to (A \times C \sim B \times D) \\
[\cdot, \cdot] & : (A \sim C) \to (B \sim C) \to (A + B \sim C) \\
(+) & : (A \sim B) \to (C \sim D) \to (A + C \sim B + D)
\end{array}
\qquad
\begin{array}{ll}
id & : A \sim A \\
\bot & : A \sim A \\
top & : A \sim A \\
\cdot^{\circ} & : (A \sim B) \to (B \sim A) \\
\pi_1 & : A \times B \sim A \\
\pi_2 & : A \times B \sim B \\
i_1 & : A \sim A + B \\
i_2 & : B \sim A + B
\end{array}
$$

Figure 1: Point-free relational combinators

The type of positions for $P\ \{F \odot G\}\ a$ is handled by unrolling the composition $F \odot G$ according to the following equations, where $\mathcal{F} \circledcirc G$ applies a functor G to the parameters of an ho-functor $\mathcal{F}$.

$$
\begin{array}{ll}
\mathit{Id} \odot G & = G \\
\underline{C} \odot G & = \underline{C} \\
(F \oplus G) \odot H & = (F \odot H) \oplus (G \odot H) \\
(F \otimes G) \odot H & = (F \odot H) \otimes (G \odot H) \\
(F \odot G) \odot H & = F \odot (G \odot H) \\
(\mu \mathcal{F}) \odot G & = \mu(\mathcal{F} \circledcirc G)
\end{array}
\qquad
\begin{array}{ll}
\mathit{Id} \circledcirc G & = \mathit{Id} \\
\underline{C} \circledcirc G & = \underline{C} \\
(\mathcal{F} \boxplus \mathcal{G}) \circledcirc G & = (\mathcal{F} \circledcirc G) \boxplus (\mathcal{G} \circledcirc G) \\
(\mathcal{F} \boxtimes \mathcal{G}) \circledcirc G & = (\mathcal{F} \circledcirc G) \boxtimes (\mathcal{G} \circledcirc G) \\
(F \boxdot \mathcal{G}) \circledcirc G & = F \boxdot (\mathcal{G} \circledcirc G) \\
\mathcal{P}\mathit{ar} \circledcirc G & = G \boxdot \mathcal{P}\mathit{ar}
\end{array}
$$

Inspired by *shapely types* [Jay95] notation, the isomorphism $T\ A \cong [\![T\ 1 \triangleright P]\!]\ A$ will be witnessed by three functions: $\mathit{shape} : T\ A \to T\ 1$ that extracts the shape, $\mathit{data} : \forall v : T\ A.\ (P\ v \to A)$ that extracts the data, and $\mathit{recover} : [\![T\ 1 \triangleright P]\!]\ A \to T\ A$ that rebuilds the data type. Being an isomorphism, these functions satisfy the equations $\mathit{recover}\ (\mathit{shape}\ x, \mathit{data}\ x) = x$ and $(\mathit{shape}\ (\mathit{recover}\ x), \mathit{data}\ (\mathit{recover}\ x)) = x$. For lists, the shape $[1]$ is isomorphic to naturals $Nat = \{0, 1, ...\}$, and thus we have $\mathit{shape}\ l = \mathit{length}\ l$, $P\ l = \{0\,..\,\mathit{length}\ l - 1\}$, and $\mathit{data}\ l = \lambda n \to l\ !!\ n$.

Deltas In our work, we model a delta $b\ \Delta\ a$ between a target value b and a source value a as a correspondence relation $P\ b \to P\ a$ (an arrow in the *Rel* category) from positions in the target value to positions in the source value. Matching lenses [BCF+10] capture the same concept but assume that a shape has a list of positions, while we formulate it in a type-safe manner with a dependent type. We will also distinguish *vertical deltas* that model updates between values of the same type, from *horizontal deltas* that establish correspondences between values of different (view and source) types [Dis11]. In our setting, this correspondence relation must be simple, i.e., each target position has non-ambiguous provenance and is related to at most one source position. In practice, this assumption does not restrict any particular kind of correspondences. For example, when constructing views every view element must necessarily be uniquely related to a source element and when performing an update we can still insert, delete and duplicate elements. The only implication is that elements must be considered atomically, i.e., we can not express for example that an element in the view is the combination of two elements in the source.

To describe deltas we will use a standard set of point-free relational combinators (Figure 1), including relational composition ($\circ$) and regular set operations such as union

(∪), intersection (∩) and difference (−). The converse of a relation R is given by R°, $\bot$ denotes the empty relation and top the largest relation, and the other combinators handle products and sums. The domain and range of a relation $r : A \sim B$ will be denoted by $\delta R \subseteq id : A \sim A$ and $\rho R \subseteq id : B \sim B$, respectively. To preserve simplicity, some combinators will only be used in controlled situations. For example, if a relation R is injective and simple (like $get_\triangle$ presented further on), then R° is also simple. By resorting to this language, we can reason about deltas using the powerful algebraic laws ruling its combinators. More details on these laws can be found in [Oli07].

3 Laying Down Delta Lenses

In Diskin's *et al* [DXC11] delta-based framework, updates are encoded as triples (s, u, s') where s, s' are the source and target values and u is a delta between elements of s and s', and lens transformations are arrows that simultaneously translate states and deltas. In our presentation, we choose to separate the state-based and delta-based components of the lenses. This, together with the dependent type notation, leads to a simpler formulation of delta lenses for polymorphic inductive data types: operationally, the delta-based components required for defining composite delta lens can be ignored by end users, which are only required to understand the more intuitive interface of the state-based components. Also, lens transformations are no longer partially defined modulo additional properties entailing preservation of the incidence between values and deltas.

Definition 1 (Delta lens - adapted from [DXC11])**.** *A delta lens l (*d-lens *for short), denoted by $l : S\ A \trianglerighteq_\blacktriangle V\ B$, is a bidirectional transformation that comprises four total functions:*

$$
\begin{aligned}
&get &&: S\ A \to V\ B\\
&get_\blacktriangle &&: \forall \{ s' : S\ A, s : S\ A \}.\ s'\ \Delta\ s \to get\ s'\ \Delta\ get\ s\\
&put &&: \forall (v, s) : V\ B\ \times\ S\ A.\ v\ \Delta\ get\ s \to S\ A\\
&put_\blacktriangle &&: \forall \{ (v, s) : V\ B\ \times\ S\ A \}, d : v\ \Delta\ get\ s.\ put\ (v, s)\ d\ \Delta\ s
\end{aligned}
$$

Note that $S\ 1$ and $V\ 1$ are the types of shapes and A and B are the types of data elements. The d-lens is called well-behaved *if it satisfies the following properties:*

$$
\begin{array}{llll}
get\ (put\ (v, s)\ d) = v & \textsc{PutGet} & get_\blacktriangle\ (put_\blacktriangle\ d) = d & \textsc{PutGet}_\blacktriangle\\
put\ (get\ s, s)\ id = s & \textsc{GetPut} & put_\blacktriangle\ id = id & \textsc{PutId}_\blacktriangle
\end{array}
$$

In the above definition, the state-based component of the d-lens is given by the functions get, that computes a view of a source value, and put, that takes a pair containing a modified view and an original source, together with a delta from the modified view to the original view, and returns a new modified value. The delta-based function $get_\blacktriangle$ translates a source delta into a delta between views produced by get, and $put_\blacktriangle$ receives a view delta and computes a delta from the new source produced by put to the original source. Properties PUTGET and GETPUT are the traditional state-based ones: view-to-view roundtrips preserve view modifications; and put must preserve the original source for identity updates. $\textsc{PutGet}_\blacktriangle$ and $\textsc{PutId}_\blacktriangle$ denote similar laws on deltas: view-to-view roundtrips preserve view updates; and $put_\blacktriangle$ must preserve identity updates. It is easy to see that our formulation is equivalent to the well-behaved d-lenses from [DXC11]. For example, their GETID property is a consequence of our axiomatization.

We can convert a d-lens $l : S\ A \trianglerighteq_{\blacktriangle} V\ B$ into a state-based lens $\lfloor l \rfloor_{diff} : S\ A \trianglerighteqslant V\ B$ that receives an alignment function $diff : \forall v' : V\ B, v : V\ B.\ v'\ \Delta\ v$, estimating a delta from the pre- and post-states of view updates, but forgets shape and alignment for further compositions. We omit its definition and properties, but they have already been studied in [DXC11] and put to practice in [BCF$^+$10].

Abstractly, d-lenses are simple to understand since they transform updates (vertical deltas) into updates. However, to propagate view updates, $put_{\blacktriangle}$ must somehow recover an horizontal delta between the non-modified view and the original source that provides the required traceability information to calculate a new source update [HHI$^+$10]. From an implementation perspective, an alternative formulation of d-lenses that compute and process these horizontal deltas explicitly is preferable (instead of, for instance, having to infer them at run-time for specific executions). As such, we propose an alternative framework of *horizontal d-lenses*, whose delta-based functions explicitly return the horizontal deltas induced by the state-based transformations. Moreover, it is convenient to include in this less abstract framework a $create$ function [BFP$^+$08] that reconstructs a default source value from a view value for situations where the original source is not available.

Definition 2 (Horizontal d-lens)**.** *An horizontal d-lens l (*hd-lens *for short), denoted by $l : S\ A \trianglerighteq_{\vartriangle} V\ B$, comprises three total functions get, put, and $create : V\ B \to S\ A$, plus three horizontal deltas:*

$$
\begin{array}{ll}
get_{\vartriangle} & : \forall \{\, s : S\ A \,\}.\ get\ s\ \Delta\ s \\
put_{\vartriangle} & : \forall \{\, (v, s) : V\ B\ \times\ S\ A \,\}, d : v\ \Delta\ get\ s.\ put\ (v, s)\ d\ \Delta\ (v, s) \\
create_{\vartriangle} & : \forall \{\, v : V\ B \,\}.\ create\ v\ \Delta\ v
\end{array}
$$

It is called well-behaved *if it satisfies* PUTGET, GETPUT *and the following properties:*

$$
\begin{array}{llll}
get\ (create\ v) = v & \textsc{CreateGet} & put_{\vartriangle}\ d \circ get_{\vartriangle} = i_1 & \textsc{PutGet}_{\vartriangle} \\
create_{\vartriangle} \circ get_{\vartriangle} = id & \textsc{CreateGet}_{\vartriangle} & [get_{\vartriangle}, id] \circ put_{\vartriangle}\ id = id & \textsc{GetPut}_{\vartriangle}
\end{array}
$$

The horizontal deltas are complements of the state-based functions that explicitly record the traceability of their execution: $get_{\vartriangle}$ denotes a delta from the resulting view to the original source and vice-versa for $create_{\vartriangle}$, while $put_{\vartriangle}$ is a delta from the new source to the input view-source pair. In practice, this will mean that the deltas of most of our lens combinators can be derived by construction by reversing their behaviors on states. The delta-based laws also dualize the state-based laws, with the insight that the type of positions of a view-source pair is the disjoint sum of the positions in the view and in the source. For example, while the CREATEGET law states that abstracting a created source shall yield the original view, the CREATEGET$_{\vartriangle}$ law evidences that the corresponding delta on views shall also preserve all view elements (identity).

We now show how hd-lenses can be used to implement the abstract framework of d-lenses:

Definition 3. *An hd-lens $l : S\ A \trianglerighteq_{\vartriangle} V\ B$ can be lifted to a d-lens $l_{\blacktriangle} : S\ A \trianglerighteq_{\blacktriangle} V\ B$ by defining $get_{\blacktriangle}\ d = get^{\circ}_{\vartriangle} \circ d \circ get_{\vartriangle}$ and $put_{\blacktriangle}\ d = [get_{\vartriangle} \circ d, id] \circ put_{\vartriangle}\ d$.*

Theorem 1. *If an hd-lens $l : S\ A \trianglerighteq_{\vartriangle} V\ B$ is well-behaved, then the d-lens $l_{\blacktriangle}$ is well-behaved.*

Proof. The state-based laws dismiss proofs. The property PUTGET$_{\blacktriangle}$ is proven using PUTGET$_{\vartriangle}$ and by knowing that $get^{\circ}_{\vartriangle} \circ get_{\vartriangle} = id$, since PUTGET$_{\vartriangle}$ entails that $get_{\vartriangle}$ is a total and injective relation. PUTID$_{\blacktriangle}$ follows directly from GETPUT$_{\vartriangle}$. □

4 Combinators for Horizontal Delta Lenses

Map A state-based lens can be lifted to a mapping hd-lens as follows:

$$\forall l : A \trianglerighteq B.\ S\ l : S_A \trianglerighteq\!\!\!\!\triangle\ S_B$$

$$\begin{array}{lll} get\ s & = S\ get_l\ s & get_\triangle\ \ = id \\ put\ (v,s)\ d & = recover\ (shape\ v, dput\ \cup\ dcreate) & put_\triangle\ d\ = i_1 \\ \quad dput & = put_l \circ \langle data\ v, data\ s \circ d\rangle & \\ \quad dcreate & = (create_l \circ data\ v) \circ (id - \delta dput) & \\ create\ v & = S\ create_l\ v & create_\triangle = id \end{array}$$

Likewise to the state-based functor mapping lens from [PC10], the get and $create$ functions simply map the components of the basic lens over the data elements, producing trivial deltas (all positions are preserved). Instead of aligning elements by their positions, put now performs global data alignment based on the view update deltas: for each view element v, if it relates to a source element s, $put\ (v,s)$ is applied[3]; otherwise a default source is generated with $create_l\ v$. The $put_\triangle$ delta is also trivial, since all elements in the new source come from elements in the view. Mapping defines a functor on the category of hd-lenses, since it preserves identity ($F\ id = id$) and composition ($F\ f \circ F\ g = F\ (f \circ g)$) laws (composition is defined below).

Natural transformation Given a natural lens that only transforms shapes (a lens whose get, put and $create$ functions are natural transformations), denoted by $F \dot{\trianglerighteq} G$, we can lift it to a reshaping hd-lens using the following combinator:

$$\forall l : S \dot{\trianglerighteq} V.\ \bar{l} : S \dot{\trianglerighteq\!\!\!\!\triangle}\ V$$

$$\begin{array}{llll} get\ s & = get_l\ s & get_\triangle\ \{s\} & = \overleftarrow{get_l}\ s \\ put\ (v,s)\ d & = put_l\ (v,s) & put_\triangle\ \{(v,s)\}\ d & = \overleftarrow{put_l}\ (v,s) \\ create\ v & = create_l\ v & create_\triangle\ \{v\} & = \overleftarrow{create_l}\ v \end{array}$$

$$\forall f : F \dot{\to} G.\ \overleftarrow{f} : \forall s : F\ A.\ f\ s\ \Delta\ s$$

$$\overleftarrow{f}\ = data\ (f\ (recover\ (shape\ s, id)))$$

The state-based components of the hd-lens are determined by the value-level functions of the argument lens. The horizontal deltas are calculated using a semantic approach inspired in [Voi09], by running the value-level functions against sources with the data elements replaced by the respective positions, thus inferring the correspondences in the target. This is performed by the auxiliary function $\overleftarrow{\cdot}$. Many useful examples of these natural hd-lens transformations are polymorphic versions of the usual isomorphisms handling the associativity and commutativity of sums and products, such as $swap : (F \otimes G)_A \trianglerighteq\!\!\!\!\triangle\ (G \otimes F)_A$. Another primitive combinator that falls under this category is the identity hd-lens $id : F_A \trianglerighteq\!\!\!\!\triangle\ F_A$. Nevertheless, this combinator is only interesting to lift lenses that already have a reasonable behavior, as is the case of isomorphisms. As long as the behavior of the d-lens is completely determined by the argument lens, the lifted versions of the *lspine* and *filter_l* examples from the introduction, though natural lenses, would not solve the alignment problem.

[3]Here, we are building a function from view positions to source elements as a relation $P\ v \sim A$. The relation $dput$ matches view elements with existing source elements and $dcreate$ creates fresh source elements for the remaining unmatched view elements. The filter $(id - \delta dput)$ guarantees that the relational union is simple.

The lifting of natural transformations to hd-lenses also defines a functor on the category of hd-lenses, since $\overline{id} = id$ and $\overline{(f \circ g)} = \overline{f} \circ \overline{g}$. Moreover, it is also a natural transformation in that category, since $\overline{f} \circ F\ g = G\ g \circ \overline{f}$. This opens the door to an interesting optimization, since in a composition involving only natural transformations and mappings, the latter can be all grouped together and trivially fused.

Composition We now show that the point-free combinators from our previous state-based lens language [PC10, PC11] are also arrows in the category of hd-lenses. Two fundamental point-free combinators are identity and composition. Identity $id : T_A \to T_A$ can be defined using a mapping. Composition can be lifted to hd-lenses as follows:

$$\forall f : V_B \trianglerighteq_{\triangle} U_C, g : S_A \trianglerighteq_{\triangle} V_B.\ (f \circ g) : S_A \trianglerighteq_{\triangle} U_C$$

$$\begin{array}{lll} get\ s & = get_f\ (get_g\ s) & get_{\triangle} = get_{\triangle g} \circ get_{\triangle f} \\ put\ (v,s)\ d_U & = put_g\ (putf, s)\ d_V & put_{\triangle}\ d_U = [putf_{\triangle}, i_2] \circ put_{\triangle g}\ d_V \\ \quad putf & = put_f\ (v, get_g\ s)\ d_U & \quad putf_{\triangle} = (id + get_{\triangle g}) \circ put_{\triangle f}\ d_U \\ \quad d_V & = [get_{\triangle f} \circ d_U, id] \circ put_{\triangle f}\ d_U & \quad d_V = [get_{\triangle f} \circ d_U, id] \circ put_{\triangle f}\ d_U \\ create\ v & = create_g\ (create_f\ v) & create_{\triangle} = create_{\triangle f} \circ create_{\triangle g} \end{array}$$

In the put direction, the intermediate delta d_V passed to put_g maps elements in the result of $put_f\ (u, get_g\ s)\ d_V$ to elements in $get_s\ s$. Since $id \circ f = f = f \circ id$ and $f \circ (g \circ h) = (f \circ g) \circ h$ we have a category of hd-lenses. To demonstrate that our design is robust, composition of d-lenses subsumes composition of hd-lenses: $(f \circ g)_{\blacktriangle} = f_{\blacktriangle} \circ g_{\blacktriangle}$. A more liberal kind of forgetful composition is also possible by first converting the d-lenses into normal lenses, as used in [BCF+10] for combining d-lenses not matching on their intermediate shapes. However, this composition is deemed ill-formed in [DXC11] since the resulting lenses may identify and align updates differently.

Product The product projection hd-lenses can be defined as follows:

$$\forall f : F_A \to G_A.\ {\pi_1}^f : (F \otimes G)_A \trianglerighteq_{\triangle} F_A$$

$$\begin{array}{lll} get\ (x,y) & = x & get_{\triangle} = i_1 \\ put\ (z,(x,y))\ d & = (z,y) & put_{\triangle}\ d = [i_1, i_2 \circ i_2] \\ create\ z & = (z, f\ z) & create_{\triangle} = {i_1}^{\circ} \end{array}$$

$$\forall f : G_A \to F_A.\ {\pi_2}^f : (F \otimes G)_A \trianglerighteq_{\triangle} G_A$$

$$\begin{array}{lll} get\ (x,y) = y & & get_{\triangle} = i_2 \\ put\ (w,(x,y))\ d & = (x,w) & put_{\triangle}\ d = [i_2 \circ i_1, i_1] \\ create\ w & = (f\ w, w) & create_{\triangle} = {i_2}^{\circ} \end{array}$$

Note that the projection lenses are not natural despite get being so: the default parameter f may be defined for a concrete type A, making $create$ non-natural. Yet, they enjoy kind of naturality laws modulo product (${\pi_1}^h \circ (f \times g) = f \circ {\pi_1}^{create_g \circ h \circ get_f}$) and modulo mapping (${\pi_1}^h \circ (F \otimes G)\ f = F\ f \circ {\pi_1}^{F\ create_f \circ h \circ F\ get_f}$), with a precise characterization of how the default generation function must be adapted.

The product bi-functor $\times$ applies two hd-lenses in parallel, and is defined as follows:

$$\forall f : F_A \trianglerighteq_{\triangle} H_B, g : G_A \trianglerighteq_{\triangle} I_B.\ f \times g : (F \otimes G)_A \trianglerighteq_{\triangle} (H \otimes I)_B$$

$$\begin{array}{lll}
get\ (x,y) & = (get_f\ x, get_g\ y) & get_{\triangle} = get_{\triangle f} + get_{\triangle g} \\
put\ ((z,w),(x,y))\ d & = (put_f\ (z,x)\ d_1, put_g\ (w,y)\ d_2) & put_{\triangle}\ d = dists \circ (put_{\triangle f}\ d_1 + put_{\triangle g}\ d_2) \\
 & d_1 = {i_1}^{\circ} \circ d \circ i_1 & \quad d_1 = {i_1}^{\circ} \circ d \circ i_1 \\
 & d_2 = {i_2}^{\circ} \circ d \circ i_2 & \quad d_2 = {i_2}^{\circ} \circ d \circ i_2 \\
create\ (z,w) & = (create_f\ z, create_g\ w) & create_{\triangle} = create_{\triangle f} + create_{\triangle g}
\end{array}$$

When computing put, the product combinator splits the delta over the view pair in two deltas mapping only left or only right elements, to be passed to put_f and put_g, respectively. The $dists$ combinator is an alias for the isomorphism $(A + B) + (C + D) \sim (A + C) + (B + D)$. By halving the deltas, the puts of the argument lenses will loose the delta correspondences for view elements that were swapped to a different side of the view pair. For example, the d-lens $\pi_1 \times \pi_1 : ((F \otimes G) \otimes (F \otimes G))\ A \trianglerighteq_{\triangle} (F \otimes F)\ A$ would only be able to restore left/right information for left/right elements. Given the polymorphic nature of this combinator, which is agnostic to the concrete instantiations of the functors F and G, this is the only reasonable behavior. A more refined behavior, involving non-trivial fitting of the right data elements of y that are related to the left view z into the original left view x, to be restored by put_f, would only be possible for very specific functor instantiations. The product hd-lens is a bi-functor in the category of hd-lenses, preserving identity and composition laws.

Sum Both the either combinator and the sum bi-functor are valid hd-lenses and can be defined as follows:

$$\forall p : H_B \to 2, f : F_A \trianglerighteq H_B, g : G_A \trianglerighteq H_B.\ [f,g]^p : (F \oplus G)_A \trianglerighteq H_B$$

$$\begin{array}{lll}
get\ (i_1\ x) & = get_f\ x & get_{\triangle}\ \{i_1\ x\} = get_{\triangle f}\ \{x\} \\
get\ (i_2\ y) & = get_g\ y & get_{\triangle}\ \{i_2\ y\} = get_{\triangle g}\ \{y\} \\
put\ (z, i_1\ x)\ d & = put_f\ (z,x)\ d & put_{\triangle}\ \{(z, i_1\ x)\}\ d = put_{\triangle f}\ \{(z,x)\}\ d \\
put\ (z, i_2\ y)\ d & = put_g\ (z,y)\ d & put_{\triangle}\ \{(z, i_2\ y)\}\ d = put_{\triangle g}\ \{(z,y)\}\ d \\
create\ z & = p\ z\ ?\ create_f\ z : create_g\ z & create_{\triangle}\ \{z\} = p\ z\ ?\ create_{\triangle f}\ \{z\} : create_{\triangle g}\ \{z\}
\end{array}$$

$$\forall f : F_A \trianglerighteqslant H_B, g : G_A \trianglerighteqslant I_B.\ f + g : (F \oplus G)_A \trianglerighteqslant (H \oplus I)_B$$

$$\begin{array}{lll}
get\ (i_1\ x) & = i_1\ (get_f\ x) & get_{\triangle}\ \{(i_1\ x)\} = get_{\triangle f}\ \{x\} \\
get\ (i_2\ y) & = i_2\ (get_g\ y) & get_{\triangle}\ \{(i_2\ y)\} = get_{\triangle g}\ \{y\} \\
put\ (i_1\ z, i_1\ x)\ d & = i_1\ (put_f\ (z,x)) & put_{\triangle}\ \{(i_1\ z, i_1\ x)\}\ d = put_{\triangle f}\ \{(z,x)\}\ d \\
put\ (i_1\ z, i_2\ y)\ d & = i_1\ (create_f\ z) & put_{\triangle}\ \{(i_1\ z, i_2\ y)\}\ d = i_1 \circ create_{\triangle f}\ \{z\} \\
put\ (i_2\ w, i_1\ x)\ d & = i_2\ (create_g\ w) & put_{\triangle}\ \{(i_2\ w, i_1\ x)\}\ d = i_1 \circ create_{\triangle g}\ \{w\} \\
put\ (i_2\ w, i_2\ y)\ d & = i_2\ (put_g\ (w,y)) & put_{\triangle}\ \{(i_2\ w, i_2\ y)\}\ d = put_{\triangle g}\ \{(w,y)\}\ d \\
create\ (i_1\ z) & = i_1\ (create_f\ z) & create_{\triangle}\ \{(i_1\ z)\} = create_{\triangle f}\ \{z\} \\
create\ (i_2\ w) & = i_2\ (create_g\ w) & create_{\triangle}\ \{(i_2\ w)\} = create_{\triangle g}\ \{w\}
\end{array}$$

For the horizontal deltas, the implicit parameters must be known to disambiguate which sides of the sums were consumed and produced by the forward transformations. Again, the sum hd-lens is a bi-functor in the category of hd-lenses, preserving identity and composition laws.

Bang Similarly to the projection lenses, it is also possible to define a combinator ! combinator that erases the source, and replaces each source with the unit value can be defined as follows:

$$
\begin{array}{l}
\forall f : \underline{1}_A \to F_A.\ !^f : F_A \trianglerighteq_\triangle \underline{1}_A \\
get\ x \qquad\quad = 1 \qquad get_\triangle = \bot \\
put\ (1, x)\ d = x \qquad put_\triangle\ d = i_1 \\
create\ 1 \qquad = f\ 1 \qquad create_\triangle = \bot
\end{array}
$$

At the value, the $!^f$ d-lens applies the argument function f to reconstruct a source value. Like in [PC11], we can phrase a lifted version of the uniqueness law $f = !^{create_f} \Leftrightarrow f : F\ A \trianglerighteq_\triangle \underline{1}$.

5 Recursion Patterns as Horizontal Delta Lenses

Apart from map, the previously presented hd-lens combinators only propagate deltas over rigid shapes (in the sense that they only process shapes polymorphically without further detail) and do not perform any sort of alignment. For mappings, updates may change the cardinality of the data (a container structure such as a list may increase or decrease in length), but alignment can be reduced to the special case of data alignment, with the shape of the update being copied to the result. This problem becomes more general whenever lenses are also allowed to restructure the types, in particular recursive ones, whose values have a more elastic shape: by changing the number of recursive steps, an update can alter the shape of the view (and thus the number of placeholders for data elements), requiring a non-trivial matching with the original source shape. If this problem of shape alignment is not addressed, then the tendency is to reflect these view modifications at the "leaves" of the source shape, causing the precise positions at which the modifications occur in the view shapes to be ignored. The goal of this section is to understand how we can use the delta information to infer meaningful shape changes. Two combinators will be introduced: catamorphisms (folds) to consume recursive sources, and anamorphisms (unfolds) to produce recursive views.

Higher-order functor mapping A ho-functor maps natural transformations to natural transformations via an operation $\forall f : F \dot{\to} G.\ \mathcal{F}\ f : \mathcal{F}\ F \dot{\to} \mathcal{F}\ G$ [JG07]. For regular ho-functors we can define a similar operation $\forall f : F\ A \to G\ A.\ \mathcal{F}\ f : \mathcal{F}\ F\ A \to \mathcal{F}\ G\ A$ for functions that are not natural transformations. Moreover, this operation can be lifted to a hd-lens $\forall f : F_A \trianglerighteq_\triangle G_A.\ \mathcal{F}\ f : \mathcal{F}\ F_A \trianglerighteq_\triangle \mathcal{F}\ G_A$ defined polytypically over the structure of the ho-functor, as follows:

$$
\begin{array}{l}
\forall f : F_A \trianglerighteq_\triangle G_A.\ \mathcal{F}\ f : \mathcal{F}\ F_A \trianglerighteq_\triangle \mathcal{F}\ G_A \\
\mathcal{Id}\ f = f \\
\mathcal{Par}\ f = id \\
\underline{C}\ f = id \\
(\mathcal{F} \boxtimes \mathcal{G})\ f = \mathcal{F}\ f \times \mathcal{G}\ g \\
(\mathcal{F} \boxplus \mathcal{G})\ f = \mathcal{F}\ f + \mathcal{G}\ f
\end{array}
$$

Similarly to the calculus of positions, for the case of composition $(F \boxdot \mathcal{G})\ f$, we can not use regular functor mapping $F\ (\mathcal{G}\ f)$ because the resulting shapes are not the same. Instead, we define inductive functions that unroll value-level compositions using the same set of equations. The same treatment will be adopted for further definitions.

Note that, unlike our primitive hd-lens functor mapping combinator, this time the transformation occurs at the level of shapes and not at the data level (the type A of elements is preserved).

Catamorphism Given an hd-lens algebra $f : \mathcal{F}\ G_A \trianglerighteq_\triangle G_A$, the catamorphism $(\!|f|\!)_\mathcal{F} : \mu\mathcal{F}_A \trianglerighteq_\triangle G_A$ could be defined as the unique hd-lens that satisfies the following equation:

$$\forall f : \mathcal{F}\ G_A \trianglerighteq_\triangle G_A.\ (\!|f|\!)_\mathcal{F} : \mu\mathcal{F}_A \trianglerighteq_\triangle G_A$$
$$(\!|f|\!)_\mathcal{F} = f \circ \mathcal{F}\ (\!|f|\!)_\mathcal{F} \circ out_\mathcal{F}$$

Although this definition receives and propagates deltas, it will not use such information to perform shape alignment. In particular, $put_{\mathcal{F}\ (\!|f|\!)_\mathcal{F}}$ will match the view and source values positionally (like the *fzip* combinator from [PC10]) and pass incomplete delta information to the f argument of the fold due to the lossy behavior of the product d-lens. Our proposal is to adapt it using deltas to recognize shape modifications. As a convention, since fold/unfolds recursively consume/produce source values, we identify insertions and deletions at the "head" of the source while proceeding recursively[4]. If none of the elements at the "head" of the view are related to source elements, then we are confident that they were created with the update and can thus be safely propagated as an insertion to the source. Conversely, if none of the elements at the "head" of the source are related to view elements, we delete such elements before proceeding. Otherwise we proceed positionally.

The "head" of a value of an arbitrary inductive data type can be considered as everything not contained in its recursive occurrences, i.e., something with the same top-level shape but with the recursive occurrences erased. Such head can be computed by the expression $\mathcal{F}\ !\circ out_\mathcal{F} : \mu\mathcal{F}\ A \to \mathcal{F}\ \underline{1}\ A$. The delta between the head elements of the source $s : \mu\mathcal{F}\ A$ and the elements of the original view $get\ s : G\ A$ can be determined by composing the delta of the previous expression with the (converse of the) delta of the catamorphism, as clarified in the following diagram[5]:

$$\mathcal{F}\ \underline{1}\ A \xleftarrow{\mathcal{F}\ !} \mathcal{F}\ \mu\mathcal{F}\ A \xleftarrow{out_\mathcal{F}} \mu\mathcal{F}\ A \xrightarrow{(\!|f|\!)_\mathcal{F}} G\ A$$

$$P\ (get_{\mathcal{F}\ !}\ (out_\mathcal{F}\ s)) \xrightarrow[get_{\triangle\,\mathcal{F}\ !}]{} P\ (out_\mathcal{F}\ s) \xrightarrow[id]{} P\ s \xleftarrow[get_{\triangle\,(\!|f|\!)_\mathcal{F}}]{} P\ (get\ s)$$

If none of the positions in the range of the delta $get^\circ_{\triangle\,(\!|f|\!)_\mathcal{F}} \circ get_{\triangle\,\mathcal{F}\ !}$ are contained in the range of the delta representing the view-update, then we can safely delete the head of the source. The concept of head of the updated view is a bit more tricky: in fact, what we need is the elements of the view that would be necessary to build a head in the source. These can be computed by issuing a create and then erasing the recursive occurrences: $get_{\mathcal{F}\ !} \circ create_f : G\ A \to \mathcal{F}\ \underline{1}\ A$. If none of the positions in the range of this delta are contained in the domain of the delta representing the view-update we proceed with an insertion.

Equipped with these procedures to detect insertions and deletions, we can specify the put of the catamorphism as follows:

[4]Note that moving is not an operation on shapes. For instance, if we swap the elements of a list $[1, 2, 3]$ to [3,2,1], the shapes of both lists are the same $[1, 1, 1]$, where 1 is the unique value inhabiting the unit type 1.

[5]Notice how the delta-component of a transformation can be obtained just by dualization of the respective state-component (reversing the order of compositions and replacing product combinators by the respective sum combinators on positions). As such, from now on, only the state-component of the transformations will be presented.

$$put_{(\!|f|\!)_{\mathcal{F}}}\,(v,s)\,d = \begin{cases} grow\,(v,s) & \text{if } V \neq \bot \wedge (\rho V \cap \delta d) = \bot \\ shrink\,(v,s) & \text{if } S \neq \bot \wedge (\rho S \cap \rho d) = \bot \\ put_{f \circ \mathcal{F}\,(\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}}\,(v,s)\,d & \text{otherwise} \end{cases}$$
$$\text{where} \quad V = create_{\triangle f} \circ get_{\triangle \mathcal{F}\,!} \quad S = get^{\circ}_{\triangle (\!|f|\!)_{\mathcal{F}}} \circ get_{\triangle \mathcal{F}\,!}$$

Insertion The head of the view can be isolated by invoking $create_f$ to produce a value of type $\mathcal{F}\ G\ A$. To propagate this newly created head, we need a way to pair each $G\ A$ inside $\mathcal{F}\ G\ A$ with the original source of type $\mu\mathcal{F}\ A$, to which we can apply $put_{(\!|f|\!)_{\mathcal{F}}}$ recursively. In category theory, a functor is said *strong* if it is equipped with a function $\sigma_F : F\ A\ \times\ B \to F\ (A\ \times\ B)$, denoted *strength*, that pairs the B with each A inside the functor. This function can easily be lifted and defined polytypically for every regular ho-functor. Not taking deltas into account, the $grow$ procedure can be specified as depicted in Figure 2. Notice that, if the functor contains more than one recursive occurrence (for trees for example), then $\sigma_{\mathcal{F}}$ will duplicate the original source for each recursive invocation of put. This is because, when invoking σ at a recursive step, the catamorphism does not know how to split the source so that each piece is related to the respective recursive view. Instead, the duplicated sources will be later aligned recursively. For example, unrelated source elements will be deleted by $shrink$. The actual implementation of $grow$ is $in_{\mathcal{F}} \circ \sigma put_{\mathcal{F}}\ (d \circ create_{\triangle f}) \circ (create_f\ \times\ id)$, where $\sigma put_{\mathcal{F}}$ is a polytypic auxiliary definition that implements the specification $\mathcal{F}\ put_{(\!|f|\!)_{\mathcal{F}}} \circ \sigma_{\mathcal{F}}$, with the necessary propagation of deltas to the recursive invocations of put. Given a lens $S_A \trianglerighteq_{\triangle} V_A$, we define the $\sigma put_{\mathcal{F}}$ combinator that receives a delta and pairs view values inside the functor with duplicated source values, invoking put (with the delta) to process recursive cases:

$$\begin{aligned}
&\sigma put_{\mathcal{F}} : \forall (v,s) : \mathcal{F}\ V_A\ \times\ S_A.\ v\,\Delta\,get\ s \to \mathcal{F}\ S_A \\
&\sigma put_{Id}\ (v,s)\ d = put\ (v,s)\ d \\
&\sigma put_{\underline{C}}\ (v,s)\ d = v \\
&\sigma put_{\mathcal{X}ar}\ (v,s)\ d = v \\
&\sigma put_{\mathcal{F} \boxtimes \mathcal{G}}\ ((x,y),s)\ d = (\sigma put_{\mathcal{F}}\ (x,s)\ (d \circ i_1), \sigma put_{\mathcal{G}}\ (y,s)\ (d \circ i_2)) \\
&\sigma put_{\mathcal{F} \boxplus \mathcal{G}}\ (i_1\ x,s)\ d = \sigma put_{\mathcal{F}}\ (x,s)\ d \\
&\sigma put_{\mathcal{F} \boxplus \mathcal{G}}\ (i_2\ y,s)\ d = \sigma put_{\mathcal{G}}\ (y,s)\ d \\
&\sigma put_{\triangle \mathcal{F}} : \forall \{(v,s) : \mathcal{F}\ V_A\ \times\ S_A\}, d : v\,\Delta\,get\ s.\ (\sigma put_{\mathcal{F}}\ (v,s)\ d)\,\Delta\,(v,s) \\
&\sigma put_{\triangle Id}\ \{(v,s)\}\ d = put_{\triangle}\ \{(v,s)\}\ d \\
&\sigma put_{\triangle \underline{C}}\ \{(v,s)\}\ d = i_1 \\
&\sigma put_{\triangle \mathcal{X}ar}\ \{(v,s)\}\ d = i_1 \\
&\sigma put_{\triangle \mathcal{F} \boxtimes \mathcal{G}}\ \{((x,y),s)\}\ d = \\
&\quad [(i_1 + id) \circ \sigma put_{\triangle \mathcal{F}}\ \{(x,s)\}\ (d \circ i_1), (i_2 + id) \circ \sigma put_{\triangle \mathcal{G}}\ \{(y,s)\}\ (d \circ i_2)] \\
&\sigma put_{\triangle \mathcal{F} \boxplus \mathcal{G}}\ \{(i_1\ x,s)\}\ d = \sigma put_{\triangle \mathcal{F}}\ \{(x,s)\}\ d \\
&\sigma put_{\triangle \mathcal{F} \boxplus \mathcal{G}}\ \{(i_2\ y,s)\}\ d = \sigma put_{\triangle \mathcal{G}}\ \{(y,s)\}\ d
\end{aligned}$$

Here, $\sigma put_{\triangle \mathcal{F}}$ is the horizontal delta induced by $\sigma put_{\mathcal{F}}$.

Deletion Again not taking deltas into account, the $shrink$ procedure can be specified as depicted in Figure 3. This time, we unfold the source value to expose its head to be erased by the auxiliary function $reduce_{\mathcal{F}}$, and then apply $put_{(\!|f|\!)_{\mathcal{F}}}$ to the argument

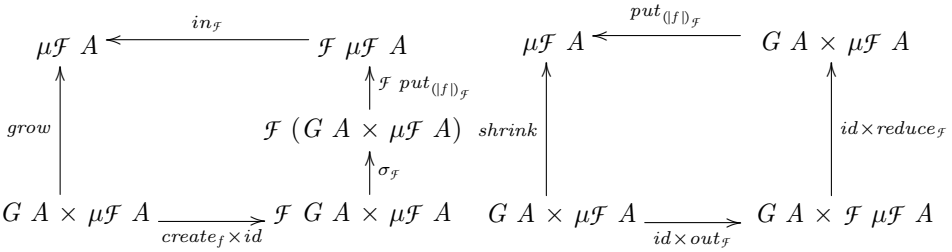


Figure 2: Specification of *grow* for folds.

Figure 3: Specification of *shrink* for folds.

view and the reduced source. In the implementation, $put_{(\!|f|\!)_{\mathcal{F}}}$ is invoked with the delta $get^{\circ}_{\triangle} \circ reduce^{\circ}_{\mathcal{F}\,\triangle} \circ get_{\triangle} \circ d$ that reflects the shape changes performed by $reduce_{\mathcal{F}}$. The function *reduce* that merges the recursive occurrences of a functor into a single value is defined as follows:

$$
\begin{aligned}
&reduce_{\mathcal{F}} : \mathcal{F}\ F_A \to F_A \\
&reduce_{Id}\ x = x \\
&reduce_{\underline{C}}\ x = zero \\
&reduce_{Par}\ x = zero \\
&reduce_{\mathcal{F} \boxtimes \mathcal{G}}\ (x, y) = plus\ (reduce_{\mathcal{F}}\ x, reduce_{\mathcal{G}}\ y) \\
&reduce_{\mathcal{F} \boxplus \mathcal{G}}\ (i_1\ x) = reduce_{\mathcal{F}}\ x \\
&reduce_{\mathcal{F} \boxplus \mathcal{G}}\ (i_2\ x) = reduce_{\mathcal{F}}\ y
\end{aligned}
$$

Here, we assume that the type $F\ A$ is a monoid, supporting two operations $zero : F\ A$ and $plus : F\ A\ \times\ F\ A \to F\ A$. Note that the horizontal delta $reduce_{\triangle\,\mathcal{F}}$ can be computed using the semantic approach since we assume that the monoid instances (and consequently $reduce_{\mathcal{F}}$) are polymorphic.

In order to erase the head, function $reduce_{\mathcal{F}}$ should merge all recursive occurrences into a single value. If the source type is a monoid, i.e., has an empty value $zero : \mu\mathcal{F}\ A$ and a binary concatenation operation $plus : \mu\mathcal{F}\ A\ \times\ \mu\mathcal{F}\ A \to \mu\mathcal{F}\ A$, then we can polytypically define $reduce_{\mathcal{F}}$ just by folding the sequence of recursive occurrences using the monoid operations. For types other than lists, there could be more than one possible monoid implementation. We provide default instances for many types, but the user is free to provide their own implementation. We require that monoid operations are natural transformations so that we can automatically compute deltas using the semantic technique presented before.

Examples We can now encode the examples from the introduction as hd-lenses. For example, the *lspine* lens can be defined as the following hd-lens fold:

$$
\begin{aligned}
&lspine : Tree_A \trianglerighteq [\,]_A \\
&lspine = (\!|in_{\angle} \circ (id + id \times \pi_1{}^{const\ [\,]})|\!)_{\mathcal{T}} \\
&plus_{Tree} : Tree \otimes Tree \dashrightarrow Tree \qquad\qquad zero_{Tree} : \forall A.\ Tree\ A \\
&plus_{Tree}\ t\ Empty = t \qquad\qquad zero_{Tree} = Empty \\
&plus_{Tree}\ t\ (Node\ x\ l\ r) = Node\ x\ (plus_{Tree}\ t\ l)\ r
\end{aligned}
$$

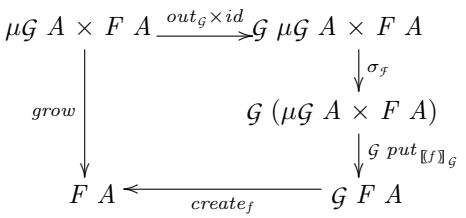


Figure 4: Specification of *grow* for unfolds.

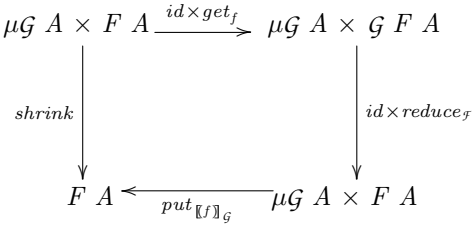


Figure 5: Specification of *shrink* for unfolds.

With the given monoid implementation, when ran against the introductory example this lens produces the desired result. Note that although no deletion is performed by the view update, the *σput* procedure for insertions will duplicate the original source tree, but the right-side duplicated tree will be successfully marked as deleted by our backward semantics and disposed of by the monoid. Also for the *lspine* example, we know that an insertion followed by a deletion (of the 'x' element for instance) would lead to no effect on the source.

In our language, a type $F\ (G\ A)$ is different in shape from an isomorphic type $(F \odot G)\ A$. For example, for the filtering lens from the introduction to produce the desired behavior the source type must be shaped as $([\,] \odot (Id \oplus \underline{B}))_A$ and not $[\,]_{A+B}$. It can then be defined using a catamorphism over the isomorphic type $\mu(\mathcal{L} \boxdot (Id \oplus \underline{B}))$:

$$\begin{aligned}&\mathit{filter\_l} : [A + B] \trianglerighteq_{\triangle} [A]\\ &\mathit{filter\_l} = (\!|(in_{\mathcal{L}} \blacktriangledown \pi_2) \circ \mathit{coassoc} \circ (id + \mathit{distl})|\!)_{\mathcal{L} \boxdot (Id \oplus \underline{B})}\end{aligned}$$

By running this lens against the example from the introduction, right values are now restored properly. Here, $f \blacktriangledown g$ is an alias for $[f, g]^{const\ True}$ and the combinators $\mathit{coassoc} : (F \oplus (G \oplus H))_A \trianglerighteq_{\triangle} ((F \oplus G) \oplus H)_A$ and $\mathit{distl} : ((F \oplus G) \otimes H)_A \trianglerighteq_{\triangle} ((F \otimes H) \oplus (G \otimes H))_A$ are hd-lens isomorphisms. We also do not provide a default function to π_2 because it would be never used by $\blacktriangledown$.

Anamorphism Analogously to catamorphisms, anamorphisms recursively construct view values. Given an hd-lens coalgebra $f : F_A \trianglerighteq_{\triangle} \mathcal{G}\ F_A$, the positional anamorphism $[\![f]\!]_{\mathcal{G}} : F_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A$ can be defined as the unique hd-lens that satisfies the following equation:

$$\begin{aligned}&\forall f : F_A \trianglerighteq_{\triangle} \mathcal{G}\ F_A.\ [\![f]\!]_{\mathcal{G}} : F_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A\\ &[\![f]\!]_{\mathcal{G}} = in_{\mathcal{G}} \circ \mathcal{G}\ [\![f]\!]_{\mathcal{G}} \circ f\end{aligned}$$

This time, we adapt it to recognize insertions and deletions at the head of the view while proceeding recursively. If none of the elements at the head of the view are related to source elements (i.e., if none of the positions in the range of the delta $get_{\triangle \mathcal{F}\ !}$ are contained in the domain of the delta representing the view-update), then we align the head as an insertion. For anamorphisms, the concept of head of the source is the more tricky one. Hence, if none of the elements of the source that would be necessary to build a head in the view are related to view elements (i.e., if none of the positions in the range

of the delta $get^{\circ}_{\triangle \llbracket f \rrbracket_{\mathcal{F}}} \circ get_{\triangle f} \circ get_{\triangle \mathcal{F}\,!}$ are contained in the range of the delta representing the view-update), then we delete the head of the source.

Formally, we specify the put of the anamorphism as follows:

$$put_{\llbracket f \rrbracket_{\mathcal{G}}}\,(v,s)\,d \;=\; \begin{cases} grow\,(v,s) & \text{if } V \neq \bot \wedge (\rho V \cap \delta d) = \bot \\ shrink\,(v,s) & \text{if } S \neq \bot \wedge (\rho S \cap \rho d) = \bot \\ put_{in_{\mathcal{G}} \circ \mathcal{G}\, \llbracket f \rrbracket_{\mathcal{G}} \circ f}\,(v,s)\,d & \text{otherwise} \end{cases}$$

$$\text{where} \quad V = get_{\triangle \mathcal{G}\,!} \quad S = get^{\circ}_{\triangle (\!| f |\!)_{\mathcal{G}}} \circ get_{\triangle f} \circ get_{\triangle \mathcal{G}\,!}$$

Insertion The $grow$ procedure is specified as depicted in Figure 4. The head of the view is isolated by applying $out_{\mathcal{G}}$ to produce a value of type $\mathcal{G}\ G\ A$. As for catamorphisms, we propagate this newly created head using the strength combinator $\sigma_{\mathcal{F}}$ followed by $\mathcal{G}\ put_{\llbracket f \rrbracket_{\mathcal{G}}}$ to apply put recursively using the original delta d. The recursively computed value of type $\mathcal{G}\ F\ A$ is then converted into a source value with a new head using $create_f$.

Deletion The $shrink$ procedure can be defined as depicted in Figure 5. To expose the head of the source, we invoke get_f to produce a value of type $\mathcal{G}\ F\ A$, that can be erased by the $reduce_{\mathcal{G}}$ function. We then apply $put_{\llbracket f \rrbracket_{\mathcal{G}}}$ to the argument view and the reduced source. In the implementation, $put_{\llbracket f \rrbracket_{\mathcal{G}}}$ is invoked with the delta $get^{\circ}_{\triangle} \circ reduce^{\circ}_{\mathcal{F}\,\triangle} \circ get^{\circ}_{\triangle f} \circ get_{\triangle} \circ d$ that reflects the shape changes performed by $reduce_{\mathcal{G}}$.

Examples Some examples of lenses that can be defined using this combinator are sieving a list to keep each second element, and list concatenation (combining a fold with an unfold):

$$sieve : A \to [\,]_A \trianglerighteq [\,]_A$$
$$sieve\ x = \llbracket (id \bullet\!\!\nabla\, \pi_2 + \pi_2{}^{const\ x}) \circ coassoc \circ (id + distr \circ (id \times out_{\mathcal{L}})) \circ out_{\mathcal{L}} \rrbracket_{\mathcal{L}}$$
$$\mathbf{data}\ NeList\ a = NeNil\ [a] \mid NeCons\ a\ (NeList\ a) \qquad NeList \cong \mu\mathcal{N}$$
$$cat : ([\,] \otimes [\,])_A \trianglerighteq [\,]_A$$
$$cat = (\!| in_{\mathcal{L}} \circ (id + id \bullet\!\!\nabla\, id) \circ coassoc \circ (out_{\mathcal{L}} + id) |\!)_{\mathcal{N}} \circ \llbracket (\pi_2{}^{!} + assoc) \circ distl \circ (out_{\mathcal{L}} \times id) \rrbracket_{\mathcal{N}}$$

6 Related Work

This paper builds on the work first presented in [PC10, PC11], describing a point-free lens language and corresponding algebraic laws. Like other state-based approaches [FGM+07, MHN+07], our previous language only considered a simple positional alignment strategy that proves to be unsatisfactory for insertion, deletion or reordering updates over arbitrary structures.

In [DXC11], Diskin *et al* discusses the inherent limitations of state-based approaches and proposed an abstract delta lens framework, whose lenses propagate deltas rather than states. They also show how delta lenses can be packaged as ordinary state-based lenses by resorting to an alignment operation that estimates deltas. Their development of the framework is mostly theoretical, focusing on the new bidirectional axioms for deltas and the relationship with ordinary lenses, and their only delta lens combinator is composition. An abstract framework where horizontal delta propagation is explicitly considered is given in [Dis11].

Matching lenses [BCF+10] extended the Boomerang domain-specific language of bidirectional string transformations [BFP+08] to consider delta-based alignment. Each matching lens separates values into a rigid shape and a list of data elements and maps an ordinary lens over the inner elements. The backward propagation can be computed using the delta associations inferred by the alignment phase. Since they focus on mappings, matching lenses assume that shape propagation is kept positional (SKELPUT law) and obey a restrictive premise enforcing the propagation of all source elements to the view (GETCHUNKS law), thus ruling out our two running examples.

The decoupling between shape and data is also at the heart of Voigtländer's semantic bidirectionalization approach [Voi09], that provides an higher-order put interpreter for polymorphic Haskell get functions. Nevertheless, this choice is motivated by different goals other than alignment, namely to avoid restricting the syntax of the forward transformations. In fact, mapping lenses are not definable in this framework, since polymorphic functions can only alter the shape, and shape alignment is kept positional even in the hybrid approach from [VHMW10], that uses a syntactically calculated state-based lens between shapes to handle shape updates.

A series of operation-based languages developed by researchers from Tokyo ([MHT04, LHT07, HHI+10] and more) treat alignment by annotating the view states with internal tags that indicate edit operations for specific types. Despite this enables put to provide a more refined type-specific behavior, it must always consider a fixed update language and more complicated updates (typically reorderings) are not supported natively and must be approximated by less exact updates.

A truly operation-based approach is the symmetric framework of *edit lenses* [HPW12], that handles updates in the form of the edits that describe the changes rather than whole annotated states. They provide combinators for inductive products, sums, lists and two particular combinators over container structures, namely mapping and restructuring. While mapping is similar to our delta-based variant, their restructuring combinator requires the positions of the transformed containers to be in bijective correspondence, meaning that it can not add nor delete elements and thus does not support our running examples. Additionally, their language of updates over containers classifies edits into insertion and deletion at the rear positions of containers and rearrangement of the elements of a container without changing its shape. This entails that shape alignment is kept positional, as insertions and deletions at arbitrary positions are always reflected at the end positions of the shape.

7 Conclusion

In this paper, we have proposed a concrete point-free delta lens language to build lenses with an explicit notion of shape and data over inductive data types, by lifting a previous state-based point-free lens language [PC10]. Our delta lens framework instantiates the abstract framework of delta lenses first introduced in [DXC11], meaning that lenses now propagate deltas to deltas and preserve additional delta-based bidirectional round-tripping axioms. In particular, we have instrumented the standard fold and unfold recursion patterns with mechanisms that use deltas to infer and propagate edit operations, thus performing the desired shape alignment. In the future, our technique could be instrumented to handle more refined edit operations that might make sense for particular types. The use of dependent types has provided a more concise formalism that simplifies the existing delta-based bidirectional theory and clarifies the connection between the state- and delta-based components of the framework. An implementation of our point-

free delta lenses, using a simple minimal edit sequence differencing algorithm [Tic84], in the Haskell non-dependently typed language is available through the Hackage package repository as part of the `pointless-lenses` library.

Likewise to matching lenses, that incorporate implicit parsing and pretty printing steps to decompose values into shape and data, a more practical implementation of delta lenses should be able to "deltify" ordinary point-free lenses by using type annotations that make the shape/data distinction explicit. We leave that extension for future work.

Acknowledgements This work is supported by Fundação para a Ciência e a Tecnologia, under grant PTDC/EIA-CCO/120838/2010 *FATBIT: Foundations, Applications and Tools for Bidirectional Transformation*. Part of this work was performed while Hugo Pacheco was visiting the National Institute of Informatics, supported by an NII Grand Challenge Project on Bidirectional Model Transformation.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.*, 342:3–27, 2005.

[2] D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *ICFP'10*, pages 193–204. ACM, 2010.

[3] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL'08*, pages 407–419. ACM, 2008.

[4] Z. Diskin. Model Synchronization: Mappings, Tiles, and Categories. In Joo Fernandes, Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *GTTSE'09*, volume 6491 of *LNCS*, pages 92–165. Springer, 2011.

[5] Z. Diskin, Y. Xiong, and K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *J Obj. Techn.*, 10:6: 1–25, 2011.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *TOPLAS'07*, 29(3):17, 2007.

[7] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP'10*, pages 205–216. ACM, 2010.

[8] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *POPL'12*. to appear, 2012.

[9] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM, 1997.

[10] C. Jay. A semantics for shape. *Sci. Comput. Program.*, 25:251–283, 1995.

[11] P. Johann and N. Ghani. Initial algebra semantics is enough! In *TLCA'07*, pages 207–222. Springer, 2007.

[12] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of xquery. In *PEPM'07*, pages 21–30. ACM, 2007.

[13] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP'07*, pages 47–58. ACM, 2007.

[14] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In Wei-Ngan Chin, editor, *APLAS'04*, volume 3302 of *LNCS*, pages 2–20. Springer, 2004.

[15] J. N. Oliveira. Data transformation by calculation. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE'07*, volume 5235 of *LNCS*, pages 139–198. Springer, 2007.

[16] H. Pacheco and A. Cunha. Generic Point-free Lenses. In Claude Bolduc, Jules Desharnais, and Bchir Ktari, editors, *MPC'10*, volume 6120 of *LNCS*, pages 331–352. Springer, 2010.

[17] H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *PEPM'11*, pages 91–100. ACM, 2011.

[18] W. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2:309–321, 1984.

[19] J. Voigtländer. Bidirectionalization for free! (Pearl). In *POPL'09*, pages 165–176. ACM, 2009.

[20] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *ICFP'10*, pages 181–192. ACM, 2010.

[21] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173. ACM, 2007.