Electronic Communications of the EASST
Volume 49 (2012)

Proceedings of the First International Workshop on Bidirectional Transformations (BX 2012)

Delta Lenses over Inductive Types

Hugo Pacheco, Alcino Cunha, Zhenjiang Hu

17 pages

Guest Editors: Frank Hermann, Janis Voigtländer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Delta Lenses over Inductive Types

Hugo Pacheco[1], **Alcino Cunha**[2], **Zhenjiang Hu**[3]

[1]hpacheco@di.uminho.pt
[2]alcino@di.uminho.pt
HASLab / INESC TEC & Universidade do Minho, Braga, Portugal
[3] hu@nii.ac.jp
National Institute of Informatics, Tokyo, Japan

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

In software engineering, data transformations are a ubiquitous tool to "bridge the gap" between technology layers and facilitate the sharing of information among software applications. However, users generally expect transformations to be bidirectional, in the sense that, if after a (forward) transformation both source and target instances co-exist and sometimes evolve independently, another (backward) transformation is able to bring the updated instances back to a consistent state.

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 views of type V (so that views contain less information than sources), 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[1]. Naturally, such synchronization is non-deterministic in general, 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.

[1] Readers unfamiliar with lenses may refer to [PC10] for a mild introduction to lenses over inductive types.

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.

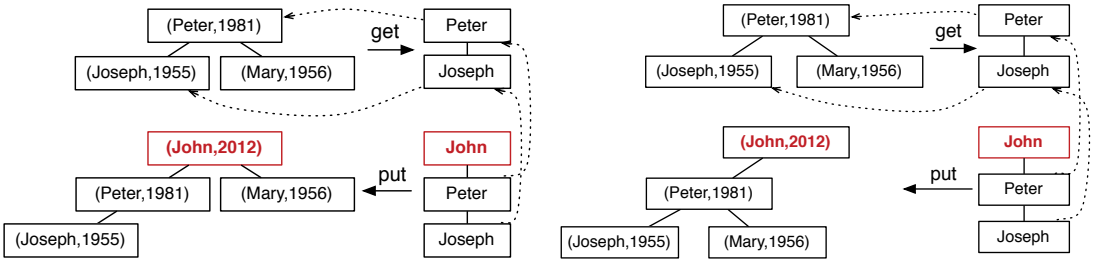
To unify both worlds and benefit from both the loose coupling of state-based approaches and the more refined updatability of operation-based approaches, Diskin *et al* [DXC11] formulated an abstract *delta lens* framework that encompasses an explicit alignment operation (that computes view deltas), and where *put* is an update-based transformation that propagates view deltas into source deltas. *Matching lenses* [BCF+10] are the first BX language that we are aware of promoting this separation principle. They generalize *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 the 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 matching lenses 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 regular state-based lens operating on its elements. In this setting, lenses are sensible to data modifications (on the types A and B of data) but not to shape modifications (on the type T of shapes) and the behavior of *put* 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 Haskell type representing a genealogical tree of persons and a corresponding transformation that computes a list of names of left ascendants in the tree:

$$\begin{array}{ll}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) & \textbf{type}\ \mathit{Person} = (\mathit{Name}, \mathit{Birth}) \\
\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) & \textbf{type}\ \mathit{Name} = \mathit{String} \quad \textbf{type}\ \mathit{Birth} = \mathit{Int} \\
\mathit{fathernames} : \mathit{Tree\ Person} \trianglerighteq \mathit{List\ Name} & \mathit{fatherline} : \mathit{Tree\ Person} \trianglerighteq \mathit{List\ Person} \\
\mathit{fathernames} = \mathit{names} \circ \mathit{fatherline} & \mathit{names} : \mathit{List\ Person} \trianglerighteq \mathit{List\ Name}
\end{array}$$

This transformation is defined in two steps: first compute the left ascendants with *fatherline*, and then select only their names using *names*. By porting the matching lens approach to this domain, we could easily define *names* using a list mapping $map : (A \to B) \to (List\ A \to List\ B)$. Unfortunately, *fatherline* does not fit the mapping corset imposed by the matching lens framework, since it reshapes the source tree into a list. Leaving *fatherline* as a standard state-based (positional)



(a) A data-aware putting back function.

(b) A data- and shape-aware putting back function.

Figure 1: A genealogical tree example.

lens would produce less-then-optimal results. For instance, for a source tree containing a person `Peter` and his parents `Joseph` and `Mary`, if we insert a new person named `John` at the head of the view, and infer a suitable view delta relating every name in the updated view but `John` to the respective person in the original view, *put* would behave as depicted in Figure 1a (where rectangles denote shape holes and dotted arrows represent deltas between shape positions).

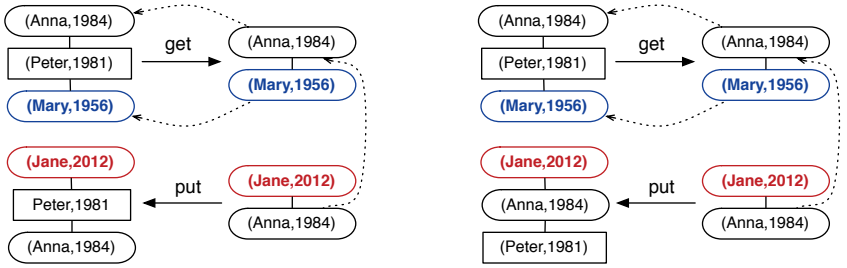
Although the order of names in the view changes, the birth years of existing persons are retrieved correctly due to the improved behavior of mapping modulo deltas (using `2012` as the default birth year for inserted persons), but the positional shape behavior of *fatherline* makes `Mary` (a parent of the first position in the original tree) an incorrect parent of `John` (the first position in the updated tree). With the extra delta information at hand we could have done better though: *fatherline* could recognize `John` as a new person and propagate his insertion to the source tree (with a default empty tree of right ascendants), as depicted in Figure 1b. It is easy to justify that this behavior on shapes induces a smaller change and is thus more predictable.

As another example, consider that we have a list of persons sorted by age, discriminating males and females, from which we filter all the females using the following transformation (where the shape is modeled by a list of optionals and data elements are either males or females in the list):

$$\textbf{data}\ \mathit{BiList}\ a\ b = \mathit{BiNil} \mid \mathit{ConsL}\ a\ (\mathit{BiList}\ a\ b) \mid \mathit{ConsR}\ b\ (\mathit{BiList}\ a\ b) \quad \textbf{type}\ \mathit{Male} = (\mathit{Name}, \mathit{Birth})$$
$$\mathit{females} : \mathit{BiList}\ \mathit{Male}\ \mathit{Female} \trianglerighteq \mathit{List}\ \mathit{Female} \qquad \textbf{type}\ \mathit{Female} = (\mathit{Name}, \mathit{Birth})$$

Again, this lens is not a mapping, as it changes the shape of the source by dropping some source elements. If we consider that it behaves positionally, inserting a new female `Jane` in the view list and deleting `Mary` would produce the source depicted in Figure 2a, where shape alignment is kept positional: the first female `Jane` in the updated view is aligned with the first female `Anna` in the source and the second female `Anna` in the updated view is aligned with the second female `Mary` in the source, while the male `Peter` is left in its original position in the source list. In the picture, rounded boxes denote females and rectangular boxes denote males. A better solution (Figure 2b) would be to use the deltas to recognize the inserted and deleted females, and propagate their modifications to the same relative positions: propagate the inserted female `Jane` to the head of the source, align `Anna` in the second position of the updated view with the first position of the source and delete `Mary` from the last position of the source. This behavior would induce a smaller source update that (for this case) would leave the source list sorted by age.

The lesson to learn is that likewise a positional *data alignment* (the matching of data elements)



(a) A positional putting back function.

(b) A shape-aware putting back function.

Figure 2: A filtering example.

is only reasonable for in-place updates, a positional behavior on shapes (that ignores the shape 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 view 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 BX 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

Functors The central requirement for this paper is the existence of types with 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 polymorphic type T is a functor $T : * \to *$ (in categorical terms, a functor between the same category $*$ with types as objects and functions as arrows) that applied to every type A returns a type $T\,A$, possessing a mapping between data elements $T\,f : T\,A \to T\,B$, for a given function $f : A \to B$. All the functors underlying data type declarations support a sums $\oplus$ of products $\otimes$ representation [PCH12]. A transformation f between functors F and G applied to data elements of type A and B (i.e., a

function with a notion of domain and target shapes) is denoted by $f : F\ A \to G\ B$, or to emphasize the shape, $f : F_A \to G_B$. Whenever a transformation η is polymorphic it is called a natural transformation, denoted by $\eta : F \dot{\to} G$, satisfying the naturality property $\eta \circ F\ f = G\ f \circ \eta$, for any function on data elements $f : A \to B$.

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[2] $\Sigma s : S.\ (P\ s \to A)$. 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], for each value $v : T\ A$. 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 $T\ 1$. For our tree and list types, positions are modeled as follows:

$$
\begin{array}{lll}
P : \forall \{T : * \to *\}, v : T\ A.\ * & & \\
P\ \{Tree\} & Empty & = 0 \\
P\ \{Tree\} & (Node\ x\ l\ r) & = 1 + P\ \{Tree\}\ l + P\ \{Tree\}\ r \\
P\ \{List\} & Nil & = 0 \\
P\ \{List\} & (Cons\ x\ t) & = 1 + P\ \{List\}\ t \\
P\ \{BiList\ B\} & BiNil & = 0 \\
P\ \{BiList\ B\} & (ConsL\ x\ t) & = P\ \{BiList\ B\}\ t \\
P\ \{BiList\ B\} & (ConsR\ y\ t) & = 1 + P\ \{BiList\ B\}\ t
\end{array}
$$

For the *BiList* type, we consider only positions on its second polymorphic argument, by fixing its first argument to the type B. The general definition for arbitrary polymorphic types is given in an accompanying technical report [PCH12]. Here, 0 is the empty type with no values and 1 is the unity type with a single value. The idea is that the (dependent) type of positions P is a tree resembling the structure of the value on which it depends, but considering only polymorphic elements. This tree representation ensures that each placeholder in the shape of a value is referenced by an unique position. Remember also that the type of positions is dependently defined for each value, and not for its shape. For example, for a list value with length n, the type of position is equivalent to the natural number n denoting the exact number of elements in the list.

Inspired by *shapely types* [Jay95] notation, a polymorphic type $T\ A$ can be characterized by three functions: $shape : T\ A \to T\ 1$ that extracts the shape, $data : \forall v : T\ A.\ (P\ v \to A)$ that extracts the data, and $recover : [\![T\ 1 \triangleright P]\!]\ A \to T\ A$ that rebuilds a value. For lists, the shape $List\ 1$ is isomorphic to naturals $Nat = \{0, 1, \ldots\}$, and thus we have $shape\ l = length\ l$, $P\ l = \{0\,..\,length\ l - 1\}$, and $data\ l = \lambda n \to l\ !!\ n$, where !! is a function that returns the element at position n in the list l.

[2] A dependent type may depend on values. The dependent function space $\forall a : A.\ B\ a$ denotes 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, as found in Agda [Nor09]. In principle, these parameters can be omitted and their value inferred from the context.

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

Figure 3: Point-free relational combinators

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 $P\ b \sim P\ a$ from positions in the target value to positions in the source value. 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 seriously restrict the kind of supported 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, this is, 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 3), whose behavior can be intuitively inferred from their signatures. Our combinators include relational composition ($\circ$) and regular set operations such as union ($\cup$), intersection ($\cap$) and difference ($-$). The converse of a relation R is given by R°, $\bot$ denotes the empty relation, and the other combinators handle products and sums[3]. The domain and range of a relation $r : A \sim B$ are coreflexive relations denoted by $\delta R \subseteq id : A \sim A$ and $\rho R \subseteq id : B \sim B$, respectively. By resorting to this language, we can reason about deltas using the powerful algebraic laws ruling its combinators. More details on this point-free relational calculus 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 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, transformations are no longer partially defined modulo

[3] To preserve simplicity, some combinators are only 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.

additional properties entailing preservation of the incidence between values and deltas. Instead, our precise characterization of deltas as relations between positions of specific values makes the difference and ensures that the domains and targets of the updates coincide with the old and new values. It also allows us to use the relational calculus to concisely express the delta-based laws. We adapt the definition of [DXC11] for our domain of polymorphic inductive types as follows:

Definition 1 (Delta lens) 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:

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

The d-lens is called *well-behaved* iff 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 fact that we require our transformations to be total, together with the state-based laws, also implies that the lenses defined with our language actually constitute views, i.e., view values always contain less information than source values. 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 $\lfloor l \rfloor_{diff} : S\,A \trianglerighteq V\,B$, 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 will define our delta lens language in 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 $l : S\,A \trianglerighteq_\triangle V\,B$, comprises three total functions *get*, *put*, and $create : V\,B \to S\,A$, plus three horizontal deltas:

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

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

$$get\ (create\ v) = v \qquad \textsc{CreateGet} \qquad\qquad put_\triangle\ d \circ get_\triangle = i_1 \qquad \textsc{PutGet}_\triangle$$
$$create_\triangle \circ get_\triangle = id \qquad \textsc{CreateGet}_\triangle \qquad\qquad [get_\triangle, id] \circ put_\triangle\ id = id \qquad \textsc{GetPut}_\triangle$$

The horizontal deltas are duals of the state-based functions that explicitly record the traceability of their execution: $get_\triangle$ denotes a delta from the original view to the original source and $create_\triangle$ conversely denotes a delta from the updated source to the updated view, while $put_\triangle$ 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 $P\,(v,s)$ is the disjoint sum of the positions in the view and in the source $P\,v + P\,s$ [PCH12]. For example, while the CREATEGET law states that abstracting a created source must yield the original view, the $\textsc{CreateGet}_\triangle$ law evidences that the delta on views must preserve all view elements (identity). The $\textsc{PutGet}_\triangle$ law states that the delta induced by a view-to-view roundtrip relates all elements in the updated view (that must be the same as the input view through PUTGET) to left elements in the input view-source pair.

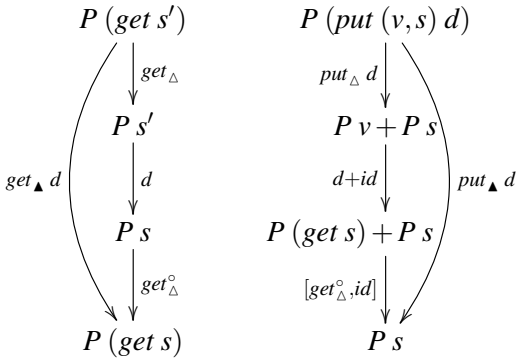


Figure 4: Illustration of the delta-level transformations from Definition 3.

We now show that hd-lenses implement the abstract framework of d-lenses:

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

To demonstrate that these deltas are composable in respect to the dependent types, their definitions are illustrated in Figure 4. The delta transformation $get_\blacktriangle$ is defined for a source update $d : s' \Delta s$, and $put_\blacktriangle$ for a view update $d : v \Delta get\ s$.

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

Proof. The state-based laws dismiss proofs. The delta-based laws can be proved by resorting

to the delta-based hd-lens laws [PCH12]. In particular, PUTGET$_\triangle$ and CREATEGET$_\triangle$ entail that $get_\triangle$ is a total and injective relation (see [Oli07]), from what we can derive that $get_\triangle^\circ \circ get_\triangle = id$, what constitutes the crucial part of the proof. □

4 Combinators for Horizontal Delta Lenses

In this section, we introduce two primitive hd-lens combinators for mapping and reshaping transformations and define liftings of our point-free lens combinators from [PC10] to hd-lens combinators that can be used to define more complex transformations in a compositional way.

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

$$\begin{array}{l} \forall l : A \trianglerighteq B.\ S\ l : S_A \trianglerighteq_\triangle S_B \\ \begin{array}{lll} get\ s & = S\ get_l\ s & get_\triangle = id \\ put\ (v,s)\ d & = recover\ (shape\ v, dput \cup dcreate) & put_\triangle\ d = i_1 \\ \quad \textbf{where}\ dput & = put_l \circ \langle data\ v, data\ s \circ d \rangle & \\ \qquad dcreate & = (create_l \circ data\ v) \circ (id - \delta dput) & \\ create\ v & = S\ create_l\ v & create_\triangle = id \end{array} \end{array}$$

Likewise 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 delta: for each view element v_e, if it relates to a source element s_e, $put\ (v_e, s_e)$ is applied[4]; otherwise a default source is generated with $create_l\ v_e$. The $put_\triangle$ delta is also trivial, since all elements in the new source come from elements in the view.

Reshaping 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:

$$\begin{array}{l} \forall \eta : S \dot{\trianglerighteq} V.\ \overline{\eta} : S \dot{\trianglerighteq}_\triangle V \\ \begin{array}{llll} get\ s & = get_\eta\ s & get_\triangle\ \{s\} & = \overleftarrow{get_\eta}\ s \\ put\ (v,s)\ d & = put_\eta\ (v,s) & put_\triangle\ \{(v,s)\}\ d & = \overleftarrow{put_\eta}\ (v,s) \\ create\ v & = create_\eta\ v & create_\triangle\ \{v\} & = \overleftarrow{create_\eta}\ v \end{array} \\ \forall \eta : F \dot{\to} G.\ \overleftarrow{\eta} : \forall s : F\ A.\ \eta\ s \Delta s \\ \overleftarrow{\eta} = data\ (\eta\ (recover\ (shape\ s, id))) \end{array}$$

Although this combinator permits defining hd-lenses that transform the shape of the source, it just infers suitable horizontal deltas for an existing state-based lens. Therefore, 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

[4] Here, $dput \cup dcreate$ builds a (total) 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.

value-level functions against sources with the data elements replaced by the respective positions, thus inferring the correspondences in the view. 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. Since the behavior of the lifted d-lens is completely determined by the argument lens, using this combinator to define the *fatherline* and *females* examples from the introduction (which indeed are natural lenses) as hd-lenses would not perform proper alignment.

Composition We now show that the point-free combinators from our previous state-based lens language [PC10, PC11] are also valid hd-lenses. Two fundamental point-free combinators are identity (trivial to define) and composition. The latter can be lifted to hd-lenses as follows:

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

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

In the *put* direction, the intermediate delta d_V passed to put_g maps elements in the result of $put_f\ (v, get_g\ s)\ d_U$ to elements in $get_s\ s$. 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 for d-lenses not matching on their intermediate shapes is also possible, by first converting them into normal lenses, as used in [BCF+10]. However, this composition is deemed ill-formed in [DXC11], since the resulting lenses may identify and align updates differently.

Product Lifting the binary product type $\times$ into a binary functor $\otimes : * \to * \to *$, we can define the following product projection hd-lenses:

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

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

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

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

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

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

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

When computing *put*, the product combinator splits the view delta 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 *put*s 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.

Similar definitions for other point-free combinators, such as sums, can be found in [PCH12].

5 Recursion Patterns as Horizontal Delta Lenses

Although useful for a combinatorial language, the previous 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 shape 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 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 shape alignment problem is not addressed, then the tendency of a positional shape alignment 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 shape to be ignored.

The goal of this section is to understand how we can use the delta information to infer meaningful shape updates. However, propagating shape updates requires knowing the behavior of the transformation, in order to establish correspondences between source and view shapes. Instead of considering arbitrary reshaping lenses, we introduce two regular structural recursion combinators that perform shape alignment: catamorphisms (folds) that consume recursive sources, and anamorphisms (unfolds) that produce recursive views.

5.1 Identifying and Propagating Shape Updates

Our general idea for shape alignment is to identify insertions and deletions at the "head" of the view shape, and propagate them to corresponding insertions and deletions at the "head" source shape. Consider, as an example, the forward transformation for the *fatherline* lens:

$$
\begin{aligned}
&get_{fatherline} : Tree\ a \to List\ a \\
&get_{fatherline}\ Empty = Nil \\
&get_{fatherline}\ (Node\ x\ l\ r) = Cons\ x\ (get_{fatherline}\ l)
\end{aligned}
$$

This function traverses the input tree and, for each non-empty node, builds a list whose head is the root element and whose tail is computed by recursively applying the transformation to the left child of the tree. The "head" of a value of an arbitrary recursive type can be considered as

everything in its type constructors besides recursive invocations, i.e., something with the same top-level shape but with the recursive occurrences erased. For non-empty trees and lists, these heads coincide with top-level elements of a value.

A suitable state-based $put_{fatherline}\ (v,s)\ d$ would then recursively match the view list v and the source tree s, consuming their respective heads at each recursive step, but disregarding the view delta d. To avoid this positional behavior, we propose to offset such default matching for insertions and deletions, using the view delta to infer shape modifications. In general, when executing *put*, if none of the elements at the "head" of the new view are related to elements in the original view, then we are confident that they were created with the update and shall be propagated as an insertion to the source. Conversely, if none of the elements at the "head" of the original view are related to elements in the new view, then such "head" shall be deleted from the original source before proceeding. Otherwise, we proceed positionally.

For the *fatherline* example, since each *Cons* in the original list came from a *Node* in the original tree, if we insert a new *Cons* at the head of the new list, then we must insert a new *Node* at the head of the new tree, with any default right child since it will be ignored by $get_{fatherline}$. For insertions, we can define $put_{fatherline}\ (Cons\ h\ t)\ (Node\ x\ l\ r) = Node\ h\ (put_{fatherline}\ t\ (Node\ x\ l\ r))\ Empty$, with *Empty* as a default right child. If we delete a *Cons* from the original view, then we must delete the corresponding *Node* from the original tree, leaving an additional choice on how to merge the children of deleted source node into a single tree. Proceeding recursively, the left spine of the new source tree will be copied from the view list and right children will be recovered from the merged tree. For deletions, we can define $put_{fatherline}\ (Cons\ h\ t)\ (Node\ x\ l\ r) = put_{fatherline}\ (Cons\ h\ t)\ (plus\ l\ r)$, where *plus* is any function that merges the left and right trees.

Next, we show how to generalize this mechanism for arbitrary folds and unfolds[5].

5.2 Generalizing Shape Alignment for Folds and Unfolds

Higher-order functors A recursive polymorphic data type T can be represented as the fixed point $\mu\,\mathcal{F}$ of a higher-order base functor $\mathcal{F} : (* \to *) \to * \to *$, together with the functions $out_{\mathcal{F}} : T \dot{\to} \mathcal{F}\ T$ and $in_{\mathcal{F}} : \mathcal{F}\ T \dot{\to} T$ that unpack and pack recursive values. In Haskell, the higher-order base functors for lists, trees and lists of optional elements can be defined as polymorphic types parameterized by a functor argument and a type argument:

data $\mathcal{L}ist\ list\ a = \mathcal{N}il \mid \mathcal{C}ons\ a\ (list\ a)$	$List\ a = (\mu\,\mathcal{L}ist)\ a$
data $\mathcal{T}ree\ tree\ a = \mathcal{E}mpty \mid \mathcal{N}ode\ a\ (tree\ a)\ (tree\ a)$	$Tree\ a = (\mu\,\mathcal{T}ree)\ a$
data $\mathcal{B}i\mathcal{L}ist\ a\ blist\ b = \mathcal{B}i\mathcal{N}il \mid \mathcal{C}ons\mathcal{L}\ a\ (blist\ b) \mid \mathcal{C}ons\mathcal{R}\ b\ (blist\ b)$	$BiList\ a\ b = (\mu(\mathcal{B}i\mathcal{L}ist\ a))\ b$

The functor arguments *list* and *tree* mark recursive invocations and the the last type variable denotes data elements. For the *BiList* type, we consider only right data elements by fixing the first parameter a. For a particular class of regular higher-order functors [PCH12], we can define a hd-lens combinator $\forall f : F_A \trianglerighteq_\triangle G_A.\ \mathcal{F}\ f : \mathcal{F}\ F_A \trianglerighteq_\triangle \mathcal{F}\ G_A$ that maps a functor transformation over the functor argument of an higher-order functor. 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).

[5] The next section assumes a background in the Algebra of Programming approach to functional programming [BM97].

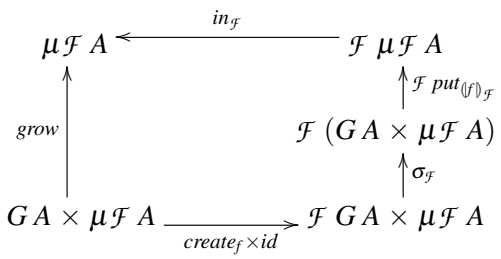


Figure 5: Specification of *grow* for folds.

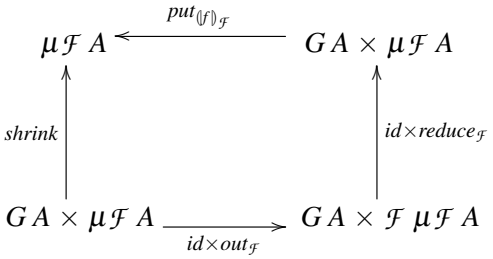


Figure 6: Specification of *shrink* for folds.

Catamorphism Given a hd-lens algebra $f : \mathcal{F}\ G_A \trianglerighteq_{\triangle} G_A$, the catamorphism $(\!|f|\!)_{\mathcal{F}} : \mu\mathcal{F}_A \trianglerighteq_{\triangle} G_A$ can 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 use them to perform shape alignment. A catamorphism recursively consumes source values, and at each iteration generates a target value for each consumed head. For recursive source values, the head can be computed generically by the expression $\mathcal{F}\ ! \circ out_{\mathcal{F}} : \mu\mathcal{F}\,A \to \mathcal{F}\ \underline{1}\,A$. where $! : F\,A \to \underline{1}\,A$ is a function that erases the functorial structure of the argument value by replacing it with the unit type lifted as a constant functor $\underline{1}$. However, the view type is not recursive in general and the notion of head of the view induced by the fold is a bit more tricky. To identify modifications at the head of the view, what we need to compare are 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 as before: $\mathcal{F}\ ! \circ create_f : G\,A \to \mathcal{F}\ \underline{1}\,A$. Formally, we specify the *put* of the catamorphism as follows:

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

Here, the V and S relations are the corresponding deltas that dualize the functions used to calculate the head of the view and the source, respectively. A more detailed explanation is given in [PCH12].

Insertion To check for an insertion, we test if none of the elements at the head of the modified view is related to the original view. To propagate a newly created head, we need a way to pair each sub-view of type $G\,A$ 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 regular higher-order functors [PCH12]. Not taking deltas into account, the *grow* procedure can be specified as depicted in Figure 5. Note that if the source functor $\mathcal{F}$ 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 $\sigma_{\mathcal{F}}$ 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*.

Deletion To check for a deletion, we test if none of the elements at the head of the original view (computed from the head of the original source) is related to the modified view. To propagate the deletion, we unfold the original source to expose its head to be erased by a function $reduce_{\mathcal{F}} : \mathcal{F}\ F\ A \to F\ A$ (defined in [PCH12]), and then apply $put_{(\!|f|\!)_{\mathcal{F}}}$ recursively to the modified view and the reduced source. Again not taking deltas into account, the *shrink* procedure can be specified as depicted in Figure 6. In order to erase the head, function $reduce_{\mathcal{F}}$ shall 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 his own implementation. We only 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 *fatherline* and *names* steps of the composite *fathernames* lens can be defined as follows:

$$
\begin{array}{ll}
fatherline : Tree\ _{Person} \trianglerighteq_{\triangle} List\ _{Person} & names : List\ _{Person} \trianglerighteq_{\triangle} List\ _{Name} \\
fatherline = (\!|in_{List} \circ (id + id \times \pi_1{}^{const\ Nil})|\!)_{Tree} & names = map\ \pi_1{}^{const\ 2012}
\end{array}
$$

$$
\begin{array}{ll}
plus_{Tree} : Tree \otimes Tree \dashrightarrow Tree & zero_{Tree} : \forall A.\ Tree\ A \\
plus_{Tree}\ Empty\ r = r & zero_{Tree} = Empty \\
plus_{Tree}\ l\ r = l &
\end{array}
$$

When ran against the introductory example from Figure 1, the composed lens produces the desired result. Note that for inserted persons, put_{names} will create a default birth year 2012 (due to *const* 2012) and $put_{fatherline}$ will generate a default empty list of right ascendants (due to *const Nil*), that when aligned with any source tree will always return an empty right tree. For deletions, the given monoid selects one of the child trees if the other is empty, or discards the right child otherwise. Also for the *fathernames* example, we know that an insertion followed by a deletion (of a person `John` for instance) would lead to no effect on the source.

The *females* example from the introduction can also be encoded as a fold:

$$
\begin{array}{l}
females : (BiList\ Male)\ _{Female} \trianglerighteq_{\triangle} List\ _{Female} \\
females = (\!|(in_{List} \bullet\nabla\ \pi_2) \circ coassoc \circ (id + coswap \circ distl)|\!)_{BiList\ Male}
\end{array}
$$

By running this lens against the example from Figure 2, males are now restored properly. The either hd-lens $\bullet\nabla$ is defined in [PCH12] and $coassoc : (F \oplus (G \oplus H))_A \trianglerighteq_{\triangle} ((F \oplus G) \oplus H)_A$, $coswap : (F \oplus G)_A \trianglerighteq_{\triangle} (G \oplus F)_A$ and $distl : ((F \oplus G) \otimes H)_A \trianglerighteq_{\triangle} ((F \otimes H) \oplus (G \otimes H))_A$ are hd-lens isomorphisms. The lifting of unfolds or *anamorphisms* into a hd-lens combinator $\forall f : F_A \trianglerighteq_{\triangle} \mathcal{G}\ F_A.\ [\![f]\!]_{\mathcal{G}} : F_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A$ can be done analogously to folds, and is presented in [PCH12].

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 synchronization framework where vertical and horizontal deltas are 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 alignment 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 as edits that describe the changes rather than whole annotated states. They provide combinators for inductive products, sums, lists and two mapping and reshaping combinators over container structures. While mapping is similar to our delta-based variant, their reshaping 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 reordering 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

The "holy grail" of BX approaches is to find solutions that mitigate the ambiguity of view-update translation, by producing minimal source updates. For the application domain of inductive data

types, we identify that a smaller update requires not only to align data elements, but also shapes.

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 on shapes, thus producing smaller source updates that best reflect a certain view update. Thus far, we only consider insertion and deletion updates on shapes, that are sufficiently generic to express modifications on a wide range of data types. Nevertheless, other more refined edit operations (like tree rotations) might make sense for particular types and application scenarios, and our technique could be instrumented to cover more edits in the future.

The use of dependent types has provided a more concise formalism that simplifies the existing delta-based BX 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 at the Hackage package repository as part of the `pointless-lenses` library.

Likewise 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 funded by the ERDF through the programme COMPETE and by the Portuguese Government through FCT (Foundation for Science and Technology), project reference FCOMP-01-0124-FEDER-020532 *FATBIT: Foundations, Applications and Tools for Bidirectional Transformation*. Part of this work was performed while Hugo Pacheco was visiting the NII, supported by a NII Grand Challenge Project on Bidirectional Model Transformation.

Bibliography

[AAG05] M. Abbott, T. Altenkirch, N. Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.* 342:3–27, 2005.

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

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

[BM97] R. Bird, O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Dis11] Z. Diskin. Model Synchronization: Mappings, Tiles, and Categories. In Fernandes et al. (eds.), *GTTSE'09*. LNCS 6491, pp. 92–165. Springer, 2011.

[DXC11] Z. Diskin, Y. Xiong, K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *J Obj. Techn.* 10:6: 1–25, 2011.

[FGM+07] 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. *TOPLAS'07* 29(3):17, 2007.

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

[HPW12] M. Hofmann, B. C. Pierce, D. Wagner. Edit lenses. In *POPL'12*. to appear, 2012.

[Jay95] C. Jay. A semantics for shape. *Sci. Comput. Program.* 25:251–283, 1995.

[LHT07] D. Liu, Z. Hu, M. Takeichi. Bidirectional interpretation of XQuery. In *PEPM'07*. Pp. 21–30. ACM, 2007.

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

[MHT04] S.-C. Mu, Z. Hu, M. Takeichi. An Algebraic Approach to Bi-Directional Updating. In Chin (ed.), *APLAS'04*. LNCS 3302, pp. 2–20. Springer, 2004.

[Nor09] U. Norell. Dependently Typed Programming in Agda. In Koopman et al. (eds.), *Advanced Functional Programming*. LNCS 5832, pp. 230–266. Springer, 2009.

[Oli07] J. N. Oliveira. Data Transformation by Calculation. In Lämmel et al. (eds.), *GTTSE'07*. LNCS 5235, pp. 139–198. Springer, 2007.

[PC10] H. Pacheco, A. Cunha. Generic Point-free Lenses. In Bolduc et al. (eds.), *MPC'10*. LNCS 6120, pp. 331–352. Springer, 2010.

[PC11] H. Pacheco, A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *PEPM'11*. Pp. 91–100. ACM, 2011.

[PCH12] H. Pacheco, A. Cunha, Z. Hu. Delta Lenses over Inductive Types. Technical report TR-HASLab:02:2012, University of Minho, Feb. 2012. Available at http://www.di.uminho.pt/~hpacheco/publications/dlenses-tr.pdf.

[Tic84] W. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.* 2:309–321, 1984.

[VHMW10] J. Voigtländer, Z. Hu, K. Matsuda, M. Wang. Combining syntactic and semantic bidirectionalization. In *ICFP'10*. Pp. 181–192. ACM, 2010.

[Voi09] J. Voigtländer. Bidirectionalization for free! (Pearl). In *POPL'09*. Pp. 165–176. ACM, 2009.

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