Supervised learning, change propagation, and delta lenses

Zinovy Diskin
McMaster University, CAS
diskinz@mcmaster.ca
Supervised learning, change propagation, and delta lenses

Zinovy Diskin
McMaster University, CAS
diskinz@mcmaster.ca

Abstract. Delta lenses are an established mathematical framework for modelling and designing bidirectional model transformations (Bx). Following the recent observations by Fong et al, the paper extends the delta lens framework with a new ingredient: learning over a parameterized space of model transformations seen as functors. We will define a notion of an asymmetric learning delta lens with amendment (ala-lens), and show how ala-lens can be organized into a symmetric monoidal category. We also show that sequential and parallel composition of well-behaved (wb) ala-lenses is also wb so that wb ala-lenses constitute a full subcategory of ala-lenses.

1 Introduction

In a seminal paper [1], Fong, Spivak and Tuyéras showed how to compose supervised machine learning (ML) algorithms so that the latter form a symmetric monoidal (sm) category \( \text{Learn} \), and built an sm-functor

\[
L_{\varepsilon, \text{err}} : \text{Para} \rightarrow \text{Learn},
\]

which maps a parameterized differentiable function \( f : P \times \mathbb{R}^m \rightarrow \mathbb{R}^n \) (with \( P = \mathbb{R}^k \) being the parameter space) to a learning algorithm called a learner; the latter improves an initially given function \( f(p, \_): \mathbb{R}^m \rightarrow \mathbb{R}^n \) by learning from a set of training pairs \((a, b) \in \mathbb{R}^m \times \mathbb{R}^n\). The functor is itself parameterized by a step size \( 0 < \varepsilon \in \mathbb{R} \) and an error function \( \text{err}: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \) needed to specify the subject function of the gradient descent procedure. Recently, Fong and Johnson noticed in [2] (quoting them directly) “surprising links between two apparently disparate areas”: ML (treated compositionally as above) and bidirectional model transformations, Bx (also treated compositionally in a framework of mathematical structures called lenses [3]), whereas “naively at least, there seemed to be little reason to expect them to be closely related mathematically”[4]

1Term \( Bx \) abbreviates “bidirectional something (or x)” and refers to bidirectional change propagation in different contexts in different domains: file synchronization in versioning, data exchange in databases, model synchronization and model transformation in Model-Driven software Engineering (MDE), see [4] for some of these contexts. Bx also refers to a community working across those domains but self-integrated by using the conceptual and terminological framework provided by lenses. The latter lay a foundational common ground for a vast variety of synchronization tasks, and a whole zoo of lenses has been created to address different particular problems (cf. [5]). Thus, within Bx the application context can vary, sometimes significantly, but in the present paper, abbreviation Bx will mainly refer to Bx in the MDE context. Sometimes we will need both a general Bx and a special Bx for MDE, then the latter will be denoted by \( Bx_{\text{MDE}} \).
The goal of the present paper is to show that incorporating the supervised learning idea into Bx is both practically useful and theoretically natural within the lens framework. We need a new species of lenses—lenses with learning capabilities or learning lenses—to be created and added to the lens zoo. In fact, learners by Fong, Spivak and Tuyéras can be seen as codiscrete learning lenses, and the category of classical (asymmetric) codiscrete lenses \(a\text{Lens}\) is a full subcategory of \(\text{Learn}\) for which the parameter space is a singleton set\(^2\). Here the attribute ‘codiscrete’ refers to the fact that spaces over which lenses operate are sets (rather than categories but are) considered as codiscrete categories: every pair of elements \((x, y)\) is an arrow \(x \to y\) and all arrows are such (the index aims to recall this type of connectivity sometimes referred to as chaotic). The prefix \(a\) in the name \(a\text{Lens}\) refers to so called asymmetric lenses (as opposed to symmetric ones); although the only lenses we will consider in the paper are asymmetric and, as a rule, we omit this attribute, it’s still useful to keep in the names of categories for future use.

The main motivational point of the present paper is that codiscrete lenses are inadequate to Bx MDE (this is discussed in detail in \([6,7]\) and briefly outlined in a “trailer” in Sect. \(2\)) and we thus need learning \(\delta\) lenses that work over categories rather than sets. The inset figure shows the story in a nutshell: we have two orthogonal ways of enriching codiscrete lenses, and the goal of the paper is to integrate them into a new type of lenses located at point 11, and analyze some of its properties. We will define the notion of \(a(n\text{ asymmetric})\) learning \(\delta\) lens (\(a\delta\text{lens}\)) and show that \(a\delta\text{lens}\) can be organized into an sm-category \(a\text{LLens}\) such that learners can be identified with twice codiscrete \(a\delta\text{lens}\): \(\text{Learn} = a\text{LLens} \subset a\text{LensaLensaLensa}\). There are also many other details and more accurate versions of the plane.

Lenses appearing in Bx applications satisfy several equational laws assuring that update propagation restores consistency (or, at least, improves it); such lenses are (often loosely) called well-behaved (\(\text{wb}\)). The laws can be either strict or laxed, in which case the equality is only achieved after a mediating update called an amendment is applied. This gives rise to a new species of lenses with amendment, \(a\text{a-lenses}\), recently formally defined in \([8]\). To follow our agenda in Fig. \(1\), we will define (asymmetric) learning lenses with amendment, \(a\alpha\text{lenses}\), and show that they form an sm-category \(a\alpha\text{LensaLensaLensa}\); lenses without amendment can be seen as \(a\alpha\text{lenses}\) with an identity amendment adn inherit the sm-structure, \(a\alpha\text{LensaLensaLensa} \subset \text{sm}\ a\alpha\text{LensaLensaLensa}\). Moreover, we show that sequential and parallel (tensor) composition of \(a\alpha\text{lenses}\) preserve two major well-behavedness laws (Stability and Putget) so that we have a full subcategory of \(\text{wb}\) (in this sense) \(a\alpha\text{lenses}\), \(a\alpha\text{LensaLensaLensa} \subset a\alpha\text{LensaLensaLensa}\). Finally, in parallel to the notion of a learning functor \([1]\) we will define the notion of a (well-behaved, compositional) model transformation language as an sm-functor

\[
L_{\text{mtl}}: p\text{Get}_{\text{nat}} \to a\alpha\text{LensaLensaLensa}_{\text{wb}},
\]

\(^2\) According to \([2]\), this was first noticed by Jules Hedges.
2 Background: Update propagation, policies, and delta lenses

whose source is a category of mtl-defined parameterized transformations \( pGet_{\text{mtl}} \) (whose arrows are called \( \text{get} \)s to be read “get the transformation done”). Functionality required in (2) should be an important requirements to a transformation language, but it seems to be missing from the current practice of model transformation language design and evaluation.

The paper is structured as follows. Section 2 briefly motivates delta lenses and explains the basic notions of the delta lens framework. Section 3 discusses why the supervised learning idea is useful and natural for Bx, and compares codiscrete and categorical learning. The cornerstone of this analysis is the category \( p\text{Cat} \) of all (small) categories and (equivalence classes of) parameterized functors between them, and its full subcategory \( p\text{Set} \) of all (small) sets and (equivalence classes of) parameterized functions between them. This material does not actually belong to lenses per se, and is placed in Appendix Sect. A. Some details behind the scene in Sect. 3 will be fully clear after ala-lenses, and their sequential and parallel compositions, are formally defined in Sect. 4 and 5 resp. The three main results of the paper are Theorems 1-2 (pages 16-16) stating that the two compositions preserve the two major ala-kens laws, and Theorem 3 showing how the universe of ala-lenses can be organized into an sm-category. All proofs (sufficiently straightforward but notationally laborious) are placed into Appendices. Section 6 reviews the related work, and Sect. 7 concludes.

About notation used in the paper. In a general context, an application of function \( f \) to argument \( x \) will be denoted by \( f(x) \). But many formulas in the paper will specify terms built from two operations going in the opposite directions (this is in the nature of the lens formalism): in our diagrams, operation \( \text{get} \) maps from the left to the right while operation \( \text{put} \) maps in the opposite direction. To minimize the number of brackets, and relate a formula to its supporting diagram, we will also use the dot notation in the following way. If \( x \) is an argument in the domain of \( \text{get} \), we tend to write formula \( x' = \text{put}(\text{get}(x)) \) as \( x' = \text{put}(x, \text{get}) \) while if \( y \) is an argument in the domain of \( \text{put} \), we tend to write the formula \( y' = \text{get}(\text{put}(y)) \) as \( (\text{put}, y, \text{get}) = y' \) or \( (\text{put}, y)\text{get} = y' \). Unfortunately, this discipline is not always well aligned with the in-fix notation for sequential (\( ; \)) and parallel/tesorial (\( || \)) composition of functions, so that some notational mix remained.

Given a category \( A \), its objects are denoted by capital letters \( A, A' \), etc. to recall that in MDE applications, objects are complex object structures (e.g., they can even be databases); the collection of all objects of \( A \) is denoted by \( |A| \). An arrow with domain \( A \in |A| \) is denoted by \( u: A \rightarrow - \) and we write \( s(u) = A \) (or sometimes \( u^A = A \)) and \( u \in A(A, -) \). Similarly \( u: - \rightarrow A' \) is an arrow with codomain \( A' \): \( t(u) = u^A = A' \) and \( u \in A(-, A') \). A subcategory \( B \subset A \) is called \( \text{wide} \) if it has the same objects.

2 Background: Update propagation, policies, and delta lenses

Classical Bx lenses specify synchronization of a pair of models \((A, B)\) connected by a transformation, say, \( A.F = B \). Think, e.g., of \( A \) as a UML model and \( B \) as a Java program generated from it, or \( A \) may be a Java program and \( B \) is its bytecode, or \( A \) can be an object model and \( B \) is its relational storage, see [9] for a variety of such examples. In more detail, we have two model spaces (think of two sets or two categories) \( A \) and \( B \), and a transformation mapping (a function
or functor) between them, \( F \colon A \to B \), such that \( F(A) = B \). If the target model \( B \) is updated to \( B' \) and consistency between the two sides is violated, the lens “sitting” over \( F \) prescribes how the source model \( A \) is to be updated to \( A' \) so that \( A', F = B' \) and consistency is restored. In the Bx jargon, one also says that the change \( B \to B' \) is (backward) propagated to change \( A \to A' \).

In this section, we will consider a simple example demonstrating main concepts and ideas of Bx\textsubscript{MDE}. Although Bx ideas work well in domains conforming to the slogan \textit{any reasonable implementation is good enough} such as code generation and model refinement in some contexts (see [9] for details and discussion), and have rather limited applications in databases (only so called updatable views can be treated in the Bx-way), we will employ a simple database example: it allows demonstrating the core ideas without any special domain knowledge required by typical Bx-amenable areas. The presentation will be semi-formal as our goal is to motivate the delta lens formalism that abstracts the details away rather than formalize the example as such.

2.1 Why deltas.

Bx -lenses first appeared in the work on file synchronization, and if we have two sets of strings, say, \( B = \{ \text{John}, \text{Mary} \} \) and \( B' = \{ \text{Jon}, \text{Mary} \} \), we can readily see the difference: \( \text{John} \neq \text{Jon} \) but \( \text{Mary} = \text{Mary} \). We thus have a structure in-between \( B \) and \( B' \) (which maybe rather complex if \( B \) and \( B' \) are big files), but this structure can be recovered by string matching and thus updates can be identified with pairs. The situation dramatically changes if \( B \) and \( B' \) are object structures, e.g., \( B = \{ o_1, o_2 \} \) with \( \text{Name}(o_1) = \text{John}, \text{Name}(o_2) = \text{Mary} \) and similarly \( B' = \{ o'_1, o'_2 \} \) with \( \text{Name}(o'_1) = \text{Jon}, \text{Name}(o'_2) = \text{Mary} \). Now string matching does not say too much: it may happen that \( o_1 \) and \( o'_1 \) are the same object (think of a typo in the dataset), while \( o_2 \) and \( o'_2 \) are different (although equally named) objects. Of course, for better matching we could use full names or ID numbers or something similar (called, in the database parlance, primary keys), but absolutely reliable keys are rare, and typos and bugs can compromise them anyway. Thus, for object structures that Bx\textsubscript{MDE} needs to keep in sync, deltas between models need to be independently specified, e.g., by specifying a sameness relation \( u \subset B \times B' \) between models. For example, \( u = \{ o_1, o'_1 \} \) says that \( \text{John}@B \) and \( \text{Jon}@B' \) are the same person while \( \text{Mary}@B \) and \( \text{Mary}@B' \) are not. Hence, model spaces in Bx\textsubscript{MDE} are categories (objects are models and arrows are update/delta specifications) rather than sets (codiscrete categories).

2.2 Update propagation with delta lenses

Example. Figure 2 presents a simple example of delta propagation for consistency restoration. Models are objects with attributes (a.k.a. labelled records), e.g., the source model \( A \) consists of three objects (identified by their OIDs \#A, \#J, \#M, think about employees of some company) with attribute values as shown in the table (attribute Expr. refers to Experience measured by a number of years, and Depart. is the column of department names). The schema of the table, i.e., the triple of attribute names (Name, Expr., Depart.) (with their domains of values String, Integer, String resp.) determine a model space \( A \). The target

\[ \begin{array}{|c|c|c|}
\hline
\text{Name} & \text{Expr.} & \text{Depart.} \\
\hline
\text{John} & 3 & \text{Depart. 1} \\
\text{Mary} & 5 & \text{Depart. 2} \\
\text{Jon} & 2 & \text{Depart. 3} \\
\hline
\end{array} \]

\[ \text{Fig. 2.} \]
2 Background: Update propagation, policies, and delta lenses

model space $\mathbf{B}$ is given by a similar schema $S_{\mathbf{B}}$ consisting of two attribute names. For any model $X \in \mathbf{A}$, we can compute its $\mathbf{B}$-view $\mathbf{get}(X)$ by selecting those OIDs $\#O \in \text{OID}^X$ for which $\text{Depart}_X(\#O) \in \{\text{Testing}, \text{ML}, \text{DB}\}$; we will refer to departments, whose names are in $\{\text{Testing}, \text{ML}, \text{DB}\}$ as to IT-departments and the view $\mathbf{get}(X)$ as the IT-view of $X$. For example, the upper part of the figure shows the IT-view $B$ of model $A$. We assume that all column names in schemas $S_A$, and $S_B$ are qualified by schema names, e.g., $\text{OID}@[S_A], \text{OID}@[S_B]$ etc, so that schemas are disjoint except elementary domain names like $\text{String}$ and $\text{Integer}$. Also disjoint are OIDs, e.g., $\#J@[A]$ and $\#J@[B]$ are different elements, but, of course, constants like John and Mary are elements of set $\text{String}$ shared by both schemas. To shorten long expressions in the diagrams, we will often omit qualifiers and write $\#J = \#J$ meaning $\#J@[A] = \#J@[B]$ or $\#J@[B] = \#J@[B]'$ depending on the context given by the diagram; often we will also write $\#J$ and $\#J'$ for such OIDs. Also, when we write $\#J = \#J$ inside block arrows denoting updates, we actually mean a pair, e.g., $([\#J@[B], \#J@[B]'])$.

Given two models over the same schema, say, $B$ and $B'$ over $S_B$, an update $v : B \to B'$ is a relation $v \subset \text{OID}^B \times \text{OID}^B$; if the schema were containing more nodes, an update should provide such a relation $v_N$ for each node $N$ in the schema. However, we do not require naturality: in the update $v_2$ specified in the figure, for object $\#J \in \text{OID}^B$, we have $\#J.v_2.\text{Name}^{B'} \neq \#J.\text{Name}^B$ but it is a legal update that modifies the value of the attribute.

Note an essential difference between the two parallel updates $v_1, v_2 : B \to B'$ specified in the figure. Update $v_1$ says that John’s name was changed to Jon (e.g., by fixing a typo), and the experience data for Mary were also corrected (either because of a typo or, e.g., because the department started to use a new ML method for which Mary has a longer experience). Update $v_2$ specifies the same story for John but a new story for Mary: it says that Mary@B left the IT-view and Mary@B’ is a new employee in one of IT-departments.

**Update propagation and update policies.** The updated view $B'$ is inconsistent with the source $S$ and the latter is to be updated accordingly — we say that update $v$ is to be propagated (put back) to $A$. Propagation of $v_1$ is easy: we just update accordingly the values of the corresponding attributes according to update $u_1 : A \to A_1'$ specified in the figure inside the black block-arrow $u_1$. Importantly, propagation needs two pieces of data: the view update $v_1$ and the original state $A$ of the source as shown in the figure by two data-flow lines into the chevron $1:\text{put}$ denoting invocation of the backward propagation operation $\text{put}$ (read “put view update back to the source”). The quadruple $1 = (v_1, A, u_1, A')$ is an instance of operation $\text{put}$, hence the notation $1:\text{put}$ (borrowed from the UML). Note that the updated source model $A'$ is actually derivable from $u_1$ as resp. This graph freely generates a category (just add four identity arrows) that we denote by $S_A$ again. We assume that a general model of such a schema is a functor $X : S_A \to \text{Rel}$ that maps arrows to relations. If we need some of these relations to be functions, we label the arrows in the schema with a special constraint symbol, say, [fun], so that schema becomes a generalized sketch in the sense of Makkai (see [10-11]). In $S_A$, all three arrows are labelled by [fun] so that a legal model must map them to functions. For example, model $A$ in the figure is given by functor $A : S_A \to \text{Rel}$ with the following values: $\text{OID}^A = \{\#A, \#J, \#M\}$, sets $\text{String}^A$ and $\text{Integer}^A$ actually do not depend on $A$—they are the predefined sets of strings and integers resp., and $\text{Name}^A(\#A) = \text{Ann}, \text{Name}^A(\#J) = \text{John}, \text{Expr}^A(\#A) = 10$, etc.
its target, but we included it explicitly into put’s output to make the meaning of the figure more immediate.

Propagation of update $v_2$ is more challenging: Mary can disappear from the IT-view because a) she quit the company, b) she transitioned to a non-IT department, and c) the view definition has changed, e.g., the view now only shows employee with experience more than 5 years (and for more complex views, the number of possibilities is much bigger). Choosing between these possibilities is often called choosing an (update) policy. We will consider the case of changing the view (conceptually, the most radical one) in Sect. 3 and below discuss policies a) and b).

For policy a) (further referred to as quitting and briefly denoted by qt), the result of update propagation is shown in the figure with green colour: notice the update (block) arrow $u_{qt}^{2}$ and its result, model $A_1^{qt}$, produced by invoking operation $\text{put}^{qt}$. Note that while we know the new employee Mary works in one of IT departments, we do not know in which one. This is specified with a special value ? (a.k.a. labelled null in the database parlance).

For policy b) (further referred to as transition and denoted tr), the result of update propagation is shown in the figure with orange colour: notice update arrow $u_{tr}^{2}$ and its result, model $A_1^{tr}$ produced by $\text{put}^{tr}$. Mary #M is the old employee who transitioned to a new non-IT department, for which her expertise is unknown. Mary #M’ is the new employee in one of IT-departments (recall that the set of departments is not exhausted by those appearing in a particular state $A \in A$). There are also updates whose backward propagation is uniquely defined and does not need a policy, e.g., update $v_1$ is such.

An important property of update propagations we considered (ignore the blue propagation in the figure that shows policy c)) is that they restore consistency:
2 Background: Update propagation, policies, and delta lenses

the view of the updated source equals to the updated view initiated the update: 
\( \text{get}^0(A') = B' \). Moreover, this equality extends for update arrows:  
\( \text{get}(u_i) = v_i, \)  
where \( i = 1, 2 \), where \( \text{get} \) is an extension of the view mapping \( \text{get} \) for update arrows.  
Such extensions can be derived from view definitions if the latter are determined 
by so called monotonic queries (which encompass a wide class of practically 
useful queries including Select-Project-Join queries); for views defined by non-
monotonic queries, in order to obtain \( \text{get} \)’s action on source updates \( u: A \rightarrow A' \), a suitable policy is to be added to the view definition (see [12,13,6] for 
a discussion). Moreover, normally \( \text{get} \) preserves identity updates, \( \text{get}(\text{id}_A) = \text{id}_{\text{get}(A)}, \)  
and update composition: for any \( u: A \rightarrow A' \) and \( u': A' \rightarrow A'' \), equality 
\( \text{get}(u; u') = \text{get}(u) ; \text{get}(u') \) holds.

Our discussion of the example can be summarized in algebraic terms as fol-

\textbf{Definition 1 (Delta Lenses ([6])} Let \( A, B \) be two categories. An \((\text{asym-
metric delta}) \) lens from \( A \) to \( B \) is a pair \( \ell = (\text{get}, \text{put}) \), where \( \text{get}: A \rightarrow B \) is a 
functor and \( \text{put} \) is a family of operations \( \text{put}_A: A(A, \_ ) \leftarrow B(A, \_ ) \) indexed 
by objects of \( A \), \( A \in |A| \). Given \( A \), operation \( \text{put}_A \) maps any arrow \( v: B \rightarrow B' \) 
where \( A, \_ = B \) to an arrow \( u: A \rightarrow A' \). We will write a lens as an arrow 
\( \ell: A \rightarrow B \) going in the direction of \( \text{get} \).

A lens is called \textit{well-behaved (wb)} if the following two equational laws hold:

\begin{align*}
\text{(Stability)} & \quad \text{id}_A = \text{put} \circ (\text{id}_A, \text{get}) \text{ for all } A \in |A| \\text{(Putget)} & \quad (\text{put}_A.v).\text{get} = v \text{ for all } A \in |A| \text{ and all } v \in B(B, \_ )
\end{align*}

Note that family \( \text{put} \) corresponds to a chosen update policy, e.g., in terms 
of the example, we have a family \( \text{put}^1_A: A(A, \_ ) \leftarrow B(A, \_ ) \) and a family 
\( \text{put}^2_A: A(A, \_ ) \leftarrow B(A, \_ ) \), over the same view functor \( \text{get} \). That is, our 
two policies determine two lenses \( \ell^1 = (\text{get}, \text{put}^1) \) and \( \ell^2 = (\text{get}, \text{put}^2) \) over the 
same get.

Asymmetric lenses are sequentially associatively composable. Having two lenses \( \ell_1: A \rightarrow B \) and \( \ell_2: B \rightarrow C \), we build a lens \( \ell = (\text{get}, \text{put}): A \rightarrow C \) with \( \text{get} = \text{get}_1 \circ \text{get}_2 \) and \( \text{put}_A \) being the family defined by composition 
as shown in the inset figure: for \( A \in |A| \) and \( w: A \rightarrow C' \), \( \text{put}_A.w = \text{put}_1.A, \text{put}_2.B, w \). The identity lens is given by identity mappings, and 
we thus have a category \( \text{aLens} \) of asymmetric delta lenses ([6,5]). It’s easy to see that sequential 
composition preserves well-behavedness; we thus have an embedding \( \text{aLens}_{\text{wb}} \subset \text{aLens} \).
2.3 Functoriality of update policies

The notion of an update policy transcends individual lenses. Fig. 4 extends the example in Fig. 2 with a new model space $C$ whose schema consists of the only attribute $Name$, and a view of the IT-view, in which only employees of the ML department are to be shown. Thus, we now have two functors, get1: $A \to B$ and get2: $B \to C$, and their composition $Get: A \to C$ (referred to as the long get). The top part of Fig. 4 shows how it works for the source model $A$ considered above.

Each of the two policies, policy $qt$ and policy $tr$, in which person’s disappearance form the view are interpreted, resp., as quitting the company and transitioning to a department not covered by the view, is applicable to the new view mappings get2 and Get, thus giving us six lenses shown in the schema Fig. 5 with solid arrows (two lenses $L^{pol}$, $pol \in \{qt, tr\}$ are obtained by applying policy $pol$ to the functor $Get$; we will call these lenses long). In addition, we can compose lenses as shown in the schema, which gives us two more lenses shown with dashed arrows (of course, we can also compose lenses of different colours ($\ell_1^{ qt}$ with $\ell_2^{ tr}$ and $\ell_1^{ tr}$ with $\ell_2^{ qt}$) but we do not need them). Now an important question is how long and composed lenses are related, i.e., whether $L^{pol}$ and $\ell_1^{ pol}$, $\ell_2^{ pol}$ are equal (perhaps up to some equivalence) or different?

Fig. 4 demonstrates how the mechanisms work with a simple example. We begin with an update $w$ of the view $C$ that says that Mary left the ML department, and a new Mary was hired for ML. Policy $qt$ interprets Mary’s disappearance as quitting the company, and hence this Mary doesn’t appear in view $B'$ produced by $put_2^{qt}$ nor in view $A'qt_{12}$ produced by $put_1^{qt}$, and updates $v^{qt}$ and $u^{qt}_{12}$ are written accordingly. Obviously, Mary also does not appear in view $A'qt$ produced by the long lens’s $Put^{qt}$. Thus, $put_1^{qt}a, put_2^{qt}a, w = Put^{qt}_A, w$, and it is easy to un-
3 Learning for Bx

3.1 Bx does need learning

Enriching delta lenses with learning capabilities has a clear practical sense for Bx. Having a lens \((\text{get}, \text{put})\): \(A \rightarrow B\) and inconsistency between \(A\) and \(B\), i.e., \(A.\text{get} \neq B\), the learning idea extends the notion of the search space and allows us to update the transformation itself so that the final consistency is achieved for a new transformation \(\text{get}' : A.\text{get}' = B\). For example, in the case shown in Fig. 2 disappearance of Mary \#M in the updated view \(B'\) can be caused by changing the view definition, which now requires to show only those employee whose experience is more than 5 years and hence Mary \#M is not an element of the view, whereas Mary \#M' is a new IT-employee whose experience satisfies the new definition. Then the update \(v_2\) can be propagated as shown in the bottom right corner of Fig. 2. To manage the extended search possibilities, it makes sense to parameterize the space of transformations as a family of mappings \(\text{get}_p : A \rightarrow B\) indexed over some parameter space \(p \in P\). For example, we may define the IT-departments view to be parameterized by the experience of employees shown in the view (including any experience as a special parameter value). Then we have two interrelated propagation operations that map an update \(B \hookrightarrow B'\) to a parameter update \(p \hookrightarrow p'\) and a source update \(A \hookrightarrow A'\) (called, resp., an update and a request in \([15]\)). Thus, the extended search space allows for new update policies that look for updating the parameter as an update propagation possibility.

Note that all transformations \(\text{get}_p, p \in P\) are elements of the same lens and operations \(\text{put}\) are not indexed by \(p\). Formally, this is exactly the ML setting.
considered above up to the currying/uncurring equivalence. The possibility to update the transformation appears to be very natural in at least two important Bx scenarios: model transformation design (cf. [10]) and evolution [?], which appear in MDE so often that perhaps we should be more surprised by why the links between learning and Bx were not discovered much earlier.

3.2 Categorical vs. codiscrete learning

If the parameter space is a set, then the search procedure can “jump” from any parameter value \( p \) to any other value \( p' \). Such freedom may not be always desirable. Suppose, e.g., that our IT-departments view is designed for evaluating available workforce for a new project, and we need to find a reasonable tuple of values \( p = (p_1, p_2, \ldots, p_n) \) (where \( p_i \) are different parameters such as experience, salary, willingness to relocate, etc) to form an integral threshold for including employees into the view. In certain contexts, the space of such \( p \)'s can be usefully supplied with a partial order \( \leq \) so that when the search procedure is looking for a better parameter \( p' \) to replace \( p \), it should only look for values \( p' \) not less than \( p \) wrt. the order. Then the parameter space \( \mathcal{P} \) is a posetal category, and a parameter update is an arrow in this category. For another example, suppose that the set \( \mathcal{P} \) of departments forming the view is itself a parameter that can be changed. Then an update from \( \mathcal{P} \) to \( \mathcal{P}' \) has a relational structure as discussed above, i.e., \( e: \mathcal{P} \to \mathcal{P}' \) is a relation (span) \( e \subseteq \mathcal{P} \times \mathcal{P}' \) specifying which of \( \mathcal{P} \)'s departments are kept in \( \mathcal{P}' \) and thus specifying which departments disappeared from the view and which are freshly added. This is a general phenomenon: as soon as parameters are structures (sets of objects or graphs of objects and attributes), a parameter change becomes a structured delta and the space of parameters gives rise to a category \( \mathcal{P} \). The search/propagation procedure returns an arrow \( e: \mathcal{P} \to \mathcal{P}' \) in this category, which updates the parameter value from \( p \) to \( p' \). Then, as our discussion in Sect. shows, a real update of the system is a pair of deltas \( (u: \mathcal{A} \to \mathcal{A}', e: \mathcal{P} \to \mathcal{P}') \) rather than a pair of pairs \( ((\mathcal{A}, \mathcal{A}'), (p, p')) \). Hence, a general model of supervised learning should assume \( \mathcal{P} \) to be a category (and we say that learning is categorical). The case of the parameter space being a set is captured by considering a codiscrete category \( \mathcal{P} \) whose only arrows are pairs of its objects we call such learning codiscrete.

As model spaces are themselves categories, the entire search space is a product of categories, \( \mathcal{P} \times \mathcal{A} \) (or even \( \mathcal{P} \times \mathcal{A} \times \mathcal{B} \) if we consider amendments), and thus codiscreteness may “affect” only \( \mathcal{P} \) or only \( \mathcal{A} \) or both. For example, for learners described in [1], spaces \( \mathcal{A}, \mathcal{B}, \mathcal{P} \) are sets, \( \text{get} \) is a parameterized function, and \( \text{put} \) consists of two families of discrete operations described in [1]; \( \text{put}^{\text{red}} \) updates the parameter and \( \text{put}^{\text{req}} \) updates the source value thus making a request to the previous layer. In general, a learning lens from a model space (category) \( \mathcal{A} \) to model space \( \mathcal{B} \) is a pair of operations \( (\text{get}, \text{put}) : \mathcal{A} \xrightarrow{\mathcal{P}} \mathcal{B} \) where \( \text{get} : \mathcal{P} \times \mathcal{A} \to \mathcal{B} \) is a functor considered (via Currying) as a family of functors \( \text{get} : \mathcal{P} \to \mathcal{B}^\mathcal{A} \), and \( \text{put} \) is a family of operations providing some sort of an inverse map for functor \( \text{get} \). (Note that the case of ordinary lenses with no learning is captured by considering category \( \mathcal{P} \) to be terminal, i.e., consisting of one object and one arrow being its identity.)

Figure 6 shows a discrete two-dimensional plane with each axis having three points: a space is a singleton, a set, a category encoded by coordinates 0,1,2 resp. Each of the points \( x_{ij} \) is then the location of a corresponding category of (asym-
3 Learning for Bx

Fig. 6: The universe of categories of learning delta lenses

metric) learning (delta) lenses (we show the names of the categories of lenses without amendments). Category \( I \) is the terminal category whose only arrow is the identity lens \((id_1, id_1) : 1 \to 1\) propagating from the terminal category to itself. (Of course, \( I \cong 1 \).) Label \( * \) refers to the codiscrete specialization of the construct being labelled: \( L^* \) means codiscrete learning (i.e., the parameter space \( P \) is a set considered as a codiscrete category) and \( aLens^* \) refers to codiscrete model spaces. We have two semi-categorized species of learning lenses: categorical learners at point \((1,2)\) and codiscretely learning delta lenses at \((2,1)\). The category of learning delta lenses (with amendments or without it) defined in the present paper (see Sections 4 and 5) is at point \((2,2)\).

3.3 Functoriality of learning algorithms in the \textit{delta} lens setting

Fig. 7: Functoriality of learning algorithms and policies

A major message of [1]—compositionality of learning algorithms— is formalized as having an sm-functor \([1]\) described on p. [1]. This functor is actually equipped with a constraint (obvious but implicit in [1]) explicated in diagram Fig. 7(a), whose nodes are sm-categories and arrows are sm-functors (further we will omit the prefix sm). Objects of \( pSet \) are all small sets and arrows are all
(equivalence classes of) parametrized functions (p-functions, see Sect. A for details). Functor $-\mathbb{R}$ embeds (non-fully) Para into $\mathbf{pSet}$ and forgets the Euclidean structure, hence the notation $-\mathbb{R}$ “minus structure” (we will use this notational rule further on). Similarly, if a functor adds a structural component $X$, we denote it by a $+X$ symbol; in this notational framework, functor $L_{e,\text{err}}$ would be denoted as shown below the arrow in diagram (a); the functor takes a p-function and adds to it a family of inverse/back-propagation operations $\text{put}^\ast$, whose double-star superscript refers to codiscrete learning over codiscrete model spaces. In its turn, functor $-\text{put}^\ast$ takes a leaner and forgets its $\text{put}^\ast$-components, thus arriving at $\mathbf{pSet}$ so that the triangle commutes.

Learning delta lenses is a categorification of learners, in which all spaces are categories, gets are functors, and families $\text{put}^{\text{upd}}, \text{put}^{\text{req}}$ are functorial operations. Lenses with amendments add one more family of functorial operations $\text{put}^{\text{self}}$, which reflectively update deltas on the $\mathbf{B}$ side. This categorification is specified in diagram Fig. 7(b) (we show the version for lenses with amendments). Euclidean spaces are replaced by model spaces definable by a model transformation language $\text{mtf}$, differentiable p-functions by $\text{mtf}$-definable p-functors, and discrete operations $\text{put}^\ast$’s by functorial operations $\text{put}$ s. Functor $L_{\text{mtf}}$ from [2] p. 2 is synonymously denoted by $+_\text{mtf} \text{put}$. We thus have a structurally similar diagram over $\mathbf{pCat}$—the foundation of the ala-lens building. In the next section, we consider the path from (a) to (b) in more detail.

4 Asymmetric Learning Lenses with Amendments

**Definition 2 (ala-lenses)** Let $S$ and $T$ be categories. An *ala-lens* from $S$ (the *source* of the lens) to $T$ (the *target*) is a pair $\ell = (\text{get}, \text{put})$ whose first component is a p-functor $\text{get}: S \rightarrow P T$ and the second component is a triple of (families of) operations $\text{put} = (\text{put}^{\text{upd}}, \text{put}^{\text{req}}, \text{put}^{\text{self}})$ indexed by pairs $p \in |P|$, $S \in |S|$; arities of the operations are specified below after we introduce some notation. Names req and upd are chosen to match the terminology in [1].

Categories $S$, $T$ are called *model spaces*, their objects are *models* and their arrows are *(model) updates* or *deltas*. Objects of $P$ are *parameters* and are denoted by small letters $p, p’, \ldots$ rather than capital ones to avoid confusion with $[1]$, in which capital $P$ is used for the entire parameter set. Arrows of $P$ are called *parameter deltas*. For a parameter $p \in |P|$, we write $\text{get}_p$ for the functor $\text{get}(p): S \rightarrow T$ (read “get $T$-views of $S$”), and if $S \in |S|$ is a source model, its $\text{get}_p$-view is denoted by $\text{get}_p(S)$ or $S.\text{get}_p$ or even $S_p$ (so that $\text{get}_p$ becomes yet another notation for functor $\text{get}_p$). Given a parameter delta $e: p \rightarrow p’$ and a source model $S \in |S|$, the model delta $\text{get}(e): \text{get}_p(S) \rightarrow \text{get}_{p’}(S)$ will be denoted by $\text{get}_e(S)$ or $e_S$ (rather than $S_e$ as we would like to keep capital letters for objects).

Since $\text{get}_e$ is a natural transformation, for any delta $u: S \rightarrow S’$, we have a commutative square $e_S; u_{p’} = u_p; e_{S’}$. We will denote the diagonal of this square by $u.\text{get}_e$ or $u_e: S_p \rightarrow S’_{p’}$. Thus, we use notation

$$
(3) \quad S_p \overset{\text{def}}{=} S.\text{get}_p \overset{\text{def}}{=} \text{get}_p(S) \overset{\text{def}}{=} \text{get}(p)(S) \\
u_e \overset{\text{def}}{=} u.\text{get}_e \overset{\text{def}}{=} \text{get}_e(u) \overset{\text{def}}{=} \text{get}(e)(u) \overset{\text{def}}{=} e_S; u_{p’} \overset{\text{nat}}{=} u_p; e_{S’} : S_p \rightarrow S’_{p’}
$$

Now we describe operations $\text{put}$. They all have the same indexing set $|P| \times |S|$, and the same domain: for any index $p, S$ and any model delta $v: S_p \rightarrow T’$ in $T$,
4 Asymmetric Learning Lenses with Amendments

the value $\text{put}_{p,S}^x(p, S)$, $x \in \{\text{req, upd, self}\}$ is defined and unique:

\[
\begin{align*}
\text{put}_{p,S}^{\text{upd}} &: p \to p' \quad \text{is a parameter delta from } p, \\
\text{put}_{p,S}^{\text{req}} &: S \to S' \quad \text{is a model delta from } S, \\
\text{put}_{p,S}^{\text{self}} &: T' \to S_{p'} \quad \text{is a model delta from } T' \\
\end{align*}
\]

called the amendment and denoted by $v^\oplus$.

Note that the definition of $\text{put}_{p,S}^{\text{self}}$ involves an equational dependency between all three operations.

We will write a lens as an arrow $\ell = (\text{get}, \text{put}): S \xrightarrow{P} T$.

A lens is called (twice) codiscrete if categories $S$, $T$, $P$ are codiscrete and thus $\text{get}: S \xrightarrow{P} T$ is a parameterized function. If only $P$ is codiscrete, we call $\ell$ a codiscretely learning delta lens, while if only model spaces are codiscrete, we call $\ell$ a categorically learning codiscrete lens.

Diagram in Fig. 8 shows how these operations are interrelated. The upper part shows an arrow $e: p \to p'$ in category $P$ and two corresponding functors from $S$ to $T$. The lower part is to be seen as a 3D-prism with visible front face $SS_p, S_p', S'$ and visible upper face $SS_p, S_{p'}$, the bottom and two back faces are invisible and the corresponding arrows are dashed. The prism denotes an algebraic term: given elements are shown with black fill and white font while derived elements are blue (recalls being mechanically computed) and blank (double-body arrows are considered as “blank”). The two pairs of arrows originating from $S$
and $S'$ are not blank because they denote pairs of nodes (the UML says links) rather than mappings/deltas between nodes.

Equational definitions of deltas $e, u, v^\delta$ are written up in three callouts near them. Four derived deltas forming the right back face of the prism are two vertical rather than mappings/deltas between nodes.

and $S'$ hold for all values of three bound variables: $p \in |P|$, $S \in |S|$ and $v: S_p \to T'$ (as in the laws below). The following two laws are crucial:

1. **Stability** if $v = \text{id}_{S_p}$ then all three propagated updates $e, u, v^\delta$ are identities:
   
   $$\text{put}_{p,S}(\text{id}_{S_p}) = \text{id}_p, \quad \text{put}_{p,S}^\text{req}(\text{id}_{S_p}) = \text{id}_S, \quad \text{put}_{p,S}^\text{self}(\text{id}_{S_p}) = \text{id}_{S_p}$$

2. **Putget** $(\text{put}_{p,S}^\text{req}, v).\text{get}_e = v; v^\delta$ where $e = \text{put}_{p,S}^\text{self}(v)$ and $v^\delta = \text{put}_{p,S}^\text{self}(v)$

A lens is **stable** if it satisfies Stability, and an $SPg$-lens if it satisfies Stability and Putget. Following the lens terminological traditions, in this paper we will also call an $SPg$ lens **well-behaved** (wb). Note however that wb-lenses defined in [8] are stronger than $SPg$.

**Remark 1.** a) Stability says that the lens does nothing if nothing happens on the target side (no trigger–no action, hence, the name of the law)

b) Putget requires the goal of update propagation to be achieved after the propagation act is finished. In the codiscrete setting, it is a notational trick rather than a law. We begin with delta $v = (S_p, T')$, which operation $\text{put}_{p,S}^\text{req}$ propagates to delta $u = (S, S')$ and function $\text{get}_e$ sends it to delta $(S_p, S'_p)$. Also, $\text{put}_{p,S}^\text{self}$ gives us an amendment delta $v^\delta = (T', S'_p)$ and $v; v^\delta = (S_p, S'_p) = \text{get}_e(S, S')$ so that Putget always holds. But in the general delta setting, there are many possible deltas between models $T'$ and $S'_p$, whereas the Putget law states the existence of some special delta $v^\delta; T' \to S'_p$, such that equation (Putget) holds.

c) Besides Stability and Putget, the MDE context for ala-lenses suggests other laws; some of them are considered in [8]

**Example 1 (Identity lenses).** Any category $A$ gives rise to an ala-lens $\text{id}_A$ with the following components. The source and target spaces are equal to $A$, and the parameter space is 1. Functor $\text{get}$ is the identity functor and all puts are identities. Obviously, this lens is $SPg$ (wb).

**Example 2 (Iso-lenses).** Let $\nu: A \to B$ be an isomorphism between model spaces. It gives rise to a wb ala-lens $\ell(\nu): A \to B$ with $P^\ell(\nu) = 1$ as follows. Given any $A$ in $A$ and $v: \nu(A) \to B'$ in $B$, we define $\text{put}_{\nu,A}^\ell, \text{req}(v) = \nu^{-1}(v)$ and the two other put operations send $v$ to identities.

**Example 3 (Bx lenses).** Examples of wb aa-lenses modelling a Bx can be found in [8]; they all can be considered as ala-lenses with a trivial parameter space 1.

### 5 Sequential and parallel composition of ala-lenses, and symmetric monoidal category \textbf{aLaLens}

**Construction 1 (Sequential composition of ala-lenses)** Let $\kappa: A \to B$ and $\ell: B \to C$ be two ala-lenses with parameterized functors $\text{get}^\ell: P \to |A, B|$ and $\text{get}^\ell: Q \to |B, C|$ resp. Their **composition** is the following ala-lens $\kappa; \ell$. 

5  Sequential and parallel composition of ala-lenses, and symmetric monoidal category aLaLens

Its parameter space is the product $P \times Q$, and the get-family is defined as follows. For any pair of parameters $(p, q)$ (we will write $pq$), $\text{get}^{\ell}_{pq} = \text{get}^{\ell}_{p} \cdot \text{get}^{\ell}_{q}$. $A \rightarrow C$. Given a pair of parameter deltas, $e: p \rightarrow p'$ in $P$ and $h: q \rightarrow q'$ in $Q$, their $\text{get}^{\ell\text{-image}}$ is the Godement product $\ast$ of natural transformations, $\text{get}^{\ell\ast}(e h) = \text{get}^{\ell}(e) \ast \text{get}^{\ell}(h)$ (we will also write $\text{get}^{\ell} || \text{get}^{\ell}$)

Now we define $\kappa;\ell$'s propagation operations $\text{puts}$. Let $(A, pq, A_{pq})$ with $A \in |A|$, $pq \in |P \times Q|$, $A.\text{get}_{pq}$.\text{get}_{pq} = A_{pq} \in |C|$ be a state of lens $\kappa;\ell$, and $w: A_{pq} \rightarrow C'$ is a target update as shown in Fig. 9. For the first propagation step, we run lens $\ell$ as shown in Fig. 9 with the blue colour for derived elements: this is just an instantiation of the pattern of Fig. 8 with the source object being $A_{p} = A.\text{get}_{p}$ and parameter $q$. The results are deltas

$$h = \text{put}^{\kappa;\ell,\text{upd}}_{q,A_{p}}(w): q \rightarrow q', \quad v = \text{put}^{\kappa;\ell,\text{req}}_{A_{p},A_{p}}(w): A_{p} \rightarrow B', \quad w^{\delta} = \text{put}^{\kappa;\ell,\text{self}}_{q,A_{p}}(w): C' \rightarrow B_{q'}.$$

Next we run lens $\kappa$ at state $(p, A)$ and the target update $v$ produced by lens $\ell$; it is yet another instantiation of pattern in Fig. 8 (this time with the green colour for derived elements), which produces three deltas

$$e = \text{put}^{\kappa;\text{upd}}_{p,A}(v): p \rightarrow p', \quad u = \text{put}^{\kappa;\text{req}}_{q,A_{p}}(v): A \rightarrow A', \quad v^{\delta} = \text{put}^{\kappa;\text{self}}_{p,A}(v): B' \rightarrow A_{p'}.$$

These data specify the green prism adjoint to the blue prism: the edge $v$ of the latter is the “first half” of the right back face diagonal $A_{p}A'_{p}$ of the former. In order to make an instance of the pattern in Fig. 8 for lens $\kappa;\ell$, we need to extend the blue-green diagram to a triangle prism by filling-in the corresponding “empty space”. These filling-in arrows are provided by functors $\text{get}^{\ell}$ and $\text{get}^{\ell}$ and shown in orange (where we have chosen one of the two equivalent ways of forming the Godement product – note two curve brown arrows). In this way we obtain yet another instantiation of the pattern in Fig. 9 denoted by $\kappa;\ell$:

$$\text{put}^{(\kappa;\ell),\text{upd}}_{A_{p},pq}(w) = (e, h), \quad \text{put}^{(\kappa;\ell),\text{req}}_{A_{p},pq}(w) = u, \quad \text{put}^{(\kappa;\ell),\text{self}}_{A_{p},pq}(w) = w^{\delta} = v^{\delta}_{q'}.$$
where \(v^0_t\) denotes \(v^0 \cdot \text{get}_t\). Thus, we built an ala-lens \(k; \ell\), which satisfies equation \(\text{Put} \cdot \text{get}_0\) by construction.

The following result is important for practical applications: it ensures that a composed synchronizer satisfies its requirements automatically as soon as its components do, which allows significant reducing of the integration testing (only connectivity is to be checked).

**Theorem 1 (Seq. composition and lens laws).** Given are ala-lenses \(k; A \rightarrow B\), \(\ell; B \rightarrow C\), and lens \(\ell; A \rightarrow C\) is their sequential composition as defined above in Constr. \(\square\). Then the lens \(k; \ell\) is SPg as soon as lenses \(k\) and \(\ell\) are such.

A proof can be found in Appendix \(\square\).

**Construction 2 (Parallel composition of ala-lenses)** Let \(\ell_i; A_i \rightarrow B_i, i = 1, 2\) be two ala-lenses with parameter spaces \(P_i\). The lens \(\ell_1 |\ell_2; A_1 \times A_2 \rightarrow B_1 \times B_2\) is defined as follows. Parameter space \(\ell_1 |\ell_2; P = P_1 \times P_2\). For any pair \((p_1|p_2) \in P_1 \times P_2\), define \(\text{get}_{\ell_1 |\ell_2}(p_1|p_2) = \text{get}_{\ell_1}(p_1) \times \text{get}_{\ell_2}(p_2)\) (we denote pairs of parameters by \(p_1|p_2\) rather than \(p_1 \otimes p_2\) as suggested by Def. ?? on p. ?? to shorten long formulas going beyond the page width). Further, for any pairs of models \(S_1|S_2 \in A_1 \times A_2\) and deltas \(v_1|v_2; (S_1|S_2)\cdot \text{get}_{\ell_1 |\ell_2} \rightarrow T_1|T_2\), we define componentwise

\[
e = \text{put}_{\ell_1 |\ell_2}(\ell_1 |\ell_2) |\ell_1 |\ell_2| (v_1|v_2); p_1|p_2 \rightarrow p_1'|p_2'
\]

by setting \(e = e_1|e_2\) where \(e_i = \text{put}_{\ell_i; S_i}(v_i), i = 1, 2\)

and similarly for \(\text{put}_{\ell_1 |\ell_2}(\ell_1 |\ell_2) |\ell_1 |\ell_2|\) and \(\text{put}_{\ell_1 |\ell_2}(\ell_1 |\ell_2) |\ell_1 |\ell_2|\) and \(\text{put}_{\ell_1 |\ell_2}(\ell_1 |\ell_2) |\ell_1 |\ell_2|\)

The following result is obvious but important (as any compositionality result—see the remark about integration testing above).

**Theorem 2 (Parallel composition and lens laws).** Lens \(\ell_1 |\ell_2\) is SPg as soon as lenses \(\ell_1\) and \(\ell_2\) are such.

Now our goal is to organize ala-lenses into an sm-category. To make seq. composition of ala-lenses associative, we need to consider them up to some equivalence (indeed, Cartesian product is not strictly associative).

**Definition 4 (Ala-lens Equivalence)** Two parallel ala-lenses \(\ell, \hat{\ell}; S \rightarrow T\) are called equivalent if their parameter spaces are isomorphic via a functor \(\nu; P \rightarrow \hat{P}\) such that for any \(S \in |S|, e; p \rightarrow p' \in P\) and \(v; (S \cdot \text{get}_p) \rightarrow T'\) the following holds:

\[
S \cdot \text{get}_e = S \cdot \text{get}_{\nu(e)}, \nu((\text{put}_{\nu(p), S}^{\text{update}}(v))) = \text{put}_{\nu(p), S}(v), \text{and } \nu(x) = \text{put}_{\nu(p), S}^x(v) \text{ for } x \in \{\text{req, self}\}
\]

**Remark 2.** It would be more categorical to require delta isomorphisms (i.e., commutative squares whose horizontal edges are isomorphisms) rather than equalities as above. However, model spaces appearing in Bx-practice are skeletal categories (and even stronger than skeletal in the sense that all isos, including iso loops, are identities), for which isos become equalities so that the generality would degenerate into equality anyway.

**Lemma 1.** Operations of lens’ sequential and parallel composition are compatible with lens’ equivalence. Hence, these operations are well-defined for equivalence classes.
Below we will identify lenses with their equivalence classes by default.

**Theorem 3 (Ala-lenses form an sm-category).** Operations of sequential and parallel composition of ala-lenses defined above give rise to an sm-category $\text{alaLens}$, whose objects are model spaces (= categories) and arrows are (equivalence classes of) ala-lenses.

**Proof.** It is easy to check that identity lenses $\text{id}_A$ defined in Example 1 are the units of the sequential lens composition defined above. The proof of associativity is rather involved and is placed into Appendix C. Thus, $\text{alaLens}$ is a category.

Next we define a monoidal structure over this category. The monoidal product of objects is Cartesian product of categories (skeletality is preserved), and the monoidal product of arrows is lens’ parallel composition defined above. The monoidal unit is the terminal category $\text{1}$. Associators, left and right unitors, and braiding are iso-lenses generated by the respective isomorphism functors (Example 2). Moreover, it is easy to see that the iso-lens construction from Example 2 is actually a functor $\text{isolens: Cat}_{\text{iso}} \to \text{alaLens}$. Then as a) $\text{Cat}$ is symmetric monoidal and fulfils all necessary monoidal equations, and b) $\text{isolens}$ is a functor, these equations hold for the ala-lens images of $\text{Cat}_{\text{iso}}$-arrows, and $\text{alaLens}$ is symmetric monoidal too (cf. a similar proof in [1] with $(\text{Set}, \times)$ instead of $(\text{Cat}, \times)$).

### 6 Related work

As suggested by Fig. 1, immediate related work should be found in areas located at points $(0,1)$ (codiscrete learning lenses) and $(1,0)$ (delta lenses) of the plane. For the point $(0,1)$, the paper [1] by Fong, Spivak and Tuyéras is fundamental: they defined the notion of a codiscrete learning lens (called a learner), proved a fundamental results about sm-functoriality (1) of the gradient descent approach to ML, and laid a foundation for the compositional approach to change propagation with learning. One follow-up of that work is paper [2] by Fong and Johnson, in which they build an sm-functor $\text{Learn} \to \text{sLens}$ which maps learners to so called symmetric lenses. That paper is probably the first one where the terms ‘lens’ and ‘learner’ are met, but an initial observation that a learner whose parameter set is a singleton is actually a lens is due to Jules Hedges, see [2]. There are conceptual and technical distinctions between [2] and the present paper. On the conceptual level, by encoding learners as symmetric lenses, they “hide” learning inside the lens framework and make it a technical rather than conceptual idea. In contrast, we consider parameterization and supervised learning as a fundamental idea and a first-class citizen for the lens framework, which grants creation of a new species of lenses. Moreover, while an ordinary lens is a way to invert a functor, a learning lens is a way to invert a parameterized functor so that learning lenses appear as an extension of the parameterization idea from functors to lenses. (This approach can probably be specified formally by treating parameterization as a suitably defined functorial construction.) Besides technical advantages (working with asymmetric lenses is simpler), the learning lens model seems more adequate to the fact that we deal with functions rather than relations. On the technical level, the lens framework we develop in the paper is much more general than in [2]: we categorificated both the parameter space and model spaces, and we work with lenses with amendment.
References

As for the delta lens roots (the point (1,0) in the figure), delta lenses were motivated and formally defined in [6] (the asymmetric case) and [7] (the symmetric one). Categorical foundations for the delta lens theory were developed by Johnson and Rosebrugh in a series of papers, see [14] for references. The lax approach to delta lens laws was proposed by the author with coauthors, see [5] for a survey and references. Specifically, the notions of a delta lens with amendments (in both asymmetric and symmetric variants) was defined in [8], and several composition results were proved. Another extensive body of work within the delta-based area is modelling and implementing model transformations with triple-graph grammars (TGG) [17,18]. TGG provides an implementation framework for delta lenses as is shown and discussed in [19,20,21], and thus inevitably consider change propagation on a much more concrete level than lenses. The author is not aware of any work of discussing functoriality of update policies developed within the TGG framework.

The present paper is probably the first one at the intersection (1,1) of the plane. The preliminary results have recently been reported at ACT’19 in Oxford to a representative lens community, and no references besides [1], [2] mentioned above were provided.

7 Conclusion

The perspective on Bx presented in the paper is an example of the fruitful interaction between two domains—ML and Bx. In order to be ported to BxMDE, the compositional approach to ML developed in [1] is to be categorified as shown in Fig. 7 (see p. 11). This opens a whole new program for Bx: checking that currently defined Bx languages and tools are compositional in the sense of functoriality of the graph morphism $L_{mtl}: pGet_{mtl} \rightarrow aLaLens_{ab}$ (see p.2). The first step of this program is performed in the paper: the category of (well-behaved) ala-lenses is defined and equipped with a symmetric monoidal structure. This allows us to formulate an important requirement to an update policy (and a tool implementing it): the policy is to be sm-functorial as prescribed by the diagram Fig. 7(b). The second step of the program is actually a family of steps: checking sm-functoriality for a given policy/tool, e.g., for TGG-based eMoflon [18] or for Haskell-based GROUNDTRAM [22]. Surprisingly, but this important requirement has been missing from the agenda of the Bx- community, e.g., an extensive endeavour of developing an effective benchmark for Bx-tools [23] does not discuss it.

Perhaps even more exiting is the possibility of using the Bx perspective in ML. Although learners are inherently codiscrete, but a codiscrete lens may actually be built on top of a (hidden internal) delta lens as explained in [6]. Most probably, this is the case for learners too, and then ML may potentially benefit from the delta lens framework developed in Bx. The interaction ML $\rightarrow$ Bx that we have explored in the paper is to be bidirectional!

References

References


A Category of parameterized functors \( \text{pCat} \) is symmetric monoidal

APPENDICES

A Category of parameterized functors \( \text{pCat} \) is symmetric monoidal

Category \( \text{pCat} \) has all small categories as objects. \( \text{pCat} \)-arrows \( A \rightarrow B \) are parameterized functors (\( \text{p-functors} \)) i.e., functors \( f: \mathcal{P} \rightarrow [A, B] \) with \( \mathcal{P} \) a small category of parameters and \( [A, B] \) the category of functors from \( A \) to \( B \) and their natural transformations. For an object \( p \) and an arrow \( e: p \rightarrow p' \) in \( \mathcal{P} \), we write \( f_p \) for the functor \( f(p): A \rightarrow B \) and \( f_e \) for the natural transformation \( f(e): f_p \Rightarrow f_{p'} \). We will write \( \text{p-functors} \) as labelled arrows \( A \overset{f}{\rightarrow} B \) and sometimes omit label \( \mathcal{P} \) over the arrow. Sequential composition of \( f: A \overset{p}{\rightarrow} B \) and \( g: B \overset{q}{\rightarrow} C \) is \( f.g: A \overset{p}{\rightarrow} B \overset{q}{\rightarrow} C \). We will write \( \text{p-functors} \) up to an equivalence of their parameter spaces. Two \( \text{p-functors} \) \( f \) and \( g \) are equivalent if there is an isomorphism \( \alpha: [A, B] \rightarrow [A, B] \) such that two parallel functors \( f: \mathcal{P} \rightarrow [A, B] \) and \( \alpha; f: [A, B] \rightarrow [A, B] \) are naturally isomorphic; then we write \( f \approx_\alpha g \). It’s easy to see that \( \text{p-functors} \) \( \text{Id} \) are units of the sequential composition. To ensure associativity we need to consider \( \text{p-functors} \) up to an equivalence of their parameter spaces. Two parallel \( \text{p-functors} \) \( f \) and \( \hat{f} \) are equivalent if there is an isomorphism \( \alpha: \mathcal{P} \rightarrow \mathcal{P} \) such that two parallel functors \( f: \mathcal{P} \rightarrow [A, B] \) and \( \alpha; f: [A, B] \rightarrow [A, B] \) are naturally isomorphic; then we write \( f \approx_\alpha \hat{f} \). It’s easy to see that if \( f \approx_\alpha \hat{f} \), \( g \approx_\beta \hat{g} \), \( \alpha; f \approx_\alpha \hat{f} \), \( f; g \approx_\alpha \hat{f} \), \( g \approx_\beta \hat{g} \), \( f; g \approx_\beta \hat{f} \) is a strict associativity of the functor composition and strict associativity of the Godement product, we conclude that sequential composition of (equivalence classes of) \( \text{p-functors} \) is strictly associative. Hence, \( \text{pCat} \) is a category.

Our next goal is to supply it with a monoidal structure. We borrow the latter from the sm-category \( (\text{Cat}, \times) \), whose tensor is given by the product. There is an identical on objects (iio) embedding \( (\text{Cat}, \times) \hookrightarrow \text{pCat} \) that maps a functor \( f: A \rightarrow B \) to a \( \text{p-functor} \) \( f: A \overset{1}{\rightarrow} B \) whose parameter space is the singleton category \( 1 \). Moreover, as this embedding is a functor, the coherence equations for the associators and unitors that hold in \( (\text{Cat}, \times) \) hold in \( \text{pCat} \) as well (this proof idea is borrowed from [2]). In this way, \( \text{pCat} \) becomes an sm-category.

In a similar way, we define the sm-category \( \text{pSet} \) of small sets and parametrized functions between them — the codiscrete version of \( \text{pCat} \). The inset diagram shows how these categories are related: labels \([w]\) and \([f]\) say that embeddings are wide or full resp.
B  Sequential composition of ala-lenses and lens laws:

Proof of Theorem 1 on page 16

Proof. Stability of $\kappa; \ell$ is obvious. To prove $\text{Put}get$ for $\kappa; \ell$, we need to prove that $(\text{put}_{pq,A}^{\ell}; \text{req}^\ell,w).\text{get}_{pq,A}^{\ell; \text{req}} = w; w^{(k\ell); \alpha}$ for any $A \in |A|$, $p \in |A|$, $q \in |Q|$ and $w: A_p \rightarrow C'$. Let $\text{put}_{pq,A}^{\ell; \text{req}}$ be pair $(e, h)$ with some $e: p \rightarrow p'$ and $h: q \rightarrow q'$. We compute:

\begin{align}
(8) \quad (\text{put}_{pq,A}^{\ell; \text{req}}, w).\text{get}_{qh}^{\ell; \text{req}} &= (\text{put}_{pq,A}^{\ell; \text{req}}. \text{put}_{q,A_p}^{\ell; \text{req}}. w).\text{get}_{qh}^{\ell; \text{req}} \quad \text{by constr. of } \kappa; \ell \\
&= (v; v^{\alpha}).\text{get}_{h}^{\ell} \quad \text{Putget for } \ell \text{ (where } v = \text{put}_{q,A_p}^{\ell; \text{req}}. w) \\
&= (v; v^{\alpha}).\text{get}_{h}^{\ell} ; h^{\beta} \quad \text{def. of } \text{get}_{h}^{\ell} \\
&= v_q ; (v^{\beta}; h^{\beta}) \quad \text{functoriality of } \text{get}_{h}^{\ell} \& \text{assoc. of } ; \\
&= v_q ; (h^{\beta}; v^{\beta}) \quad \text{naturality of } \text{get}_{h}^{\ell} \\
&= (w; w^{\beta}) ; v^{\beta} \quad \text{Putget for } \ell \\
&= w; (w^{\beta}; v^{\beta}) \quad \text{associativity of } ; \\
&= w; w^{\beta} \quad \text{as the most convenient):}
\end{align}

\[ C \]

Sequential ala-lens composition is associative

Let $\kappa; A \rightarrow B$, $\ell; B \rightarrow C$, $\mu; C \rightarrow D$ be three consecutive lenses with parameter spaces $P$, $Q$, $R$ resp. We will denote their components by an upper script, e.g., $\text{get}_{P}^\ell$ or $\text{put}_{P,A}^{\ell; \text{req}}$, and lens composition by concatenation: $\kappa \ell$ is $\kappa; \ell$ etc; $\text{put}_{P,A}^{(k\ell); \text{req}}$ denotes $\text{put}_{(P,A)}^{(k\ell); \text{req}}$.

We need to prove $(k\ell)\mu = (k\mu)\ell$. We easily have associativity for the get part of the construction: $(P \times Q) \times R \cong P \times (Q \times R)$ (to be identified for equivalence classes), and $(\text{get}_{P}^{k}; \text{get}_{\ell}^{\mu}).\text{get}_{P}^{\ell} = (\text{get}_{P}^{\ell}; \text{get}_{\mu}^{\ell}).\text{get}_{P}^{\ell}$, which means that $\text{get}_{(P,Q)}^{(k\ell); \alpha} = \text{get}_{\mu}^{(k\ell); \alpha}$, where $p, q, r$ are parameters (objects) from $|P|$, $|Q|$, $|R|$ resp., and pairing is denoted by concatenation.

Associativity of puts is more involved. Suppose that we extended the diagram in Fig. 9 with lens $\mu$ data on the right, i.e., with a triangle prism, whose right face is a square $D_{pq}D_rD_pD_{\alpha}$ with diagonal $\omega^{\alpha}$: $D_{pq} \rightarrow D^{\alpha}$ where $r \in R$ is a parameter, $D_{pq} = \text{get}_{P}^{p}(C_{pq})$ and $\omega: D_{pq} \rightarrow D'$ is an arbitrary delta to be propagated to $P$ and $A$, and reflected with amendment $\omega^{\alpha} = \text{put}_{\mu}^{\alpha}$. Below we will omit parameter subindexes near $B$ and $C$.

We begin with term substitution in equations (5,7) in Constr. 1 which gives us equational definitions of all put operations (we use the function application notation $f.x$ as the most convenient):

\[ A \]

References

C  Sequential ala-lens composition is associative

Now we apply these definitions to the lens \((k\ell)\mu\) and substitute. Checking \(\text{put}^{(k\ell)\mu,\text{req}}\) is straightforward similarly to associativity of \(\text{gets}\), but we will present its inference to show how the notation works (recall that \(\omega; D_{pqr} \rightarrow D'\) is an arbitrary delta to be propagated).

\[
\begin{align*}
\text{put}^{(k\ell)\mu,\text{req}} \cdot \omega &= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \quad \text{by (9)} \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot (\text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega) \quad \text{by (9)} \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \omega \quad \text{by (9)} \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \omega \quad \text{by (9)}.
\end{align*}
\]

Computing of \(\text{put}^{(k\ell)\mu,\text{upd}}\) is more involved (below a pair \((x, y)\) will be denoted as either \(xy\) or \(x'y\) depending on the context).

\[
\begin{align*}
\text{put}^{(k\ell)\mu,\text{upd}}_{(pq)r,A} \cdot \omega &= \left(\text{put}^{(k\ell)\mu,\text{upd}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega: p|q \rightarrow p'|q'\right) \parallel \left(\text{put}^{(k\ell)\mu,\text{upd}}_{r,C} \cdot \omega: r \rightarrow r'\right) \quad \text{by (10)} \\
&= \left(\text{put}^{(k\ell)\mu,\text{upd}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \parallel \text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \right) \parallel \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \quad \text{by (10)} \quad \text{same} \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \omega \parallel \left(\text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \parallel \text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega \right) \quad \text{by assoc. of} \parallel \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \text{put}^{(k\ell)\mu,\text{req}}_{q,B} \cdot \omega \quad \text{by (9)} \parallel \text{by (10)} \\
&= \text{put}^{(k\ell)\mu,\text{req}}_{pq,A} \cdot \omega \quad \text{again by (10)}.
\end{align*}
\]

Associativity of \(\text{put}^{(k\ell)\mu,\text{self}}\) can be proved in a similar manner using associativity of \(\parallel\) (see (11)) rather than associativity of \(\cdot\) (see (10)) used above. Below \(w\) stands for \(\text{put}^{(k\ell)\mu,\text{req}}_{r,C} \cdot \omega\).

\[
\begin{align*}
\text{put}^{(k\ell)\mu,\text{self}}_{(pq)r,A} \cdot \omega &= \left(\text{put}^{(k\ell)\mu,\text{self}}_{r,C} \cdot \omega: \text{get}^{(k\ell)\mu,\text{self}}_{pq,A} \cdot \omega \right) \quad \text{by (11)} \\
&= \left(\text{put}^{(k\ell)\mu,\text{self}}_{r,C} \cdot \omega: \text{get}^{(k\ell)\mu,\text{self}}_{q,B} \cdot \omega \parallel \text{get}^{(k\ell)\mu,\text{self}}_{q,B} \cdot \text{get}^{(k\ell)\mu,\text{self}}_{r,C} \cdot \omega \right) \quad \text{by (11)} \\
&= \left(\text{put}^{(k\ell)\mu,\text{self}}_{r,C} \cdot \omega: \text{get}^{(k\ell)\mu,\text{self}}_{q,B} \cdot \text{get}^{(k\ell)\mu,\text{self}}_{r,C} \cdot \omega \right) \parallel \text{get}^{(k\ell)\mu,\text{self}}_{p,A} \cdot \text{get}^{(k\ell)\mu,\text{self}}_{q,B} \cdot \omega \quad \text{by def. of} \: w \\
&= \text{put}^{(k\ell)\mu,\text{self}}_{p(A)} \cdot \omega \quad \text{by def. of} \: w \quad \text{and} \quad \text{by (12)} \quad \text{applied twice} \quad \text{by (11)}.
\end{align*}
\]