Tiling the view update problem

Zinovy Diskin

Dec 7, 2009

Abstract

The paper describes a novel algebraic framework for specifying backward translations of view updates. It is based on diagram operations and can be considered an essential generalization of *dynamic* views (known in the literature since the eighties). A distinctive feature of the framework is that both view and update mappings are first class citizens and explicitly occur in the arity shape of the translation operation. We show that dynamic views augmented with mappings give rise to double categories, and discuss the benefits of this formalism.

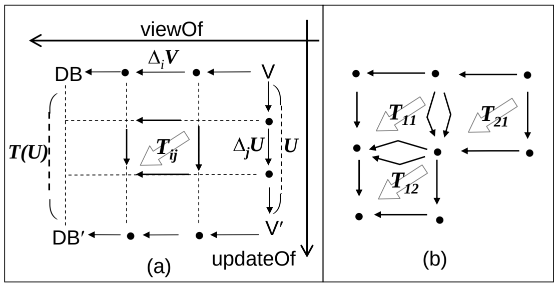


Figure 1: Tiling update translation

1 Introduction

The *view update problem* refers to the task of translating updates to a view to updates on the database. Such translations are *correct* if the view of the updated base equals to the updated view. Since the view abstracts away some information, many (or none) correct updates on the base may result in the given view update. Finding a unique backward translation is a difficult problem, which does not have universal solutions. Yet for special classes of queries and updates, reasonable translation procedures ensuring uniqueness (so called *update policies*) could be found. Design of update policies is a delicate business that needs a careful inspection of the syntactical (the base schema and the view definition) and the semantic (conditions on instances) sides of the problem [4, 12, 14].

To manage complexity, it is reasonable to split the task into simpler components. A complex view definition V is decomposed into a sequence of simple blocks $\Delta_i V$ (so that V becomes a view of a view of...), and similarly a complex update U is presented as a sequence of elementary deletions, insertions and modifications $\Delta_j U$. For every pair $(\Delta_i V, \Delta_j U)$ a correct translator T_{ij} is a priori designed and stored in a library. Then translation $T(U)$ of the update U over the view definition V can be composed from elementary blocks T_{ij} as schematically shown in Fig. 1(a). This mechanism works if we can guarantee correctness of $T(U)$ as soon as all "tiles" T_{ij} are correct. In overall, the goal of the paper is to make the sketch above mathematically precise: to explicate the underlying algebraic structures and show their adequacy to the problem.

In more detail, the tiling idea separates the view update problem into two parts. The hard one is to create the very building blocks (tiles), i.e., to design update policies and write translators T_{ij} — we are not dealing with it in this paper. The other part is to design a composition mechanism for joining elementary translators together (tiling). We need to specify a common interface for tiles T_{ij} so that tiling would be implementation-independent and correctness-preserving. Although easier, this part is not straightforward and exposes several issues to be solved.

As mentioned above, update policies are inherently specific and dependent on particular kinds of schema constraints (dependencies), queries, updates, and their interaction. The tile interface should capture peculiarities of individual policies in an abstract way. On the other hand, the interface should *not* be overly abstract and skip essential constructs because it may corrupt tile composition (tiling must be "tight"). Note also that the very composition is two-dimensional, which requires special algebraic means beyond ordinary composion of functions. We also aim to formulate tiling mechanism in a syntax-independent way to make it applicable for models beyond relational. Indeed, "viewOf" relationships are ubiquitous in software engineering and different versions of the view update problem appear in different contexts beyond the traditional database area, for example, in bi-directional programming [7, 18], optimistic replication [8] and model synchronization [17, 5].

Use of algebra and compositionality is, of course, a known idea that can be traced back to the first works on the view update problem. Compositionality of updates is a basic assumption in founding works on constant complement [1] and dynamic views [10]. Compositionality of views is an evident requirement, but it has not been explicitly formulated until the recent work on lenses [2]: sequential lens composition [7] includes a trivial discrete treatment of view composition. Unfortunately, algebraic structures used in these works are too rough for our goals: they model the view and base datasets as discrete points with nothing in-between. That is, in terms of Fig. 1(a), arrows between nodes have no extension: they are just pairs of nodes and denote "viewOf" and "updateOf" relationships. The reality is richer. If a dataset B is a view of A, there is a *view mapping* $v\colon B \to A$ tracing the origin of B's elements. Similarly, if a dataset A' is an update of A, there is an *update mapping* $u\colon A \to A'$ specifying which elements were kept unchanged and which ones were modified in the update. In section 2 we consider an example justifying the claim. Thus, arrows in Fig. 1(a) do have extension and the tile composition mechanism must ensure their coherence. Figure 1(b) shows that translations well composable in the discrete framework may be not composable if mappings are taken into account.

Another key point to be addressed is syntax. To broaden applicability and avoid peculiarities of a particular syntactic mechanism, both lenses and dynamic views only consider views semantically, i.e., as functions from one space of datasets to another. We will say these frameworks are *syntax-free.* However, policy design does account for typing: data elements are typed by schema elements, view traceability links are typed by view definitions and update links are typed by update definitions ([4] is a notable example of how it works). Thus, in an accurate specification of update translation, syntax must be captured but in an abstract way. The framework is to be *syntax-independent* rather than syntax-free.

To summarize, we aim to specify a mathematical structure satisfying the following requirements:

(a) Be concrete enough to capture the following constructs essential for the view update problem: data, schemas and typing, view and update definitions, view and update mappings, and update translation all working in concert;

(b) Be abstract enough to model these constructs in a syntax-independent way;

(c) Address compositions of views and of updates, and their interaction between themselves and with update translations;

(e) Be algebraic to continue the "product line" of algebraic models for the view update problem.

Appearance of mappings in the requirement (a) and especially requirement (c) make use of category theory practically inevitable. Our main algebraic vehicle is the notion of double category, which is nothing but a systematic algebraic treatment of two-dimensional composition. In the course of the paper we explain categorical ideas in parallel with their application.

Our plan is as follows. In section 2 we consider a detailed example of relational view update to show that mappings do matter. We also demonstrate the deficiency of fixing one predefined update policy, and argue that a possibility to choose from a range of polices works better (sections 2.3,2.4). In section 3 we specify a mathematical structure satisfying the requirements; we try to reduce categorical "abstract nonsense" to a minimum and focus on ideas rather than technicalities.

ref to lechtenborger pods paper is lost

2 Mappings and view updates

Figure 2 presents an example of relational update propagation, inspired by the main example of [4].[1] Our goal in this section is first to illustrate the main concepts and constructs of the view update translation, and then abstract them in semi-formal terms to prepare and motivate the subsequent syntax-independent formalization (in section 3). Each subsection thus begins with considering a corresponding fragment of the example, and then proceeds to its semi-formal abstract arrangement.

Distinction between a mapping (function) and its instance (a link or tuple) is crucial for this section. Given a mapping $f\colon X \Rightarrow Y$ between sets or structures X, Y, we write the value of f at argument $x \in X$ as either $f(x)$ or $(x)f$ or $x.f$ choosing notation that makes formulas easier to read. If $y = f(x)$, the pair (x, y) is called an *(application) instance* of f. In instance diagrams like Fig. 2, we show such pairs by arrows $x \to y$ and call them *links.* Thus, a mapping f appears as a set of links, and in instance diagrams we show mappings by ovals encompassing their links. In

[1]We refer to [4] for science-historical explanations and possible complaints :).

the text, we distinguish mappings and links by using double- and single-line arrows respectively.

2.1 View definition

Consider three tables in the left top corner of Fig. 2. The database *schema* $\boldsymbol{\mathcal{S}}$ consists of two relation *schemes*, EDS(Empl,Dept,Sal) and DM(Dept,Mngr), and its possible state A is presented by the corresponding tables (with an intended typo in the Newton's name). The columns of table EDS are displayed in the "reverse order" to reduce clutter later when we will show mappings between relations. The names of key columns are underlined. The bullets in the columns under relation names denote row identifiers (rids), which will be referred to as EDS.row1, EDS.row2 etc. counting from the top. The set of rids of table T is denoted by $[\![T]\!]^{rid}$ and called the *rid extension* of T. The extension of the entire table (all columns including rid) is denoted by $[\![T]\!]$.

Table EDS⊗DM is derived by the following query Q:

$$EDS{\otimes}DM \overset{\text{def}}{=} (EDS \otimes_{Dept=Dept} DM) \upharpoonright_{(Empl,Mngr)}$$

where $\otimes_{\_}$ denotes the relational join operator (subindexed by the condition) and $\upharpoonright_{\_}$ denotes projection to the set of attributes in the subindex. More accurately, the syntactical side of the table (the attribute and relation names, metadata) is not derived and is a part of the query definition. The very data in the table are computed/derived by query execution, correspondingly, the respective part of the table is framed with dashed lines (blue with a color display). Rids are also derived: each one is a pair of rids from the respective operands, (EDS⊗DM).row1=(EDS.row1,DM.row1) and (EDS⊗DM).row2=(EDS.row2,DM.row2), which is also shown by dashed blue projection arrows. In general, the rid extension of a join table is a subset of the Cartesian product of the operands, $[\![T_1 \otimes_P T_2]\!]^{rid} \subset [\![T_1]\!]^{rid} \times [\![T_2]\!]^{rid}$ with the subset selection condition given by predicate P. We thus have two projection mappings $p_i\colon [\![T_1 \otimes T_2]\!]^{rid} \Rightarrow [\![T_i]\!]^{rid}$, $i = 1, 2$.

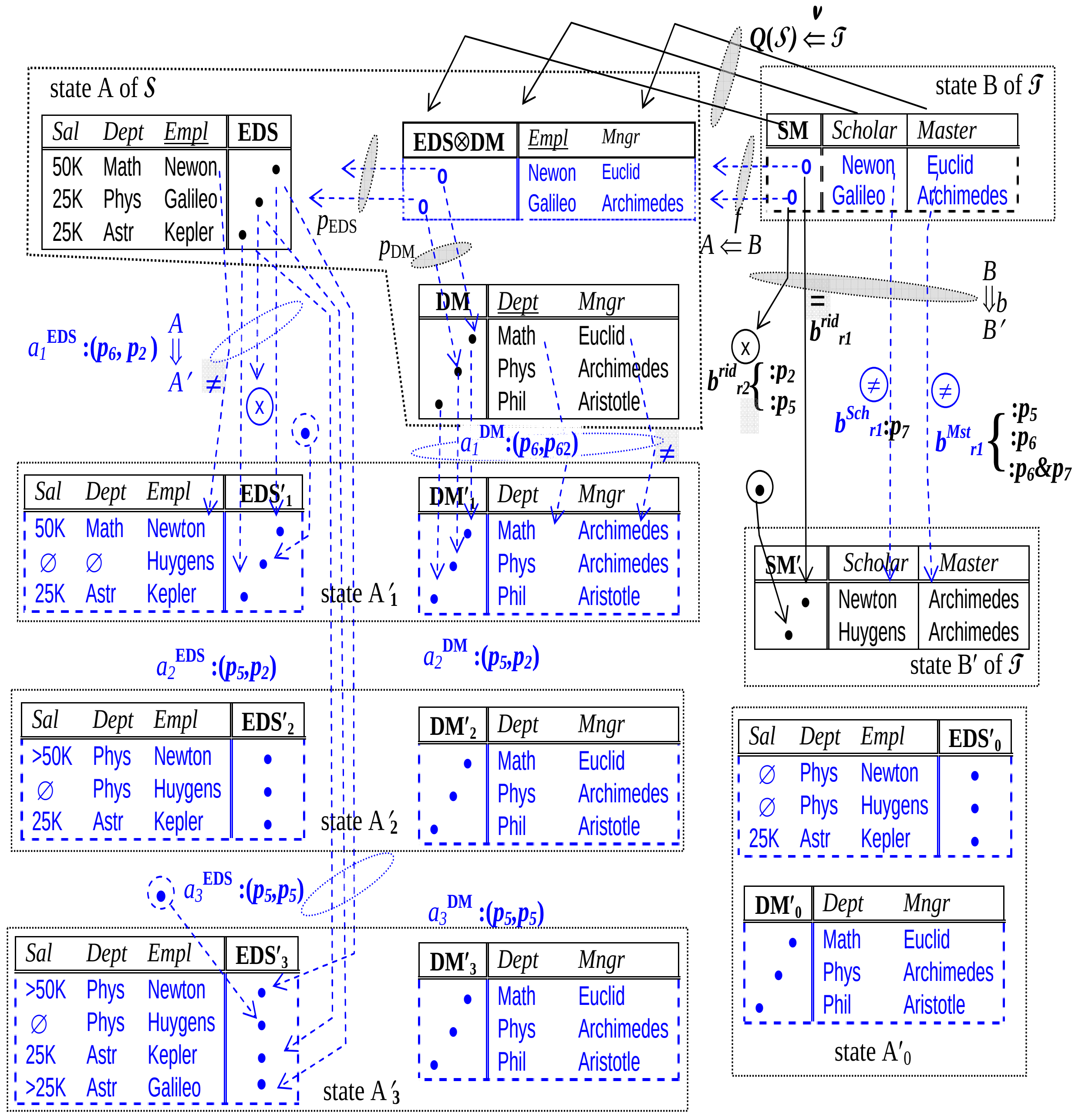
A view to $\boldsymbol{\mathcal{S}}$ is specified in the right top corner of the figure. The view schema $\boldsymbol{\mathcal{T}}$ consists of a single relational scheme SM(Scholar,Master), which is mapped to the base schema $\boldsymbol{\mathcal{S}}$ by three bent links at the very top of the figure. These three links form a mapping $\boldsymbol{v}\colon \boldsymbol{\mathcal{T}} \Rightarrow Q(\boldsymbol{\mathcal{S}})$, where $Q(\boldsymbol{\mathcal{S}})$ denotes the union of $\boldsymbol{\mathcal{S}}$ with the query schema $\boldsymbol{\mathcal{S}}_Q$ (in our example, $Q(\boldsymbol{\mathcal{S}})$ consists of three relation schemes EDS, DM, EDS⊗DM). We need the union rather than only $\boldsymbol{\mathcal{S}}_Q$ because in general some of the view schema relations are mapped to basic relations of $\boldsymbol{\mathcal{S}}$. Thus, a view definition is a triple $(\boldsymbol{\mathcal{T}}, Q, \boldsymbol{v})$ with $\boldsymbol{\mathcal{T}}$ the view schema, Q a query against $\boldsymbol{\mathcal{S}}$, and $\boldsymbol{v}$ a mapping as above. Importantly, the view mapping must respect the constraints, e.g., key columns are mapped to key columns.

Given a view definition, the extension of the view schema is computed by, first, executing the query (and getting table EDS⊗DM) and then coping the result to the view location (the table SM). More accurately, first an isomorphic set of fresh rids is created and then data in the attribute columns are copied. We thus have a mapping $f\colon [\![\mathrm{SM}]\!] \Rightarrow [\![\mathrm{EDS{\otimes}DM}]\!]$, whose rid part is shown in the figure.

2.2 Updates and update propagation

Let B denotes the initial state of the view shown in the figure. Suppose that the view is updated and comes to a state B' (table SM$'$). Let us first ignore all arrows in-between the two states and inspect the states immediately. We see that two old rows were deleted and two new rows are inserted. Deletions can be propagated to the database by deleting the respective rows in either one or both of the base relations EDS, DM (or by their modifications destroying the joining condition). Assume, for example, that deletions in a join table are realized by suitable deletions in the left operand of the term $T_1{\otimes}T_2$, (cf.[2]), i.e., table EDS in our case. We also assume that insertions into SM are realized by insertions into EDS, and so we add two new rows to EDS with Nulls in the Salary column denoting unknown values. The resulting database state A'_0 is shown on the right of the figure under the table SM'.

However, here is another interpretation of the passage from SM to SM'. Suppose that the row1 was modified rather than deleted: the typo in the name Newon was fixed, and Newton's Master changed from Euclid to Archimedes. We show this interpretation by linking the rids: arrow b^{rid}_{r1} linking SM.row1 and SM'.row1 means that two rows refer to the same real world object (note the =-label near the link). The absence of such linking for SM.row2 is syntactically sugared by a quasi-link b^{rid}_{r2} from the row to the "crash-symbol" ⊗. Similarly, the absence of links going *to* SM'.row2 is sugared by a "creation" link from symbol ⊙. Due to rigidity of the relational model, links for the rids determine the corresponding links for the values in the attribute columns, which may be immediately tested for equality and we get ≠-links from Newon to New*t*on and from Euclid to Archimedes.



Update Translation Table

Action over $\mathcal{T}$ (i.e., **SM**) \ Action over $\mathcal{S}$	Hire Empl. p_1	Fire Empl. p_2	Create Dept. p_3	Delete Dept. p_4	ChangeOf Dept. p_5	ChangeOf Mngr. p_6	Modify Attr. p_7
Delete row (u_1)	-	+	-	+	+ (moveToNomng.)	+ (Del Mngr.)	-
Insert row (u_2)	+	-	+	-	+ (moveFromNomng.)	+ (Get Mngr.)	-
Modify *Scholar* clmn. (u_3)	-	-	-	-	-	-	+
Modify *Master* clmn. (u_4)	-	-	-	-	+	+	-

Figure 2: Examples of relational view updates: the base schema $\mathcal{S}$={EDS,DM} and the view schema $\mathcal{T} = \{SM\}$

In this way we obtain an *update mapping* $b\colon B \Rightarrow B'$ consisting of links b^x_{ri}, $i = 1, 2$, $x = rid, Sch, Mst$. We will see in the next subsection that this interpretation of the update has several translation strategies that result in states different from A'_0.

We can imagine yet another update mapping consisting of the link b^{rid}_{r1} as above, and the link from SM.row2 to SM$'$.row2: think of the case when Galileo was erroneously placed into Physics instead of Huygens. Backward translation of this update would result in table EDS$'$ different from EDS$'_0$ by the value 25K rather than $\varnothing$ in Salary column. Thus, update mappings do matter.

Note that in general update mappings cannot be derived from the states. If we attempt to identify rows automatically by primary keys, then it would be impossible to modify the key (e.g., to fix a typo). If, on the contrary, we attempt to identify rows automatically by internal identifiers (rids), it would not work too because the user can delete a row but later recognize that was a mistake and restore the same row — however, the system would assign this same row a new identifier. Automatic rid-identification would not also work when the view is copied to another computer. Thus, update mappings is an independent part of update specification.

Update mappings reduce uncertainty in interpreting updates but not entirely. For example, there is a number of different database changes that cause the change of Newton's Master from Euclid to Archimedes in the view, and a number of different changes that cause deletion of the row (Galileo, Archimedes). We first fix one possible interpretation (cause 1) to see how the translation works, and later will address variability (causes 2,3,...).

Suppose that update $b\colon B \Rightarrow B'$ shown in Fig. 2 is caused by the change of the Math manager from Euclid to Archimedes and by firing Galileo. The corresponding base update is specified by mapping $a_1\colon A \Rightarrow A'_1$, whose components, $a^{\mathrm{EDS}}_1\colon A^{\mathrm{EDS}} \Rightarrow A'^{\mathrm{EDS}}_1$ and $a^{\mathrm{DM}}_1\colon A^{\mathrm{DM}} \Rightarrow A'^{\mathrm{DM}}_1$, are shown in the figure (some links are missing to avoid clutter but we remind that a complete set of rid-links determines the rest. Note that Huygens' department is unknown because now Archimedes manages both Math and Physics). These two component are actually a projection of a bigger mapping $Q(a_1)\colon Q(A) \Rightarrow Q(A'_1)$ including also the third component $a^{\mathrm{EMS}\otimes\mathrm{DM}}_1\colon \mathrm{EMS}\otimes\mathrm{DM} \Rightarrow \mathrm{EMS}'\otimes\mathrm{DM}'$ not shown in the figure.

The entire procedure of view computation and backward update propagation is specified in Fig. 3 in abstract terms of objects (data structures) and mappings. Horizontal and vertical arrows denote mappings related to views and updates respectively. "Depth" arrows going from the front to the back side are typing mappings: they provide types (relation and attribute names) for data elements. The user-defined elements are shown by black solid lines, the computed elements are dashed (blue with a color display); basic nodes are shaded and derived ones are blank. Computation consists of five consecutive operations (denoted by double-headed arrows with numbers), which input two arrows of a ceratin incidence, and output other two arrows that extend the input configuration up to a square. Substituting results of one operation to another amounts to tiling. This is a key advantage of the notation: substitution becomes geometrically evident.

The operations (steps) are as follows.

Step 1^1. *Query execution.* A query again schema $\boldsymbol{\mathcal{S}}$ is specified by adding to $\boldsymbol{\mathcal{S}}$ new derived elements to be computed by the query. Hence, we have an inclusion mapping $\mathsf{i}_Q\colon \boldsymbol{\mathcal{S}} \hookrightarrow Q(\boldsymbol{\mathcal{S}})$. A database state is a typing mapping $\tau_A\colon A \Rightarrow \boldsymbol{\mathcal{S}}$ (see the top horizontal left square in the figure). Given a query definition i_Q and data τ_A, the query can be executed; adding the execution results to the initial dataset can be specified by inclusion $\mathsf{i}^A_Q\colon A \hookrightarrow Q(A)$.

Step 1^2. *Coping and relabeling.* The results of the query are copied to the view according to the view mapping $\boldsymbol{v}$.

Step 2^1. *Update execution.* An update request $\boldsymbol{u}$ is submitted to the view and executed producing an update mapping b (the rightmost face of the parallelepiped).

Step 2^2. *Backward relabeling.* The update results are relabeled according to the augmented schema $Q(\boldsymbol{\mathcal{S}})$ (and may be thought of as "copied" back to virtual slots storing the query results). We may also consider this operation as backward projection of B' along f, hence the projection notation $\_\restriction_f$ for the results.

Step 3. *Backward propagation.* The query schema extension is backward propagated to the database. This propagation can be considered as finding solution A' to algebraic equation $Q(A') = B'\restriction_f$. As in general there are many solutions, to achieve uniqueness some update policy is needed. This is the most non-trivial part of the procedure.

Diagram Fig. 2(b) presents an abbreviated version of diagram (a): we hide the query mechanism inside the view mappings $\boldsymbol{v}_Q$, f_Q and operation vExe of view execution which includes query execution. The idea is made precise in categorical algebra, and is known

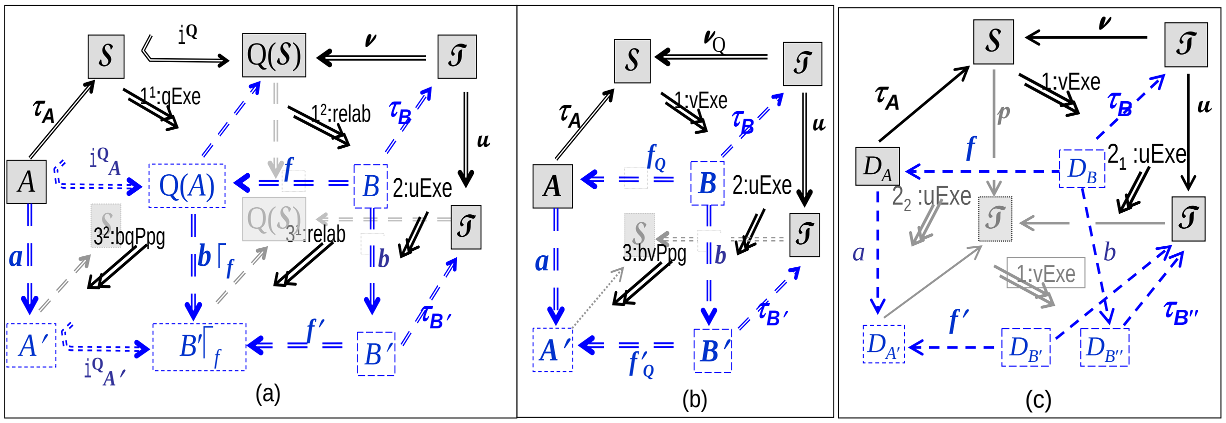


Figure 3: Updates via mappings and tiles

under the name of *Kleisli mappings* ([15]). Thus, we may understand view mappings as Kleisli mappings (for some unspecified query langauge) and leave the Q-part of the machinery implicit (including the subindex Q).

Apart of variability of update policies, diagram in Fig. 3(a) gives a sufficiently accurate abstract specification of the content of Fig. 2 and captures all but one sort of mappings appearing in the figure. What is missing is links and mappings relating query results with basic data — note projection mappings p_{EDS}, p_{DM} in Fig. 2. These mapping are important for update translation but are often implicit in the literature ([4] is a notable exception). Recently, they have been considered in the literature on data lineage and uncertainty [16]. An accurate formalization of lineage mappings is a whole story because they are multi-valued, but for the present paper we note the following. Let $\boldsymbol{v}^{+}\colon \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ denotes the composition of the view definition mapping as above with (multi-valued) lineage mapping $l\colon Q(\boldsymbol{\mathcal{S}}) \Rightarrow \boldsymbol{\mathcal{S}}$ induced by query Q; correspondingly, we have view mapping $f^{+}\colon B \Rightarrow A$. Now, if $\boldsymbol{v}'\colon \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ is a parallel view definition, we have $\boldsymbol{v}'^{+}{=}\boldsymbol{v}^{+}$ and $f'^{+}{=}f'$ as soon as $\boldsymbol{v}{=}\boldsymbol{v}'$ because our lineage mappings are entirely determined by queries. That is, if two tiles coincide on a common sub-edge $\boldsymbol{v}$ or f, they have to coincide on the entire edge including lineage. It allows us to hide lineage inside of view mappings $\boldsymbol{v}$ and f.

2.3 Variability of update policies...

We remind that the view update $b\colon B \Rightarrow B'$ was translated into the base update $a_1\colon A \Rightarrow A'_1$ under assumption that b is caused by the change of the Math manager from Euclid to Archimedes and by firing Galileo. However, several other causes are possible.

For example, Newton may move to Physics department while managers are not changed: the resulting base state is A'_2 with the corresponding update mapping (not shown in the figure to avoid clutter). We also assume a policy that any move to another department leads to a salary increase, hence the constraint >50K. Yet another possibility is when Newton moved to Physics *and* Archimedes starts to manage Math (it is not shown in the figure).

All these base updates were computed with the assumption that row (Galileo, Archimedes) disappeared since Galileo quitted the enterprize. Now assume that Galileo moved to Astronomy successfully performing without a manager. This move together with Newton's move to Physics give us the updated state A'_3 together with update mapping $a_3\colon A \Rightarrow A'_3$ (see the figure). There are other possibilities of base updates providing the same view update. These possibilities can be enumerated with a table like the Translation table in the bottom of Fig. 2. The leftmost column lists four possible view update actions $\boldsymbol{u}_i$, $i = 1..4$. The topmost row lists seven related database update actions $\boldsymbol{p}_j$, $j = 1..7$. Plus symbols in the cells mark possible backward interpretations of view updates. In some cases these interpretations can be composed: consider, for example, the double move ($\boldsymbol{p}_5$&$\boldsymbol{p}_6$) mentioned above, or compositions of department deletion with its employees firing, $\boldsymbol{p}_4$&$\boldsymbol{p}_2$, or with employees move to a non-managed department $\boldsymbol{p}_4$&$\boldsymbol{p}_5$. When choosing an update policy, we may exclude such combined interpretations as non-minimal, but there is still a multitude of minimal (non-comparable) interpretations encoded by the table.

In Fig. 2, each of the links b^{X}_{ri}, $i{=}1,2,$

$x{=}rid, Sch, Mst$ constituting the update $b\colon B \Rightarrow B'$ is labeled with a set of possible database update $\boldsymbol{p}_i$ causing this link to appear. By choosing one of the labels from the set, we set a unique backward interpretation of the view update, for example, taking $\boldsymbol{p}_5$ for b_{r1}^{Mst} and $\boldsymbol{p}_2$ for b_{r2}^{rid} states that Newton moved to the department managed by Archimedes and Galileo quitted. It does not provide a unique database update (if there are several departments managed by Archimedes), but essentially narrows the range of possibilities. This range can be described by conditional values, for example, we place the value "a dept. managed by Archimedes" in the column EDS$'_2$Dept for row Newton. In our case, this value is uniquely evaluated to Physics and we come to update $a_2\colon A \Rightarrow A_2$ (note its label $(\boldsymbol{p}_5,\boldsymbol{p}_2)$). Thus, we have six possibilities of backward translations (six combinations of labels for the links), three of which are shown in the figure. In general, the same update $b\colon B \Rightarrow B'$ may have a number of quite reasonable and correct backward interpretations $a\colon A \Rightarrow A'$.

2.4 ... and changing the focus.

The conclusion above is well-known, and it is what constitutes the view update problem. Our goal in the paper is to provide general and unifying mathematical foundations for different approaches to the problem. To this end, we consider the most general situation when the view user is allowed to label view updates with possible database updates to inform the system about the user's interpretation (or intention) with the update. To subsume the case of a predefined update policy, we may introduce a special null label "no interpretation", in which case the system translates the update according to some predefined "null" policy.[2]

Thus, a view update request becomes a pair $(\boldsymbol{u}_i,\boldsymbol{p}_j)$ with $\boldsymbol{u}_i$ the view update definition and $\boldsymbol{p}_j$ its intended backward interpretation. This settings changes the focus of the view update problem. The main task is the design of update definition spaces, $U_{\boldsymbol{\mathcal{S}}}$ and $U_{\boldsymbol{\mathcal{T}}}$, and enumeration of reasonable correspondences between them (the Translation Table of our example). The previous focus becomes a particular branch related to the "null"-label, that is, to backward translation of pairs $(\boldsymbol{u},\emptyset){\in}U_{\boldsymbol{\mathcal{S}}}{\times}U_{\boldsymbol{\mathcal{T}}}$. The approach outlined above seems appealing but its precise mathematical underpinning is not straightforward. Considering view updates as pairs $(\boldsymbol{u}_i,\boldsymbol{p}_j)$ means that the translation mapping is a binary relation $\boldsymbol{T} \subset U_{\boldsymbol{\mathcal{T}}}{\times}U_{\boldsymbol{\mathcal{S}}}$ rather than a dynamic view function $\boldsymbol{T}: U_{\boldsymbol{\mathcal{T}}} \Rightarrow U_{\boldsymbol{\mathcal{S}}}$ [10]. Moreover, our example shows that the update spaces are graphs of states and update mappings rather than plain sets of states. Thus, $\boldsymbol{T}$ is a binary relation between graphs. In addition, we have seen that the spaces are interconnected by view mappings $f\colon B \Rightarrow A$. Finally, both update and view mappings are composable and translation should be compatible with these compositions in some way. In the next section we specify a suitable mathematical framework.

[2]By endowing the system with inductive learning and other AI capabilities, we may gradually make the backward translation more reasonable by using heuristics, or/and modifying the interface for update request.

3 Well-defined view systems

We begin with a short primer on graphs and categories. To ease reading, we will often write "a set X of *widgets*" instead of "an abstract set X whose elements are called *widgets*".

3.1 From graphs to double categories

Graphs. A *graph* G consists of a set of *nodes* G_0 and a set of *arrows* G_1 together with two mappings $\partial_i\colon G_1 \to G_0$, $i = 0, 1$. As usual, we write $a\colon N \to N'$ if $\partial_0 a = N$ and $\partial_1 a = N'$. Sometimes we write $a\colon N \to N'{::}G$ to remind that $a{\in} G_1$. Two arrows a, b are called *composable* if $a\partial_1 = \partial_0 b$. A graph is called *thin* if for any pair of nodes (N, N') there is at most one arrow $a\colon N \to N'$. A *graph mapping (morphism)* $f : G \to G'$ is a pair (f_0, f_1) of mappings (functions) $f_i\colon G_i \to G'_i$ that preserves the incidence relations between nodes and arrows ($a.f_1.\partial_0{=}a.\partial_0.f_0$, etc.).

Tiles. Let $\Box$ denotes a fixed graph shown in Fig. 4(a) with fixed names for its nodes and arrows. A *tile* in a graph G is a graph mapping $T\colon \Box \to G$. It is convenient to denote it as shown in Fig. 4(b) (ignore symbol $\mathbb{P}$ for a moment), which means that $T(12) = f$, $T(13) = b$, $T(34) = f'$, and $T(24) = a$. In other words, a tile is a quadruple of arrows (f, b, f', a) subject to incidence relationships shown in diagram (b) and encoded by diagram (a). If we write a tile as a quadruple of arrows, we always follow the order used above (12,13,34,24). Some nodes and some arrows in a tile may be equal (it's a tuple, not a set), e.g., it may happen that $T(2){=}T(4) = A$, in which case both left nodes in diagram (b) will be A. Nodes A, B.. and arrows f, b.. are *elements* of the tile T. They will be denoted by ${}^{\bullet}T$, $T^{\bullet}$,.. and $\overline{T}$, $T|$,... respectively. If $\mathbb{P}$ is a class of tiles (a predicate that defines the class), we write $T{:}\mathbb{P}$ for $T \in \mathbb{P}$ and place this declaration at the center of the diagram. The name of the tile may

$$\begin{tikzcd}
2 & 1 \arrow[l, "12"'] \arrow[d, "13"] \\
4 & 3 \arrow[l, "34"]
\arrow[from=1-1, to=2-1, "24"']
\end{tikzcd}$$
(a)

$$\begin{tikzcd}
A \arrow[d, "a"'] \arrow[dr, phantom, "T{:}\mathbb{P}"] & B \arrow[l, "f"'] \arrow[d, "b"] \\
A' & B' \arrow[l, "f'"]
\end{tikzcd}$$
(b)

$$\begin{tikzcd}
A \arrow[d, "a"'] \arrow[dr, phantom, "\Delta\swarrow"] & B \arrow[l, "f"'] \arrow[d, "b"] \\
A' & B' \arrow[l, "f'"]
\end{tikzcd}$$
(c)

Figure 4: Tile (a,b) and db-arrow (c) definitions

be left implicit but the predicate is always there.

Categories and functors. A *category* $\boldsymbol{C}$ is a graph with
(i) an associative operation of arrow composition denoted by ; (semi-colon), that is, any pair of composable arrows $A \xrightarrow{a} B \xrightarrow{b} C$ is assigned with one and only one arrow $A \xrightarrow{c} C$ denoted by $a;b$, and
(ii) a unary operation $\mathbf{1}\_$ that assigns to every node N an *identity/idle* loop $\mathbf{1}_N : N \to N$. These loops are units of the composition: $\mathbf{1}_{\partial_0 a}; a = a = a; \mathbf{1}_{\partial_1 a}$. Nodes in a category are called *objects*, and arrows are *morphisms*. The graph underlying category $\boldsymbol{C}$ is denoted by $|\boldsymbol{C}|$, but we will often omit the bars. Hence, $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ denote the classes of all objects and all morphisms resp. The former can be also considered as a category, whose only arrows are identities (called *discrete category*).

A *functor* $f\colon \boldsymbol{C} \to \boldsymbol{C'}$ is a graph morphism $f\colon |\boldsymbol{C}| \to |\boldsymbol{C'}|$ that is compatible with arrow composition $(a;b).f_1{=}(a.f_1);(b.f_1)$ and idle loops $f_1(\mathbf{1}_N){=}\mathbf{1}_{N.f_0}$.

Double graphs, categories, functors. A *double graph* $\mathbb{D}$ is a four-sorted algebraic structure comprising a class $\mathbb{D}_0$ of *nodes* (also called *0-cells*), classes $\mathbb{D}_1^{\mathsf{h}}$ of *horizontal* and $\mathbb{D}_1^{\mathsf{v}}$ of *vertical arrows (1-cells)* and a class $\mathbb{D}_2$ of *double arrows (2-cells)* subject to incidence (source and target) conditions shown in diagram Fig. 4(c). To wit: if Δ denotes a double arrow, then its *vertical source* is arrow f, the *horizontal target* is a etc. These data define a tile $\partial\Delta$ called the *boundary* of Δ. Below we use abbreviations *h-arrow*, *v-arrow*, *db-arrow* or *db-graph*.

A db-graph is called *thin* if for any tile T there is at most one db-arrow Δ such that $\partial\Delta = T$.

A *double category* is db-graph satisfying the following requirements. Nodes and h-arrows form a category denoted, again, by $\mathbb{D}_1^{\mathsf{h}}$, and similarly, we have a category $\mathbb{D}_1^{\mathsf{v}}$ so that $(\mathbb{D}_1^{\mathsf{h}})_0 = (\mathbb{D}_1^{\mathsf{v}})_0 = \mathbb{D}_0$. H-arrows considered as *nodes* and db-arrows as *arrows* form a category $\mathbb{D}_2^{\mathsf{v}}$, in which db-arrows are composed *vertically.* similarly, v-arrows as *nodes* and db-arrows as *arrows* form category $\mathbb{D}_2^{\mathsf{h}}$ in which db-arrows are composed *horizontally.* Thus, both categories have the same set of arrows, $(\mathbb{D}_2^{\mathsf{h}})_1 = (\mathbb{D}_2^{\mathsf{v}})_1 = \mathbb{D}_2$, but compositions are different. Nevertheless we denote them by the same symbol, semi-colon, because the context is always clear.

The fact that $\mathbb{D}_2^{\mathsf{x}}$, x=h,v, are categories means that there are *h-idle* and *v-idle* db-arrows, which are the units of the respective compositions. Horizontal edges of h-idle db-arrows must be idle h-arrows; same holds for v-idle db-arrows. (These definitions become quite clear when the diagrams are drawn on a paper, see [13] or [6]). Finally, horizontal and vertical db-arrow compositions are coordinated between themselves by an *interchange law*: for a regular net of db-arrows like in Fig. 1(a), any order of composition (first horizontally and then vertically, or the other way round, or in a mixed way) gives the same result (see [13] for details).[3]

A *double functor* $\boldsymbol{f} : \mathbb{D} \to \mathbb{E}$ is a quadruple of functions $\boldsymbol{f}_0, \boldsymbol{f}_1^{\mathsf{v}}, \boldsymbol{f}_1^{\mathsf{h}}, \boldsymbol{f}_2$, sending i-cells in $\mathbb{D}$ to respective i-cells in $\mathbb{E}$ $(i = 0, 1^{\mathsf{v}}, 1^{\mathsf{h}}, 2)$, with preservation of all incidence relationships and such that pairs $(\boldsymbol{f}_0, \boldsymbol{f}_1^{\mathsf{x}})\colon \mathbb{D}_1^{\mathsf{x}} \to \mathbb{E}_1^{\mathsf{x}}$ and $(\boldsymbol{f}_1^{\mathsf{x}}, \boldsymbol{f}_2)\colon \mathbb{D}_2^{\mathsf{x}} \to \mathbb{E}_2^{\mathsf{x}}$ (x=v,h) are ordinary functors.

3.2 Building the definition

We aim to define a mathematical structure satisfying the list of requirements described in Introduction. We approach the task with a sequence of steps that consecutively introduce new "pieces" of structure until the requirements (a,c) are satisfied.

3.2.1 The universe

We take some infinite directed graph $\mathscr{U}$ to be the carrier of the mathematical structure we are going to build. Nodes of $\mathscr{U}$ are interpreted as objects with structure of some kind (e.g., graphs with additional structure). Arrows of $\mathscr{U}$ are interpreted as mappings between these objects compatible with their structure. We will use terms node and object, arrow and mapping interchangeably.

Some classes of $\mathscr{U}$-object may have additional structure, and some classes of mappings may respect this additional structure. We will thus have several sorts of objects and mappings. Some pairs of composable mappings can be indeed composed into a "long" arrow while others cannot depending on sorting. We say that $\mathscr{U}$ is an infinite graph with *partially composable* arrows.

[3]Appendix reproduces this material for the reviewers' convenience but is excluded from the paper due to space limitations.

$$\begin{tikzcd}
\boldsymbol{\mathcal{S}} \arrow[d, "\boldsymbol{p}"'] \arrow[dr, phantom, "\boldsymbol{w}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{T}} \arrow[l, "\boldsymbol{v}"'] \arrow[d, "\boldsymbol{u}"] \\
\boldsymbol{\mathcal{S}} & \boldsymbol{\mathcal{T}} \arrow[l, "\boldsymbol{v}"]
\end{tikzcd}$$

(a) general

$$\begin{tikzcd}
\boldsymbol{\mathcal{S}} \arrow[d, "\mathbf{1}^{\mathsf{v}}_{\boldsymbol{\mathcal{S}}}"'] \arrow[dr, phantom, "\mathbb{1}^{\mathsf{v}}_{\boldsymbol{v}}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{T}} \arrow[l, "\boldsymbol{v}"'] \arrow[d, "\mathbf{1}^{\mathsf{v}}_{\boldsymbol{\mathcal{T}}}"] \\
\boldsymbol{\mathcal{S}} & \boldsymbol{\mathcal{T}} \arrow[l, "\boldsymbol{v}"]
\end{tikzcd}$$

(b) v-idle

$$\begin{tikzcd}
\boldsymbol{\mathcal{S}} \arrow[d, "\boldsymbol{u}"'] \arrow[dr, phantom, "\mathbb{1}^{\mathsf{h}}_{\boldsymbol{u}}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}} \arrow[l, "\mathbf{1}^{\mathsf{h}}_{\boldsymbol{\mathcal{S}}}"'] \arrow[d, "\boldsymbol{u}"] \\
\boldsymbol{\mathcal{S}} & \boldsymbol{\mathcal{S}} \arrow[l, "\mathbf{1}^{\mathsf{h}}_{\boldsymbol{\mathcal{S}}}"]
\end{tikzcd}$$

(c) h-idle

$$\begin{tikzcd}
\boldsymbol{\mathcal{S}} \arrow[d, "\mathbf{1}^{\mathsf{v}}_{\boldsymbol{\mathcal{S}}}"'] \arrow[dr, phantom, "\mathbb{1}^{\mathsf{vh}}_{\boldsymbol{\mathcal{S}}}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}} \arrow[l, "\mathbf{1}^{\mathsf{h}}_{\boldsymbol{\mathcal{S}}}"'] \arrow[d, "\mathbf{1}^{\mathsf{v}}_{\boldsymbol{\mathcal{S}}}"] \\
\boldsymbol{\mathcal{S}} & \boldsymbol{\mathcal{S}} \arrow[l, "\mathbf{1}^{\mathsf{h}}_{\boldsymbol{\mathcal{S}}}"]
\end{tikzcd}$$

(b+c) vh-idle

Figure 5: Class of $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$-tiles

3.2.2 Metadata

A: Data, view and update definitions. There is a subclass ***dataDef*** of $\boldsymbol{\mathscr{U}}$-objects called *data definitions* or *schemas.* There are two sorts of arrows between schemas: *update definitions* and *view definitions* (the former are loops). When talking about metadata, we will shortly say updates and views. Each schema $\boldsymbol{\mathcal{S}}$ is assigned with two special loop-arrows: *idle update* $\mathbf{1}^{\boldsymbol{\mathcal{S}}}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}}$ (that defines an idle action doing nothing) and *identical view* $\mathbf{1}_{\boldsymbol{\mathcal{S}}}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}}$ (that defines $\boldsymbol{\mathcal{S}}$ as a view of itself). Updates can be composed associatively, and the idle update is neutral w.r.t. composition. It means that schemas and updates form a category, ***updDef***. Similarly, we postulate a category of schemas and views ***viewDef***. These two categories have the same class of nodes but different arrows. Together they form a subgraph $\boldsymbol{\mathscr{U}}^{\mathrm{def}}$ of $\boldsymbol{\mathscr{U}}$. We additionally assume that each schema $\boldsymbol{\mathcal{S}}$ has a designated set of basic updates $\mathbf{U}_{\boldsymbol{\mathcal{S}}}$ so that all $\boldsymbol{\mathcal{S}}$-updates are freely generated by $\mathbf{U}_{\boldsymbol{\mathcal{S}}}$. To ease reading the diagrams below, we will always draw view arrows horizontally and updates vertically.

B: Forward update translation. An ordinary view definition specifies how to transform data. We consider an extended notion of view, which specifies also how to transform updates.

We assume defined a class $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$ of tiles of shape specified in Fig. 5(a). It is interpreted as that update $\boldsymbol{p}$ on the base is translated to update $\boldsymbol{u}$ on the view. Note that two horizontal arrows are equal: translation does not change the view definition.

Class $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$ satisfies the following conditions.

(o) We assume that any basic view update $\boldsymbol{u}$ is correct and hence there is at least one $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$-tile having $\boldsymbol{u}$ as its right edge.

$$\begin{tikzcd}
\boldsymbol{\mathcal{S}}_1 \arrow[d, "\boldsymbol{u}_1"'] \arrow[dr, phantom, "\boldsymbol{w}_{21}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}}_2 \arrow[l, "\boldsymbol{v}_{21}"'] \arrow[d, "\boldsymbol{u}_2"] \arrow[dr, phantom, "\boldsymbol{w}_{32}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}}_3 \arrow[l, "\boldsymbol{v}_{32}"'] \arrow[d, "\boldsymbol{u}_3"] \\
\boldsymbol{\mathcal{S}}_1 \arrow[d, "\boldsymbol{u}'_1"'] \arrow[dr, phantom, "\boldsymbol{w}'_{21}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}}_2 \arrow[l, "\boldsymbol{v}_{21}" description] \arrow[d, "\boldsymbol{u}'_2"] \arrow[dr, phantom, "\boldsymbol{w}'_{32}{:}\boldsymbol{uv}\mathbb{D}\mathbf{ef}"] & \boldsymbol{\mathcal{S}}_3 \arrow[l, "\boldsymbol{v}_{32}" description] \arrow[d, "\boldsymbol{u}'_3"] \\
\boldsymbol{\mathcal{S}}_1 & \boldsymbol{\mathcal{S}}_2 \arrow[l, "\boldsymbol{v}_{21}"] & \boldsymbol{\mathcal{S}}_3 \arrow[l, "\boldsymbol{v}_{32}"]
\end{tikzcd}$$

Figure 6: $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$-tile composition

(a) For any update $\boldsymbol{p}$, there is one and only one $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$-tile. Hence, we may consider $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$ as an operation producing arrow $\boldsymbol{u}$ from the other elements of the tile. Condition (o) states that this operation is surjective.

(b) The class contains special tiles specified in diagrams Fig. 5(b,c). That is, an idle update is forward translated into the corresponding idle update, and identical views do nothing with updates. It implies that all tiles consisting of four idle arrows belong to $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$ (diagram (b+c)).

(c) $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$-tiles can be composed horizontally and vertically as shown in Fig. 6. In more detail, we define outer tiles $\boldsymbol{w}_{32};\boldsymbol{w}_{21} \overset{\mathrm{def}}{=} [(\boldsymbol{v}_{32};\boldsymbol{v}_{21}), \boldsymbol{u}_3, (\boldsymbol{v}_{32};\boldsymbol{v}_{21}), \boldsymbol{u}_1]$ and $\boldsymbol{w}_{21};\boldsymbol{w}'_{21} \overset{\mathrm{def}}{=} [\boldsymbol{v}_{21}, (\boldsymbol{u}_2;\boldsymbol{u}'_2), \boldsymbol{v}_{21}, (\boldsymbol{u}_1;\boldsymbol{u}'_1)]$, and the condition states they belong to $\boldsymbol{uv}\mathbb{D}\mathbf{ef}$ as soon as the inner tiles are such.

Below, names of definitions (data, update, view) are bold italic.

3.2.3 Data vs. metadata

For any schema $\boldsymbol{\mathcal{S}}$, there is a class of arrows into $\boldsymbol{\mathcal{S}}$ disjoint to $\boldsymbol{\mathscr{U}}^{\mathrm{def}}$ and called *typing mappings.* A pair $A = (D_A, \tau_A)$ with $\tau_A\colon D_A \to \boldsymbol{\mathcal{S}}$ a typing mapping is to be thought of as *data object* D_A schematized by $\boldsymbol{\mathcal{S}}$. We also call such pairs *datasets* over $\boldsymbol{\mathcal{S}}$, or *(data) instances* of $\boldsymbol{\mathcal{S}}$, or *(database) states* when $\boldsymbol{\mathcal{S}}$ is considered a database schema. The class of $\boldsymbol{\mathcal{S}}$-instances is denoted by $[\![\boldsymbol{\mathcal{S}}]\!]$. We will also write $A{:}\boldsymbol{\mathcal{S}}$ for $A \in \boldsymbol{\mathcal{S}}$.

3.2.4 Execution mechanisms

A1: Update execution. We assume defined some class $\mathbf{u}\mathbb{E}\mathbf{xe}$ of tiles of the shape shown in Fig. 7(a): the left arrow is an update definition, the horizontal arrows are typing mappings, and arrow a is to be thought of as execution of $\boldsymbol{u}$ for dataset $A{=}(D_A, \tau_A)$.

Class $\mathbf{u}\mathbb{E}\mathbf{xe}$ satisfies the following conditions.

(a) For every arrow $\boldsymbol{p}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}}$ and data instance $A = (D_A, \tau_A)$ of $\boldsymbol{\mathcal{S}}$, there one and only one $\mathbf{u}\mathbb{E}\mathbf{xe}$-tile

$$\begin{tikzcd}
\mathcal{S} \arrow[d, "\boldsymbol{p}"'] \arrow[dr, phantom, ":\mathbb{uExe}"] & D_A \arrow[l, "\tau_A"'] \arrow[d, "a"] \\
\mathcal{S} & D_{A'} \arrow[l, "\tau_{A'}"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathcal{S} \arrow[d, "1^{\mathsf{v}}_{\mathcal{S}}"'] \arrow[dr, phantom, "\mathbf{1}_A{:}\mathbb{uExe}"] & D_A \arrow[l, "\tau_A"'] \arrow[d, "a"] \\
\mathcal{S} & D_A \arrow[l, "\tau_A"']
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\mathcal{S} \arrow[d, "\boldsymbol{p}"'] \arrow[dr, phantom, ":\mathbb{uExe}"] & D_A \arrow[l, "\tau_A"'] \arrow[d, "a"] \\
\mathcal{S} \arrow[d, "\boldsymbol{p}'"'] \arrow[dr, phantom, ":\mathbb{uExe}"] & D_{A'} \arrow[l, "\tau_{A'}"'] \arrow[d, "a'"] \\
\mathcal{S} & D_{A''} \arrow[l, "\tau_{A''}"']
\end{tikzcd}$$

(c)

Figure 7: Update execution

$$\begin{tikzcd}
\mathcal{S} & \mathcal{T} \arrow[l, "\boldsymbol{v}"'] \\
D_A \arrow[u, "\tau_A"] \arrow[ur, phantom, ":\mathbb{vExe}"] & D_B \arrow[l, "f"'] \arrow[u, "\tau_B"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathcal{S} & \mathcal{S} \arrow[l, "1^{\mathsf{h}}_{\mathcal{S}}"'] \\
D_A \arrow[u] \arrow[ur, phantom, "\mathbf{1}_A{:}\mathbb{vExe}"] & D_A \arrow[l, "f"] \arrow[u]
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\mathcal{S} & \mathcal{T} \arrow[l, "\boldsymbol{v}"'] & \boldsymbol{u} \arrow[l, "\boldsymbol{w}"'] \\
D_A \arrow[u, "\tau_A"] \arrow[ur, phantom, ":\mathbb{vExe}"] & D_B \arrow[l, "f"'] \arrow[u, "\tau_B"] \arrow[ur, phantom, ":\mathbb{vExe}"] & D_C \arrow[l, "g"'] \arrow[u, "\tau_C"']
\end{tikzcd}$$

(c)

Figure 8: View execution

having $\boldsymbol{p}$ and τ_A at its left top corner as shown in diagram (a). Hence, we may consider $\mathbb{uExe}$ as an operation producing instance A' and vertical arrow a from the other elements of the tile. We denote instance A' by $A^{\boldsymbol{p}}$, arrow a by $\overline{\boldsymbol{p}_A}$, and the entire tile by $[\boldsymbol{p}]_A$. Arrows like a are called *update mappings*

(b) All tiles with idle update definitions Fig. 7(b) are in $\mathbb{uExe}$. In this way a subclass of *idle* update mappings denoted by $\mathbf{1}_A$ is defined.

(c) Class $\mathbb{uExe}$ is closed under vertical tile composition Fig. 7(c), That is, if two inner tiles are from $\mathbb{uExe}$, composition $a; a'$ is defined in $\mathcal{U}$ and the outer tile is also in $\mathbb{uExe}$. We assume that composition so defined is associative, and tiles specified by diagram (b) are units. The latter means that idle update definition produce idle update mappings. Hence, our assumptions can be summarized by saying that instances as objects and $\mathbb{uExe}$-tiles as arrows between them form a category. We will denote it by ***updMap***.

A2: View execution. We assume defined some class $\mathbb{vExe}$ of tiles of the shape shown in Fig. 8(a). The upper horizontal arrow is a view definition, the vertical arrows are typing mappings—the right one is computed by executing the view definition for dataset $A{=}(D_A, \tau_A)$, the bottom arrow is to be thought of as the view traceability mapping. We denote the entire tile by $[\boldsymbol{v}]_A$ and write $A^{\boldsymbol{v}}$ for B and $\overline{\boldsymbol{v}_A}$ for *view mapping* f.

Class $\mathbb{vExe}$ satisfies several conditions being the horizontal analogs of conditions for class $\mathbb{uExe}$; Fig. 8 explains the idea. Hence, we have a category of data objects and *view mappings*, ***viewMap***, whose arrows are view execution tiles.

B: Forward update translation. Let $\boldsymbol{w}$ be a $\mathbb{wDef}$-tile in Fig. 5(a). It provides a view definition and two update definitions, which can be executed for a given data instance $A{=}(D_A, \tau_A)$. There are two concurrent computation path: We can get a pair of instances B and B' by executing view $\boldsymbol{v}$ for A and for $A' = A^{\boldsymbol{v}}$ respectively (see see Fig. 9(a)). Alternatively, we may execute update $\boldsymbol{u}$ for view $B = A^{\boldsymbol{v}}$ and get another instance B'' Fig. 3(c).

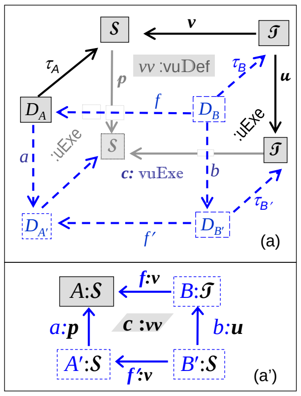


Figure 9: Forward update translation

We call a tile $\boldsymbol{w}$ *well-defined* if instances B' and B'' are equal. Note that instance equality means both $D_{B'} = D_{B''}$ and $\tau_{B"} = \tau_{B''}$. In the string-base notation, the law takes the form

$$(\mathsf{Sync})_{\boldsymbol{w}} \quad (A^{\boldsymbol{v}})^{\boldsymbol{u}} = (A^{\boldsymbol{u}})^{\boldsymbol{p}}$$

where $A \in [\![{}^{\bullet}\boldsymbol{w}]\!]$, $\boldsymbol{p}{=}|\boldsymbol{w}$, $\boldsymbol{u}{=}\boldsymbol{w}|$ and $\overline{\boldsymbol{w}} = \underline{\boldsymbol{w}} = \boldsymbol{v}$.

In this case, we may say that the view definition tile $\boldsymbol{w}$ is executed for instance A and completes the pair $(\boldsymbol{w}, A)$ up to an arrow *translation* cube shown in Fig. 9. We denote this cube by $[\boldsymbol{w}]_A$. Thus, a well-defined $\mathbb{wDef}$-tile gives rise to a correct translation cube.

We assume that every $\mathbb{wDef}$-tile is well-defined, and we thus have a class

$$\mathbb{wExe} = \{[\boldsymbol{w}]_A : \ \boldsymbol{w} \in \mathbb{wDef} \text{ and } A \in [\![{}^{\bullet}\boldsymbol{w}]\!]\}$$

of forward translation cubes: left face is translated into the right face.

Definition 1 *A* well-defined *view system* $\mathcal{V}$ *is a graph* $\mathcal{U}$ *with partial composition carrying the structure specified above: metadata constructs* ***dataDef***, ***updDef***, ***viewDef*** *and* $\mathbb{wDef}$ *(consisting of well-defined tiles), instance mapping assigning to each schema* $\boldsymbol{\mathcal{S}}$ *its set of instances* $[\![\boldsymbol{\mathcal{S}}]\!]$, *and execution mechanisms* $\mathbb{uExe}$, $\mathbb{vExe}$ *that result in data constructs* ***updMap***, ***viewMap*** *and* $\mathbb{wExe}$*; the latter defines an update* translation execution *mechanism.*

3.3 Update policies

To set our main results about update policies, we will need the following categorical reformulation of the main definition.

Proposition 1 *A well-defined view system gives rise to a double functor $\pi\!\!\pi\colon \mathbb{I}\mathbf{nst} \to \mathbb{MD}\mathbf{ata}$ between two thin double categories.*

Proof (sketch). We need to define two thin double categories and a functor between them. A thin double category is defined by its vertical and horizontal categories, and a class of tiles. For $\mathbb{MD}\mathbf{ata}$, we take categories ***updDef*** and ***viewDef*** as $\mathbb{MD}\mathbf{ata}_1^{\mathsf{v}}$ and $\mathbb{MD}\mathbf{ata}_1^{\mathsf{h}}$ respectively. *Double arrows* are $\mathbf{w}\mathbb{D}\mathbf{ef}$-tiles, and all the necessary special (idle and identity) tiles are there (see Fig. 5). The associativity of tile compositions is immediately follows from associativity of composition in categories ***updDef*** and ***viewDef***, and idle/identity tiles are neutral units because idle/identity arrows are such.

Definition of $\mathbb{I}\mathbf{nst}$ is a bit more intricate. 0-cells are instances, $\mathbb{I}\mathbf{nst}_0 \overset{\text{def}}{=} \bigcup_{\boldsymbol{\mathcal{S}}\in\mathbb{MD}\mathbf{ata}_0} [\![\boldsymbol{\mathcal{S}}]\!]$. *Vertical arrows* are update execution tiles, $\mathbb{I}\mathbf{nst}_1^{\mathsf{v}} \overset{\text{def}}{=} \mathbf{u}\mathbb{E}\mathbf{xe}$, and they form a category by conditions in Section 3.2.4(A1). *Horizontal arrows* are view execution tiles, $\mathbb{I}\mathbf{nst}_1^{\mathsf{h}} \overset{\text{def}}{=} \mathbf{v}\mathbb{E}\mathbf{xe}$, and they form a category by conditions in Section 3.2.4(A2). *Double arrows* are synchronous cubical diagrams from $\mathbf{w}\mathbb{E}\mathbf{xe}$, $\mathbb{I}\mathbf{nst}_2 \overset{\text{def}}{=} \mathbf{w}\mathbb{E}\mathbf{xe}$. Availability of all necessary idle/identity tiles (cubes), composition and its properties (associativity and units) are immediately checked because double arrows are thin (coincide with their cubical boundaries). Hence, we get a thin double category.

Typing double functor $\pi\!\!\pi$ is nothing but a projection that maps tiles constituting $\mathbb{I}\mathbf{nst}$-arrows to their meta-sides, and maps cubes constituting $\mathbb{I}\mathbf{nst}$-tiles to their meta-faces. Compatibility of this mapping with composition and identities is evident. □

The following result is crucial but is proved by immediate checking the definitions.

Proposition 2 *The double functor $\pi\!\!\pi\colon \mathbb{I}\mathbf{nst} \to \mathbb{MD}\mathbf{ata}$ determined by a well-defined view system possesses the following inverting facilities (we write $A{:}\boldsymbol{\mathcal{S}}$ to say $\pi\!\!\pi(A){=}\boldsymbol{\mathcal{S}}$):*

(a) Update execution mechanism*, which assigns to any vertical arrow (an update definition) $\boldsymbol{u}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}}{:}{:}\mathbb{MD}\mathbf{ata}_1^{\mathsf{v}}$ and data instance $A{:}\boldsymbol{\mathcal{S}}$, an arrow $[\boldsymbol{u}]_A : A{:}\boldsymbol{\mathcal{S}} \to A'{:}\boldsymbol{\mathcal{S}}{:}{:}\mathbb{I}\mathbf{nst}_1^{\mathsf{v}}$;*

(b) View execution mechanism*, which assigns to any horizontal arrow (view definition) $\boldsymbol{v}\colon \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{S}}{:}{:}\mathbb{MD}\mathbf{ata}_1^{\mathsf{h}}$ and instance $A{:}\boldsymbol{\mathcal{S}}$, an arrow $[\boldsymbol{v}]_A : B{:}\boldsymbol{\mathcal{T}} \to A{:}\boldsymbol{\mathcal{S}}{:}{:}\mathbb{I}\mathbf{nst}_1^{\mathsf{h}}$;*

(c) Translation execution mechanism*, which assigns to any tile (translation definition) $\boldsymbol{w}{=}(\boldsymbol{v},\boldsymbol{u},\boldsymbol{v},\boldsymbol{p}){\in}\mathbb{MD}\mathbf{ata}_2$ and instance $A{:}^{\bullet}\boldsymbol{w}$, a tile*

$[\boldsymbol{w}]_A = ([\boldsymbol{v}]_A, [\boldsymbol{u}]_{A^{\boldsymbol{v}}}, [\boldsymbol{v}]_{A^{\boldsymbol{u}}}, [\boldsymbol{p}]_A) \in \mathbb{I}\mathbf{nst}_2$.

Moreover, these execution operations are compatible with compositions:

(d) $[x1;x2]_A{=}[x1]_B \;;\; [x2]_A$

where $x1, x2$ are similar cells (either similar arrows or tiles), semicolon ; is composition (vertical or horizontal), and $B = A^{x2}$. In categorical terms, it means that functors $\pi\!\!\pi_1^{\mathsf{v}}$ and $\pi\!\!\pi_1^{\mathsf{h}}$ are (split) opfibration and fibration respectively, and $\pi\!\!\pi$ is a double fibration [11].

Corollary 3 *Let $\boldsymbol{w}_{ij}\,(i = 1..m, j = 1..n)$ be a net of $\mathbf{w}\mathbb{D}\mathbf{ef}$-tiles like shown in Fig. 1. Let $\boldsymbol{w} = \sum_{ij} \boldsymbol{w}_{ij}$ denotes the entire outer tile produced by composition of the inner tiles, and $A \in [\![{}^{\bullet}\boldsymbol{w}]\!]$. Then $[\boldsymbol{w}]_A = \sum_{ij} [\boldsymbol{w}_{ij}]_{A_{ij}}$, where A_{ij} is the net of instances computed from $A = A_{11}$ ($A_{i+1,j} = A_i^{\boldsymbol{v}_i}$, $A_{i,j+1} = A_j^{\boldsymbol{u}_j}$).*

Let $\mathcal{V}$ be a well-defined view system, $\boldsymbol{v} = \boldsymbol{v}_m; \boldsymbol{v}_{m-1}; ..; \boldsymbol{v}_1 : \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{S}}$ a complex view definition in $\mathcal{V}$, and $\mathbf{U}_{\boldsymbol{\mathcal{T}}}$ is the class of basic update definitions over $\boldsymbol{\mathcal{T}}$. We remind that for any pair $(\boldsymbol{v},\boldsymbol{u}){\in}\boldsymbol{viewDef}_1{\times}\boldsymbol{updDef}_1$, there is at least one $\mathbf{w}\mathbb{D}\mathbf{ef}$-tile having $\boldsymbol{u}$ as its right edge.

Definition 2 *A* backward translation policy *for $\boldsymbol{\mathcal{T}}$ is a mapping $\boldsymbol{T}\colon \boldsymbol{U}_{\boldsymbol{\mathcal{T}}}{\times}\underline{m} \to \mathbf{w}\mathbb{D}\mathbf{ef}$ with $\underline{m} \overset{\text{def}}{=} \{1..m\}$. A policy is called* well-designed *if it ensures tight tiling (see Fig. 1), i.e., formally, $\boldsymbol{T}^u_{i+1}| = |\boldsymbol{T}^u_i$ for all $i{=}1,2,..,m{-}1$ and $\boldsymbol{T}^u_m| = \boldsymbol{u}$.*

Proposition 4 *Let $\boldsymbol{T}$ a well-designed update policy for view $\boldsymbol{v}\colon \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{S}}$ as above, and $\boldsymbol{u} = \boldsymbol{u}_1; \boldsymbol{u}_2..; \boldsymbol{u}_n$, $\boldsymbol{u}_j{\in}\boldsymbol{U}_{\boldsymbol{\mathcal{T}}}$ a complex update over $\boldsymbol{\mathcal{T}}$. Then for any instance $A{:}\boldsymbol{\mathcal{S}}$ and its view $B = \boldsymbol{v}^A$, the view update $\overline{\boldsymbol{u}_B}\colon B \to B'$ is uniquely and correctly translated back to an update $a\colon A \to A'$.*

Proof. Decompositions $\boldsymbol{v} = \boldsymbol{v}_m; \boldsymbol{v}_{m-1}; ..; \boldsymbol{v}_1$ and $\boldsymbol{u} = \boldsymbol{u}_1; \boldsymbol{u}_2..; \boldsymbol{u}_n$ form a net, and the update policy "fills" each cell of this net with a translation definition $\mathbf{w}\mathbb{D}\mathbf{ef}$-tile $\boldsymbol{T}^j_i$. Let $\boldsymbol{w} = \sum_{ij} \boldsymbol{T}^j_i$ denotes the total composition of all these tiles, $\boldsymbol{p} = |\boldsymbol{w}$ be its leftmost edge, and $a = \overline{\boldsymbol{p}_A}\colon A \to A'$ is the corresponding update. We need to prove that translation is correct, that is, $\overline{\boldsymbol{v}_{A'}} = B'$ or $[\boldsymbol{w}]_{A}.| = \overline{\boldsymbol{u}_B}$. We compute

$$\begin{aligned}
[\boldsymbol{w}]_{A}.| &= [\,\textstyle\sum_{ij} \boldsymbol{T}_i^j]_{A}.| && \text{by definition of } \boldsymbol{w}\\
&= \textstyle\sum_{ij}\, [\boldsymbol{T}_i^j]_{A_{ij}}.| && \text{by the corollary}\\
&= \textstyle\sum_{j} \left(\sum_{i}\, [\boldsymbol{T}_i^j]_{A_{ij}}\right).| && \text{by the interchnage law}\\
&= \textstyle\sum_{j} \left([\,\sum_{i} \boldsymbol{T}_i^j]_{B_j}\right).| && \text{by the corollary again}\\
&= \textstyle\sum_{j} \overline{\boldsymbol{u}_{j\,B_j}} && \text{by definition of well designed policy}\\
&= \overline{\textstyle\sum_{j} \boldsymbol{u}_{j\,B_j}} && \text{by Proposition 2(d)}\\
&= \overline{\boldsymbol{u}_B} && \text{by definition of } \boldsymbol{u}
\end{aligned}$$

Importantly, the proof shows that the result of backward translation does not depend on the way of composing tiles to obtain the total composition $\boldsymbol{w}$. This provides a space for optimization of update translations. The Pasting Lemma [13] allows us to extend the space for non-regular nets of view and update blocks.

References

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

[2] A. Bohannon, B. Pierce, and J. Vaughan. Relational lenses: a language for updatable views. In *PODS*, 2006.

[3] R. Dawson and R. Pare. General associativity and general compostion for double categories. *Cahiers de topologie et géométrie différentielle catégoriques*, 34:57–79, 1993.

[4] U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, 1982.

[5] Z. Diskin. Algebraic models for bidirectional model synchronization. In *MoDELS*, pages 21–36, 2008.

[6] Z. Diskin, K. Czarnecki, and M. Antkiewicz. Model-versioning-in-the-large: Algebraic foundations and the tile notation. In *ICSE 2009 Workshop on Comparison and Versioning of Software Models*, pages 7–12, 2009. DOI: 10.1109/CVSM.2009.5071715.

[7] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246, 2005.

[8] J. N. Foster, M. Greenwald, C. Kirkegaard, B. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *J. Comput. Syst. Sci.*, 73(4):669–689, 2007.

[9] J. N. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[10] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM TODS*, 13(4):486–524, 1988.

[11] B. Jacobs. *Categorical logic and type theory.* Elsevier Science Publishers, 1999.

[12] A. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, 1985.

[13] G.M. Kelly and R. Street. Review of the elements of 2-categories. In *Category Seminar, Sydney 1972/73*, Lecture Notes in Math., 420, pages 75–103, 1974.

[14] R. Langerak. View updates in relational databases with an independent scheme. *ACM TODS*, 15(1):40–66, 1990.

[15] E. Manes. *Algebraic Theories.* Graduate Text in Mathematics. Springer Verlag, 1976.

[16] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, Shubha U. Nabar, and Jennifer Widom. Representing uncertain data: models, properties, and algorithms. *VLDB J.*, 18(5):989–1019, 2009.

[17] P. Stevens. Bidirectional model transformation in QVT: Semantic issues and open questions. In *MoDELS*, 2007.

[18] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE*, 2007.