GSDLAB TECHNICAL REPORT

Lax Lenses

Zinovy Diskin

GSDLAB–TR 2013–03–10 March 2013

Generative Software
Development Lab

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Lax Lenses

Zinovy Diskin

Generative Software Development Lab.,
Department of Electrical and Computer Engineering,
University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

1 Introduction

This report aims to support the paper [1] with an accurate formal definition of a new family of lens structures called *lax lenses*. Their main distinction from ordinary lenses (either state-based or delta-based) is a weakened version of PutGet law. Ordinary lenses require equality of the original view and the view provided by the updated source: $A = \mathsf{get}(\mathsf{put}(A, B)$ for state-based lenses, and similarly for delta lenses. However, in many applications we only have an inclusion $A \subset \mathsf{get}(\mathsf{put}(A, B)$. Appendix provides an example (borrowed from the paper). Such transitions from strict equality to inclusion (or, more generally, a delta) have been studied in several contexts in category theory under the name of "laxity": equality is *relaxed* and becomes a delta, particularly, inclusion. Correspondingly, a construct X, whose definition requires equality, but the latter is replaced by a delta, is called *lax* X. Hence, *lax lenses*.

2 Asymmetric Lenses

Let M be be a view schema, N a source schema, and $\mathsf{get}\colon \mathsf{Inst}(\mathsf{M}) \leftarrow \mathsf{Inst}(\mathsf{N})$ a totally defined operation *get-the-view* (we direct the arrow backward to sync the presentation with examples in Section 2). Below we will not directly use metamodels as syntactical objects, and symbols M, N will denote the respective model spaces $\mathsf{Inst}(\mathsf{M})$,$\mathsf{Inst}(\mathsf{N})$.

2.1 Non-incremental BX

Let M and N be two model spaces: the latter consists of *source* models, and the former contains their *views* computed according to some view definition. We thus have a total unary operation $\mathsf{get}\colon \mathsf{M} \leftarrow \mathsf{N}$ *get-the-view* (we direct the arrow backward to sync the presentation with examples in Section 2).

A *bi-directional (BXed)* view definition provides, in addition, another total operation $\mathsf{gen}\colon \mathsf{M} \rightarrow \mathsf{N}$ (*generate-the-source-*), which is *inverse* to get in a certain sense. For simple view definitions, invertibility means equality $A = \mathsf{get}(A.\mathsf{gen})$ for any model A in M, where application of a function to an argument is denoted by two ways: for get, the argument is written on the right of the function symbol

in brackets, and for gen, the argument is written on the left of the function symbol separated by a dot; then direction of the operations in the example is respected in the linear notation and makes it more suggestive.

The equality above is a basic law of the lens framework; in the lens jargon, it should be phrased as the GenGet law. However, as the example in Appendix shows, for views defined by complex queries like relational join, we may only have subsetting $A \subseteq A.\mathsf{gen}.\mathsf{get}$. This is a general rule: due to some conditions on the implementation side, the view to the generated source contains everything specified in the original view plus, perhaps, something extra. Hence, we need a generalization of lenses, in which GenGet law would be relaxed.

Definition 1 (Model spaces). A *model space* is a partially ordered set $\mathsf{M} = (|\mathsf{M}|, \sqsubseteq_{\mathsf{M}})$, whose elements are called *models*, and relationship $A_1 \sqsubseteq_{\mathsf{M}} A_2$ is interpreted as model A_1 is contained in model A_2. We will often skip bars and subindex, if they are clear from the context.

Definition 2 (Non-incremental lenses). A *non-incremental lax lens* is a tuple $L = (\mathsf{M}, \mathsf{N}, \mathsf{get}, \mathsf{gen})$ with M and N *model spaces*, and (get, gen) a pair of monotonic (i.e., order-preserving) unary operations as above, which satisfy the lax GenGet law:
(GenGet) $A \sqsubseteq \mathsf{get}(A.\mathsf{gen})$ for all $A \in \mathsf{M}$.

In addition, to prevent infinite loops of updating, we also require two weak invertibility laws to hold:
($\mathsf{GetGenGet}$) $\mathsf{get}((\mathsf{get}(B).\mathsf{gen}) = \mathsf{get}(B)$ for all $B \in \mathsf{N}$
($\mathsf{GenGetGen}$) $(\mathsf{get}(A.\mathsf{gen})).\mathsf{gen} = A.\mathsf{gen}$ for all $A \in \mathsf{M}$

It is convenient to denote a lens by a double arrow $L^{\leq} : \mathsf{M} \rightleftharpoons \mathsf{N}$ (think of two operations), and use the upper index to point to the view space. Then $L^{\geq} : \mathsf{M} \rightleftharpoons \mathsf{N}$ will be a lens with N being the view space. That is, the upper index points to the direction of get.

2.2 Incremental BX

Discrete incrementality To save private changes on the B-side and generate B-models incrementally, we need to enrich the computational framework with a binary operation $\mathsf{put} : \mathsf{M} \times \mathsf{N} \to \mathsf{N}$ (called *put (the view update back to source)* in the lens jargon), whose second argument is to be thought of as the original source whose private data is to be preserved. As the first step towards an algebraic model of incrementality, we consider the *discrete case*, when an update from model A to model A' is just a pair (A, A') rather than delta $\Delta_{AA'}$ really specifying the update.

Definition 3 (Discrete lax lenses). A *discretely-incremental* or just a *discrete* lax lens is a pair $L^{\leq} = (L_0^{\leq}, \mathsf{put})$ with $L_0^{\leq} = (\mathsf{M}, \mathsf{N}, \mathsf{get}, \mathsf{gen})$ a non-incremental lens as above, and put a binary operation as above, which is *monotonic* in the following sense:

$A \sqsubseteq \mathsf{get}(B) \Rightarrow (A, B).\mathsf{put} \sqsubseteq B$ and $A \sqsupseteq \mathsf{get}(B) \Rightarrow (A, B).\mathsf{put} \sqsupseteq B$.

That is, an insertion (deletion) on the A-side is propagated to an insertion (deletion) on the B-side.

Below we will often write $A.\mathsf{put}_B$ for $(A,B).\mathsf{put}$.

These data must satisfy three laws:
(GetPut) $(\mathsf{get}(B)).\mathsf{put}_B = B$ for all $B \in \mathsf{N}$
(PutGet) $A \sqsubseteq \mathsf{get}(A.\mathsf{put}_B)$ for all $A \in \mathsf{M}$
(PutGetPut) $A.\mathsf{put}_B = (\mathsf{get}(A.\mathsf{put}_B)).\mathsf{put}_B$ for all $A \in \mathsf{M}$

We can consider source generation operation gen as a special case of put. Assume that every model space M has some fixed minimal model 0_M contained in any M-model A, $0_\mathsf{M} \sqsubseteq A$. We then require $0_\mathsf{M} = \mathsf{get}(0_\mathsf{N})$, and define for any model $A \in \mathsf{M}$, $\mathsf{gen}(A) \stackrel{\text{def}}{=} \mathsf{put}_{0_\mathsf{N}}(A)$. Thus, a discretely incremental lens can be defined as a tuple $L = (\mathsf{M}, \mathsf{N}, \mathsf{get}, \mathsf{put})$, where model spaces are assumed to have minimal models.

Delta-incrementality: Updates are deltas between models To model delta-incremental transformations, we make model spaces M, N directed graphs rather than sets. Arrows in these graphs are embeddings of one model into another. By the abuse of notation, we will denote the states of a model A before and after update by A and A' resp. (thus, A denotes both the model and its initial state). The diagram $A \xrightarrow{u} A'$ shows that A' is bigger than A and the update is an addition. Dually, the diagram $A \xleftarrow{u} A'$ specifies a deletion. Finally, diagram $A \xleftarrow{u_1} K \xrightarrow{u_2} A'$ specifies a modification, i.e., a deletion followed by insertion with K denoting the part of the model kept unchanged. We will call all such diagrams *deltas*.

As arrow-updates can be composed, and there is an idle (do-nothing) arrow for every model, it's reasonable to assume model spaces to be *categories*, and functional mappings between them to be *functors* (precise definitions can be found in [5]). If M is a category, $\mathsf{M}_\bullet$ denotes its set of nodes (models), and M_Δ is its set of arrows (model embeddings). Correspondingly, the node and the arrow parts of a functor $f\colon \mathsf{M} \to \mathsf{N}$ are denoted by $f_\bullet\colon \mathsf{M}_\bullet \to \mathsf{N}_\bullet$ and $f_\Delta\colon \mathsf{M}_\Delta \to \mathsf{N}_\Delta$ resp.

We also assume that a categorical model space M has a *minimal* (categoricians say *initial*) object 0_M such that for any object $A \in \mathsf{M}_\bullet$ there is a unique delta $o_A\colon 0_\mathsf{M} \to A$. Now propagation operation become functorial and act over both models and deltas.

Definition 4 (a sketch). A *delta lens* is a tuple $L^{\leq} = (\mathsf{M}, \mathsf{N}, \mathsf{get}, \mathsf{put})$ with M and N categories, $\mathsf{get}\colon \mathsf{M} \leftarrow \mathsf{N}$ a functor, and $\mathsf{put}\colon \mathsf{M}_\Delta \times \mathsf{N}_\bullet \to \mathsf{N}_\Delta$ an operation, which takes a view delta $a\colon A \to A'$ and an original source B such that $A = \mathsf{get}(B)$, and produce a source delta $b\colon B \to B'$ such that $a = \mathsf{get}(b)$ by the PutGet law, particularly, $A' = \mathsf{get}(B')$. Precise definitions and details can be found in [5].

For lax delta lenses, equality is to be relaxed, i.e., replaced by embedding $e\colon A' \to \mathsf{get}(B')$, which is, in fact, the second component of the propagation's output

Definition 5 (Lax delta lenses). A *lax delta lens* is a pair $L^{\leq} = (L_1^{\leq}, \mathsf{put}^{\Delta})$ where $L_1 = (\mathsf{M}, \mathsf{N}, \mathsf{get}, \mathsf{put})$ is a delta lens as above, and $\mathsf{put}^{\Delta}\colon \mathsf{M}_{\Delta}\times\mathsf{N}_{\bullet} \to \mathsf{M}_{\Delta}$ is an operation, which takes a view delta $a\colon A \to A'$ and an original source B such that $A = \mathsf{get}(B)$ exactly like operation put above, but produces a view delta $a' \stackrel{\text{def}}{=} a.\mathsf{put}^{\Delta}_B\colon A' \to \mathsf{get}(B')$, which matches the updated view A' with its implementation B'. We can merge put and put^{Δ} in one two-valued operation $\mathsf{Put}\colon \mathsf{M}_{\Delta}\times\mathsf{N}_{\bullet} \to \mathsf{N}_{\Delta}\times\mathsf{M}_{\Delta}$

The PutGet law now states that delta a' is a monic arrow, which is the categorical way to say that this arrow is an injection.

In addition, weak invertibility holds:
($\mathsf{GetPutGet}$) $\mathsf{get}(\mathsf{get}(b).\mathsf{put}_B) = \mathsf{get}(b)$ for any $b\colon B \to B'$
($\mathsf{PutGetPut}$) $(a; a').\mathsf{put}_B = a.\mathsf{put}_B$ for any $a\colon A \to A'$ and B, a' as defined above.

To have a unified notation for the three types of lenses we introduced, we will denote them by symbols $L^{\leq}_{\mathsf{incr}}$, where incr is incrementality index set to 0, 1/2, or 1 for non-incremental, discrete or delta lax lenses resp.

3 Symmetric lenses via asymmetric ones

We do not strive for maximal generality to define symmetric lenses abstractly as in [6,2,4]. Instead, we consider a bit less general but covering the majority of practically interesting cases setting, when symmetric lenses are built on top of asymmetric ones.

Definition 6. A *symmetric lax lens* of incremental type $\mathsf{incr} \in \{0, \frac{1}{2}, 1\}$ is a pair of lenses, the *left* one, $L^{\geq}_{\mathsf{incr}}\colon \mathsf{M} \rightleftharpoons \mathsf{S}$, and the *right* one, $R^{\leq}_{\mathsf{incr}}\colon \mathsf{S} \rightleftharpoons \mathsf{N}$, working over a shared view space S. We denote such a pair by $S^{\geq\leq}_{\mathsf{incr}}\colon \mathsf{M} \rightleftharpoons \mathsf{N}$.

The operations of forward, from M to N and backward, from N to M, update propagation are defined by composing the corresponding gets and puts. For example, for the discrete case, we have binary operations $\mathsf{fPpg}\colon \mathsf{M}\times\mathsf{N} \to \mathsf{N}$ and $\mathsf{bPpg}\colon \mathsf{M} \leftarrow \mathsf{N}\times\mathsf{M}$ defined for any $A \in \mathsf{M}$ and $B \in \mathsf{N}$ as follows: $(A, B).\mathsf{fPpg} \stackrel{\text{def}}{=} (A.\mathsf{get}_L, B).\mathsf{put}_R$ and $\mathsf{bPpg}(B, A) \stackrel{\text{def}}{=} \mathsf{put}_L(\mathsf{get}_R(B), A)$.

The same idea works for the delta-case, but now we compose operations via deltas as shown in Fig.22 of paper [3] (the reader may ignore the middle tile).

The following result is straightforward.

Proposition 1. Operations fPpg and bPpg are monotonic, and propagate identity to identity. Particularly, for the discrete case, if $A.\mathsf{get}_L = \mathsf{get}_R(B)$, then $(A, B).\mathsf{fPpg} = B$ and $A = \mathsf{bPpg}(B, A)$.

Remark 1. For incrementality index incr=1/2, very mild conditions for equivalence of an abstract definition of symmetric lenses and ours $S^{\geq\leq}_{1/2}$ are given in [7]. Stating such an equivalence for richer delta settings is an open question.

References

1. Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: Symmetrization of Model Transformations: Towards a Rational Taxonomy of Model Synchronization Types. In: Submitted to: MODELS 2013 - 16th International Conference on Model Driven Engineering Languages and Systems, Miami, Florida, USA, Sept 29 - Oct 4, 2013 (2013)
2. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS. Lecture Notes in Computer Science, vol. 6981, pp. 304–318. Springer (2011)
3. Diskin, Z.: Model synchronization: Mappings, tiles, and categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 6491, pp. 92–165. Springer (2009)
4. Diskin, Z., Maibaum, T.S.E.: Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In: Golas, U., Soboll, T. (eds.) ACCAT. EPTCS, vol. 93, pp. 1–21 (2012)
5. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology 10, 6: 1–25 (2011)
6. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. In: POPL (2011)
7. Stevens, P.: Observations relating to the equivalences induced on model sets by bidirectional transformations. ECEASST 49 (2012)

A Symmetrization via Examples

We will consider several examples of synchronization scenarios to illustrate what we mean by symmetrization.

A.1 Getting Started: "Fly with comfort!"

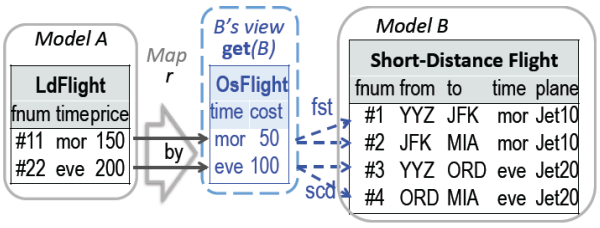

Fig. 1: Model B and mapping r implement model A

A Canadian air carrier is anticipating a significant increase in the passenger flow from Toronto to Miami due to the Models'13 conference. The company decides to organize several charter one-stop flights. Two teams are created to work on the project. The first is responsible for the marketing: the number of flights to be added, their time (morning or evening), and prices. Model A in Fig. 1 (left) is a sample model the marketing team developed. The second team is technical: it works on implementation of marketing decisions, and deals with an optimal choice of intermediate airports and airplanes to use. Having data about airports and planes, the team can compute a crucial flight parameter: cost-per-passenger, or

just cost. An example of technical model implementing model A is also shown in Fig. 1. The model consists of a base table B of short-distance flights (sd-flights)[1], and a derived table of one-stop flights (os-flights), obtained by joining sd-Flight with themselves: an os-Flight flight self is a pair of sd-flights (self.fst, self.scd) satisfying two conditions:

(Q) self.fst.to = self.snd.from and self.fst.time ≤ self.snd.time.

The first one defines relational join, and the second one assumes ordering mor< eve. We also define self.time=self.fst.time. Computing self.cost is done by some procedure using airport and airplane data. We thus have a function get (read "get the view") that computes derived table OsFlight from base table SdFlight.

Two 'by'-links relate Ld-flights to their one-stop implementations. Together they form an inter-table (sub)mapping by: LdFlight → OsFlight, which constitutes a *model-correspondence mapping* $r\colon A \to \mathsf{get}(B)$ (mapping r could contain more submappings like by, if model A would have more tables/classes). Note that implementation is a pair (B, r), but we will often say 'implementation' B, leaving the correspondence mapping implicit. We consider implementation (B, r) to be *correct*, if for each ld-flight self we have:

(C) self.time = self.by.time and self.price ≥ self.by.cost + 100.

Many different correct implementations of the same model A are possible. For example, Fig. 2 shows a variant with JFK serving as a hub for several flights. It implies that the relational join table OsFlight has extra flights, and thus mapping r is not surjective. This is a typical situation: an implementation platform normally provides many possibilities, not all of which are used in a concrete implementation.

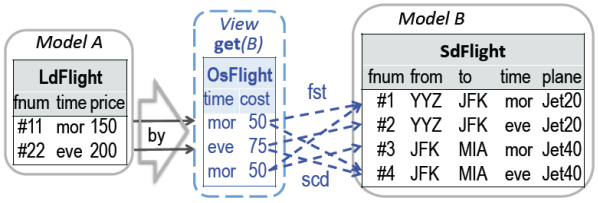

Fig. 2: Another Implementation

Also, to be precise, what is implemented is not the entire model A but its projection $\mathsf{get}^*(A)$ on the 'time' and 'price' columns. Indeed, attribute 'fnum' does not play any role in creation of model B. Figure 3 refines Fig. 2 and makes mapping 'by' totally defined. View $\mathsf{get}^*(A)$ can be more complex if there are other attributes affecting the price (wholesales, promotions, etc) but not relevant for side B.

Since model $A^* = \mathsf{get}^*(A)$ is fully determined by model A, we say that A *informationally dominates* A^* and write $A \geq_{\mathsf{inf}} A^*$. We will also say that model A has its *private* information invisible to A^*. Similarly, $\mathsf{get}(B) \leq_{\mathsf{inf}} B$. However, neither of models A and B fully dominates the other as they both have their own private information. We term this as *informational symmetry*, $A \asymp_{\mathsf{inf}} B$.

[1] We use standard airport codes: YYZ – for Pearson Int., Toronto, JFK – John F. Kennedy, New York, MIA – Miami, Florida, and ORD — Chicago O'Hare

A.2 A tour of synchronization symmetries and asymmetries.

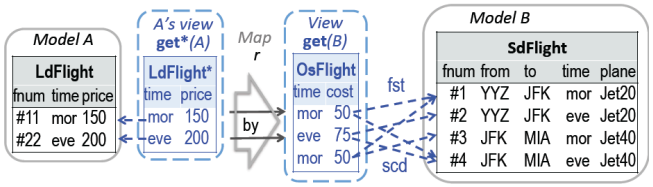

Fig. 3: Model B and mapping r implement model $\mathsf{get}^*(A)$

There may be several ways of approaching the intended extension of the schedule, and finding its best implementation. We will distinguish five basic patterns amongst them, which are consecutively considered in the next five subsections.

A.2.1 Any implementation is good enough, and the user does not care which one is chosen. For example, either of the two possibilities shown in Fig. 1, 2, or yet another one, would suit the user. This is an unlikely scenario for flight implementation, but it can be often encountered in code generation, when the user does not have an access to code. We will refer to this situation by saying that model A *strongly dominates* B *organizationally*, and write $A \gg_{\mathsf{org}} B$ (or, equivalently, $B \ll_{\mathsf{org}} A$).

A.2.2 Implementation is an asset. Now suppose that the technical team works intensively with model B, tries different variants, analyzes them, and strives to find an optimal solution. Discarding results of these efforts with every change in model A would be discouraging for side B. A much better solution is to implement changes on side A incrementally as shown in Fig. 4: the change, or *delta*, on side A, Δ_A, is propagated to a delta on side B, Δ_B, which together with the original implementation B provides an updated implementation. In the figure, solid lines and shaded tables refer to given data, and dashed lines and blank tables denote data produced by the operation of delta propagation. In more detail, Δ_A makes explicit that flight #11 is preserved and flight #22 is added. Correspondingly, delta Δ_B keeps flights #1 and #2, and adds two Sd-flights implementing the required one-stop flight. The range of possible implementations is captured by placing *labeled nulls* $?_i$ into the table: nulls with the same label must be substituted with the same value (unknown so far), nulls with different labels are independent, but may be also substituted with the same value. In this way, uncertain model B' captures the implementation in Fig. 1 with $?_1$=ORD, and that one in Fig. 2 with $?_1$=JFK, and others possibilities as well. Correspondingly, the derived OsFlight table is also uncertain: the cost value is given by applying some known procedure say, F, to unknown argument values $?_i, i = 1, 2, 3$.

Incremental propagation as described above gives model B more independence than in case A.2.1. Suppose that model B is the result of customizing some previous model B^-, i.e., the update delta Δ^- leading to B was independently produced on side B rather than propagated. We assume some policy that does allow such changes on side B if they do not affect side A, that is, original model A^-=A (consistent with B^-) and updated model B are still consistent.

Then customization on side B will be saved if A changes and the change is propagated incrementally as described by Fig. 4. However, the policy may be prohibitive towards B, if updated B^- (i.e., B) and A^- are inconsistent: then to restore consistency, the policy would require to roll back Δ^-. In other words, while update propagation from A to B is allowed, update propagation from B to A is prohibited. We will refer to this situation by saying that model A dominates B *organizationally* (but not strongly), and write $A >_{\text{org}} B$ or $B <_{\text{org}} A$. We will also term the case as *organizational asymmetry*; then we may call the case in A.2.1 *strong* organizational asymmetry.

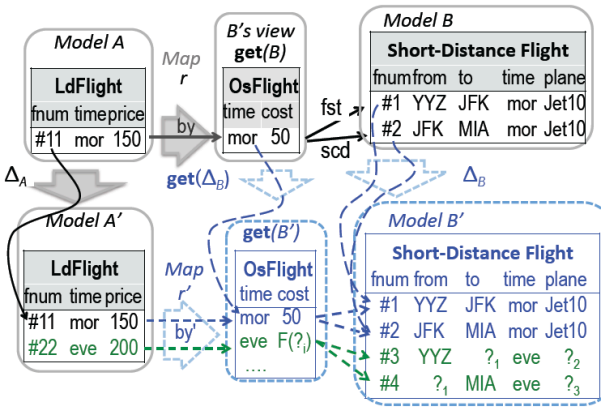

Fig. 4: Incremental implementation

A.2.3 Implementation is as important as specification: Round-tripping. Consider a different business context, in which the technical team gains a greater authority and administrative weight than before. Now, if while working with the B-model originated from model A, the team would find a good modification B', e.g., new profitable os-flights, but B' is inconsistent with A, then the B-team may require to modify model A to a state A' consistent with B'. In other words, updates now can be propagated in both directions. We will refer to the case as *organizational symmetry* of A and B and write $A \asymp_{\text{org}} B$.

A.2.4 Concurrent updates. For ap-symmetric synchronization, it is natural to allow concurrent (simultaneous) updates on the two sides. It brings two new challenges. First, independently updated models are to be matched (aligned) for specifying the mapping r. For example, if several new ld-flights appeared on side SA and, independently, several new os-flights are added on side B, we need to establish correspondences between them. In simple cases, a one hour meeting of the teams would be enough. In more complex cases (big industrial models comprising thousands of elements, teams with different background, each one working on several projects etc), matching should be done automatically with use of various heuristics and AI algorithms. Second, after mapping r is discovered, we may find that the two concurrent updates are in conflict (i.e., condition (C) on p.A.1 in violated). To resolve the conflict, we need to change either model A or B, or both. We will write $A \curlywedge_{\text{org}} B$ for this case; the symbol $\curlywedge$ is meant to recall divergence.

A.2.5 In-between organizational asymmetry and symmetry. Assume that model B is a database of SdFlights, and A is its materialized view comprising *all* OsFlights. Of course, the attribute 'fnum' is to be skipped, and mapping 'by' is bijective: $A \cong \mathsf{get}(B)$.

In our previous examples, view A was *prescriptive* in the sense that model B was thought of as an implementation of A. Now we consider view A as an ordinary *descriptive* view on the data source B as is typical for databases. Yet we still want to allow the user to update the view, say, to state A', and propagate the update back to the source. The problem is that there are many states B' such that $\mathsf{get}(B') = A'$, but we cannot arbitrarily choose one of them: updated state A' reflects an updated state of the world, which is, in turn, reflected in the unique updated source B' (recall that B is a view of the real world data). In other words, choosing an arbitrary update policy does not work anymore.

If the view update is insertion, we can manage the uniqueness problem by filling unknown slots with labeled nulls (Section 2.2.2). Suppose, however, that the view update is a deletion, for example, ld-flight #11 in Fig. 4 is deleted. This deletion can be caused by a real world deletion of sd-flight #11.by.fst: YYZ–JFK, or #11.by.scd: JFK–MIA, or both. We cannot arbitrarily choose one of them, because objects of class SdFlight must represent actual flights existing in the schedule. Hence, deletions in the view A must be prohibited.

Thus, some of view updates are propagatable, and some are not; however, any view update is reachable from the source side B. The case is thus more symmetric than organizational asymmetry in 2.2.2, but less symmetric than roundtripping in 2.2.3 We will term the situation as organizational *semi-symmetry* (or *partial* round-tripping) and write $A \leq_{\mathsf{org}} B$.

A dual semi-symmetry case, $A \geq_{\mathsf{org}} B$ is also possible. It means that all view updates are propagatable, but only some of the source updates are allowed. For example, we can imagine a propagation policy when sd-flight updates changing costs, and hence possibly affecting prices, are propagatable, whereas updates that imply deletions or additions of ld-flights are prohibited.