Bidirectional Transformations - Exercises

Harald König
FHDW Hannover / HVL Bergen
harald.koenig@fhdw.de

March 23, 2021

Exercises marked with $(*)$ are a bit harder and require a little extra contemplation. Exercises marked with $(**)$ are harder and are optional.

Exercise 1: Below, you find a list of transformations. For each case you will have to decide in what direction you can define a get and in which direction you require a put. Maybe it is also sometimes possible to define a get in both directions or the lens may even be symmetric? You should also propose an implementation for get and put.

a) Pairs of numbers (Int,Int) $\leftrightarrow$ their product (Int)

b) Person entities with firstName and lastName $\leftrightarrow$ Persons with name

c) Sets (unordered, unique) $\leftrightarrow$ lists (ordered, non-unique)

Solution to Exercise 1:

a) Clearly $\mathsf{get}(x, y) := xy$, but put needs some effort: We define $(x', y') := \mathsf{put}((x, y), z)$ by approximating x with x' in the best way: Let x' be a factor of z' for which $|x' - x|$ is minimal, then define $\mathsf{put}((x, y), z) = (x', z'/x')$.

b) An easy way would be to have two get's: $\mathsf{get}(fn, ln) := \mathsf{concat}(\mathsf{concat}(\mathsf{fn}, \text{' '}), \mathsf{ln})$, $\mathsf{get}(n) := n.splitAt(\text{' '})$, if there is only one ' ' in n. Otherwise one needs to decide which ' ' to take for splitting. If we know that there is exactly one last name after some first names, we take the last one. Otherwise an automatic back propragation is difficult and probably needs user support.

c) Here we have a get from lists to sets due to informational asymmetry, because ordering is private for lists. Let $\mathbb{S}$ be the collection of all lists and $\mathbb{V}$ be the collection of all sets. Then get : $\mathbb{S} \to \mathbb{V}$ takes a list and transforms it into a set by iterating over the list and adding each element to the output set. If, during iteration, a list element has been added before, it is not added again.

Vice versa, we need put : $\mathbb{S} \times \mathbb{V} \to \mathbb{S}$, because given a set in $\mathbb{V}$, in order to preserve the ordering, one needs to check for each element in the set, whether it is already present in the list. If not, one adds it according to the ordering. Moreover, each element not in the set has to be removed from the list. In either case, we need the previous state of the list.

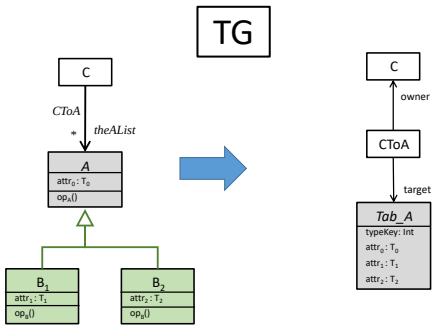


Figure 1: Object Relational Mapping

Exercise 2: We assume class and data models to be structured according to the meta-models from slide 32 and we consider the following object-relational mapping from class models to data models (TG, see Figure 1):

- Each association with target a class is transformed into a (linking) table with two foreign key columns, e.g. table CToA in Figure 1.
- Each class inheritance hierarchy is mapped to exactly one table with an extra column, in which an identifier for the concrete class is stored. Values in columns may have null values, if they don't belong to the identified class.
- Associations with target a PrimType (aka attributes) are stored as usual columns.

- Of course operations are not mapped to the data model.

Primary keys are not shown in the figure and foreign keys are depicted as arrows between the tables.

In order to keep both sides in sync after a change of a class model or a data model, would you propose to use a symmetric or an asymmetric lens? How do you implement the $\overrightarrow{R}/\overleftarrow{R}$, the put/get, resp.? If implementations of some restorers are not unique, you must only give one reasonable proposal.

Solution to Exercise 2:
In general one needs a symmetric lens because each time a data model has to be generated from a class model, we'd like to keep the name and type of primary keys from the previous version. To make things a little bit simpler, we will nevertheless use get : $\mathbb{S} \to \mathbb{V}$ and put : $\mathbb{S} \times \mathbb{V} \to \mathbb{S}$, where $\mathbb{S} = TypedGraphs^{!}(CMM)$ and $\mathbb{V} = TypedGraphs^{!}(DMM)$ constantly generating a primary key column with name *id* and type *Int* in each table.

Then get is just as described in the exercise. Let's investigate possibilities for put. Given class model $s \in \mathbb{S}$ and a data model $v \in \mathbb{V}$. For put, we have to analyse the differences of s and v by distinguishing the types in v. We call associations, whose target is a PrimType, an *attribute*.

1. Let t be a table in v, whose name does not start with "Tab" and which is not a linking table. If there is no class with this name, add a class to s with associations according to the respective relation tables and foreign keys and attributes according to columns' names and types. If there is a class with that name, adjust associations and attributes accordingly.

2. If $t.name$ ="Tab_A" and there is a corresponding class A, remove attributes and associations in class A and subclasses, if there is no corresponding column in Tab_A, add new attributes and associations always in A.

3. If $t.name$ ="Tab_A" and there is no corresponding class A, do the same as in 1), but make it abstract (this hints the modeler at the fact that the new table shall serve as data storage for objects of classes in a class hierarchy).

4. New or missing PrimTypes as well as mismatch of column's types must exactly be adjusted in s.

5. Alignment of linking tables is carried out by similar comparison with associations (with target a class). There is consistency, if for each association with target class B, there is a linking table with target pointing to the table representing the topmost class in the class hierarchy, to which B belongs to. Make sure to have steps 1 - 4 carried out before step 5.

6. Each class/association whose get-image is not present in v must then be removed from s.

Exercise 3: About the composer example:

a) Justify correctness and hippocraticness for the implementations of $\overrightarrow{R}$ and $\overleftarrow{R}$ on Slide 40.

b) Let (A_1, A_2) be a consistent pair and $A_1 \to A_1'$ be an update. Describe the requirement given by the following equation:

$$\overrightarrow{R}(A_1, \overrightarrow{R}(A_1', A_2)) = A_2$$

and check whether it is satisfied in the composer example.

Solution to Exercise 3:

a) An easy task: The first step (adding to A_2) yields

(name,nationality)-pairs of $A_1 \subseteq$ (name,nationality)-pairs of A_2,

the second step (deleting from A_2) yields $\supseteq$ between these sets.

b) The requirement says that in a consistent state the two updates $A_1 \to A_1'$ and $A_1' \to A_1$ yield $A_2 \in \mathfrak{M}_2$, i.e. each update must be *undoable*! Unfortunately, it is not satisfied in the composer example. This can be seen using the same argumentation as in Slide 52: Removal of a composer in $\mathfrak{M}_1$ and subsequent adding of the same data cannot restore the interim removed private data in $\mathfrak{M}_2$.

Exercise 4: A mathematical example: We consider the following asymmetric lens: $\mathbb{S}$ contains pairs (X, Y) of arbitrary sets X and Y. $\mathbb{V}$ just contains all sets. Furthermore let $\mathsf{get}(X, Y) := X \cap Y$ and

$$\mathsf{put}((X, Y), Z') := (Z' \cup (X - Y), Z' \cup Y)$$

a) Why is information asymmetrically distributed?

b) Is the lens hippocratic and correct?

c) $(*)$ Is it history ignorant?

Solution to Exercise 4:

a) Because we can always calculate the intersection of two sets, but given a set Z, being the intersection of some two sets, the information, which elements *outside* Z also belong to one of the two sets, is not present in $\mathbb{V}$.

b) Hippocraticness:

$$\mathsf{put}((X,Y), X \cap Y) = ((X \cap Y) \cup (X - Y), (X \cap Y) \cup Y) = (X,Y),$$

because an element $x \in X$ is either in Y or not (equality of first component) and each $y \in Y$ is in X or not (second component).
Correctness:

$$\mathsf{get}(\mathsf{put}((X,Y), Z') = \mathsf{get}(Z' \cup (X - Y), Z' \cup Y)) = (Z' \cup (X - Y)) \cap (Z' \cup Y).$$

The latter set equals Z':
"$\subseteq$": For z contained in this set there are four cases ($z \in Z'$ or $z \in X - Y$) combined with ($z \in Z'$ or $z \in Y$), of which the case $z \in X - Y \wedge z \in Y$ is impossible. In the three other cases $z \in Z'$.
"$\supseteq$" holds, because both sets in the intersection are a superset of Z'.

c) No! There is the following counterexample: Given consistent state

$$(X_0 = \{1\}, Y_0 = \emptyset), Z_0 = \emptyset.$$

Adding 2 to Z_0 yields new consistent state

$$(X_1 = \{1,2\}, Y_1 = \{2\}), Z_1 = \{2\}.$$

Removing 2 again from Z_1 results in

$$(X_2 = \{1\}, Y_2 = \{2\}), Z_2 = \emptyset,$$

because put preserves Y as best as possible. On the other hand a direct update $Z_0 \to Z_2$ yields $(X_2' = X_0, Y_2' = Y_0 \neq Y_2)$ by hippocraticness.

Exercise 5: The special setting $\mathbb{S} \cong \mathbb{V} \times C$ for some set C with

$$\mathsf{get}(v, c) = v \text{ and } \mathsf{put}((v, c), v') = (v', c) \tag{1}$$

for an asymmetric lens was dubbed "the constant complement setting". We pointed out that such a lens is *history ignorant* (the put-put-law).

a) Is a constant complement decomposition of $\mathfrak{M}_1$ into $\mathfrak{M}_2$ and C possible for the composer example on slide 52? (If yes, by the above statement, there would be a very simple implementation of put!)

b) Let in the constant complement setting $\mathbb{V}$ be the collection of all class models, elements of $\mathbb{S}$ the resp. generated code (e.g. by EMF), which has regions (elements in C) that can manually be enhanced and are protected during re-generation. Describe the practical action of get, put in this case, especially why manual enhancements are really protected. What means history ignorance in this case? Why is the put still problematic w.r.t. syntax errors?

c) (∗∗) History ignorance is a nice feature, because one needs to synchronize only once in the end after several updates, saving a lot of performance. The question is, for how many situations this feature holds. By the above statement it holds at least for the lenses with constant complement property.

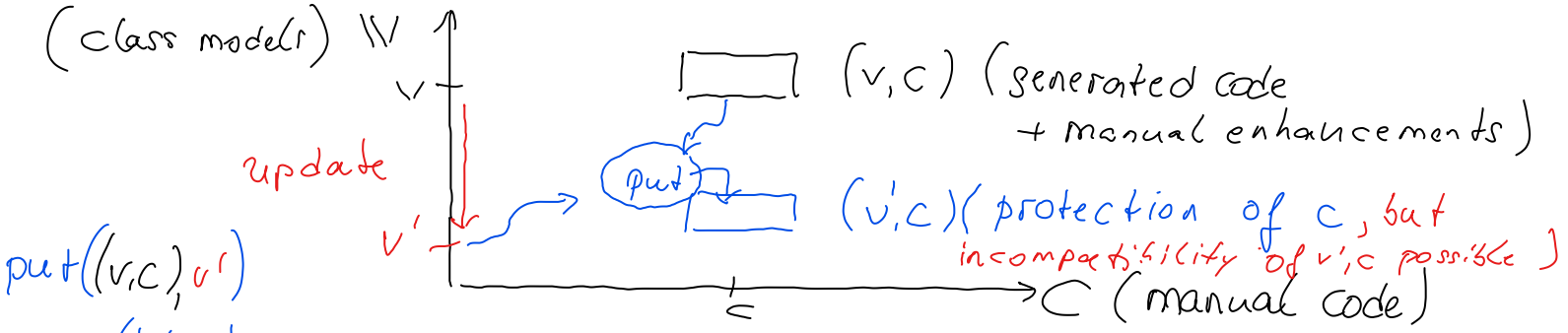
   Show that – unfortunately – no other asymmetric lens L as the ones, for which the constant complement property holds, are history ignorant, i.e. the converse of the above statement holds:

$$L \text{ is history ignorant } \Rightarrow L \text{ must have the form (1)}$$

   Hint: For $v \in \mathbb{V}$ define $\mathbb{S}_v = \{s \in \mathbb{S} \mid get(s) = v\}$ and show that for any $v_1, v_2 \in \mathbb{V}$ the function $\mathsf{put}(s, v_2) : \mathbb{S}_{v_1} \to \mathbb{S}_{v_2}$ is bijective by the put-put-law. From this, you can determine C: Look at the modulo arithmetic example and find the bijection $\mathbb{S} \cong \mathbb{V} \times C$.

Solution to Exercise 5:

a) Assume it would be possible, then the variant of the composer scenario would be history ignorant, which is false by the considerations on slide 52.



b) We may assume that there is a 1:1-correspondence between class models and their generated code (an element of some space $\mathbb{V}'$), i.e. $\mathbb{V} \cong \mathbb{V}'$ with $gen : \mathbb{V} \to \mathbb{V}'$ being the bijection. Then $\mathbb{S} \cong \mathbb{V} \times C$ and get forgets the contents of the protected regions, i.e. $\mathsf{get}(genCode, manual) = gen^{-1}(genCode)$. $\mathsf{put}((gen, manual), mod) = (gen(mod), manual)$ preserves the manually added code, but there may be syntax errors because in the manually added code features of a class may have been used, which are not present any more in new model mod.

c) First, by correctness

$$\mathsf{get}(\mathsf{put}(s, v_2)) = v_2,$$

whenever $s \in \mathbb{S}_{v_1}$, i.e. $f := \mathsf{put}(\_, v_2)$ indeed maps into $\mathbb{S}_{v_2}$ by the definition of $\mathbb{S}_{v_2}$. That f has invserse $g := \mathsf{put}(\_, v_1)$ follows from history ignorance, i.e. the putput-law: For all $s \in \mathbb{S}_{v_1}$, i.e. for all s with $\mathsf{get}(s) = v_1$:

$$g(f(s)) = \mathsf{put}(\mathsf{put}(s, v_2), v_1) \overset{!}{=} \mathsf{put}(s, v_1) = \mathsf{put}(s, \mathsf{get}(s)) = s$$

the last equality by hippocraticness. Thus $g \circ f = id_{\mathbb{S}_{v_1}}$. In the same way, one proves $f \circ g = id_{\mathbb{S}_{v_2}}$, such that for all $v_1, v_2 \in \mathbb{V}$, we obtain $S_{v_1} \cong S_{v_2}$. We choose any of these sets and call it C, then we can define a bijection $h : \mathbb{V} \times C \to \mathbb{S}$ as follows. Let $v \in \mathbb{V}$ and $c \in C$ be given, then since $C \cong \mathbb{S}_v$, there is a unique counterpart $s \in S_v$ of c. We let $h(v, c) := s$. Since all sets $(\mathbb{S}_v)_{v \in V}$ are mutually disjoint, it can easily be seen that this yields a bijection. Hence

$$\mathbb{S} \cong \mathbb{V} \times C$$

as desired.