Bidirectional Transformations - Exercises

March 19, 2021

Exercise 1

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

$Get : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
$Get((x, y)) = x * y,$

$Put : (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$
Put((factor1, factor2), product) =
if (product % factor2 == 0) return (product/factor2, factor2);
if (product % factor1 == 0) return (factor1, product/factor1);
return (product, 1);

b) `Person` entities with `firstName` and `lastName` $\leftrightarrow$ `Persons` with `name`[1]:

$Get : PersonEntity \rightarrow Person$
Get(pE) = new Person(pE.firstName + " " + pE.lastName)

$Get : Person \rightarrow PersonEntity$
Get(p) = new PersonEntity(
p.name.substring(0, p.name.indexOf(" "),
p.name.substring(p.name.indexOf(" "));

We assume that we do not store any middle names. Complex last names with white spaces in between are allowed. The method indexOf(" ") looks for the first occurrence of a whitespace, which must separate first name and last name. So, we have the same information in both models, only in a different form.

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

$Get : List \rightarrow Set$
Get(list) = new Set(list)
Start with an empty set and add all the list items to it.

$Put : List \times Set \rightarrow List$
Put(list, set) = Copy the whole list and check the set for elements not already contained in the list. Add these elements in a deterministic way at the end of the copied list. Return the copied list.

[1] I assume `name` is `firstName` and `lastName` combined.

Exercise 2

I propose an asymmetric lens:

$Get : CD \rightarrow DS$

$Put : CD \times DS \rightarrow CD$

CD stands for class diagram and DS for database schema.

To implement **Get**, we use the OR-Mapping defined in the exercise. Additionally, we add a primary key with the name *id* and type *integer* (use a fixed-length) for every generated table.

Since we have a deterministic way to calculate the only information labeled as private on slide 32, we get an asymmetric lens. However, if we were to include stored procedures, triggers, or other database-specific things in the metamodel, we would have to use a symmetric lens.

To implement **Put**, we need to do the reverse OR-Mapping from a database schema to a class diagram ignoring each table's primary key. For each table, we have to check if there exists an inheritance hierarchy in the old class diagram. If so, we keep it and always add new attributes/associations from new DB-columns/linking tables to the hierarchy's root. If the table does not match any class, we create a new class with all the attributes/associations for the DB-columns/linking tables.

Nitpicking: Figure 1 is probably only shown for illustrative purposes, but there are no operations, methods, and multiplicities in the class diagram metamodel.

Exercise 3:

a.) Correctness:

$(A_1', \vec{R}(A_1', A_2)) \in C$

Let A_1' be a set of composers typed in m_1:

$\vec{R}(A_1', A_2)$ has to contain a composer with the same name and nationality as in A_1'.

Let $a_1 \in A_1'$. Why do we have a $a_2 \in A_2$ with the same name and nationality?

Case 1: $a_2 \notin A_2$. But then the object gets added.

by $\overrightarrow{R}$.

Case 2: $a_2 \in A_2$. $\overrightarrow{R}$ keeps a_2.

Let $a_2 \in \overrightarrow{R}(A_1', A_2)$. Why is there a matching $a_1 \in A_1$?

a_2 was either added or kept by $\overrightarrow{R}$ (case 1 and case 2, above. All other $a_2 \in \overrightarrow{R}$ were deleted.

So consistency holds for $\overrightarrow{R}$.

$(\overleftarrow{R}(A_1, A_2'), A_2) \in C$

It also holds for a similar $\overleftarrow{R}$.

Hippocraticness: $(A_1, A_2) \in C \Rightarrow \overrightarrow{R}(A_1, A_2) = A_2$

$$(A_1, A_2) \in C \Rightarrow \overleftarrow{R}(A_1, A_2) = A_1$$

For $\vec{R}$, we always end up in case 2 above.
Since there is always a consistent pair of
compresses no deletions happen afterwards.
So we get the same A_2 back ($\vec{R}(A_1, A_2) = A_2$).
Same for a similar $\overleftarrow{R}$.

b.) $(A_1, A_2) \in C$, $A_1 \to A_1'$ is an update

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

The resulting decoder

the equation describes an endo-operation. After the forward restoration is applied for the updated A_1', it should be reversible by forward restoration with the pre-update A_1. Reversible such that the old A_2 is the result of this operation.

Not the case for $\vec{R}$ in the composer example:

$A_1 = \{(\text{"Carlsen"}, 1990, \text{Norge})\}$

$A_2 = \{(\text{"Magnus"}, \text{"Carlsen"}, \text{Norge})\}$ $\Rightarrow (A_1, A_2) \in C$

$A_1' = \emptyset$ $\quad (A_1 \rightarrow A_1'$: delete Carlsen).

$\vec{\Omega}(A_1', A_2) = \emptyset$

$\vec{R}(A_1, \emptyset) = \{("", \text{"Carsten"}, \text{Norge})\}$

$A_2 \neq \vec{R}(A_1, \vec{R}(A_1', A_2))$

The requirement is not satisfied.

Exercise 4:

$Get(X, Y) = X \cap Y$

$Put((X, Y), Z') = (Z' \cup (X - Y), Z' \cup Y)$

a.) The view only shows $x \in X \cap Y$ but not elements which are only in X or Y.

b.) Hint:

$\text{Get}((x,y),z) \in C$, which means: $((x,y), \text{Get}(x,y)) \in C$, so

$z = \text{Get}(x,y)$ (1)

$\vec{R}((x,y),z) \overset{?}{=} z$

$\vec{R}((x,y),z) \overset{\text{Def. } \vec{R}}{=} \text{Get}(x,y) \overset{(1)}{=} z$

$\overleftarrow{R}((x,y),z) \overset{?}{=} (x,y)$

$\overleftarrow{R}((x,y),z)$

$= \text{Put}((x,y),z)$

$\overset{\text{Def Put}}{=} (z \cup (x-y), z \cup y)$

$\overset{(1)}{=} (GetA(X,Y) \cup (X - Y),\ GetA(X,Y) \cup Y)$

$\overset{\text{Def. Get}}{=} (X \cap Y \cup (X - Y),\ (X \cap Y) \cup Y)$

$= (X, Y)$, intuitively clear and proven

with the help of propositional logic in MPS.

Correct:

$Get(Put((X,Y),Z)) \overset{?}{=} Z$

$Get(Put((X,Y),Z))$

$= Get(Z \cup (X - Y),\ Z \cup Y)$

$= (Z \cup (X - Y)) \cap (Z \cup Y)$

$= Z$, no other overlap than the elements of Z.

(proven in MPS).

c.) $put(put((X,Y),Z),Z') \stackrel{?}{=} put((X,Y),Z')$

$put((X,Y),Z') = (Z' \cup (X-Y), Z' \cup Y)$

$put(put((X,Y),Z),Z')$

$= put((Z \cup (X-Y), Z \cup Y), Z')$

$= (Z' \cup ((Z \cup (X-Y)) - Z \cup Y), Z' \cup Z \cup Y)$

looking at the second component in the tuple:

$Z' \cup Y \overset{?}{=} Z' \cup Z \cup Y$. If $Z \neq \emptyset$, $Z \neq Z'$ and $Z \neq Y$

the second component is different!

It is not history ignorant.

Example: $X = Y = Z' = \emptyset$, $Z = \{a\}$

$$mut((\emptyset, \emptyset), \emptyset) = (\emptyset, \emptyset)$$

$$mut(mut((\emptyset, \emptyset), \{a\}), \emptyset)$$

$$= mut((\{a\} \cup (\emptyset - \emptyset), \{a\} \cup \emptyset))$$

$$= mut((\{a\}, \{a\}), \emptyset)$$

$$= (\emptyset \cup (\{a\} - \{a\}), \emptyset \cup \{a\})$$

$$= (\phi, \{a\}) \neq (\phi, \phi) .$$

Exercise 5

a) $\mathbb{S} = Name \times Nationality \times YearOfBirth$
$\mathbb{V} = Name \times Nationality$

YearOfBirth is a candidate for the complement type C. The implementation of put on slide 52 would then be the one on slide 52. However, the complement YearOfBirth is *not* constant in the given implementation since it is set to `0000` if addition was detected in the view. It is also altered if elements are deleted from the view. This violates history-ignorance and the put-definition as stated in (1).

The problem is that the consistency definition makes it impossible to keep the YearOfBirth of old entries around, i.e., a *constant* complement.

b) `Get`: Updates the class model so that it reflects the EMF-code. For example, new classes, associations, and attributes in the EMF-code are added in the class diagram.

`Put`: Regenerates the EMF-code bases on the class diagram. However, the generation does not change code inside protected regions.

We can see in (1) that the put-function does not touch the complement c. Since manual enhancements are part of the complement, they will never be altered if (1) holds.

History ignorance means regenerating the EMF-code after each change in the class diagram or after a sequence of changes will result in the same code, including identical content in the protected regions.

We could, for example, rename an association in the class diagram, which would trigger syntax errors in protected regions that use them since they are untouched by `put`.