Solving the TTC Families to Persons Case with FunnyQT

Tassilo Horn
tsdh@gnu.org

The GNU Project

Abstract

This paper describes the FunnyQT solution to the bidirectional TTC 2017 Families to Persons transformation case. The solution is really simple and concise and passes all batch transformation and some of the incremental tests.

1 Introduction

This paper describes the FunnyQT[1] [Hor16, Hor15] solution of the TTC 2017 Families to Persons case [ABW17]. With only 52 lines of declarative code, the solution is able to pass all batch transformation tests and some of the incremental tests. The solution project is available on Github[2], and there's a SHARE demo[3] which includes both the transformation source code and the *benchmarx*[4] testing framework with the integrated FunnyQT solution.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[5]. Queries and transformations are Clojure programs using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSL) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF [**?**] and JGraLab [6] TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases, e.g., model management, functional querying, polymorphic functions, relational querying, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and some more. For solving the families to persons case, its bidirectional transformation and relational model querying DSLs have been used.

2 Solution Description

This section explains the FunnyQT solution. First, section 2.1 introduces the basic syntax and semantics of its embedded bidirectional transformation DSL. Thereafter, 2.2 explains the actual transformation solving the case. Lastly, 2.3 explains how the solution is integrated into the *benchmarx* framework.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1] `http://funnyqt.org`
[2] `https://github.com/tsdh/ttc17-families2persons-bx`
[3] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_BenchmarX_FunnyQT.vdi`
[4] `https://github.com/eMoflon/benchmarx/`
[5] `http://clojure.org`
[6] `https://github.com/jgralab/jgralab`

2.1 FunnyQT's Bidirectional Transformation DSL

FunnyQT's bidirectional transformation DSL is based on its relational model querying DSL which in turn is based on the Clojure port of *miniKanren*[7] which is called *core.logic*[8]. FunnyQT is able to generate a relational querying API for a given metamodel. Using that, a custom relation like the following can be written. In there, the namespace alias `ccl/` denotes *core.logic* constructs and the alias `f/` denotes relations generated by FunnyQT.

```
(defn same-named-mother-and-childo [f family mother child name]
  (ccl/all
   (f/Family f family)
   (f/->mother f family mother)
   (f/name f mother name)
   (ccl/conde
    [(f/->daughters f family child)]
    [(f/->sons f family child)])
   (f/name f child name)))
```

A relation like `same-named-mother-and-childo`[9] is defined as a plain function with a name and a argument vector. By convention, all generated relations require the model being queried as first argument, i.e., `f` denotes a family model. The relation describes a `family` with a `mother` having some `name`. Additionally, the `family` has a daughter or son[10] `child` which happens to have the same `name` as the mother.

An application of such a relation (called a *goal*) delivers all possible solutions, i.e., all tuples of a family, its mother, a child of the family, and a name which is the name of both mother and child. Any parameter can be both input and output. If it is *fresh* (not bound already), it'll be bound to every possible solution one after the other. If it is *ground* (already bound to some concrete value), it restricts the sequence of possible solutions of the remaining fresh variables.

Based on the relational model querying API described above, FunnyQT provides a bidirectional transformation DSL. A bidirectional transformation is defined with the `bx/deftransformation`[11] macro, it has a name, and an argument vector.

```
(bx/deftransformation something2anything [st at]
  ;; t-relations...
  )
```

The first and second argument always denote the models the transformation handles. In the example, we say that `st` is the *left* model and `at` is the *right* model[12].

Such a transformation definition gets compiled to a plain Clojure function which receives the left and the right models, then an additional argument denoting the direction in which the transformation is to be executed, and then any additional arguments the transformation might declare (none in the example). Thus, it can simply be executed like shown in the next listing.

```
(something2anything my-left-model my-right-model :right)
;; Valid directions:
;; :right: enforce in the direction of the right model
;; :left:  enforce in the direction of the left model
;; :right-checkonly: check if the right model corresponds to the left
;;         Returns trace information for any t-relation showing both
;;         elements that could and could not be related to elements
;;         in the other model.
;; :right-checkonly: check if the left model corresponds to the right
```

A bidirectional transformation consists of named transformation relations which define correlations between elements in the left model and elements in the right model using a `:left` and a `right` clause, each being a vector of relational goals forming an implicit conjunction. All goals have to succeed for a valid correlation to be established.

[7] `http://minikanren.org/`

[8] `https://github.com/clojure/core.logic`

[9] As miniKanren was originally implemented for Scheme which doesn't require a namespace system, it has become a convention to suffix relations with "o" to disambiguate them from functions.

[10] `ccl/conde` is a disjunction where both clauses may succeed.

[11] In all listings, the namespace alias `bx` prefixes constructs from FunnyQTs bidirectional transformation namespace.

[12] For a truly bidirectional approach, I think the terminology *source*/*target* models and transforming in *forward* or *backward* direction is at odds. FunnyQT speaks of a left and a right model and transforms in the direction of either of those.

```
(any2some
 :left  [(s/Some st ?some)
         (s/name st ?some ?value)]
 :right [(a/Any at ?any)
         (a/name at ?any ?value)])
```

The `any2some` transformation relation describes that an element `?some` of type `Some` in the left model corresponds to an element `?any` of type `Any` in the right model in case the `name` attribute of both elements has the same `?value`. In a transformation relation, all logic variables are prefixed with a question mark. The variables `st` and `at` aren't because they are the transformation's input arguments, the left and the right model.

Transformation relations have a forall-there-exists semantics, i.e., when the example transformation is enforced in the direction of the right model, it ensures that for every `Some` element there will be an `Any` element with the same name. If there are multiple `Some` elements with the same name, then just one `Any` element with that name will suffice.

Transformation relations may also have preconditions defined as a `:when` clause. Like `:left` and `:right`, it is a vector of relational goals forming a conjunction. However, whereas either the `:left` or `:right` clause may lead to creation or modification of elements depending on the direction the transformation is executed in, the `:when` clause is always exectued in check-only mode. It is a typical place to define goals which query the transformation's trace model or perform computations on plain values like translating between ages and years of birth.

To define the control flow inside a transformation, there are two mechanisms available. First, at least one t-relation must be annotated with `^:top` metadata. These are executed implicitly in declaration order. Secondly, a t-relation may have a `:where` clause containing arbitrary code. Usually, this clause is used to call other t-relations with elements matched or created by the calling t-relation.

The FunnyQT bidirectional transformation DSL has many more features not discussed in this short intro like inheritance between transformations and extension of t-relations. Some of them are used and described in the actual transformation below. For the others, refer to [Hor16] and the documentation linked from `http://funnyqt.org`.

2.2 The Families to Persons Transformation

In this section, the actual FunnyQT transformation is going to be discussed. As a first step, relational querying APIs for the two metamodels are generated.

```
(rel/generate-metamodel-relations "metamodels/Families.ecore" f)
(rel/generate-metamodel-relations "metamodels/Persons.ecore" p)
```

This makes the relations for the families metamodel available with the namespace alias `f` and those of the persons metamodel with alias `p`.

Next, we define some helper relation which defines the possible kinds of relationships between a `family` and a family `member` depending on if we prefer to create parents over creating children (parameter `pref-parent`). This is a higher-order relation in that the two remaining parameters are a parent relation `prel` (either `f/->father` or `f/->mother` has to be given) and a child relation (either `f/->daughters` or `f/->sons` has to be given).

```
(defn relationshipo [pref-parent f family member prel crel]
  (ccl/conda
   [(bx/target-directiono :right)     ;; (1)
    (ccl/conde
     [(prel f family member)]
     [(crel f family member)])]
   [(bx/existing-elemento? member)]   ;; (2)
   [(ccl/== pref-parent false)        ;; (3)
    (crel f family member)]
   [(bx/unseto? f family prel member) ;; (4)
    (prel f family member)]
   [(crel f family member)]))         ;; (5)
```

`ccl/conda` is like a short-cutting logical OR. The n-th clause is only tried if all preceeding clauses fail[13]. The first clause succeeds when we are transforming into the direction of the right model, i.e., the person register. In this case, `member` may be in a parental role of family (`prel`), or it might be in a child role (`crel`). We don't really care but want to ensure that all members of the given `family` are reachable, thus we use a non-short-cutting `ccl/conde`. All other clauses deal with transforming in the direction of the family model.

[13] In contrast to `ccl/conde` which gives every clause a chance to succeed.

The second clause deals with the case where `member` is an already existing element, i.e., not coming into being by the current execution of the transformation. Here, we assume that this member is already properly assigned to a family, so we simply succeed without doing anything.

In clause three, if we do not prefer assigning to parental roles, then the child relation `crel` must succeed between the `family` and the `member`.

In the fourth clause, if the `family`'s parental role is still unset or already assigned to `member`, then the parental relation must succeed between the `family` and the `member`.

Lastly, if no clause has succeeded until now, then the child relation has to succeed. Since a family can have an arbitrary number of children, this goal can always succeed.

In the following, the actual transformation definition is explained. It starts with the following.

```
15 (bx/deftransformation families2persons [f p prefer-parent prefer-ex-family]
16   :delete-unmatched-target-elements true
17   :id-init-fn bx/number-all-source-model-elements
```

The transformation's name is `families2persons` and it declares four parameters. The parameter `f` is the family model (the left model), `p` is the persons model (the right model), `prefer-parent` is a boolean flag determining if we prefer creating parents to creating children, and `prefer-ex-family` is a boolean flag, too, determining if we prefer re-using existing families over creating new families for new family members.

By default, bidirectional FunnyQT transformations will never delete elements from the current target model, i.e., the model in whose direction the synchronization is performed. The reason for that behavior is that it allows to run the transformation first in one direction and then in the other direction in order to perform a full synchronization where missing elements are created in each of the two models. Thus, after running a transformation, e.g., in the direction of the right model, it is only ensured that for each element (considered by the transformation's rules), there is a corresponding counterpart in the right model. However, the right model might still contain elements which have no counterpart in the left model. With option `:delete-unmatched-target-elements`set to `true`, this behavior is changed. Elements in the current target model which are not required by the current source model and the transformation relations are deleted.

The next option, `:id-init-fn`, has the following purpose. In this transformation case, family members and persons don't have some kind of unique identity. For example, it is allowed to have two members named Jim with the same name in the very same family Smith. With FunnyQT BX transformations' forall-there-exists semantics, it would suffice to create just one person in the right model with the name set to "Smith, Jim". However, the case description mandates that we create one person for every member and vice versa, no matter if they can be distinguished based on their attribute values. For such scenarios, FunnyQT's bidirectional transformation DSL provides a concept of synthetic ID attributes. The value of `:id-init-fn` has to be a function which returns a map from elements to their synthetic IDs. The built-in function `bx/number-all-source-model-elements` returns a map where every element in the source model gets assigned a unique integer number. These synthetic IDs are then used in a transformation relation which is discussed further below.

The first transformation relation, `family-register2person-register`, transforms between family and person registers.

```
18   (^:top family-register2person-register
19    :left  [(f/FamilyRegister f ?family-register)]
20    :right [(p/PersonRegister p ?person-register)]
21    :where [(member2female :?family-register ?family-register :?person-register ?person-register)
22            (member2male :?family-register ?family-register :?person-register ?person-register)])
```

It is defined as a top-level rule meaning that it'll be executed as the transformation's entry point. It's `:left` and `:right` clauses describe that for every `?family-register` there has to be a `?person-register` and vice versa. We assume that there's always just one register in each model.

The `:where` clause defines that after this relation has been enforced (or checked in checkonly mode), then the two transformation relations `member2female` and `member2male` have to be enforced (or tested) between the current `?family-register` and `?person-register`[14].

The next transformation relation, `member2person`, describes how family members of a family contained in a family register in the left model correspond to persons contained in a person register in the right model. As

[14]Transformation relations are called with keyword parameters. The two calls in the `:where` clause say that the current `?family-register` will be bound to the logic variable with the same name in the called relation, and the same is true for the `?person-register`.

can be seen, there's no goal describing how the `?family` and the `?member` are connected in the `:left` clause, and in the `:right` clause we're dealing just with a `?person` of class `Person` which is abstract. As such, this relation is not sufficient for the complete synchronization between members in the different roles of a family to females and males. Instead, it only captures the aspects that are common in the cases where mothers and daughters are synchronized with females and fathers and sons are synchronized with males. Therefore, this relation is declared abstract.

```
23  (^:abstract member2person
24   :left  [(f/->families f ?family-register ?family)
25           (f/Family f ?family)
26           (f/name f ?family ?last-name)
27           (f/FamilyMember f ?member)
28           (f/name f ?member ?first-name)
29           (id ?member ?id)
30           (ccl/conda
31            [(ccl/== prefer-ex-family true)]
32            [(bx/existing-elemento? ?member)
33             (id ?family ?last-name)]
34            [(id ?family ?id)])]
35   :right [(p/->persons p ?person-register ?person)
36           (p/Person p ?person)
37           (p/name p ?person ?full-name)
38           (id ?person ?id)]
39   :when  [(rel/stro ?last-name ", " ?first-name ?full-name)])
```

So what are these common aspects? Well, a `?member` of a `?family` (where we haven't determined the role, yet) contained in the `?family-register` passed in as parameter from `family-register2person-register` corresponds to a `?person` (where we haven't determined the gender, yet) contained in the `?person-register` passed in as the other parameter from `family-register2person-register`. The `:when` clause defines that the concatenation of the `?family`'s `?last-name`, the string `", "` and the `?member`'s `?first-name` gives the `?full-name` of the `?person`.

What hasn't been described so far are the `id` goals in lines 29, and 34 and the `ccl/conda` goal starting in line 30. The first two define that the `?member` and the corresponding `?person` must have the same synthetic ID. Remember the `:id-init-fn` in line 17 which assigned a unique number to every element in the respective source model of the transformation. With these synthetic IDs, the transformation is able to create one person for every member and vice versa even in the case where two elements are equal based on attribute values.

Lastly, the `ccl/conda` goal starting in line 30 of the `:left` clause handles the preference of re-using existing families, i.e., assigning new members to existing families, over creating new families for new members. By default, FunnyQT would always try to re-use an existing family. Thus, if the `prefer-ex-family` parameter is `true`, nothing needs to be done. Likewise, if `?member` is an existing element for which we assume she's already assigned to some family, we can also just stick to the default behavior but define the `?family`'s ID to be its name (although it's probably not unique). If the first two `ccl/conda` clauses fail, i.e., `prefer-ex-family` is `false` and `?member` is a new member which is just going to be created by the enforcement of this relation, then we define that the `?family`'s ID must equal the IDs of the `?member` and `?person`. Thus, in this case and only in this case, new members force the creation of a new family even when there already is a family with the right name.

The last two transformation relations extend the `member2person` relation for synchronizing between members in the role of a family mother or dauhter and female persons, and between family fathers or sons and male persons.

```
40  (member2female
41   :extends [(member2person)]
42   :left  [(relationshipo prefer-parent f ?family ?member f/->mother f/->daughters)]
43   :right [(p/Female p ?person)])
44  (member2male
45   :extends [(member2person)]
46   :left  [(relationshipo prefer-parent f ?family ?member f/->father f/->sons)]
47   :right [(p/Male p ?person)]))
```

In the `:left` clauses we use the `relationshipo` helper relation described in the beginning of this section which chooses the right female or male role based on the preference parameter `prefer-parent` and the current state of the family, i.e., by checking if the respective parental role is still unset. In the two `:right` clauses, we only need to specify that the `Person ?person` is actually a `Female` or `Male`.

These 33 lines of transformation specification plus the 12 lines for the `relationshipo` helper, and two lines for the generation of the metamodel-specific relational querying APIs form the complete functional parts of the solution. The only thing omitted from the paper are the namespace declaration[15] consisting of 5 lines of code.

[15]The Clojure equivalent of Java's package statement and imports.

2.3 Gluing the Solution with the Framework

Typically, open-source Clojure libraries and programs are distributed as JAR files that contain the source files rather than byte-compiled class files. This solution does almost the same except that the JAR contains the solution source code, FunnyQT itself (also as sources) and every dependency of FunnyQT (like Clojure) except for EMF which the *benchmarx* project already provides.

Calling Clojure functions from Java is really easy and FunnyQT transformations are no exception because they are plain Clojure functions, too. The FunnyQT solution's `BXTool` implementation `FunnyQTFamiliesToPerson` extends the `BXToolForEMF` class. Essentially, it just has a static member `T` which is set to the transformation.

```
public class FunnyQTFamiliesToPerson extends BXToolForEMF<FamilyRegister, PersonRegister, Decisions> {
    private final static Keyword LEFT = (Keyword) Clojure.read(":left");
    private final static Keyword RIGHT = (Keyword) Clojure.read(":right");

    private final static IFn T;

    static {
        final String transformationNamespace = "ttc17-families2persons-bx.core";
        // Clojure's require is similar to Java's import.  However, it also loads the required
        // namespace from a source code file and immediately compiles it.
        final IFn require = Clojure.var("clojure.core", "require");
        require.invoke(Clojure.read(transformationNamespace));
        T = Clojure.var(transformationNamespace, "families2persons");
    }
```

All Clojure functions implement the `IFn` interface and can be called using `invoke()`. And exactly this is done to call the transformation.

```
    private void transform(Keyword direction) {
        T.invoke(srcModel, trgModel, direction,
                 configurator.decide(Decisions.PREFER_CREATING_PARENT_TO_CHILD),
                 configurator.decide(Decisions.PREFER_EXISTING_FAMILY_TO_NEW));
    }
```

This corresponds to a call `(families2persons src trg dir prefer-parent prefer-ex-family)` directly in Clojure.

3 Evaluation & Conclusion

BatchForward.* All tests result in an *expected pass*.

BatchBwdEandP.* All tests result in an *expected pass*.

BatchBwdEnotP.* All tests result in an *expected pass*.

BatchBwdNotEandP.* All tests result in an *expected pass*.

BatchBwdNotEnotP.* All tests result in an *expected pass*.

IncrementalForward.testStability This is an *expected pass*; re-running the transformation after a no-op operation doesn't change the target model.

IncrementalForward.testIncrementalMixed This is an *expected fail*. In this test, after transforming forward, father Homer's birthday is changed in the target model, then father Homer is deleted and immediately re-created, and the transformation is run again. Since the transformation doesn't consider birthdays at all, the existing homer with the changed birthday is still ok and not modified. Thus, he still has the changed and not the default birthday.

IncrementalForward.testHippocraticness This is an *expected pass*. Empty families are of no relevance to the transformation, so creating an empty family and re-running the transformation has no effect on the target model.

IncrementalForward.testIncrementalMove This is an *expected fail*. Here, the birthdays are set manually in the persons model. Then two members are moved in the source model and the transformation is re-executed. This leads to creating two new persons and deleting the old ones in the target model. Of course, the two new persons again have the default birthday.

IncrementalForward.testIncrementalDeletions This is an *expected fail*. Here, after the initial transformation which creates two Barts in the persons model, their birthdays are set to different values. Then, one Bart is deleted in the family model and the transformation is re-executed. The solution deletes the wrong one because from the perspective of the transformation, both are completely equivalent (since it doesn't consider birthdays).

IncrementalForward.testIncrementalRename This is an *expected fail*. After an intial transformation, all birthdays are changed in the person model. Then, the source model family is renamed, and the transformation is re-executed. This leads to deletion of all persons and their re-creation with the new full names. Obviously, the manually set birthdays are lost.

IncrementalForward.testIncrementalInserts This is an *expected pass*. After an intial transformation, all birthdays are changed in the person model. Then, a new member is added to the source model family and the transformation is re-executed. A new person is created and the others aren't changed, i.e., they also keep their manually set birthdays.

IncrementalForward.testIncrementalMoveRoleChange This is an *expected fail*. After an intial transformation, all birthdays are changed in the person model. Then, daughter is moved to a different family as a son, and the transformation is re-executed. This leads to the deletion of the corresponding female and creation of a new male and the manually set birthday is lost.

IncrementalBackward.testIncrementalInsertsDynamicConfig This is kind of an *unexpected fail*. When we neither prefer existing families nor assigning to parental roles, it still adds the new Seymore to an existing family in a parental role. I haven't found the cause yet[16].

IncrementalBackward.testStability This is an *expected pass*; re-running the transformation after a no-op operation doesn't change the target model.

IncrementalBackward.testIncrementalInsertsFixedConfig This is an *expected pass*.

IncrementalBackward.testIncrementalMixedDynamic This is an *expected fail*. Here, Homer is deleted and re-created anew, and then the transformation is re-executed with preference for child roles. Since FunnyQT is not incremental, the model looks unchanged and Home stays in his original father role.

IncrementalBackward.testIncrementalOperational This is probably an *expected pass* because FunnyQT uses standard EMF iterators which iterate elements in insertion order (unless the ELists are sorted afterwards).

IncrementalBackward.testRenamingDynamic This is an *expected fail*. Since FunnyQT's bx transformations are state-based, renaming leads to deletion and re-creation of members if only the first name changed, or complete families including the changed members if their family name changed.

IncrementalBackward.testHippocraticness This is an *expected pass*. Re-running the transformation a second time with unchanged source model won't change the target model.

IncrementalBackward.testIncrementalDeletions This is an *expected fail*. Every person is deleted. Therefore, no family is required by the transformation in the family model and we end up with an empty family register, not with a family register still containing an empty family.

References

[ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The family to persons case. In *Transformation Tool Contest*, 2017.

[Hor15] Tassilo Horn. Graph pattern matching as an embedded clojure dsl. In *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*, 2015.

[Hor16] Tassilo Horn. *A Functional, Comprehensive and Extensible Multi-Platform Querying and Transformation Approach*. PhD thesis, University Koblenz-Landau, Germany, 2016.

[16] The author has just become father a second time, so there have been more important things to do.