Solving the TTC Movie Database Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure providing a rich and efficient querying and transformation API. This paper describes the FunnyQT solution to the TTC 2014 Movie Database transformation case. All core tasks and all extension tasks have been solved.

1 Introduction

This paper describes a solution of the TTC 2014 Movie Database Case [3]. All core and extension tasks have been solved. The solution project is available on Github[1], and it is set up for easy reproduction on the SHARE[2] image.

The solution is implemented using FunnyQT [2] which is a model querying and transformation library for the functional Lisp dialect Clojure[3]. Queries and transformations are plain Clojure programs using the features provided by the FunnyQT API. This API is structured into several task-specific sub-APIs/namespaces, e.g., there is a namespace *funnyqt.in-place* containing constructs for writing in-place transformations, a namespace *funnyqt.model2model* containing constructs for model-to-model transformations, a namespace *funnyqt.bidi* containing constructs for bidirectional transformations, and so forth.

As a Lisp, Clojure provides strong metaprogramming capabilities that are exploited by FunnyQT in order to define several *embedded domain-specific languages* (DSL, [1]) for different tasks. For example, the pattern matching constructs used in this solution is provided in terms of a task-oriented DSL.

2 Solution Description

In this section, the transformation and query specification for all core and extension tasks are going to be explained. In the listings given in the following, all function calls are shown in a namespace-qualified form to make it explicit in which Clojure or FunnyQT namespace those functions are defined. Clojure allows to define short aliases for used namespaces in order to allow qualification while still being concise, e.g., `(emf/eget o :prop)` where `emf` is an alias for the namespace `funnyqt.emf` and `eget` is the function name. All functions with namespace aliases `emf`, `ip`, `poly`, and `u` are FunnyQT functions, all others are either core Clojure or Clojure standard library functions, or functions defined in the transformation namespace itself.

[1] https://github.com/tsdh/ttc14-movie-couples
[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_FunnyQT4.vdi
[3] http://clojure.org

Submitted to:
TTC 2014

2.1 Task 1: Generating Test Data

The first task is generating test data. The case description [3] illustrates the task with Henshin rules. Since the rules actually don't match anything but simply create new elements in the model, we have implemented them as plain functions receiving the model and an integer parameter `i` from which the movie ratings and actor names are derived. The function `create-positive!` creates 5 movies, 3 actors, and 2 actresses, sets their attributes, and links them as requested.

```
(defn ^:private create-positive! [model i]
  (let [m1 (emf/ecreate! model 'Movie {:rating (+ 0.0 (* 10 i))})
        m2 (emf/ecreate! model 'Movie {:rating (+ 1.0 (* 10 i))})
        m3 (emf/ecreate! model 'Movie {:rating (+ 2.0 (* 10 i))})
        m4 (emf/ecreate! model 'Movie {:rating (+ 3.0 (* 10 i))})
        m5 (emf/ecreate! model 'Movie {:rating (+ 4.0 (* 10 i))})]
    (emf/ecreate! model 'Actor   {:name (str "a" (* 10 i))       :movies [m1 m2 m3 m4]})
    (emf/ecreate! model 'Actor   {:name (str "a" (+ 1 (* 10 i))) :movies [m1 m2 m3 m4]})
    (emf/ecreate! model 'Actor   {:name (str "a" (+ 2 (* 10 i))) :movies [m2 m3 m4]})
    (emf/ecreate! model 'Actress {:name (str "a" (+ 3 (* 10 i))) :movies [m2 m3 m4 m5]})
    (emf/ecreate! model 'Actress {:name (str "a" (+ 4 (* 10 i))) :movies [m2 m3 m4 m5]})))
```

The `create-negative!` function is defined similarly, so it is skipped here for brevity.

2.2 Task 2/3 and Extension Task 2/3: Finding Couples/Cliques and Compute Average Rankings

For finding cliques of arbitrary sizes $n \geq 3$, a higher-order transformation should be defined that generates a transformation rule for that n. The FunnyQT solution also allows for $n = 2$ and deals with the fact that in this case, `Couple` elements should be created rather than `Clique` elements. Also, the *computation of the average rankings* of a couple's or clique's common movies is done while creating the `Couple` or `Clique` element instead of doing it separately in a further step.

Before discussing this higher-order transformation, a few helper functions are going to be introduced which will be used as constraints in the patterns of the generated rules. First, there's a function `movie-count` that gets some person and returns the number of movies that this person has acted in. Secondly, the `person-count` function returns the number of persons that acted in a given movie. Thirdly, `movie-set` gets a person element and returns the movies that person acted in as a set. And lastly, a function `avg-rating` gets a collection of movies and returns their average rating.

The function `n-common-movies?` printed in the next listing gets an integer `n`, a person element `p`, and a sequence of `more` person elements [4]. `loop` and `recur` implement a local tail-recursion. Initially, the set of common movies `common` is bound to the set of `p`'s movies, and the remaining persons are bound to `more` shadowing the function's parameter of the same name. If there are more persons, the `recur` in line 5 jumps back to the `loop` in line 2 where `common` is rebound to the intersection of `common` and the movies of the first person in `more`. Likewise, `mode` is rebound to the remainder of `more`. Thus, if all given persons act together in at least `n` movies, the set of common movies is returned. Otherwise, `nil` is returned. Since in Clojure the values `nil` and `false` are falsy while every other value is truthy, this function can act as a predicate and still return more information, i.e., the common movies, in the positive case.

```
1 (defn n-common-movies? [n p & more]
2   (loop [common (movie-set p), more more]
3     (when (>= (count common) n)
4       (if (seq more)
5         (recur (set/intersection common (movie-set (first more))) (rest more))
6         common))))
```

[4]The Clojure varargs syntax `& els` is similar to Java's `Type... els` syntax.

The higher-order transformation generating a FunnyQT in-place transformation rule for a given $n \geq 2$ is a Clojure *macro*. A macro is a function which is executed at compile-time by the Clojure compiler. It receives code passed to it as arguments, processes it, and returns new code that takes the place of it's call. This new code is called the macro's *expansion*. Because like all Lisps, Clojure is *homoiconic*, i.e., Clojure code is represented using Clojure datastructures (literals, symbols, lists, vectors), a macro is essentially a transformation on the abstract syntax tree of the Clojure code that's passed to the macro.

Listing 1 in the appendix on page 6 shows the `define-groups-rule` macro which is the higher-order transformation solving the tasks. It receives an parameter `n` and, as its name suggests, expands into a rule for finding couples if `n` equals 2 or cliques of size `n` for larger values of `n`.

We're not going to discuss the macro in details, however the central idea of the Clojure (or Lisp) macrosystem is that one defines the basic structure of the macro's expansion using a *quasi-quoted* (back-ticked) form as a kind of template. In this quasi-quoted form, values computed at compile-time can be inserted using the *unquote* (`~`) and *unquote-splicing* (`~@`) operators to fill in the template's variable parts.

The last part of the implementation of the tasks 2 and 3 and the extension tasks 2 and 3 is to actually invoke the macro to create the transformation rules for couples and cliques of 3, 4, and 5 persons.

```
(define-group-rule 2) ;; make-groups-of-2!: The Couples rule
(define-group-rule 3) ;; make-groups-of-3!: The Cliques of Three rule
(define-group-rule 4) ;; make-groups-of-4!: The Cliques of Four rule
(define-group-rule 5) ;; make-groups-of-5!: The Cliques of Five rule
```

Instead of discussing the rule generation macro in details, it makes more sense to have an in-depth look at one of its expansion like the one for `n` being 3 shown below. A FunnyQT in-place transformation rule is defined whose name is `make-groups-of-3!`, and it gets as arguments the `model` on which it should be applied, and an integer `c` which determines how many common movies a clique of three persons needs to have. The case description fixes `c` to 3, but with this parameter, we allow for a bit more generality.

```
 1 (ip/defrule make-groups-of-3!
 2   {:forall true}
 3   [model c]
 4   [m<Movie>            :when (>= (person-count m) 3)
 5    m -<persons>-> p0   :when (>= (movie-count p0) c)
 6    m -<persons>-> p1   :when (>= (movie-count p1) c)
 7                        :when (neg? (compare (emf/eget-raw p0 :name) (emf/eget-raw p1 :name)))
 8                        :when (n-common-movies? c p0 p1)
 9    m -<persons>-> p2   :when (>= (movie-count p2) c)
10                        :when (neg? (compare (emf/eget-raw p1 :name) (emf/eget-raw p2 :name)))
11    :when-let [cms (n-common-movies? c p0 p1 p2)]
12    :as [cms p0 p1 p2] :distinct]
13   (emf/ecreate! model 'Clique {:avgRating (avg-rating cms), :persons [p0 p1 p2], :commonMovies cms}))
```

Lines 4 to 12 define the rule's pattern. The structural part defines that it matches a `Movie` element `m` which references three `Person` elements `p0`, `p1`, and `p2` using its `persons` reference.

Additionally, the pattern defines several constraints using the `:when` keyword. The movie `m` needs to have at least three acting persons (line 4), and all persons need to act in at least `c` movies (lines 5, 6, and 9). To avoid duplicate matches where only the order of the three person elements differs, the constraints in line 7 and 10 enforce a lexicographical order of the names of the persons `p0`, `p1`, and `p2`.

Line 8 ensures that `p0` and `p1` have at least `c` common movies. The same for the complete clique of three persons is also asserted in line 11, where the common movies are also bound to the variable `cms`. The first constraint is there only for performance reasons. Clearly, if `p0` and `p1` already have less than `c` common movies, then `p0`, `p1`, and `p2` cannot have more. This test ensures that the pattern matching process stops for the combination of `p0` and `p1` as soon as possible.

The last line of the pattern, line 12, defines that each match should be represented as a vector containing the set of common movies `cms` and the three persons. The keyword `:distinct` specifies that only distinct matches should be found. The reason is that if some clique of three acts in x common movies, there are exactly *x* matches that differ only in the movie `m`. By omitting the movie from the match representation and specifying that we are only interested in distinct matches, those duplicates are suppressed.

The last two lines define the action that should be applied on matches. A new `Clique` element is created that gets assigned the found persons with their common movies and average rating.

What has been skipped from explanation until now is the rule's `:forall` option. It specifies that calling the rule finds all matches at once and then applys the action to each of them. FunnyQT performs the pattern matching process in parallel for such `:forall`-rules.

2.3 Extension Task 1/4: Compute Top-15 Couples/Cliques

The case description demands for the Extension Tasks 1 and 4 the computation of the top-15 groups according to the criteria (a) *average rating of common movies*, and (b) *number of common movies*. If there's a tie between two groups for the current criterium, the respective other criterium is used to cut it. If that doesn't suffice, i.e., both groups have the same average rating and number of common movies, the names of the group's members are compared as a fallback. Since the person names are unique in the models, there is no chance that no distinction can be made.

The implementation is simple in that the sequence of all couples (or cliques of a given size) are sorted using a comparator. Like in Java, a Clojure comparator is a function that receives two objects and returns a negative integer if the first object should be sorted before the second, a positive integer if the first object should be sorted after the second item, and zero if both objects are equal with respect to order.

The comparators for the average rating, number of common movies, and the group's member names are shown in the next listing.

```clojure
(defn rating-comparator [a b]
  (compare (emf/eget b :avgRating) (emf/eget a :avgRating)))
(defn common-movies-comparator [a b]
  (compare (.size ^java.util.Collection (emf/eget-raw b :commonMovies))
           (.size ^java.util.Collection (emf/eget-raw a :commonMovies))))
(defn names-comparator [a b]
  (compare (str/join ";" (map #(emf/eget % :name) (actors a)))
           (str/join ";" (map #(emf/eget % :name) (actors b)))))
```

They get two objects `a` and `b` (two couples or cliques) and compare them using Clojure's standard `compare` function which works for objects of any class implementing the `java.lang.Comparable` interface.

Until now, there are only three individual comparators, but sorting is always done with one single comparator. So the following listing defines a higher-order comparator, e.g., a function that receives arbitrary many comparators and returns a new comparator which compares using the given ones.

```clojure
(defn comparator-combinator [& comparators]
  (fn [a b]
    (loop [cs comparators]
      (if (seq cs)
        (let [r ((first cs) a b)]
          (if (zero? r) (recur (rest cs)) r))
        (u/errorf "%s and %s are incomparable!" a b)))))
```

The function `comparator-combinator` returns an anonymous function with two arguments `a` and `b`. This function recurses[5] over the given `comparators` applying one after the other until one returns a non-zero result. So finally, here are the two top-15 groups functions.

```clojure
(defn groups-by-avg-rating [groups]
  (sort (comparator-combinator rating-comparator common-movies-comparator names-comparator) groups))
(defn groups-by-common-movies [groups]
  (sort (comparator-combinator common-movies-comparator rating-comparator names-comparator) groups))
```

[5]Clojure's `(loop [<bindings>] ... (recur <newvals>))` is a local tail-recursion. `loop` establishes bindings just like `let`, and `recur` jumps back to the `loop` providing new values for the variables.

`groups-by-avg-rating` gets a collection of groups `groups` and then sorts them by the combined comparator first taking the average rating into account, then the number of common movies, and eventually the names of the groups' actors if neither of the two former comparators could decide on the two groups order. `group-by-common-movies` is defined analoguously with the `common-movies-comparator` taking precedence over the `rating-comparator`.

3 Evaluation and Conclusion

With respect to correctness, the FunnyQT solution computes the same numbers of couples and cliques of various sizes as printed in the case description. Also, the top-15 lists are identical for all models.

The table below shows the execution times for the IMDb models. They include the pattern matching time, the time needed for creating the `Couple` and `Clique` elements, and the time needed for setting their attributes and references including the computation of the average ratings. The benchmarks were run on a GNU/Linux machine with eight 2.8GHz cores with 30GB of RAM dedicated to the JVM process.

Model	**Couples (secs)**	**3-Cliques (secs)**
imdb-0005000-49930.movies.bin	1.677278992	6.957570992
imdb-0010000-98168.movies.bin	1.702668160	11.333623024
imdb-0045000-299504.movies.bin	3.947932624	15.714349728
imdb-0085000-499995.movies.bin	9.012942560	26.388751712
imdb-0200000-1004463.movies.bin	35.409998560	117.833811360
imdb-0495000-2000900.movies.bin	159.973018224	757.280006768
imdb-all-3257145.movies.bin	619.160156640	4295.030516512

The main bottleneck of the FunnyQT transformation is the required memory. The generated rules for finding groups of a given size first compute all matches and then generate one new couple or clique element for each match. This means that the original model, all matches, and also all new elements reside in memory at the same time.

Another strong point of the solution is its conciseness. All in all, it consists of 152 lines of code (including boilerplace code like namespace definitions) in three source files, one for the generation of the synthetic test models (30 LOC), one for the couple and cliques rules (52 LOC), and one for the queries (70 LOC, most of which are concerned with pretty-printing the results into files).

References

[1] Martin Fowler (2010): *Domain-Specific Languages*. Addison-Wesley Professional.

[2] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[3] Christian Krause, Tassilo Horn & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*. In: *Transformation Tool Contest 2014*.

```
(defmacro define-group-rule [n]
  (let [psyms (map #(symbol (str "p" %)) (range n))]
    `(ip/defrule ~(symbol (str "make-groups-of-" n "!"))
       {:forall true}
       [~'model ~'c]
       [~'m<Movie> :when (>= (person-count ~'m) ~n)
        ~@(mapcat (fn [i]
                    (let [ps (nth psyms i)]
                      `[~'m -<persons>-> ~ps
                        :when (>= (movie-count ~ps) ~'c)
                        ~@(when-not (zero? i)
                            `[:when (neg? (compare (emf/eget-raw ~(nth psyms (dec i)) :name)
                                                   (emf/eget-raw ~ps :name)))])
                        ~@(when-not (or (zero? i) (= i (dec n)))
                            `[:when (n-common-movies? ~'c ~@(take (inc i) psyms))])]))
                  (range n))
        :when-let [~'cms (n-common-movies? ~'c ~@psyms)]
        :as [~'cms ~@psyms]
        :distinct]
       (emf/ecreate! ~'model ~(if (= n 2) `'Couple `'Clique)
                     ~(if (= n 2)
                        `{:commonMovies ~'cms :avgRating (avg-rating ~'cms)
                          :p1 ~(first psyms) :p2 ~(second psyms)}
                        `{:commonMovies ~'cms :avgRating (avg-rating ~'cms)
                          :persons [~@psyms]})))))
```

Listing 1: The higher-order transformation generating couple and cliques rules