Solving the TTC Train Benchmark Case with FunnyQT

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

This paper describes the FunnyQT solution to the TTC 2015 Train Benchmark transformation case. The solution solves all core and all extension tasks, and it won the *overall quality award*.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Train Benchmark Case [3]. All core and extension tasks have been solved. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3]. This solution won the *overall quality award* for this case.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. 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 models and JGraLab 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 train benchmark case, especially its in-place transformation DSL has been used.

2 Solution Description

In this section, the individual tasks are discussed one by one. They are all implemented as in-place transformation rules supported by FunnyQT's *funnyqt.in-place* transformation DSL. The rules' repair actions simply call the CRUD functions of the EMF-specific *funnyqt.emf* namespace.

Task 1: PosLength. The transformation rule realizing the *PosLength* task is given below.

```
1 (defrule pos-length {:forall true :recheck true} [g]
2   [segment<Segment>
3    :when (<= (eget-raw segment :length) 0)]
4   (eset! segment :length (inc (- (eget-raw segment :length)))))
```

[1] http://funnyqt.org
[2] https://github.com/tsdh/ttc15-train-benchmark-funnyqt
[3] The SHARE image name is ArchLinux64_TTC15-FunnyQT_2
[4] http://clojure.org

Submitted to:
TTC 2015

The `defrule` macro defines a new in-place transformation rule with the given name (`pos-length`), an optional map of options (`{:forall true, ...}`) a vector of formal parameters (`[g]`), a pattern (`[segment<Segment>...]`), and one or many actions to be applied to the pattern's matches (`(eset! ...)`). The first formal parameter must denote the model the rule is applied to, so here the argument `g` denotes the train model when the rule is applied using `(pos-length my-train-model)`.

The pattern matches a node called `segment` of metamodel class `Segment`. Additionally, the segment's length must be less or equal to zero as defined by the `:when` constraint. The action says that the segment's `length` attribute should be set to the incremented negation of the current length.

The normal semantics of applying a rule is to find one single match of the rule's pattern and then execute the rule's actions on the matched elements. The `:forall` option changes this behavior to finding all matches first, and then applying the actions to each match one after the other. FunnyQT automatically parallelizes the pattern matching process of such forall-rules under certain circumstances like the JVM having more than one CPU available and the pattern declaring at least two elements to be matched.

The `:recheck` option causes the rule to recheck if a pre-calculated match is still conforming the pattern just before executing the rule's actions on it. This can be needed for forall-rules whose actions possibly invalidate matches of the same rule's pattern, e.g., when the application of the action to a match m_i cause another match m_j to be no valid match any longer[5].

Task 2: SwitchSensor. The transformation rule realizing the *SwitchSensor* task is given below.

```
5 (defrule switch-sensor {:forall true :recheck true} [g]
6   [sw<Switch> -!<:sensor>-> <>]
7   (eset! sw :sensor (ecreate! nil 'Sensor)))
```

It matches a switch `sw` which is not contained by some sensor. The exclamation mark of the edge symbol `-!<:sensor>->` specifies that no such reference must exist, i.e., it specifies a negative application condition. The action fixes this problem by creating a new `Sensor` and assigning that to the switch `sw`.

Task 3: SwitchSet. The `switch-set` rule realizes the *SwitchSet* task. Its definition is given below.

```
8 (def Signal-GO (eenum-literal 'Signal.GO))

9 (defrule switch-set {:forall true :recheck true} [g]
10   [route<Route> -<:entry>-> semaphore
11    :when (= (eget-raw semaphore :signal) Signal-GO)
12    route -<:follows>-> swp -<:switch>-> sw
13    :let [swp-pos (eget-raw swp :position)]
14    :when (not= (eget-raw sw :currentPosition) swp-pos)]
15   (eset! sw :currentPosition swp-pos))
```

It matches a `route` with its entry `semaphore` where the semaphore's signal is `Signal.GO`. The route follows some switch position `swp` whose switch `sw`'s current position is different from that of the switch position. The fix is to set the switch's current position to the position of the switch position `swp`.

Note that there are no metamodel types specified for the elements `semaphore`, `swp`, and `sw` because those are already defined implicitly by the references leading to them, e.g., all elements referenced by a route's `follows` reference can only be instances of `SwitchPosition` according to the metamodel. FunnyQT doesn't require the transformation writer to encode tautologies in her patterns[6].

[5] This cannot happen for the `pos-length` rule, however the case description demands matches to be revalidated before applying the repair actions.

[6] In fact, if there are types specified, those will be checked. So omitting them when they are not needed also results in slightly faster patterns.

Extension Task 1: RouteSensor. The extension task *RouteSensor* is realized by the `route-sensor` rule given below.

```
16 (defrule route-sensor {:forall true :recheck true} [g]
17   [route<Route> -<:follows>-> swp -<:switch>-> sw
18    -<:sensor>-> sensor --!<> route]
19   (eadd! route :definedBy sensor))
```

It matches a `route` that follows some switch position `swp` whose switch `sw`'s `sensor` is not contained by the `route`. The repair action is to assign the `sensor` to the `route`.

Extension Task 2: SemaphoreNeighbor. The second and last extension task *SemaphoreNeighbor* is realized by the `semaphore-neighbor` rule defined as shown below.

```
20 (defrule semaphore-neighbor {:forall true :recheck true} [g]
21   [route1<Route> -<:exit>-> semaphore
22    route1 -<:definedBy>-> sensor1 -<:elements>-> te1
23    -<:connectsTo>-> te2 -<:sensor>-> sensor2
24    --<> route2<Route> -!<:entry>-> semaphore
25    :when (not= route1 route2)]
26   (eset! route2 :entry semaphore))
```

It matches a route `route1` which has an exit `semaphore`. Additionally, `route1` is defined by a sensor `sensor1` which contains some track element `te1` that connects to some track element `te2` whose sensor is `sensor2`. This `sensor2` is contained by some other route `route2` which does not have `semaphore` as entry semaphore. The fix is to set `route2`'s entry reference to `semaphore`.

2.1 Deferred Rule Actions

As mentioned above, the normal semantics of a forall-rule is to compute all matches of the rule's pattern first (possibly in parallel), and then apply the rule's actions on every match one after the other. However, the case description strictly separates the computation of matches from the repair transformations.

FunnyQT also provides stand-alone patterns. Using them, one could have defined patterns for finding occurrences of the five problematic situations in a train model, and separate functions for the repair actions where the latter receive one match of the corresponding pattern and fix that.

But for in-place transformation rules, FunnyQT also provides *rule application modifiers*. Concretely, any in-place transformation rule `r` can be called as `(as-pattern (r model))` in which case it behaves as a pattern. That is, where a normal rule would usually find one match and apply its actions on it and a forall-rule would usually find all matches and apply its actions to each of them, when called with `as-pattern`, a rule simply returns the sequence of its matches. With a normal rule, this sequence is a lazy sequence, i.e., the matches are not computed until they are consumed. With a forall-rule, the sequence is fully realized, i.e., all matches are already pre-calculated (possibly in parallel).

The second FunnyQT rule application modifier is `as-test`, and this is what is highly suitable for this transformation case. When a rule `r` is applied using `(as-test (r model))`, it behaves almost as without modifier but instead of applying the rule's actions immediately, it returns a closure of arity zero (a so-called *thunk*) which captures the rule's match and the rule's actions. Invoking the thunk causes the actions to be applied on the match. Thus, the caller of the rule gets the information if the rule was applicable at all, and if it was applicable, she can decide if she wants to apply it or not. And when she applies it, the pattern matching part is already finished and only the actions are applied on the pre-calculated match the thunk closes over.

In case of a forall-rule `r`, `(as-test (r model))` doesn't return a single thunk but a vector of thunks, one thunk per match of the rule's pattern. This is exactly what is needed for solving this transformation

case. Using this feature, a final function is defined that receives a rule `r` and a train model `g` and executes the rule as a test.

```
27 (defn call-rule-as-test [r g]
28   (as-test (r g)))
```

This function is then called with the transformation rules from the Java trainbenchmark framework. The given rule gets applied and returns a sequence of thunks which will apply the actions to the match they are wrapping. Thus, the only thing the framework has to do is to apply the thunks corresponding to the matches which are going to be repaired in the current repair phase.

These 28 lines of Clojure code form the complete functional part of the FunnyQT solution that solves all core and extension tasks. There is also a plain-Java glue project which implements the interfaces required by the benchmark framework and simply delegates to the Clojure/FunnyQT part of the solution. This glue project is briefly discussed in the following section.

2.2 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 the same, and that JAR is deployed to a local Maven repository from which the Maven build infrastructure of the benchmark framework can pick it up.

Then, in the FunnyQT glue project the rules and functions from above are referred to like shown in the next listing.

```
private final static String SOLUTION_NS = "ttc15-train-benchmark-funnyqt.core";
Clojure.var("clojure.core", "require").invoke(Clojure.read(SOLUTION_NS));
final static IFn POS_LENGTH = Clojure.var(SOLUTION_NS, "pos-length");
...
final static IFn CALL_RULE_AS_TEST = Clojure.var(SOLUTION_NS, "call-rule-as-test");
```

In line 2, the solution namespace `ttc15-train-benchmark-funnyqt.core` is required[7]. The `Clojure` class provides a minimal API for loading Clojure code from Java. When requiring a namespace as above, it will be parsed and compiled to JVM byte-code just in time[8].

Thereafter, the solution's in-place transformation rules and the `call-rule-as-test` function are referred to. `IFn` is a Clojure interface whose instances are Clojure functions that can be called using the `invoke()` method as can be seen in the definition of the glue project's `BenchmarkCase.check()` method shown below.

```
@Override
protected final Collection<Object> check() throws IOException {
    matches = (Collection<Object>) FunnyQTBenchmarkLogic.CALL_RULE_AS_TEST
                                    .invoke(rule, this.resource);
    // If the rule has no matches it returns nil/null but the framework
    // wants a Collection.
    if (matches == null) {
        matches = new LinkedList<Object>();
    }
    return matches;
}
```

In that code, `rule` is one of the rule `IFn`s `POS_LENGTH`, `SWITCH_SET`, et cetera, and they are called via the `call-rule-as-test` function to make them return one thunk per match instead of performing the rules' repair actions immediately.

The implementation of the `BenchmarkCase.modify()` method is even simpler.

[7] `require` is kind of Clojure's equivalent to Java's `import` statement.

[8] If the Clojure code was distributed in a pre-compiled form, the resulting classes would simply be loaded.

```
@Override
protected final void modify(Collection<Object> matches) {
    for (Object m : matches) {
        ((IFn) m).invoke();
    }
}
```

Since the rules are called as tests and thus return thunks that apply the rule's actions, those simply need to be invoked.

3 Evaluation & Conclusion

The FunnyQT solution implements all core and all extension tasks exactly as demanded by the case description, thus it is *complete*. When run in the benchmark framework, all assertion it checks are satisfied, thus the solution is also *correct*.

The FunnyQT solution consists of 28 NCLOC of FunnyQT/Clojure code for the five rules with their patterns and repair actions, and the function `call-rule-as-test`. Therefore, it is very *concise*.

Readability is a very subjective matter, and not everyone is fond of Lisp syntax. However, there are some strong points with respect to readability. (1) The queries (patterns) and repair actions are bundled in one in-place transformation rule each keeping the definition of cause and effect localized. (2) FunnyQT's pattern matching DSL used to specify the rules' patterns is both concise and readable. It should be easy to understand for graph transformation experts especially if they have used other textual graph transformation languages such as *GrGen.NET* before. It should also be easy to understand for any Clojure programmer because it strictly conforms to the style guidelines and best practices there.

FunnyQT implements pattern matching as a local search. Thus, each *recheck* phase take approximately as much time as the initial *check* phase. In contrast, with an incremental approach like *EMF-IncQuery*, the rechecking the patterns is not needed because all matches of all patterns are cached and updated when the model changes. This makes FunnyQT not especially suited for incremental model validation scenarios. However, given the fact that the evaluation of forall-patterns is automatically parallelized on multi-core machines, the *performance* is still reasonable. The benefit of FunnyQT's search-based approach is that it has far less memory requirements than an incremental approach. When comparing the performance with the EMF-IncQuery solution on an 8-core machine with 32 GB RAM, FunnyQT was only about 15% slower and could still transform models which already caused an `OutOfMemoryError` with EMF-IncQuery. But of course, when increasing the number of iterations, the performance benefit of incremental approaches will increase, too.

References

[1] 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.

[2] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*.

[3] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation**. In: *Transformation Tool Contest 2015*.