Solving the TTC Java Refactoring 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 Java Refactoring transformation case. The solution solves all core tasks and also the extension tasks 1 and 2, and it has been elected as overall winner of this case.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Java Refactoring Case [3]. It solves all core and exception tasks with the exception of *Extension 3: Detecting Refactoring Conflicts* and has been elected as overall winner of the case. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3].

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[5] that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSLs) 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 one providing constructs supporting a concrete use-cases, e.g., model management, visualization, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and co-evolution transformations. For solving this case, FunnyQT's out-place and in-place transformation DSLs have been used.

2 Solution Description

The solution consists of three steps. (1) Converting the Java code to a program graph, (2) refactoring the program graph, and (3) propagating changes in the program graph back to the Java code. These steps are discussed in the following sections.

2.1 Step 1: Java Code to Program Graph

The first step in the transformation chain is to create an instance model conforming to the program graph metamodel predefined in the case description from the Java source code that should be subject to refactoring. The FunnyQT solution does that in two substeps.

---

[1] `http://funnyqt.org`
[2] `https://github.com/tsdh/ttc15-java-refactoring-funnyqt`
[3] The SHARE image name is `ArchLinux64_TTC15-FunnyQT_2`
[4] `http://clojure.org`
[5] The abstract syntax of a program can be accessed as data and manipulated at compile-time.

(a) Parse the Java source code into a model conforming to the EMFText JaMoPP[6] metamodel.

(b) Transform the JaMoPP model to a program graph using a FunnyQT out-place transformation.

Step (a) is implemented in the solution namespace *ttc15-java-refactoring-funnyqt.jamopp*. It simply sets up JaMoPP and defines two functions, one for parsing a source tree to a JaMoPP model, and a second one to synchronize the changes in a JaMoPP model back to the source tree. Both just access JaMoPP built-in functionality. Being able to seamlessly interoperate with Java is a feature FunnyQT gets for free from its host language Clojure.

Step (b) is implemented as a FunnyQT out-place transformation which creates a program graph from the parsed JaMoPP model.

The transformation also tries to keep the target program graph minimal. The source JaMoPP model contains the complete syntax graph of the parsed Java sources including all their dependencies. In contrast, the program graph created by the transformation only contains TClass elements for the Java classes parsed from source code and direct dependencies used as field type or method parameter or method return type. TMember elements are only created for the methods of directly parsed Java classes, and then only for those members that are not static because the case description explicitly excludes those. As a result, the program graph contains only the information relevant to the refactorings and is reasonably small so that it can be visualized by FunnyQT which is helpful for debugging purposes.

The FunnyQT out-place transformation API used for implementing this task is quite similar to ATL or QVT Operational Mappings. There are mapping rules which receive one or many JaMoPP source elements and create one or many target program graph elements.

A cutout of the transformation showing the rules responsible for transforming fields is given below. The transformation receives one single source model `jamopp` and one single target model `pg`.

```
1  (deftransformation jamopp2pg [[jamopp] [pg]]
2    ...
3    (field2tfielddef
4      :from [f 'Field]
5      :when (not (static? f))
6      :to   [tfd 'TFieldDefinition {:signature (get-tfieldsig f)}])
7    (get-tfieldsig
8      :from [f 'Field]
9      :id   [sig (str (type-name (get-type f)) " " (j/name f))]
10     :to   [tfs 'TFieldSignature {:field (get-tfield f)
11                                  :type  (type2tclass (get-type f))}])
12   (get-tfield
13     :from [f 'Field]
14     :id   [n (j/name f)]
15     :to   [tf 'TField {:tName n}]
16     (pg/->add-fields! *tg* tf))
17   (type2tclass
18     :from [t 'Type]
19     :disjuncts [class2tclass primitive2tclass])
20     ...)
```

For each non-static field in the JaMoPP model, the `field2tfielddef` rule creates one TFieldDefinition element in the program graph. The signature of this TFieldDefinition is set to the result of calling the `get-tfieldsig` rule.

This rule uses the `:id` feature to implement a n:1 semantics. Only for each unique string `sig` created by concatenating the field's type and name, a new TFieldSignature is created. If the rule is called thereafter for some other field with the same type and name, the existing field signature created at the first call is returned. The field signature's field and type references pointing to a TField and a TClass respectively are set by calling the two other rules `get-tfield` and `type2tclass`. This latter rule is a disjunctive rule which delegates to either the `class2tclass` or the `primitive2tclass` rule[7].

---

[6]http://www.jamopp.org/index.php/JaMoPP

[7]Rule disjunction is a feature borrowed from QVTo

In total, the transformation consists of 10 rules summing up to 71 lines of code. In addition, there are five simple helper functions like `static?`, `get-type`, and `type-name` that have been used in the above rules already.

A FunnyQT out-place transformation like the one briefly discussed above returns a map of traceability information. This traceability map is used in step 3 of the overall procedure, i.e., the back-propagation of changes in the program graph to the Java source code.

2.2 Step 2: Refactoring of the Program Graph

The refactorings are implemented in the solution namespace *ttc15-java-refactoring-funnyqt.refactor* using FunnyQT in-place transformation rules which combine patterns to be matched in the model with actions to be applied to the matched elements.

All rules defined in the following have a parameter `pg2jamopp-map-atom` which is essentially the inverse of the traceability map created by the JaMoPP to program graph transformation from step 1, i.e., it allows to translate program graph TClass and TMember elements to the corresponding JaMoPP Class and Member elements.

Pull Up Member. The case description requests *pull-up method* as the first refactoring core task. However, with respect to the program graph metamodel, there is actually no difference in pulling up a method (TMethodDefinition) or a field (TFieldDefinition), i.e., it is possible to define the refactoring more generally as *pull-up member* (TMember) and have it work for both fields and methods. This is what the FunnyQT solution does.

The corresponding `pull-up-member` rule is shown in the next listing. The rule is overloaded on arity. There is the version (1) of arity three which receives the program graph `pg`, the inverse lookup map `pg2jamopp-map-atom`, and the JaMoPP resource set `jamopp`, and there is the version (2) of arity four which receives the program graph `pg`, the inverse lookup map atom `pg2jamopp-map-atom`, a TClass `super`, and a TSignature `sig`.

```
21 (defrule pull-up-member
22   ([pg pg2jamopp-map-atom jamopp]                               ;; (1)
23    [:extends [(pull-up-member 1)]]                              ;; pattern
24    ((do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)  ;; action
25     jamopp))
26   ([pg pg2jamopp-map-atom super sig]                            ;; (2)
27    [super<TClass> -<:childClasses>-> sub -<:signature>-> sig    ;; pattern
28     sub -<:defines>-> member<TMember> -<:signature>-> sig
29     :nested [others [super -<:childClasses>-> osub
30                      :when (not= sub osub)
31                      osub -<:signature>-> sig
32                      osub -<:defines>-> omember<TMember> -<:signature>-> sig]]
33     :when (seq others)                                         ;; (a)
34     super -!<:signature>-> sig                                 ;; (b)
35     :when (= (count (pg/->childClasses super)) (inc (count others)))  ;; (c)
36     :when (forall? (partial accessible-from? super)           ;; (d)
37                    (mapcat pg/->access (conj (map :omember others) member)))]
38    (do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)))  ;; action
```

The version (2) is the one which is called by the ARTE test framework whereas the first version is called when performing the interactive refactoring extension.

The pattern of the version (2) matches a subclass `sub` of class `super` where `sub` defines a `member` of the given signature `sig`. A nested pattern is used to match all other subclasses of `super` which also define a member with that signature. The constraint (a) ensures that there are in fact other subclasses declaring a member with signature `sig`. Then the negative application condition (b) defines that the superclass `super` must not define a member of the given `sig` already. The constraint (c) ensures that all subclasses define a member of the given `sig`, i.e., not only a subset of all subclasses do so. Lastly, the constraint

(d) makes sure that all field and method definitions accessed by the member to be pulled up are already accessible from the superclass[8].

The pattern of the arity three variant (1) of the `pull-up-member` rule contains just an `:extends` clause specifying that its pattern equals the pattern defined for the arity four variant. As said, this variant is used by the extension task 2 where possible refactorings are to be proposed to the user. The difference between the overloaded versions of the `pull-up-member` rule is that version (1) matches `super` and `sig` itself whereas these two elements are parameters provided by the caller (i.e., ARTE) in version (2).

When a match is found, both versions of the rule call the function `do-pull-up-member!` which is defined as follows.

```
39 (defn do-pull-up-member! [pg pg2jamopp-map-atom super sub member sig others]
40   (doseq [o others]                                       ;; PG modification
41     (doseq [acc (find-accessors pg (:omember o))]
42       (pg/->remove-access! acc (:omember o))
43       (pg/->add-access! acc member))
44     (edelete! (:omember o))
45     (pg/->remove-signature! (:osub o) sig))
46   (pg/->remove-signature! sub sig)
47   (pg/->add-defines! super member)
48   (pg/->add-signature! super sig)
49   (fn [_]                                                 ;; JaMoPP modification
50     (doseq [o others]
51       (edelete! (@pg2jamopp-map-atom (:omember o)))
52       (swap! pg2jamopp-map-atom dissoc (:omember o)))
53     (j/->add-members! (@pg2jamopp-map-atom super) (@pg2jamopp-map-atom member))))

54 (defn find-accessors [pg tmember]
55   (filter #(member? tmember (pg/->access %))
56           (pg/all-TMembers pg)))
```

It first applies the changes to the program graph by deleting all duplicate member definitions from all other subclasses of `super` and pulling up the selected member into `super`. It also updates all accessors of the old members in order to have them access the single pulled up member. Lastly, it returns a closure which performs the equivalent changes in the JaMoPP model and updates the reference to the inverse lookup map when being called.

A function encapsulating the changes is returned here instead of simply applying the changes also to the JaMoPP model because the ARTE TestInterface defines that the back-propagation of changes happens at a different point in time than the refactoring of the program graph. Thus, the solution's TestInterface implementation simply collects the closures returned by appling the rules in a collection and invokes them in its `synchronizeChanges()` implementation.

Note that the rule's variant (1) immediately invokes the function returned by `do-pull-up-member!`. This is because this variant is not called by ARTE but is intended for extension task 2, and with that there is no need to defer back-propagation.

The rule `create-superclass` implementing the other core task is defined analogously, and the extension task 1 rule `extract-superclass` simply combines `create-superclass` with `pull-up-member`.

FunnyQT provides built-in functionality to let users steer rule application, i.e., choose an applicable rule and one of its matches and then apply the rule to that match. This feature is used for solving the second extension task of proposing refactorings to the user.

2.3   Step 3: Program Graph to Java Code

The core `pull-up-member` and `create-superclass` rules return closures which perform the refactoring's actions in the JaMoPP model when ARTE calls the `TestInterface`'s `synchronizeChanges()` method.

---

[8]The `accessible-from?` predicate has been skipped for brevity.

Then, the JaMoPP model needs to be saved to reflect those changes also in the Java source code files. This is done by the `synchronizeChanges()` method of the solution's `TestInterface` implementation.

```java
public boolean synchronizeChanges() {
    try {
        for (IFn synchronizer : synchronizeFns) { synchronizer.invoke(jamoppRS); }
        SAVE_JAVA_RESOURCE_SET.invoke(jamoppRS);
        return true;
    } catch (Exception e) { return false; }
      finally { synchronizeFns.clear(); }
}
```

`synchronizedFns` is the list of closures returned by the rules which simply get invoked and perform the same changes to the JaMoPP model which have previously been applied to the program graph. Thereafter, the JaMoPP resource set is saved which means that the source code files are updated accordingly.

3 Evaluation & Conclusion

In this section, the FunnyQT solution is evaluated according to the criteria suggested in the case description which was also used as the basis for the open peer review.

The FunnyQT solution is *correct*, i.e., all tests performed by ARTE pass, and it implements all core tasks. Thus, it is also *complete* and received a full score for the correctness and completeness criterium.

According to ARTE, the FunnyQT solution runs in less than a tenth of a second for all test cases on an off-the-shelf laptop so the *performance* seems to be good. Nevertheless, the benchmarking performed by the case authors suggested that all other solutions except for NMF perform even better. However, all the ARTE test cases are actually too small to provide meaningful numbers. And in any case, the execution time of the actual refactorings on the program graph and the back-propagation into the JaMoPP model are completely negligible when being compared to the time JaMoPP needs to parse the Java sources, resolve references in the created model, and serialize the model back to Java again.

Another strong point of the solution is its *conciseness*. It consists of only 271 NCLOC of FunnyQT code for all core and the two solved extension tasks and 145 NCLOC of Java code for the `TestInterface` implementation class required by ARTE.

The FunnyQT solution also received a high *extension score* because it provides runnable implementations for the extensions 1 (*extract superclass*) and 2 (*propose refactoring*).

A *main critique* of the solution and FunnyQT in general is that many developers used to languages with C-like syntax such as Java dislike FunnyQT's Lisp-syntax. Additionally, its functional emphasis where transformations and rules are essentially functions which might get composed and passed to higher-order functions requires a shift from the object-oriented to the functional paradigm. Although this provides several benefits it also requires more learning effort and might hinder the adoption of FunnyQT.

Nevertheless, the FunnyQT solution received a reasonably good reviewer score which paired with its correctness and completeness resulted in letting it carry off the overall winner award for this case.

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] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation*. In: *Transformation Tool Contest 2015*.