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 selected 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*. 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 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, pattern matching, out-place transformations, in-place transformations, bidirectional transformations, and some more. 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
3. Propagating changes in the program graph back to the Java code

All 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

Submitted to:
TTC 2015

(a) Parse the Java source code into a model conforming to the EMFText JaMoPP[5] metamodel.
(b) Transform the JaMoPP model to a program graph metamodel 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.

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

The transformation also minimizes the target program graph. 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 statics. As a result, the program graph contains only the information relevant to the refactorings and is reasonably small so that it can be visualized which is nice especially 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 depicting the rules responsible for transforming fields is given in the following listing. The transformation receives one single source model `jamopp` and one single target model `pg`.

```
(deftransformation jamopp2pg [[jamopp] [pg]]
  ...
  (field2tfielddef
   :from [f 'Field]
   :when (not (static? f))
   :to   [tfd 'TFieldDefinition {:signature (get-tfieldsig f)}])
  (get-tfieldsig
   :from [f 'Field]
   :id   [sig (str (type-name (get-type f)) " " (j/name f))]
   :to   [tfs 'TFieldSignature {:field (get-tfield f)
                                :type  (type2tclass (get-type f))}])
  (get-tfield
   :from [f 'Field]
   :id   [n (j/name f)]
   :to   [tf 'TField {:tName n}]
   (pg/->add-fields! *tg* tf))
  (type2tclass
   :from [t 'Type]
   :disjuncts [class2tclass primitive2tclass])
  ...)
```

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 `get-tfield` and `type2tclass` rules which are not shown because of space limitations.

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

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 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 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`.

```
(defrule pull-up-member
  ([pg pg2jamopp-map-atom jamopp]                                          ;; (1)
   [:extends [(pull-up-member 1)]]                                         ;; pattern
   ((do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)  ;; action
    jamopp))
  ([pg pg2jamopp-map-atom super sig]                                       ;; (2)
   [super<TClass> -<:childClasses>-> sub -<:signature>-> sig               ;; pattern
    sub -<:defines>-> member<TMember> -<:signature>-> sig
    :nested [others [super -<:childClasses>-> osub
                     :when (not= sub osub)
                     osub -<:signature>-> sig
                     osub -<:defines>-> omember<TMember> -<:signature>-> sig]]
    :when (seq others)                                                     ;; (a)
    super -!<:signature>-> sig                                             ;; (b)
    :when (= (count (pg/->childClasses super)) (inc (count others)))       ;; (c)
    :when (forall? (partial accessible-from? super)                        ;; (d)
                   (mapcat pg/->access (conj (map :omember others) member)))]
   (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 is already accessible from the superclass[6].

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.

```
(defn do-pull-up-member! [pg pg2jamopp-map-atom super sub member sig others]
  (doseq [o others]                                          ;; PG modification
    (doseq [acc (find-accessors pg (:omember o))]
      (pg/->remove-access! acc (:omember o))
      (pg/->add-access! acc member))
    (edelete! (:omember o))
    (pg/->remove-signature! (:osub o) sig))
  (pg/->remove-signature! sub sig)
  (pg/->add-defines! super member)
  (pg/->add-signature! super sig)
  (fn [_]                                                    ;; JaMoPP modification
    (doseq [o others]
      (edelete! (@pg2jamopp-map-atom (:omember o)))
      (swap! pg2jamopp-map-atom dissoc (:omember o)))
    (j/->add-members! (@pg2jamopp-map-atom super) (@pg2jamopp-map-atom member))))

(defn find-accessors [pg tmember]
  (filter #(member? tmember (pg/->access %))
          (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 other core rule `create-superclass` is defined analogously, and the extension 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 both return closures which perform the refactoring's actions in the JaMoPP model when ARTE calls the TestInterface's `synchronizeChanges()` method.

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

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

```
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(); }
}
```

In there, `synchronizedFns` is the list of functions 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

In this section, the FunnyQT solution is evaluated according to the criteria suggested in the case description.

Correctness and completeness (max. 60 points). The FunnyQT solution passes all test cases provided by the ARTE testing framework thus it seems to be correct and complete with respect to the restricted subset of Java detailed in the case description. Thus, there is no obvious reason why it shouldn't earn the maximum of 60 points here.

Performance (max. 10 points). 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 except for *pub_pum3_1* where the execution takes 0.67 seconds. However, when running only that test (`execute --test pub_pum3_1`) its execution time is measured with 0.02 seconds which matches the execution time of the other cases. So this single outlier with `execute --all` seems to be a GC hiccup or something alike. However, the execution times for the toy examples tested by ARTE are not very significant anyhow. It would be very interesting to have tests covering larger code bases on which multiple refactorings depending on each other are performed.

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. As a reference, on a medium-sized project with 1358 files amounting to 257267 LOC, JaMoPP takes about two minutes for parsing and reference resolution.

Since the performance score will be assessed in comparison to other solutions, no value can be suggested here.

Reviewer opinion (max. 2 x 15 points). The strongest point of the solution is its completeness in that it solves all core tasks and two out of three extension tasks. FunnyQT's rule overloading and pattern inheritance features helped here a lot in order to avoid duplication of large parts of patterns. However, pattern inheritance trades comprehensibility for conciseness. The extended pattern is not visible in the extending pattern, thus the latter isn't understandable without the former. This is actually the same in OOP where the members inherited from superclasses aren't obviously visible in the subclasses.

A strong point of the solution is its conciseness. It weights only 271 NCLOC of FunnyQT/Clojure code for all core and extension tasks and 145 NCLOC of Java code for the `TestInterface` implementation class required by ARTE.

The performance is also good for the provided test cases although the EMFText JaMoPP which is used by the solution might the bottleneck when applying the refactorings to larger code bases.

With respect to debuggability, debugging FunnyQT rules quite doable. The interactive rule application combinator `interactive-rule` used for solving extension 2 is actually a high-level debugging tool which lets users steer rule application manually, inspect matches, and visualize (parts of) the model under transformation.

A weak point of the solution and FunnyQT (Clojure) in general can be seen in that it is dynamically typed, and thus type errors are runtime errors. Of course, the type world implied by a metamodel is different than the type world of Java (at least unless classes are generated from the metamodel). But for example, Henshin requires the metamodel to be known when specifying patterns and rules, and then the visual Henshin editor makes it impossible to define patterns which use types, references, or attributes which aren't defined by the metamodel.

Extensions (max. 15 points). The FunnyQT solution provides runnable implementations for the extensions 1 (*extract superclass*) and 2 (*propose refactoring*). Extension 3 (*detect refactoring conflicts*) hasn't been solved practically but an idea for its solution has been sketched. However, that requires some further additions to FunnyQT's state space generation facility. So the FunnyQT solution should score at least 10 out of 15 points for the extension task score.

4 Conclusion

This paper discussed the FunnyQT solution to the TTC 2015 Java Refactoring case. The solution solves all core tasks and also two out of three extension tasks, namely extensions 1 (*extract superclass*) and 2 (*propose refactoring*).

The solution is correct and complete. All tests performed by the ARTE testing framework pass.

The solution is also quite concise summing up to 271 lines of FunnyQT/Clojure code for the actual realization and 145 lines of Java code for the `TestInterface` implementation required by ARTE.

The performance of the solution is also good. All test cases are executed in small fractions of a second on an off-the-shelf laptop. However, all tests performed by ARTE are executed on very small toy programs so the significance of the measured execution times with respect to real-world code bases is questionable.

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.* To appear.

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