Software Evolution

Assignment 2

16-03-2018

Robin Esposito (1280155) - r.esposito@student.tue.nl
Mattijs Jansen (0748947) - m.j.jansen.1@student.tue.nl

1. Introduction

The goal of this assignment is to implement a tool that automates the modernization of Java source code. More precisely, the tool will indicate the lines in the code where a weakly typed container is present, suggesting the appropriate object type. Moreover, it will generate new source code in which said lines will be replaced by their modernized version.
The tool will be developed using the meta-programming language *Rascal*[1] and it will be evaluated on the *eLib*[2] project.

2. Assumptions

The following assumptions were made regarding the analyzed source code: all the source code files are directly contained in the project folder, with no subdirectories. Each Java file contains a single class and inner classes are not supported. Lambda expressions are not supported either.

3. Tool

3.1. Architecture overview

The implementation of the tool is divided over three Rascal modules, plus the *Main* module.

3.1.1. FlowGraphsAndClassDiagram

The first module, *"FlowGraphsAndClassDiagram"* was not implemented by us, but we obtained the source code from the GitHub repository published by Jurgen J. Vinju[3]. We did not reproduce all the code in the module: we only used the *buildGraph* function, which takes a *FlowProgram* object as input and creates the actual OFG graph.

3.1.2. SuggestionGenerator

The second module we implemetned was *"SuggestionGenerator".* This module contains most of the logic of our application, structured in six methods: *getSingleSuggestions, doSingleCorrections, doSingleMethodCorrections, getDoubleSuggestions, doDoubleCorrections, doDoubleMethodCorrections.* Let us examine them one by one.

[1] Rascal - https://www.rascal-mpl.org/about/
[2] eLib project - https://github.com/cwi-swat/rascal-demo-workspace/tree/master/eLib
[3] Example code to compute flow graphs - https://gist.github.com/jurgenvinju/8972255

- *getSingleSuggestions:* this method takes as input a set of location objects, pointing to weakly typed collections in the code, where a suggestion can possibly be made. For each of these collections, we use the OGF graph and we track down what are the types of the objects that get assigned to said collections. These types are stored as a set, since we do not know how many different types (or subtypes) will be assigned to the same collection. Then, if at least one type assignment has been found, we use the *Util* module to retrieve the smallest common superclass of all the assigned types. Finally, we return a *Relation* object with all the type suggestions that we found.

- *doSingleCorrections:* this method takes as input the list of type suggestions we previously retrieved. For each suggestion, firstly we use the *Location* object to read the exact line from the source file where the problem is located. Then, we parse this line and we edit it to include the type parameter. We retrieve the corresponding file in the duplicated project folder and we update it with the edited line. Lastly, we append some new lines to the *"suggestions.txt"* file, to keep a trace of every modification.

- *doSingleMethodCorrections:* we considered that the type parametrization could be applied not only to fields and variables, but also to the return type of methods. Therefore, in this function we use the OFG graph to track down the methods in the project that return weakly typed collections. We use the list of suggestions obtained from the *getSingleSuggestions* function to link every method to the correct type. Then, as we did in the previous function, we read the relative line from the source code, we edit it and we modify it in the duplicated project. Lasly, we print a trace of the modification to the *"suggestions.txt"* file.

The remaining three methods are similar to the ones seen above, with one substantial difference: they do not apply to collections with a single type parameterization (such as *Set* or *List*), but to collections where two types are required (such as *Maps*). The type suggestion for the values in the map is obtained as we have seen before, while the key type is obtained by using a specific function from the *Util* module, as we will see in next paragraph. The rest of the implementation is identical, except for the obvious differences in parsing and printing the lines of code.

3.1.3. Util

The *Util* module contains two functions which we considered to be a bit more general in purpose, and were therefore separated from the previous module: *getSmallestCommonSuperClass and getKeyTypes*.

- *getSmallestCommonSuperClass:* this method takes as input a set of location, representing various classes of the analyzed project. We use the *M3* model to analyze the inheritance hierarchy of each class. Than, we compare the hierarchies of all the classes in the set to find the first common element, which is the smallest common superclass, and we return it. If the set contains a single class, that class is returned by the method. If no common superclasses are found, the method return the generic Java type *Object*.

- *getKeyTypes:* this method takes as input an object of type *Location,* pointing to a weakly typed collection in the code, and uses the *M3* model to get the type of keys assigned to that collection. This is done by parsing the source code, in order to find either a line in which the key is instantiated at the same time as it is inserted in the map, or a different line in which the type of the key is declared. We collect all the found key types in a set. Since the keys can belong to different subclasses, we use the function seen in the previous point to find the smallest common super class, which is then returned by the method.

3.1.4. Main

In the *Main* module, all the methods seen before are combined together in the *"get_suggestions"* function. Firstly, we obtain the *M3* model and the OFG graph. We create a duplicate of the project that we are going to analyze, where we are going to carry out the modernization: we copy the whole folder and all the contained source files. We also create (or reset, if it already exists) a text file named "*suggestions.txt",* where we are going to print a trace of the modifications. Then, we use the *M3* model to obtain a set of the weakly type collections in the project that require a single type suggestion. Using the methods in the *SuggestionGenerator* module, we generate a list of type suggestion and we carry out the corrections for parameters, fields and methods. We repeat the same process for the collections requiring a double type suggestion.

3.2. Tool structure

The tools consists of four source files, corresponding to four different Rascal modules: *Main.rsc* (60 lines), *FlowGraphsAndClassDiagram.rsc* (21 lines), *SuggestionGenerator.rsc* (223 lines) and *Util.rsc* (104 lines).

3.3. Running the tool

To successfully run the tool, the project that is going to be used as input (in our case *eLib*) has to be already contained in the Eclipse workspace directory. The tool is going to create a new folder in the same workspace, with the suffix "_modernized" (e.g. *eLib_modernized*), containing the modernized version of the code. The tool is also going to output a text file with a trace of the carried out modifications. For every suggestion the file is going to show the location and the line number of where the problem was found, the original line of code and the edited version.

To run the tool, after importing the *Main* module in the *Rascal* console, the user can call the function *generate_suggestions(loc project)*, where the variable *"project"* is an object of type *location,* indicating the directory of the project to analyze. For reasons of practicality, the tool also includes a run() function, to quickly run it with the default value of *"|project://eLib/|"*.

4. Results

4.1. Obtained suggestions

Here are reported the suggestions identified and carried out by our tool. For the *eLib* project, 16 possible modernizations were found: 7 regarding weakly typed containers (5 lists and 2 maps), 6 regarding *Iterator* objects and 3 regarding methods.

```
@|project://eLib/Library.java|(2936,1000,<116,6>,<116,34>)
  List docsFound = new LinkedList();
  List<Document> docsFound = new LinkedList<>();

@|project://eLib/Library.java|(3456,1000,<137,10>,<137,30>)
  Iterator i = loans.iterator();
  Iterator<Loan> i = loans.iterator();

@|project://eLib/Library.java|(3234,1000,<127,10>,<127,43>)
  Iterator i = documents.values().iterator();
  Iterator<Document> i = documents.values().iterator();

@|project://eLib/Library.java|(2637,1000,<105,6>,<105,34>)
  List docsFound = new LinkedList();
  List<Document> docsFound = new LinkedList<>();

@|project://eLib/Library.java|(2674,1000,<106,10>,<106,43>)
  Iterator i = documents.values().iterator();
  Iterator<Document> i = documents.values().iterator();

@|project://eLib/Library.java|(2349,1000,<94,6>,<94,35>)
  List usersFound = new LinkedList();
  List<User> usersFound = new LinkedList<>();

@|project://eLib/User.java|(115,1000,<8,12>,<8,36>)
  Collection loans = new LinkedList();
  Collection<Loan> loans = new LinkedList<>();

@|project://eLib/User.java|(1190,1000,<60,11>,<60,31>)
        Iterator i = loans.iterator();
        Iterator<Loan> i = loans.iterator();

@|project://eLib/Library.java|(2973,1000,<117,10>,<117,43>)
  Iterator i = documents.values().iterator();
  Iterator<Document> i = documents.values().iterator();
```

```
@|project://eLib/Library.java|(2387,1000,<95,10>,<95,39>)
  Iterator i = users.values().iterator();
  Iterator<User> i = users.values().iterator();

@|project://eLib/Library.java|(124,1000,<7,12>,<7,36>)
  Collection loans = new LinkedList();
  Collection<Loan> loans = new LinkedList<>();

@|project://eLib/Library.java|(2880,1000,<115,1>,<124,2>)
 public List searchDocumentByAuthors(String authors) {
 public List<Document> searchDocumentByAuthors(String authors) {

@|project://eLib/Library.java|(2585,1000,<104,1>,<113,2>)
 public List searchDocumentByTitle(String title) {
 public List<Document> searchDocumentByTitle(String title) {

@|project://eLib/Library.java|(2309,1000,<93,1>,<102,2>)
 public List searchUser(String name) {
 public List<User> searchUser(String name) {

@|project://eLib/Library.java|(62,1000,<5,5>,<5,30>)
  Map documents = new HashMap();
  Map<Integer, Document> documents = new HashMap<>();

@|project://eLib/Library.java|(95,1000,<6,5>,<6,26>)
  Map users = new HashMap();
  Map<Integer, User> users = new HashMap<>();
```

4.2. Performances

The execution time of the tool resulted to be between 15 and 20 seconds, on average. The slowness of the execution was quite predictable, as the result of numerous operations of scanning and parsing of the source files. Considering that the project that was used to test it consists of a small number of files of relatively small dimension, we can conclude that this tool could be successfully applied on a fairly simple real Java software. In case of a complex project, with a considerable volume of source code, we believe that the tool would still produce useful suggestions, but the excessive execution time would prevent it from being efficient.

5. Discussion

5.1. Strengths and weaknesses

To achieve a reliable performance, we firstly focused on precision when suggesting possible modernizations, because we wanted to guarantee the soundness of our system, so that the edited version of the project could still be compiled and executed. Then, we tried to improve the recall of the tool by adding other possible suggestions. In light of this way of proceeding, a weakness of the tool is that it does not find all possible modernizations in the *eLib* project: in *Main.java* it did not find the type parameters of two weakly-typed containers and of two *Iterators*. All the other possible modernizations were found and carried out correctly.

A strength of this tool is that the suggestions are not limited to weakly typed collections used as fields or parameters, but it also suggests the correct type parametrization for *Iterator* objects and for the return type of methods. Moreover, it is also to suggest types for containers with multiple parameters, such as *Maps*. Another strength of the tool is that its execution time makes it possible to use it on real world java program, if not excessively complex. On the other hand, since the tool relies on numerous parsing operations it is strictly connected to the Java syntax, which implies a high maintenance effort to support future evolutions of the language. An example of this problem are the lambda expressions, which are not supported by our version of the tool.

5.2. Further applications

Object flow graphs can be used to check how many times a method is called or how many objects of a particular type are created. This information can be used to find possible dead code in a program. From the object flow graph it can be retrieved that there are no links between certain methods or classes and the rest of the code. If this is the case, possible dead code is found. Moreover, the tool could be used to analyze the architecture of a project: by studying the connection between the various classes and the code files in which they are used, it is possible to get an insight on the modularity of the program.

6. Conclusion

In this report we have shown the implementation of a tool to modernize Java programs. The tool uses an *Object Flow Graph* to find parameter types of weakly-typed containers. It outputs a text file with the lists of the retrieved suggestions and at the same time it implements the modernization, by creating a new project, duplicating the original source code and modifying it appropriately.