Prediction of code evolution

Viktor Karabut

November 9, 2015

Contents

Abstract

Goal of this work is develop computer system that predicts possible improvements for specific Java code fragment. System parses Java method into ACT (approximate code trees) fingerprints and searches for similar code from database of fingerprinted, versioned and indexed OSS (open source software) projects. During this work I plan to obtains OSS code from Maven Central and construct chains of versioned ACT fingerprints. After that develop search engine for finding similar code in fingerprint database.

Introduction

Supreme goal of my work is an implementation of IDE plugin which check writed code and gives suggestion about code organization. For example some similar routines from different modules of large project can be refactored into one utility method. Another usage scenario is a code fragments which can be replaced with utilities available from some existing library. Often code reuse is limited by legal issues, such as patents, copyrights. However there is a wide variety of an open source projects, from where code can be used as an inspiration or even be directly used.

Our goal is to find not intentionally copied code, but code fragment which is similar because they implements similar algorithms or similar code routines. In many previous works (cite?) code clones considered as textual copies of code with changed formatting, added comments or renamed variables. In this article we emphasise on semantically similar code, which cannot even have similar statements, but which does semantically same work. For example for every while-cycle we can write semantically equivalent while-cycle or replace if-else with switch-case statement.

In the first chapter we consider questions about code similarity. In order to detect similar code fragments we will parse Java language code into AST. After that we introduce intermediate representation of parsed code: Abstract Code Trees (ACT). During translation to ACT we get rid of reduntant for our purpose information.

In second chapter we will consider fast search engine for code fragments. ACT each-to-each comparison complexity is $O(n^2 \cdot m^2)$ where n is number of trees and m is size of each treee. For large code fragment database speed of quadratic algorithms is unacceptable. To solve this issue we use a hashing mechanism. For this we use fingerprinting approach. Fingerprint is a generalized version of ACT, where removed some information. Fingerprinting allows to find in constant time get some relatively small set of matching ACT from where we can find needed code fragments using an quadratic algorithms.

Finally in third chapter we describe experiments with finding code fragments in open source projects fetched from Maven Central database.

1 Similar code detection

Simplest possible way to find code duplication would be line-by-line comparison of source lines. However code can differ by an indentation, comments, variable names.

Other approach is parse language construct into language-specific Abstract Syntax Trees (AST) and then compare each node by node using recursive similarity definition, as it did Baxter in his work [1]:

$Similarity = \frac{2 \cdot S}{2 \cdot S + L + R}$

where S is a number of shared nodes. L is a number of different nodes in left subtree, and R - in right subtree.

In this work we will use similar approach, but instead of working with language-specific AST we will translate it to language-agnostic intermediate code representation: Abstract Code Trees (ACT). ACT is laconic imperatice language. ACT doesn't meant to be executed, althroug this possible. Main reason of introducing ACT is a give formal definition of 'code similarity' which allows us to reason about effectiveness of our search methods.

Intermediate representation will allow to make language-agnostic search engine, so it would be possible to migrate to another language. Laconic by nature ACT will help to make search engine simple and effective.

As a bonus at Java translation step we get rid of various Java 'sugar' constructs, such as introduced in Java 6 'foreach' cycle for collection traversing. All possible

Not all Java constructs are translated to ACT. Some drop

1.1 Description of ACT

ACT (Abstract Code Tree) is a intermediate representation of code. ACT was inspired by Nielson's while language (cite?). ACT doesn't meant to be executed, however execution is also possible. Language describes these EBNF rules:

```
stmt = block | if | while | expr | try-catch| return | throw
block = begin, { stmt }, end
if = "if", expr, "then", block, "else", block
while = "while", expr, "do", block
expr = assign | var | call | str_literal | num_literal
assign = expr ":=" expr
call = "f(", str_literal, {, expr}, ")"
try-catch = "try", block, {"catch ", str_literal, block}
return = "return", expr
throw = "throw", expr
```

1.2 Translation ACT

ACT have much less constructs than Java. In some cases we express different Java language statements with same ACT constructs, or, in rare cases, we just drop them. Our goal is to get ACT representation for our similar-code search engine, so we intentionally loose some Java semantics, like 'goto' statements.

ACT doesn't have notion of unary or binary operators, so all basic arithmetic, logic and comparison operators are translated to special method call, for example Java expression '3*(2+1)' will be translated to 'f(*,3,f(+,2,1)'.

ACT if-statement is same as in Java except it always have else branch. Java switch-case statement are translated to series of consecutive if-statements. Translator expects that every case block are followed by 'break' statement. If break statement is absent translator would add it artificially, it isn't semantically correct behaviour but good-enough for our similar code detection.

```
switch(a): {
  case 1:
   doA(); break;
  case 2:
    doB(); break;
  default:
   doC(); break;
}
```

```
if (f('eq', a, 1) then
  begin f('doA') end
else
  begin
    if (f('eq', b, 1) then
      begin f('doB') end
    else
      begin f('doC') end
  end
```

Java language has 4 different cycle statements. Whyle-cycle translates as is. Java do-cycle converted to while by repeating the cycle body. For-cycle initialization moves outside of cycle and counter expression are appended to the end of body.

```
do {
  doA();
} while (cond());
```

```
doA();
while f('cond') do begin
  f('doA')
end
```

```
for (init(); cond(); counter()){
  doA();
}
```

```
f('init')
while f('cond') do begin
  f('doA')
  f('counter')
end
```

Added in Java 6 enhanced for-cycle is a syntax sugar for Java SE collection framework API. We replace them with equivalent for-cycle and then translate it to ACT while. For example these two Java code fragment are equivalent in sense of generated bytecode:

```
for (X x : xs){
  do(x);
}
```

```
for (Iterator it = xs.iterator(); it.hasNext();) {
  X x = (X) it.next();
  do(x);
}
```

Some statements are completely dropped during translation: different 'goto' statements, assertions and synchronization blocks. 'Finally' blocks isn't present on ACT, but its content added just after try-catch statement.

1.3 Definition of similarity

We define similarity between two ACT nodes is a float in interval from 0 to 1. Similarity should be equal 1 if ACT is identical, and be 0 if we compare nodes of different types. Now we can define similarity recursively for every node.

Two numerical literals are similar if and only if they are equal:

$$Sim(x_a, x_b) = \begin{cases} 1 & \text{if } x_a = x_b \\ 0 & \text{otherwise} \end{cases}$$

For string literals good candidate is Levenshtein edit distance. The Levenshtein distance between two words is the minimum number of single-character insertions, deletions or substitutions required to change one string into the other (cite?). To get value in needed interval, we will normalize output of Levenshtein function by dividing to string length.

$$Sim(s_a, s_b) = 1 - \frac{lev(s_a, s_b)}{max(length(s_a), length(s_b))}$$

Using same idea we can define editional distance for two code blocks: edit distance of two

code blocks if the minimum number of single statement insertions, deletions or substitutions required to change one code block into other. Insertions and deletions are count as 1 operation, substitutions cound as $1-sim(stm_a, stm_b)$ operations. Such approach will help to handle similar code fragment where only few lines were removed or added.

$$Sim(block_a, block_b) = 1 - \frac{lev(block_a, block_b)}{max(length(block_a), length(block_b))}$$

We define similarity of 'if' node as a linear composition of similarities of child nodes.

$$\begin{aligned} Sim(if(expr_a, block_{a,then}, block_{a,else}), if(expr_b, block_{b,then}, block_{b,else}))) = \\ C_{if,0}+ \\ C_{if,1}Sim(expr_a, expr_b)+ \\ C_{if,2}Sim(block_{a,then}, block_{b,then})+ \\ C_{if,3}Sim(block_{a,else}, block_{b,else}) \end{aligned}$$

Constant $C_{if,0}, C_{if,1}, C_{if,2}, C_{if,3}$ can be selected empirically such as $C_{if,0}, C_{if,1}, C_{if,2}, C_{if,3} >= 0$ and $C_{if,0} + C_{if,1} + C_{if,2} + C_{if,3} = 1$ to ensure that similarity always in interval 0..1.

Same approach we use to define similarity of 'while' nodes:

$$\begin{aligned} Sim(while(expr_a, block_a), while(expr_b, block_b)) = \\ C_{while,0}+ \\ C_{while,1}Sim(expr_a, expr_b)+ \\ C_{while,2}Sim(block_a, block_b) \end{aligned}$$

2 AST search engine

2.1 Hashing of code fragments

One to one comparisons of all ACT nodes required quadratic time and isn't possible for large code base. We need hashing mechanism for ACT where similar ACT have same hash codes. For this we introduce ACT fingerprinting structure. We produce fingerprint throwing away irrelevant information, generalizing types, normalizing expressions and replacing some construction with wildcards. We work at method level, so as a disadvantage we cannot handle refactorings such as a method inlining or method extraction.

First in our hashing function we can ignore variable names. The primitive types we preserve as is. In Java there are 8 primitive data types which we should preserve in our hashing mechanism:

boolean type, Unicode character type, 8-, 16-, 32-, 64-bit signed integers, IEEE 754 double- and single-precision floating numbers. As an option we can generalize all integer and floating types which are semantically similar.

Class type handling is different. To properly handle the class types we need to have information about full project class hierarchy. It means that we should parse not only whole project but all project dependencies. Moreover similar classes in different project can have different names, so considering class types would significantly limit variety of similar code. By considered above reason we throw away object information in our search engine. But there is one exclusion: we preserve Java standard library classes, such as 'java.lang.String' which are the same for every project.

Some expressions is normalized, for example chain of assignments, if assignment are not related with each other we can safely reorder them. We fingerprint ACT with every possible reorder and add them into our fingerprint database.

Some kinds of expression are removed while ACT processing. This is a simple log method calls, if statements which checks does logging is enabled or nor, assertions and class casts. To detect logging we just have base of known logging frameworks. Most of Java projects use SL4J, java.logging, commons-logging etc.

2.2 Database and Maven

Apache Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: First, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven Central Repository, and stores them in a local cache.

Build artifacts in maven repository is versioned and often provided with source code. It allows as to trace evolution of project. In this paper we user artifacts from Maven Central Repository. Maven Central Repository is the default repository for Apache Maven and it serves only Open Source projects. According Maven Central Repository statistics in November 2014 there were over 800 thousands artifacts.

For accessing Maven repository there is well documented Maven Plugin API for fetching and managing artifacts.

3 Experiments

About experiments, benchmarks, examples etc.

4 Conclusion and Future Work

How does class file look like? Check out JVMS [2].

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society Press, 1998.

[2] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, February 2012.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

Appendices

Appendix A: A very important messy details

Text of the appendix A.