Clone Detection in Haskell

Mariam Vardishvili, Lu Jin, Gohar Irfan Chaudhry

Background:

Reusing code fragments during programming is a common activity. Copying and pasting creates multiple fragments that share common features. While, this is important and useful feature, having similar fragments in the code makes software maintenance harder. Various researchers [1–5] have reported more than 20–59% code replication. Clone detection is an open research question and it is not a trivial task, since code clones can be modified in many different ways, detecting them requires different approaches. The evidence is provided by the fact that around 66% of cloned code is further modified [6].

Definitions (according to [7]):

Definition 1: Code Fragment. A code fragment (CF) is any sequence of code lines (with or without comments).

Definition 2: Code Clone. A code fragment CF2 is a clone of another code fragment CF1 if they are similar by some given definition of similarity, that is, f(CF1) = f(CF2) where f is the similarity function.

Definition 3: Clone Types. There are two main kinds of similarity between code fragments:

- similarity of program text
    - Type-1 (Exact Clone): Identical code fragments except for variations in whitespace, layout and comments
    - Type-2 (Syntactical Code): Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
    - Type-3 (approximate clones): Copied fragments with further modifications such as changed, added or removed statements
- similarity based on their functionality (independent of their text)
    - Type-4 (behaviorally equivalent fragment): Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Clone detection process [7]:

1. **Preprocessing:**
    - Remove uninteresting parts
    - Determine source units
    - Determine comparison units / granularity
2. **Transformation**
    - Extraction – transforms source code to the form suitable as input to the actual algorithm:
        - Tokenization
        - Parsing

        - Control and Data flow analysis
    - Normalization – optional step to eliminate superficial differences:
        - Removal of whitespace
        - Removal of comments
          These steps are optional, and some of the researchers use whitespaces and comments as their metrics.
        - Normalizing identifiers
        - Pretty-printing of source code
        - Structural transformations
3. **Match detection**
4. **Formatting**
5. **Post-processing / Filtering**
    - Manual Analysis
    - Automated Heuristics

Different approaches for clone detection [7]

Text-based Techniques: Looks for strict identity of lines. Program is considered a sequence of strings and files are checked for identical sequence of strings.

Token-based Techniques (lexical approach): After tokenization, tokens of lines or files form the token sequence are compared for clone detection

Tree-based Techniques: The source units to be compared are subtrees of the parse tree or the AST. Comparison algorithms look for similar subtrees to mark as clones.

The semantic/PDG-based approach: Source units to be compared are represented as subgraphs of PDGs. This can detect non-contiguous clones but requires more time for large systems.

The syntactic/metric-based approach: In metric-based methods dissimilar metrics are assembled such as number of functions, number of lines etc. from code segments and then evaluates that metrics in spite of assessments of source code directly.

Hybrid Techniques: The clone detection in Wrangler, [Li and Thompson 2009, 2010], uses a hybrid technique. Clone candidates – which include false positives – are identified using the token stream, rendered as a suffix tree; the AST is then used to check which of these candidates is a true clone.

Properties of clone detection tools

1. Usage Facets:
    - Platform: the execution platform for which the tool is available.
    - External Dependencies: states whether the tool requires a special environment or additional other tools to work.
    - Availability: the kind of license under which the tool is made available.
2. Interaction Facets
    - User Interface

    - Output
    - IDE Support
3. Language Facets
    - Language Paradigm
    - Language Support
4. Clone Information Facets
    - Clone Relation
    - Clone Granularity
    - Clone Type
5. Technical Aspect Facets
    - Comparison Algorithm
    - Comparison Granularity
    - Worst Case Computational Complexity
6. Adjustment Facets
    - Pre-/Post-Processing
    - Heuristics/Thresholds
7. Processing Facets
    - Basic Transformation/Normalization
    - Code Representation
    - Program Analysis
8. Evaluation Facets

One sentence description of some existing tools [7]:

Text-Based:

One Sentence Description	**Tool/1st Author**
Text-Based	
Hashing of strings per line, then textual comparison	Johnson
Finds similar files with approximate fingerprints	sif
Data structure of an inverted index and an index with n-neighbor distance concept	SDD
Token-Based	
Suffix trees for tokens per line	Dup
Normalized token comparison integrated with Visual Studio	Clone Detective
Normalized token comparison with suffix-tree	clones
Tree-Based	
Hashing of syntax trees and tree comparison	CloneDr
XML representation of ASTs and anti-unification/code abstraction	CloneDigger
Serialization of syntax trees and suffix-tree detection	cpdetector
Metrics-based	
Clustering feature vector of procedures with neural net	Davey
Graph-based	
Approximative search for similar subgraphs in PDGs	Duplix
Searching similar subgraphs in PDGs with slicing	Komondoor

Clone Removal: transformation stage of clone detection

This step transforms identified clones from the clone classes in the first stage into calls to an appropriate abstraction. There are three important points in this part:

- Clone Refactoring Supports
  Implemented in HaRe: Folding, merging
  Anti-unification
- General Function for Abstraction
- Step-by-step Clone Detection
  Identify particular areas and invoke the refactorings;
  Semi-automatic detector: whether or not to create an abstraction

Clone Refactoring Supports

1. **Implemented in HaRe**

   Folding
   1) Function folding

   Function folding will substitute all sub expressions in a program. This could eliminate duplicate code by identify instances of common higher-order functions, such as foldr.
   For example, we can select the expression **sum [1..10]** in **1 + (sum [1..10])** and choose to fold against the definition of foldr. This would produce the expression **1 + (foldr (+) 0 [1..10])**.

   It is important to note that sum is an instance of foldr and is not a clone.

   2) As-pattern folding
   Sub-expression refers to a pattern binding in the same scope. As-pattern assigns the list pattern after "@" to the symbol “a" so that this symbol can be used in the right-hand side expression.
   As-pattern folding replaces particular sub expressions in a program.
   Example: **f (x:xs) = length (x:xs)** -> **f a@(x:xs) = length a**

   Merging

   Merging creates a new tuple-returning definition; the constituent components of the tuple are extracted from a number of identified definitions introducing sharing.
   Eg. Let’s consider two definitions: take aims to extract the first n elements from a list, while drop tends to drop the first n elements.
   line 3 to 6 are identical except for the function name, but line 2 in the two functions are different.

Merging will combine the definitions of take and drop together and form a new, recursive function splitAt. Line 2 contains the instance from take function, and line 4 comes from the function drop. The other lines below are the duplicated definitions in the two functions.

2. Anti-unification

Anti-unification was first described by Plotkin and Reynolds, which is a means to measure the structural differences.

Two terms: E1 and E2
Generalization term: **E**
Two substitutions: σ1 and σ2
σ1(E) = **E1** σ2(E) = **E2**

It naturally captures the most specific pattern which matches two similar abstract syntax trees, and provides a means to measure their structural differences. The process of finding an anti-unifier is called anti-unification.
E.g.

(p,n+m) **: alphaMerge** xs ys
(p,n) **: alphaMerge** xs ((q,m):ys)
(q,m) **: alphaMerg**e ((p,n):xs) ys

The expressions (:) and alphaMerge occur in the same place in each expression, and these parts are the anti-unifier.

Usage of anti-unification in HaRe
a.Calculate the distance between two similar AST from code fragments;
b.Capture the most specific pattern which matches the two AST;
c.Group similar trees into equivalent classes called clusters.

3. **Summary**
   All ways of refactoring help to remove duplicated code. This stage relies more on the users to locate and invoke the refactorings accordingly. Also, users to locate and invoke the refactorings accordingly.

General Function for Abstraction

This is a new refactoring for HaRe in the form of function abstraction considering both Instance of an expression and exact duplicates.

This refactoring will abstract expressions within a clone class, after the detection, HaRe will create a report and aks users for decision. Finally, it will create a new function call.
The refactoring takes into account instance of an expression and exact duplicates of the expression.
A new function is created automatically with the expression in the clone class;
Any formal parameters are added to the function abstraction.
i.e. Clone differs in a literal at each instance -> made into a parameter

Advantages of general function for abstraction

- Remove duplicate code within the program.
- Encourage code reuse and maintenance. Easy to maintain the functions at a later stage in the program (3 calls VS 1 call).
- Rename refactoring in HaRe[Li 2006]: introduce appropriate names for the abstraction and its arguments.

Step-by-step Clone Detection

One important symbol of HaRe detector: semi-automatic.

- **Clone detection**: fully automatic in the form of a report.
- **Clone refactoring**: Clone detection is designed to be fully automatic. Clone classes that are detected are presented to the user in the form of a report, where the user can then step through the instances, deciding whether or not they should be abstracted; this can be done by tabbing through the instances in an editor.

Benefit of step-by-step detection:

- Only want to replace some of the instance;
- HaRe reports clones classes with a large number of clones;
- Reply “A” for replacing all clones.

Performance

In order to test the true performance of the clone analysis, the HaRe clone detector was ran over a number of Haskell 98 programs varying in program size.

Table shows the results of applying the HaRe clone detector to the Haskell programs mentioned above. The first column shows the program's size by the number of tokens over the whole program.
The number of clones and the time taken in seconds is shown in Table 1 for the clone detection of expressions with token size larger than 3, 6 and 10 respectively.

For all applications, the clone detector was able to finish the detection in a reasonable time. The running time seems to be affected by the size of the program, and the number of candidates grouped as clones. This suggests that the algorithm for detecting clones is O(n2) in the worst case.
Interestingly, the clone detector reported more clone candidates when comparing all expressions with a token size ≥ 3 and many fewer candidates for expressions with a token size ≥ 10. This suggests that the clone detector gives more accurate results without giving false positives when expressions with a token size ≥ 6.
the performance could be greatly enhanced by making the comparison parallelizable. Another would be to use suffix trees to do an initial comparison, and then use the more costly AST to check clone candidates

Overall summary

The clone detector makes use of an AST-based approach to detect clones, and allows the user to select which clones to eliminate
The benefit of using an AST based approach is that it tends to lead to more accurate results over token-based methods.
1. The first stage was an automatic clone detection system. Clones are reported to a text file.
2. allows the user to highlight a particular instance of a clone. HaRe then asks the user whether or not they would like to replace the clones with a call to an abstraction in the identified class.

Future works

In the future the clone detection work can continue in a number of directions:
1. extend the tool to be used to identify the redefinition of library functions.
2. expect to look at a hybrid approach comparable to the Wrangler approach to let it be suitable for larger Haskell projects.

3. extend the clone analysis to consider the type information, which would help to avoid problems with generalisation and could build upon the existing type analysis infrastructure of HaRe.

References:

1. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd IEEE Working Conference on Reverse Engineering, Toronto, Ontario, Canada, pp. 86–95, July 1995

2. Ducasse, S., Rieger, M., Demeyer, S.: A Language independent approach for detecting duplicated code. In: Proceedings of 1st IEEE International Conference on Software Maintenance, Oxford, UK, ICSM 1999, pp. 109–118 (1999)

3. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of 1st IEEE International Conference on Software Maintenance, Monterey, CA, pp. 244–254 (1996)

4. Kontogiannis, K., Mori, R.D., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. J. Autom. Softw. Eng. 3(1), 79–108 (1996)

5. Lague, B., Proulx, D., Mayrand, J., Merlo, E.J., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: Proceedings of 1st IEEE International Conference on Software Maintenance, Washington, DC, USA, pp. 314–321 (1997)

6. A. Puri and S. Kumar, “Software Code Clone Detection Model,” International Journal of Computers and Distributed Systems, vol. 1, no. 3, pp. 69–74, 2012

7. Chanchal K. Roy, James R. Cordy and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming, 74(7), 2009.

8. H. Li and S. Thompson, Clone Detection and Removal for Erlang/OTP within a Refactoring Environment, in: ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation, in: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 169–178 (2009).

9 Bulychev, Peter, and Marius Minea. "An evaluation of duplicate code detection using anti-unification." Proc. 3rd International Workshop on Software Clones. 2009.