UNIVERSITY OF CALIFORNIA

Los Angeles

Clone Detection in R

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Valeed Malik

2017

ABSTRACT OF THE THESIS

Clone Detection in R

by

Valeed Malik
Master of Science in Computer Science
University of California, Los Angeles, 2017
Professor Miryung Kim, Chair

Copy-and-paste code offers an immediate convenience in exchange for latent risk. Clones scattered across a project become difficult to modify consistently. Simple awareness can mitigate this condition. A clone detection tool can identify code duplicates, empowering the user to eliminate the clone.

Despite the rise of popularity in data science, the R community has yet to see an effective industrial strength clone detector. This report presents a clone detection process specialized to R. Our tool is based on a metric-based approach with a post-processing step inspired by token-based techniques. Adapted from Deckard [JMS07], R source code is converted to an abstract syntax tree. Subtrees are encoded with characteristic vectors. These vectors are compared, offering a scalable and effective similarity calculation. To better compare code structure, we derive a program abstraction technique from CCFinder [KKI02]. String comparison is applied on generalized source code which has been stripped of superficial identifiers.

A systematic mutation test by Roy et al. [RCK09] is adapted to evaluate RClone's performance. The tool is also applied to 43K SLOC of production source code of R libraries: GGPlot, Broom and Knitr. RClone was able to effectively detect useful Type-1, Type-2 and Type-3 across the production source code. A sensitivity analysis, based on the Broom library, suggests an optimal threshold of vector distance at 7.5% and least common sequence at 67%. Evaluation also reveals potential opportunities to decrease false positive rate and to prune to the most useful results.

The thesis of Valeed Malik is approved.

University of California, Los Angeles

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Code clones, like any code smell, violate design principles and negatively impact code quality. Duplicated fragments can be dispersed to different lines, functions, files or codebases making code maintenance increasingly problematic. The notion of copy-and-paste code contradicts the principles of abstraction and single responsibility. However, there are only a few justified reasons to use code clones such as performance.

A clone detector is a useful tool in a developer's toolkit, enabling the developer to make the informed decision to refactor a clone. An effective clone detector can identify similar code despite any formatting, renaming and structural edits. The need for this fundamental tool will become increasingly necessary as R increases in popularity and becomes a leading language for statistical analysis & machine learning. R attracts users from various fields and skillsets, many with minimal programming experience. R lacks a comprehensive toolset, resulting in limited programmer productivity.

First we establish the taxonomy of a code clone. Clones exist to a varying degree. These classifications help identify pairs giving a relative sense of clone similarity.

Type-1: *Identical code fragments except for variations in whitespace, layout and comments.*
Type-2: *Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.*
Type-3: *Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.*
Type-4: *Two or more code fragments that perform the same computation but are implemented by different syntactic variants.* [RCK09].

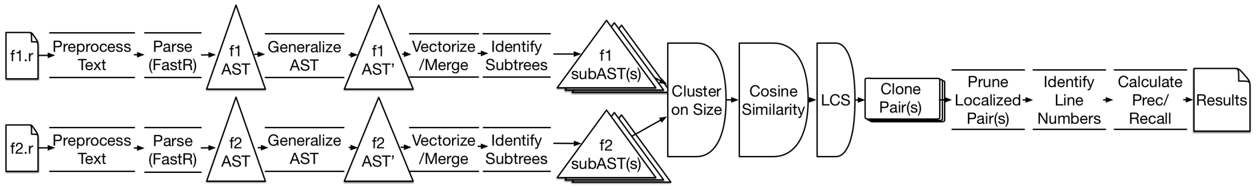
To construct an effective industrial scale process tailored for R, we must consider the various existing clone detection approaches. Each technique comes with tradeoffs between precision, scalability and applicability to R. Type-4 clone detection requires a semantic analysis approach which has shown to not scale to large programs [JMS07]. Thus we chose to focus on Type-1, Type-2 and Type-3 clones. With normalization/regularization techniques, generally all approaches are able to detect Type-1 clones independent of language. There are various approaches capable of detecting Type-2 and Type-3 clones. Many are specialized to common languages such as C, C++ and Java. Syntactic and token-based techniques abstract code to compare programs based on their underlying structure. This has been found to an effective method to detect Type-1, Type-2 and Type-3 clones however is prone to produce false positives. An effective approach must operate at a low false positive rate but also detect clones at scale.

Several clone detection approaches have contributed to the formulation of RClone. Existing text-based tools are able to support R but lack accuracy and scalability [RCK09]. Token-based tools introduce program normalization/abstraction/regularization to strip away concrete names and values. This method yields higher accuracy as the underlying structure is observed. Tree-based methods offer an alternative representative form by parse or abstract syntax tree. Tree matching algorithms can detect similar subtrees. Although tree-matching algorithms are computationally intensive, a metric can be derived from trees as an alternative lightweight representation of programs.

RClone is a hybrid of syntactic and lexical approaches. Our tool translates source code to an AST. Each subtree within an AST is approximated to a characteristic vector. This metric is the primary medium of a full subtree comparison [JMS07]. Moreover, RClone applies transformations to node identifiers with an encoding to strip away superficial details while also retaining syntactic structure [KKI02]. The AST is pretty printed with the modified identifiers to generate an abstracted source code. RClone applies the least common subsequence algorithm to the abstracted source code to generate the second similarity measure. Our approach is enumerate as follows :

- Source code is preprocessed. Syntax not supported by the parser, is removed
- Source code is parsed into ASTs.

Figure 1.1: Pipeline Overview



- Nodes not relevant to overall structure or characteristic vector are eliminated.

- Parser node type is mapped to characteristic vector elements.

- Characteristic vectors are initialized. For example, $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ indicates 1 occurrence of particular syntax type node within a single subtree.

- Characteristic vectors are merged bottom up. Now, each subtree root node node contains a vector approximation.

- Subtrees in similar size are compared with cosine similarity.

- Like token-based techniques, parameter nodes (identifiers and literals) are encoded by their position index among the reoccurrences in a tree.

- Since this comparison is made on vector approximations, source code pretty printed with replaced parameter node encodings is compared by a string comparison technique.

- Finally redundant result pairs are eliminated and aggregated into a text file.

We adapt the mutation test from Roy et al. [RCK09]. This systematic test is composed of fifteen scenarios, each with one unique mutation. Furthermore, RClone is also applied to a R production source code of some 43K LOC.

RClone is able to identify all fifteen scenarios from the systematic mutation test. However the tool also identifies smaller subtrees of lesser use, indicating a need for filtering and threshold refinement. The case studies validated the merit of each algorithms adopted. RClone identifies several notable clone pairs, each highlighting strengths and weaknesses of the approach. Characteristic vectors efficiently identified Type-1, Type-2 and Type-3 clones while also detecting many false positives. String comparison of abstracted source code supported the metric-based approach to reduce false positives. A sensitivity analysis, based on the Broom library, indicate the optimal threshold of distance threshold at 7.5% and least common sequence thresholds at 67%.

As of now, RClone is limited to single tree comparison and cannot detect a series of statements. Comparison of forests will enable detection of larger Type-3 clones. In order to increase precision, the vector similarity algorithm and post-processing must be refined. The efficacy of LSH must be empirically evaluated against the current cluster/cosine-similarity technique. Post-processing may be improved with a fully token-based technique instead of the current textual comparison. Finally, we can advance our hyper-parameter tuning. More sophisticated statistical methods can facilitate in developing a weighted scheme within the characteristic vector. This can help tailor our approach to R.

The paper is organized as follows. Section 1.1 surveys related works that have contributed to the formulation of our approach. In section 2 we expand on our approach. Section 3 presents an empirical study of RClone's performance. Section ?? discusses limitation observed. Section 4.1 suggests future work to advance our work and clone detection as a whole. Finally, section 4 concludes the paper.

1.1 Related Work

Most existing clone detection are implemented with a small set of languages. Majority of tools do not support R, let alone have a tailored clone detection process for the language.

However, language independent clone detectors can generically be applied to R programs. These tools follow a textual-approach, applying minimal transformation to the source code before the comparison stage [RCK09]. With comparison based on character strings, this approach can generally only find Type-1 clones [SFZ10]. Simian uses a regular expression technique, but it is a line-based technique which include comments and identifiers in comparisons [RCK09]. Normalization/abstraction/transformation expands the capabilities of Simian to enable detection of Type-2 clones. SDD on the other hand uses a *n*-neighbor approach to find near-miss clones.

This tool tolerates some gaps in similarity which can help detect near-miss clones, however this may cause lower precision [RCK09]. In general, text-based techniques are ill-suited to detect Type-3 clones [RCK09]. These limitations make text-based approaches undesirable as a primary algorithm for RClone.

Source code introduces variability far too great for a string comparison. Textual comparison is especially not suitable for a data science languages where constant values are common. Token-based tools are usually more effective and efficient but have limited language support [JMS07]. Parameter tokens (identifiers and literals) are encoded, abstracting away concrete names and values but retaining token order. Dup [Bak07] is a seminal work in token-based clone detection. Just as any token-based approach, Dup begins with lexing of source code into a sequence of tokens. Parameter tokens (identifiers and literals) are encoded by their position index among the reoccurrences in a line. Sequences are then converted into a suffix tree form where a matching algorithm extracts similar subsequences. CCFinder [KKI02] is a representative work of the token-based technique. CCFinder extends on Dup's normalization by defining specialized sets of transformation rules for C++ and Java. For example, namespace/library attributions, initialization lists and accessibility keywords are removed. Furthermore, all statements following an $if()$, do, $else$, $for()$ and $while()$ are automatically transformed to compound blocks. Parameter tokens are more finely encoded to express type for a constant or variable. The notion of normalization which abstracts raw identifiers but retains order is a major strength of token-based approaches. These approaches can identify Type-1 clones and scale well [RCK09] [JMS07]. Some token-based tools however may fall victim to minor edits and code restructuring due to a lack of normalization [JMS07].

Contrastingly, syntactic(tree-based) is a more robust approach. This technique requires source code to be translated to a parse tree or abstract syntax tree form. This allows tree-based tools to ignore formatting differences and comments [RCK09]. With source abstracted into a tree representation, more sophisticated matching algorithms such as Yang et al.'s Tree_Matching algorithm which offer a more accurate comparison [Yan91]. CDiff presents a dynamic programming approach for handling syntactic differences between subtrees. An alternative tree-based technique by Koschke et al. [KFF06] converts AST subtrees into node sequences and identifies subsequences with a suffix tree technique similar to the token-based Dup. The syntactic-approach allows source code to be expressed at a finer level than tokens but also be able to identify clones with less time complexity. Because tree-based techniques computationally expensive, tree fingerprinting and efficient vector clustering techniques are viable alternatives [JMS07]. A metric-based approach can are typically derived from tree-based techniques. Deckard is based on a characterization of trees as vectors in $\mathbb{R}^n$, where n is the number of different syntactic types. A characteristic vector is less sensitive to code restructuring edits [JMS07]. Deckard relies on vector similarity by a hashing and near-neighbor querying algorithm and eliminates the tree matching process, allowing scalability to millions of lines of code [JMS07].

We found Deckard's approach to be favorable as it highly extensible. The characteristic metric offers a level of generality which enables robust detection of Type-1, Type-2 and Type-3 clones. It is also scalable and can be highly specialized for a language. Vector comparison can be limited in effectiveness when comparing vectors of high dimensionality in euclidean distance. Deckard uses locality sensitive hashing(LSH) to cluster vectors. Alternatively, RClone initially filters potential clones by vector size and then applies cosine similarity. This approach is prone to false positives as different clones can yield similar metrics [RCK09]. Metric-based approaches abstract away not only concrete identifiers and values but also order. To retain order, RClone is complemented with token-based post-processing derived from CCFinder. Parameter nodes identifiers are encoded by syntax type and their position index for their occurrence in the tree. The transformation and parameter replacement process is applied to AST nodes. Pretty printed strings with the normalized node identifiers are used for string similarity comparison by least common subsequence.

CHAPTER 2

Approach

Here we present the techniques involved in each phase of the detection process. RClone detects clones through two steps: (1) a syntax-based technique, therefore, beginning with translation of source code to an AST which can be fingerprinted for comparison and (2) a token-based technique by generalizing the AST node identifiers to produce abstracted source code as an alternative form of comparison. Finally, we wrap up by explaining our result pruning process.

2.1 Preprocessing

To eliminate superficial differences, each program must be normalized to remove extra whitespace and comments. Moreover, parser limitations necessitate structural transformations. Lines containing library import statements $library(...)$ and $install.packages(...)$ are removed. Library resolution and data frame element extraction operators, :: and $ respectively,. Lines containing the operators cannot be entirely eliminated as they may also contain single parentheses or braces. RClone simply replaces the operators with a generic parsable name. These normalizations remove minor variations and ensures full parsable files.

2.2 Parsing

Since RClone is based on a syntactic approach, it requires a parser to convert R source code into an abstract syntax tree. The RClone parser is derived from an open source implementation of R named FastR [KMM14]. We utilize the Java parser from ANTLR 3.4. The code follows a visitor design pattern, offering a couple benefits in development.

```
[function, identifier, value, sequence, assignment, binary op,unary op, if, for, while, array]
```

Figure 2.1: Characteristic Vector Structure

The design pattern provides an interface for extensibility and adaptability for modifications in AST output or future R grammar support. A proprietary node class is designed to encapsulate data relevant to the RClone process such as characteristic vectors, file identification and concrete identifiers. To produce an AST better fit for the metric-based clone detection approach, FastR defined types are remapped to the syntax types defining the characteristic vector. RClone nodes are mapped to one of the following types: constant, variable, vector, operator(assignment, arithmetic), method(standard library, user defined) or control flow(if, for, while). The size of the set of types supported are sufficient enough to characterize R code and keep dimensionality at a minimum. The parser supports a high dimensionality of syntax types, sometimes applying an undesirable degree of syntax type decomposition, which requires some types to be mapped to umbrella terms.

Once a node's type is established, the respective node's characteristic vector is initialized. For example, the characteristic vector $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ indicates the presence of an identifier type note. However, vectors do not characterize their subtree until vectors are merged.

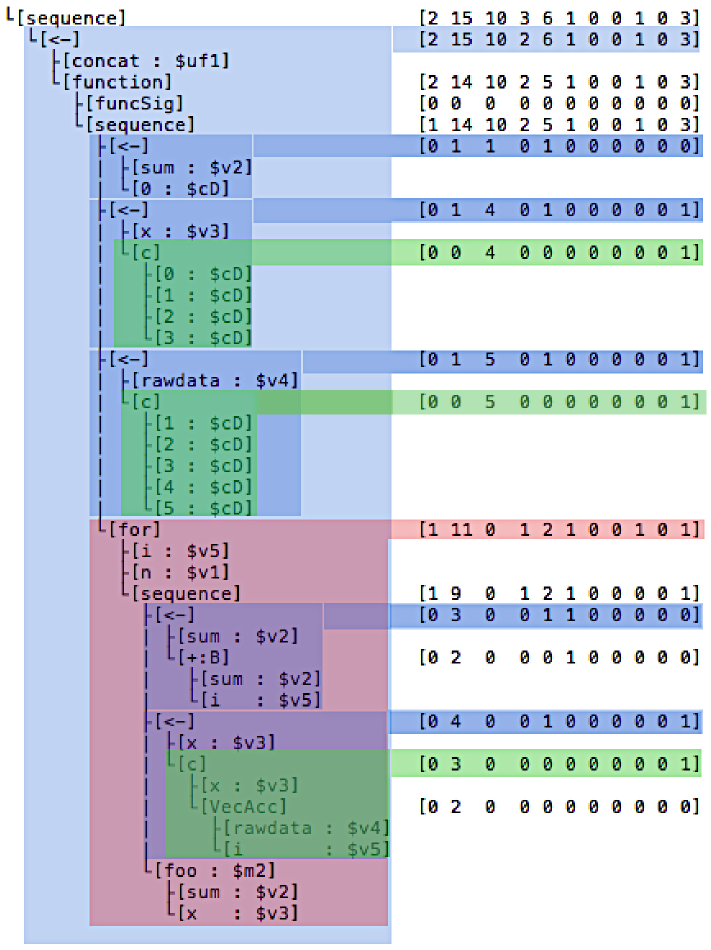
Moreover, this parser code is the mechanism driving AST traversal which can be replicated and repurposed, serving as the template for utilities. The first classed derived is the pretty printer. Given a node, the pretty printer will traverse and generate original or abstracted source code versions of the tree. The abstracted source is utilized for string comparison in the post-processing phase.

The second derived class traverses the AST to produce a text-based tree view. The tree is produced in a depth first search vertical representation, as seen in figure 2.3. Each AST node lies on a separate line. This enabled tree-outputs to display node level details such as node id, raw and abstracted identifiers and characteristic vectors as demonstrated in figure 2.3. This human friendly tree output provides intuition of syntactic structure of R functions. The tree printer also served as a helpful utility during development, debugging and reporting.

Figure 2.2: Sample.r - Raw

```
conCat <- function(n) {
    sum <- 0.0  #C1
    x <- c(0.0, 1.0, 2.0, 3.0)
    rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.
    for(i in n){
        sum <- sum + i
        x <- c(x,rawdata[i])
        foo(sum, x)
    }
}
```

Figure 2.3: Sample.r - Parsed & Vectorized



```
└[sequence]                                [2 15 10 3 6 1 0 0 1 0 3]
 └[<-]                                     [2 15 10 2 6 1 0 0 1 0 3]
   ├[concat : $uf1]
   └[function]                             [2 14 10 2 5 1 0 0 1 0 3]
     ├[funcSig]                            [0 0  0  0 0 0 0 0 0 0 0]
     └[sequence]                           [1 14 10 2 5 1 0 0 1 0 3]
       ├[<-]                               [0 1  1  0 1 0 0 0 0 0 0]
       | ├[sum : $v2]
       | └[0 : $cD]
       ├[<-]                               [0 1  4  0 1 0 0 0 0 0 1]
       | ├[x : $v3]
       | └[c]                              [0 0  4  0 0 0 0 0 0 0 1]
       |   ├[0 : $cD]
       |   ├[1 : $cD]
       |   ├[2 : $cD]
       |   └[3 : $cD]
       ├[<-]                               [0 1  5  0 1 0 0 0 0 0 1]
       | ├[rawdata : $v4]
       | └[c]                              [0 0  5  0 0 0 0 0 0 0 1]
       |   ├[1 : $cD]
       |   ├[2 : $cD]
       |   ├[3 : $cD]
       |   ├[4 : $cD]
       |   └[5 : $cD]
       └[for]                              [1 11 0  1 2 1 0 0 1 0 1]
         ├[i : $v5]
         ├[n : $v1]
         └[sequence]                       [1 9  0  1 2 1 0 0 0 0 1]
           ├[<-]                           [0 3  0  0 1 1 0 0 0 0 0]
           | ├[sum : $v2]
           | └[+:B]                        [0 2  0  0 0 1 0 0 0 0 0]
           |   ├[sum : $v2]
           |   └[i   : $v5]
           ├[<-]                           [0 4  0  0 1 0 0 0 0 0 1]
           | ├[x : $v3]
           | └[c]                          [0 3  0  0 0 0 0 0 0 0 1]
           |   ├[x : $v3]
           |   └[VecAcc]                   [0 2  0  0 0 0 0 0 0 0 0]
           |     ├[rawdata : $v4]
           |     └[i       : $v5]
           └[foo : $m2]
             ├[sum : $v2]
             └[x   : $v3]
```

2.3 Extraction

RClone would be able to compare similarity between ASTs however this is a computationally intensive approach [RCK09] .

We adapt Deckard's method of metric extraction. RClone transforms source code to characteristic vectors, a form proper for vector similarity algorithms. An n-dimensional characteristic vector approximates a tree where n denotes the number of syntactic types. Types are mapped to an Index as shown in figure 2.1. A node's vector is first initialized according to the immediate node type. Vectors are then summed to their parent node vector in a post order process. Once the vector merging fully propagates to the top, any node can be observed as a subtree. Observable in figure 2.3, characteristic vectors show a node type composition of any given subtree.

Vector similarity algorithms can fully compare all subtrees by still utilizing tree based data without the tree-matching cost. Assuming two trees of n and m nodes, a constant time vector similarity calculation avert the $O(n * m)$ full-subtree simultaneous traversal comparison. Calculation is dependent on the number of syntax types defined within the characteristic vector. We have committed smaller vectors in this figure for readability.

Furthermore, a node-type specific weighting scheme can easily be applied, an optimization to explore for future work.

2.4 Similarity Calculation

Vector Comparison To extract similar subtrees between two programs, RClone searches the space for a pair of similar characteristic vectors. A similarity value of one indicates Type-1 clones whereas a zero implies that the two fragments have no structural relation.

As we developed the characteristic vector for RClone to fit R, the dimensionality of the vector continued to increase. Having a high dimensional metric which characterizes a R program well requires higher complexity in the vector distance algorithm. Euclidean distance tends to perform poorly under high dimensionality. With the addition of each metric, the volume of space increases rapidly and depicts a sparsity within the characteristic vectors.

Dimensionality reduction is a simple solution, however this compromises the fine grained fit to R. Thus RClone instead compares vectors by cosine similarity, which is more resistant to the curse of dimensionality. This measure finds the cosine of the angle between two vectors. Since magnitude is not considered, RClone applies a constraint on size disparity between potential vector pairs. The size of a fragment is approximated by element-wise summation.

2.4.1 Post-Process: Textual Comparison

The metric-based approach allows comparison at scale however the metrics derived from the tree is an estimation. The characteristic vector is a composition of node types but eliminates the

Rule #	Rule description
R1 Remove Comments	**Comment → ϕ** Phi is a null sequences.
R2 Remove Import Statements	**'library(' Name ') ' → ϕ** Phi is a null sequence 'Name' is an arbitrary library name. Ex. $library(gdata)$ is deleted
R3 Generalize Literals	**Literal → '$c' TypeIdentifier** All literals are replaced label denoting const status along with literal data type. The identifier consists of a '$c' denoting constant followed by the type, which can be 'D' for double, 'S' for string, 'L' for logical and 'N' for null. Ex. $1 + TRUE$ is translated to $\$cD + \$cL$
R4 Generalize Variable Names	**Name → '$v' UniqueIdentifier** All names are replaced with a systematic unique label. The identifier consists of a '$v' denoting variable status followed by a unique variable number. The unique id replaces every occurrence of the particular variable within the span of the file. Ex. $sum < -1 + TRUE$ is translated to $v2 < -\$cD + \$cL$. This instance of sum is the second unique variable.
R5 Generalize Method Names	**stdR '()' → stdR '()'** **userDefined '()' → '$uf' UniqueIdentifier** **Unknown '()' → '$m' UniqueIdentifier** User defined method identifiers consist of a '$uf' followed by a unique method number. Standard function ids are syntactically untouched but still assigned a unique method number. Ids for methods falling in neither group consist of '$m' followed by a unique method number. Ex. $foo()$ is translated to $m1()$ where foo is the first occurring method in the file.

Table 2.1: Transformation Rules

structural relationships between them. To reduce false positives, pairs detected by the vector comparison are filtered by a token-based technique able to find structural similarities.

Structure is obtained by abstracting superficial differences which make detection of basic clones difficult. We rely on generalization techniques from CCFinder. Just like parameter tokens in a token-based approach, parameter nodes(identifiers and values) are encoded with a generalized identifier. Without effecting code structure, concrete names and values are abstracted away. This technique enables the pretty printer to generate code with generalized parameter nodes.

The following abstractions are formally defined in table 2.1. First, each variable and all reassurances within a tree are encoded with the same identifier. All variable nodes follow the same naming convention with v representing variable and followed by a unique numerical id. The id is allocated based on the order of unique occurrence. All repeat occurrences within the scope of a single file are assigned a single id.

This notation is shown in figure 2.4. $ is a denotation of an abstracted identifier. There are five unique variables in this fragment as indicated by $\$v5$, the highest valued variable identifier. The first variable, n which is encoded as $\$v1$, is a method argument. The previous explanation

```
conCat <- function(n) {
    sum <- 0.0 #C1
    x <- c(0.0, 1.0, 2.0, 3.0)
    rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
    for(i in n){
        sum <- sum + i
        x <- c(x,rawdata[i])
        foo(sum, x)
    }
}
```

```
$uf1 <- function($v1) {
    $v2 <- $cD
    $v3 <- c($cD, $cD, $cD, $cD)
    $v4 <- c($cD, $cD, $cD, $cD, $cD)
    for($v5 in $v1) {
        $v2 <- $v2 + $v5
        $v3 <- c($v3, $v4[$v5])
        $m2($v2, $v3)
    }
}
```

Figure 2.4: Sample.r - Generalized Pretty Print

indicates that the method's single parameter serves as the upper limit of the for loop and the identifier reoccurs in line 5.

Second, unlike the characteristic vector for dimensionality limitations, this technique can afford a more fine level of type classification. Constant nodes are decomposed to double, string, logical or null. This information is extracted from the original FastR node. All const nodes follow a naming convention with *c* representing const and the later letter indicating the constant subtype. In line two of figure 2.4, the identifier *$cD* is the indication of a *const* node of the type *double*.

Third, method nodes are mostly encoded in a similar fashion to variable nodes. Abstracted method nodes follow a naming convention with *m* representing method and followed by a unique numerical id. Exceptions to this convention are small set of standard methods which have a large influence on R code. To include them for comparison, these nodes are not generalized. Standard methods retain the raw identifier.

User defined methods, on the other hand, introduce too much variability for similarity analysis. User defined function nodes follow the naming convention with "uf" representing user define function, followed by a unique numerical id.

FastR grammar defines the variable assignment rule: *variable* ← *method*(). RClone discriminates between user defined, standard and other methods by evaluating the raw identifier of the method node. For example, a method identifier 'function' implies the l-value *variable* is a user defined function and must be encoded as such.

Generalized Pretty Print These node abstractions are included in the pretty print strings, as seen in figure 2.4, are included in the pretty print strings. Closely resembling the original, this code retains a syntactically equivalent structure with superficial details omitted. These strings are passed into a least common subsequence algorithm which provides another similarity measure between zero and one.

2.5 Pair Pruning

Although RClone compares subtrees within a size radius, more false positives can be removed. Full-subtree comparisons develop clusters of pairs which localize to a particular pair of nodes. Pair clusters are identified and pruned to a single pair which yields optimal similarity. A reduced number of repeating results yields the most useful clones but also reduces manual inspection work.

CHAPTER 3

Evaluation & Results

Evaluation of clone detection techniques is challenging as there is no agreed upon evaluation criteria or representative benchmark [RCK09]. Many evaluations are tailored to observe the technique and its parameters [RCK09].

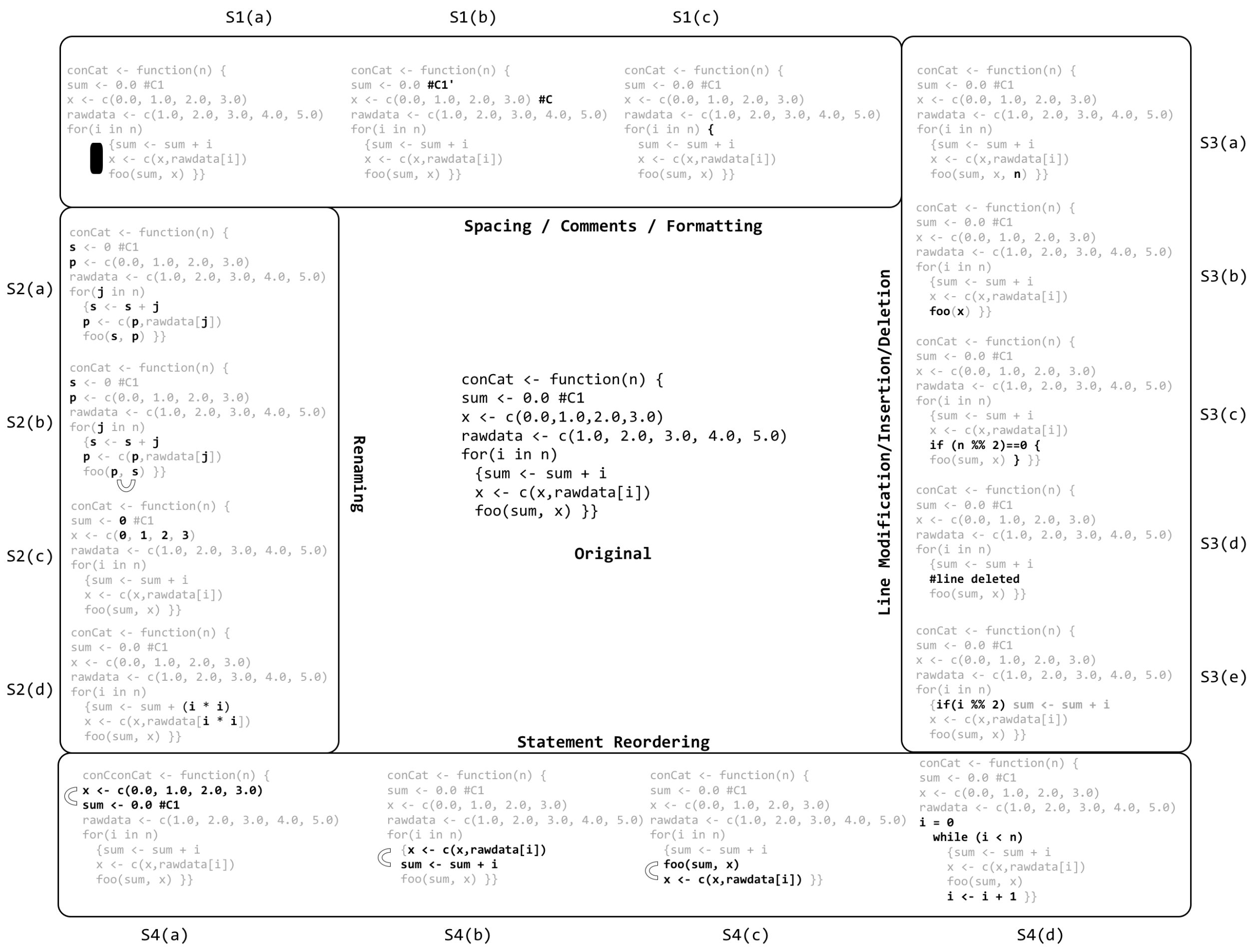
First we adapt the most extensive study to date, by Roy et al. [RCK09] which offers a universal criterion. The evaluation consists of a systematic mutation experiment composed of a series of hypothetical program editing scenarios representative of typical changes to copy/pasted code [RCK09]. Second we conduct a case study on prevalent open source R libraries and then manually inspect results.

3.1 Systematic Evaluation

We adapt the qualitative systematic mutation experiment by Roy et al. [RCK09] for R. The original study evaluates 42 existing tools in the clone detection space. Although not a concrete evaluation, this study provides a high level view of the potential of RClone in handling of the scenarios enumerated. Originally based on C code, performing an adapted study provides a relative measure against other established approaches.

The base C function is adapted for R which is embodied by sample.r(figure 2.2). This 12-line user-defined method is expressive of a diverse set of syntactic structures.

Figure 3.1: Mutation Taxonomy



S1(a)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
     {sum <- sum + i
     x <- c(x,rawdata[i])
     foo(sum, x) }}
```

S1(b)
```
conCat <- function(n) {
sum <- 0.0 #C1'
x <- c(0.0, 1.0, 2.0, 3.0) #C
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S1(c)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n) {
  sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S2(a)
```
conCat <- function(n) {
s <- 0 #C1
p <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(j in n)
  {s <- s + j
  p <- c(p,rawdata[j])
  foo(s, p) }}
```

S2(b)
```
conCat <- function(n) {
s <- 0 #C1
p <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(j in n)
  {s <- s + j
  p <- c(p,rawdata[j])
  foo(p, s) }}
```

S2(c)
```
conCat <- function(n) {
sum <- 0 #C1
x <- c(0, 1, 2, 3)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S2(d)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + (i * i)
  x <- c(x,rawdata[i * i])
  foo(sum, x) }}
```

Original
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0,1.0,2.0,3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S3(a)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x, n) }}
```

S3(b)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(x) }}
```

S3(c)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  if (n %% 2)==0 {
  foo(sum, x) } }}
```

S3(d)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  #line deleted
  foo(sum, x) }}
```

S3(e)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {if(i %% 2) sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S4(a)
```
conCconCat <- function(n) {
x <- c(0.0, 1.0, 2.0, 3.0)
sum <- 0.0 #C1
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  x <- c(x,rawdata[i])
  foo(sum, x) }}
```

S4(b)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {x <- c(x,rawdata[i])
  sum <- sum + i
  foo(sum, x) }}
```

S4(c)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
for(i in n)
  {sum <- sum + i
  foo(sum, x)
  x <- c(x,rawdata[i]) }}
```

S4(d)
```
conCat <- function(n) {
sum <- 0.0 #C1
x <- c(0.0, 1.0, 2.0, 3.0)
rawdata <- c(1.0, 2.0, 3.0, 4.0, 5.0)
i = 0
  while (i < n)
    {sum <- sum + i
    x <- c(x,rawdata[i])
    foo(sum, x)
    i <- i + 1 }}
```

Mutation Description 15 mutations are applied separately to the base method. Edits cover 4 categories of edit scenarios:

1. spacing/comment edits
2. renamed identifiers
3. line edits: insertion/deletion/modifications
4. semantically equivalent edits

Each category maps to Type-1, Type-2, Type-3 and Type-4 clones respectively and consists of several sub-scenarios.

Scenario Series 1 We test the tool's ability to detect clones in spite of Type-1 edits such as extra whitespace, styling/formatting and comment changes. RClone identifies all three copy/-pasted/modified fragments as clones of the original code.

As for any tree-based technique, the parsing phase is responsible for normalizing programs programs. Neither comments nor formatting changes effect AST production and are effectively removed before comparison. In fact, all methods including the base and the three sub-scenarios converge to the same AST.

To exclusively target scenario 1 related clones, more post-processing is needed [RCK09]. Characteristic vectors provide insufficient data as they are unable to detect internal positioning changes. Exclusive detection can be achieved within the current framework by searching for identical pretty printed source code without generalization.

Textual and syntactic techniques both identify Type-1 clones without difficulty. Cosine similarity and least common subsequence both indicate similarity of 100%.

Scenario Series 2 Edits commonly seen with IDE refactoring tool usage such identifier renaming, type modifying and argument swapping are applied.

RClone finds each of the three pairs to be identical with a cosine similarity of 100%. LCS notices the identifier renaming returning similarity values ranging from 91% to 98% with an average of 95%.

The tool is resistant to these Type-2 clone mutations such as renaming, as identifiers are abstracted away in the characteristic vector. In fact, it is due to the abstraction the metric provides, the tool cannot make a distinction between Type-1 and Type2 clone. [RCK09] Scenario-2(a), Scenario-2(b) and Scenario-2(c) all yield identical metrics. Token-based techniques such as Dup [Bak07] are not able to differentiate Scenario-1 clones from Scenario-2 due to initial normalization/abstraction before comparison. The AST retains raw identifiers enabling RClone to distinguish identifier changes through post-processing of pretty printed source code.

Scenario-2(d) tests a techniques ability to discern identifier changes from arithmetic modifications within the context of an argument. The tree-based approach is sensitive to syntactic constructs. The characteristic vector aptly represents both sets of binary operation and respective additional identifiers.

Type modification detection is limited by the types observed by the parser: double, logical, string and boolean. As exemplified by Scenario-2(c), a modification which stays within the types expressed above will not be recognized. Here, an integer changed to a double will be identified as a double. Another issue to note is R's dynamic typing. This will require semantic analysis, a challenge addressed for future work.

Scenario Series 3 This scenario introduces Type-3 clones with line(s) insertions, line(s) deletions and modification of whole lines. The average cosine similarity is 99% with least common subsequence of 96%.

When the only edit consists of an identifier being added or removed, as arguments are being modified in scenarios A and B, similarity is not greatly effected. A near miss clone with a single identifier modification will be observed within the metric for comparison. Depending on fragment size, the metric may not be able to emphasize a modification. Furthermore, an addition or removal may not be large enough to indicate a change due to calculations rounding up to 100% This is a vulnerability of the metric-based approach also noted by Roy et al. [RCK09].

This begs the question, what kind of modification should have greater influence? The addition of five values should not be as impactful as five while loops. What would be the proper set of weights for each syntax type? A weighting scheme that differentiates identifiers from types of greater structural influence can provide more accurate similarity levels. Types such as identifiers and values are more freely used with minimal influence on code behavior. Giving an emphasis to types which influence overall syntactical structure and execution should be able to sway similarity calculation more than an existence of a variable.

Currently, we mitigate this limitation of syntax-based techniques by post processing by least common subsequence. RClone's textual analysis performs better than vector comparison when detection of near miss clones. A single line modification within a twelve line fragment will yield a 90% similarity with least common and contrastingly 99% by metric analysis.

Scenario Series 4 Scenario 4 explores the rearrangements of data independent and dependent statements and declarations. RClone yields average cosine similarity is 99% with least common subsequence yielding 93%.

Syntax order is not retained with the metric-based approach. All rearrangement edits result in converging characteristic vectors. A robust clone detection should identify this change.

Vector analysis is able to identify Scenario-4 clones at a high level however least common subsequence provides a more accurate measure of similarity.

However data dependance is not addressed by our approach. This is observable in Scenario-4(d) where control flow is substituted with a semantically equivalent modification. A for loop is replaced with a $while$ along with compensations of iterator declaration and incrementation. Our metrics differentiates between control statements however a subset of these changes effect vector similarity calculation. Changing the for loop to a $while$ merely transfers the occurrence to another element position within the vector. The value is being passed from one dimension to another. Thus, the added identifiers are the modifications which are distinguishable by vector comparison.

Overall Performance RClone exhibits high recall with strict thresholds. The tool is able to cast a wide net and detect all possible clones. This however is done without a focus on particular clone types. RClone does not provide tunable parameters to target sub-scenarios individually. A focus on disseminating detected types would increase precision and produce more useful clones.

Roy et al. [RCK09] defines a rating between seven various levels of performance: *very well, well medium, low probably can, probably cannot and cannot* Based on this criteria, the current approach requires further tuning as it can detect at a level between medium and low proficiency. This score is justified as each scenario is detected with high recall exceeding low criteria.

Table 3.1: Subject Libraries

	File Count	LOC	Average LOC / File	LOC of Largest File
GGPlot2(2.1.0)	250	25,051	100	590
Broom(0.4.2)	84	8,383	99	683
Knitr(1.15.1)	70	9,781	139	903

3.2 Manual Inspection

To get a finer grain analysis, we must manually inspect results As there doesn't exist ground truth by past studies of this scope, .

Condensing results to the most relevant subset of detected pairs is a necessity to enable effective human inspection. Comparing every subtree pair possible through brute force produces many similar results that are localized around a true clone pair. Clustering similar results and finding a single optimal pair helped reduce repetitive results.

Another way we reduce results to a manageable size is to compare a subset of files to the entire codebase as a subset-to-all relationship. To get a rough understanding of subject libraries, we take the first 5 files and compare them to the entire codebase.

GGPlot2 We apply our tool on the prevalent ggplot2 graphics library. RClone constructed 441 ASTs and performed 97,461 pair unique comparisons. While scanning through output, single line clones are most prevalent. Type-3 clones detected are limited.

Figure 3.3 resembles scenario 2 as the difference between the two fragments is a single identifier. The metric-based algorithm fails to differentiate between a duplicate clone and this Type-3 clone. Figure 3.3's similarity measure suggests absolute similarity however the LCS shows 64% similarity. This is representative pattern among a large number of results as they differ in only a couple of identifiers.

Single line pairs do not give insight since they converge because they are based on funda-

Figure 3.2: Detected Pair A (GGPlot2)

Source Level

```
scale_linetype <- function(..., na.value="blank") {
  discrete_scale("linetype", "linetype_d", linetype_pal(), na.value=na.value, ...)
}
scale_linetype_continuous <- function(...) {
  stop("A␣continuous␣variable␣can␣not␣be␣mapped␣to␣linetype", call.=FALSE)
}
scale_linetype_discrete <- scale_linetype
```

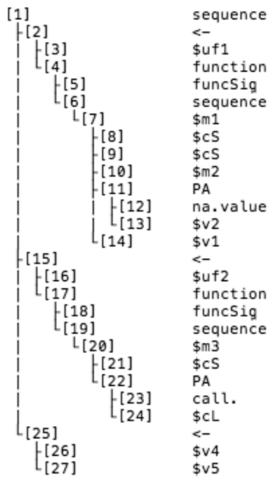
```
scale_shape <- function(..., solid=TRUE) {
  discrete_scale("shape", "shape_d", shape_pal(solid), ...)
}
scale_shape_discrete <- scale_shape
scale_shape_continuous <- function(...) {
  stop("A␣continuous␣variable␣can␣not␣be␣mapped␣to␣shape", call.=FALSE)
}
```

Abstracted Source Level

```
$uf1 <- function(..., na.value=$cS) {
        $m1($cS, $cS, $m2(), na.value=$v2, $v1)
}
$uf2 <- function(...) {
        $m3($cS, call.=$cL)
}
$v4 <- $v5
```

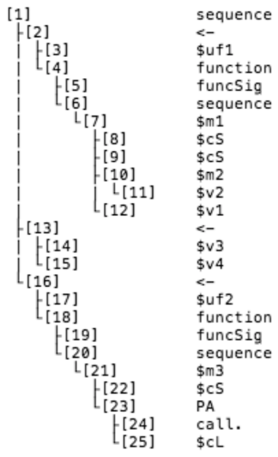
```
$uf1 <- function(..., solid=$cL) {
        $m1($cS, $cS, $m2($v2), $v1)
}
$v3 <- $v4
$uf2 <- function(...) {
        $m3($cS, call.=$cL)
}
```

AST Level



```
[1]                       sequence
├[2]                      <-
| ├[3]                    $uf1
| └[4]                    function
|   ├[5]                  funcSig
|   └[6]                  sequence
|     └[7]                $m1
|       ├[8]              $cS
|       ├[9]              $cS
|       ├[10]             $m2
|       ├[11]             PA
|       | ├[12]           na.value
|       | └[13]           $v2
|       └[14]             $v1
├[15]                     <-
| ├[16]                   $uf2
| └[17]                   function
|   ├[18]                 funcSig
|   └[19]                 sequence
|     └[20]               $m3
|       ├[21]             $cS
|       └[22]             PA
|         ├[23]           call.
|         └[24]           $cL
└[25]                     <-
  ├[26]                   $v4
  └[27]                   $v5
```

(a) scale-linetype-p.r



```
[1]                       sequence
├[2]                      <-
| ├[3]                    $uf1
| └[4]                    function
|   ├[5]                  funcSig
|   └[6]                  sequence
|     └[7]                $m1
|       ├[8]              $cS
|       ├[9]              $cS
|       ├[10]             $m2
|       | └[11]           $v2
|       └[12]             $v1
├[13]                     <-
| ├[14]                   $v3
| └[15]                   $v4
└[16]                     <-
  ├[17]                   $uf2
  └[18]                   function
    ├[19]                 funcSig
    └[20]                 sequence
      └[21]               $m3
        ├[22]             $cS
        └[23]             PA
          ├[24]           call.
          └[25]           $cL
```

(b) scale-shape-p.r

Figure 3.3: Detected Pair B (GGPlot2)

```
Similarity: 1.00 // LDMetric: 0.48 // AvgSize: 12  // LCS Pecentage: 0.642
(func: 2, id: 5, value: 0, seq: 0, assign: 3, binOp: 1, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
(func: 2, id: 6, value: 0, seq: 0, assign: 4, binOp: 1, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
-----------------------------------------------------------------------annotation-logticks-p.r
gpar(col=alpha(colour, alpha), lty=linetype, lwd=size * .pt)
--------------------------------------------------------------------------------geom-curve-p.r
gpar(col=alpha(transcolour, transalpha), lwd=transsize * .pt, lty=translinetype, lineend=
    translineend)
```

mental syntactical structure. RClone must support forest comparison to identify multi statement clones.

Figure 3.4 is a Type-3 clone. Here, a fragment has been duplicated in another file along with

```
Similarity: 0.99  // LDMetric: 0.65 // AvgSize: 52  // LCS Pecentage: 0.773
(func:11, id:17, value:5, seq:1, assign:11, binOp: 1, unOp: 0, if: 3, for: 0, while: 0, vec: 1)
(func:11, id:21, value:3, seq:1, assign:13, binOp: 1, unOp: 0, if: 3, for: 0, while: 0, vec: 1)
-----------------------------------------------------------------------------annotation-map-p.r
annotation_map <- function(map, ...) {
    stopifnot(is.data.frame(map))
    if(!is.null(maplat)) mapy <- maplat
    if(!is.null(maplong)) mapx <- maplong
    if(!is.null(mapregion)) mapid <- mapregion
    stopifnot(all((c("x", "y", "id")) %in% (names(map))))
    layer(data=NULL, stat=StatIdentity, geom=GeomAnnotationMap, position=PositionIdentity,
        inherit.aes=FALSE, params=list(map=map, ...))
}
-------------------------------------------------------------------------------------geom-map-p.r
function(mapping=NULL, data=NULL, stat="identity", ..., map, na.rm=FALSE, show.legend=NA, inherit
    .aes=TRUE) {
    stopifnot(is.data.frame(map))
    if(!is.null(maplat)) mapy <- maplat
    if(!is.null(maplong)) mapx <- maplong
    if(!is.null(mapregion)) mapid <- mapregion
    stopifnot(all((c("x", "y", "id")) %in% (names(map))))
    layer(data=data, mapping=mapping, stat=stat, geom=GeomMap, position=PositionIdentity, show.
        legend=show.legend, inherit.aes=inherit.aes, params=list(map=map, na.rm=na.rm, ...))
}
```

Figure 3.4: Detected Pair C (GGPlot2)

the modification of one line. These fragments are large enough for the similarity function to withstand the Type-3 edits whereas LCS is greatly effected. Clones similar to figure 3.4 one bring into question whether utilizing LCS is helpful. We are able to eliminate false positives however run the risk of creating false negatives.

The tool suffers from a large proportion of false positives as seen in figure 3.5. This can be due to the terse nature of R where many statements are limited to assignment-function-identifier node type combinations. However examples such as this one provide insight on how to improve our approach. The later fragment contains an if and also return statement that the other lacks.

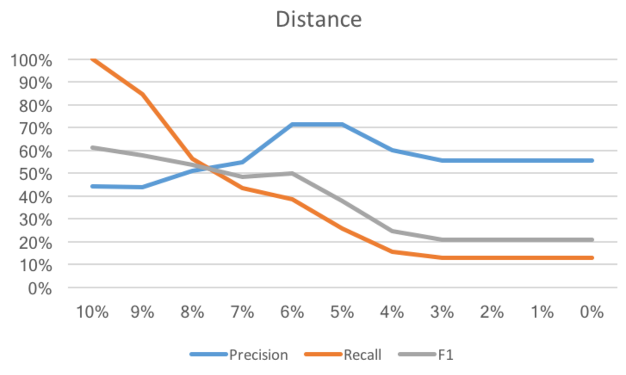
```
Similarity: 0.98  // LDMetric: 0.31  // AvgSize: 51 // LCS Pecentage: 0.527
(func:9,  id:19, value:3, seq:2, assign:14, binOp:1, unOp: 0, if:0, for:0, while:0, vec:1)
(func:10, id:24, value:3, seq:1, assign:12, binOp:2, unOp: 0, if:1, for:0, while:0, vec:0)
---------------------------------------------------------------------------annotation-map-p.r
GeomAnnotationMap <- ggproto("GeomAnnotationMap", GeomMap, extra_params="", handle_na=function(
    data, params) { data }, draw_panel=function(data, panel_scales, coord, map) {
    coords <- coord_munch(coord, map, panel_scales)
    coordsgroup <-
    grob_id <- match(coordsgroup, unique(coordsgroup))
    polygonGrob(coordsx, coordsy, default.units="native", id=grob_id, gp=gpar(col=datacolour,
        fill=alpha(datafill, dataalpha), lwd=datasize * .pt))
}, required_aes=c())
---------------------------------------------------------------------------geom-polygon-p.r
{
n <- nrow(data)
if(n == 1) return(zeroGrob())
munched <- coord_munch(coord, data, panel_scales)
munched <- munched[order(munchedgroup), ]
first_idx <- !duplicated(munchedgroup)
first_rows <- munched[first_idx, ]
ggname("geom_polygon", polygonGrob(munchedx, munchedy, default.units="native", id=munchedgroup,
    gp=gpar(col=first_rowscolour, fill=alpha(first_rowsfill, first_rowsalpha), lwd=first_rowssize
     * .pt, lty=first_rowslinetype)))
}
```

Figure 3.5: Detected Pair D (GGPlot2)

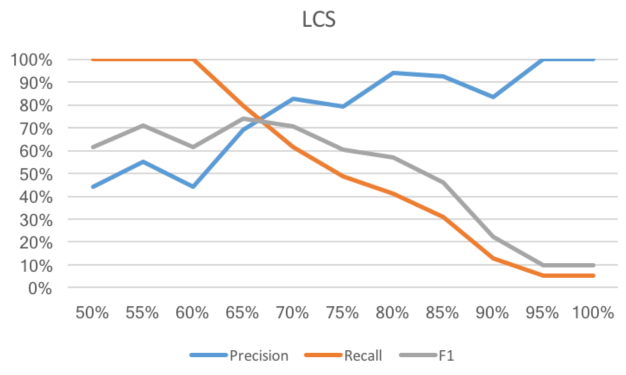
3.3 Sensitivity Analysis

The threshold variables which govern the characteristics of resulting pairs must be optimized to yield not only obscure but useful results.

Vector Distance There needs to be a distance, δ between the characteristic vectors of two trees that crosses a threshold and should no longer be considered to be a likely code clone. Threshold δ is tested between the bounds of one and zero. Results quickly showed that vastly differing trees would be matched since vectors may have the same distance from each other and at the same time have completely different internal values.



(a)



(b)

Figure 3.6: Threshold Sensitivity Analysis on Broom

Tree Size A minimum tree size limit is also optimized. With the limit being too small, the majority of the results get populated by small trivial trees lacking structure within its nodes, which are often lists of identifiers. For instance, with the *size* threshold at 3, the results contain trees that are too simple and common to the language. The *size* must be large enough to detect tree pairs based on a programmer's larger syntactic patterns rather than common compound syntactic terms defined in the language. An example of this case is shown when two trees are matched since they are calls to functions with the same number of arguments. However with a large minimum limit, say one hundred nodes, it is likely that many potential pairs would not be found. The issue with comparison by a distance formula can be exacerbated by the magnitude of *size*. As *size* gets larger the space for potential matches expands making a greater number of trees matches more likely Larger distances increase the number of possible vector formations that can achieve same distance. To mitigate, *size* must be limited to an optimal range.

Least Common Subsequence Similarity between trees is alternatively calculated by the longest common subsequence algorithm. This measure also ranges from one to zero. Having a threshold of one results in pairs that have identical code. Setting the threshold to zero allows any pair that matches by similar characteristic vectors but does not have to necessarily have a single character match between either source code.

CHAPTER 4

Conclusion

RClone combines techniques from the syntactic and token-based approaches. While tree-based fingerprinting allows fragment comparison at a scalable level, it must be supplemented with a more fine grained comparison. A token-based generalization is included to mitigate performance limitations related to program approximation. Mutation analysis considering fifteen typical edits indicate that RClone is not vulnerable to typical edits between clones. With a high recall, this approach has potential however incurs low precision which must be addressed in future studies.

4.1 Future Work

The vector similarity step can be improved by further minimizing the effect of high dimensionality. Our approach must be compared to techniques such as hashing as applied in CloneDr [BYM98] and Deckard [JMS07]. A viable technique, locality sensitive hashing hashes the input and can map similar items to the same bucket with high probability. Improving the vector similarity phase will enable the expansion of the characteristic vector express R code in more specific syntactic types. Node types such as'SimpleAssignVariable', 'UpdateVector' and 'UpdateExpression' would not have to be generalized under a single term.

We also would like to improve our string analysis technique. Instead of comparing string representation of the generalized fragments, token-based comparison would be a positive incremental step.

By complementing vector similarity with abstracted source code comparison, we learned that no single technique will produce perfect results. Apart from iterating on current algorithms, more techniques must be integrated to increase diversity in results.

Further tailoring to R is another priority. One point of improvement with respect to R is to achieve full language support. Currently, for source code to be parsable, the library resolution and data frame element extraction operators(::, \$, $library()$, $library.packages()$) must be extracted or modified. By modifying the FastR parser and AST output, files can be parsed without the need for such preprocessing. The other alternative is to build from scratch through ANTLR, a parser generator. Apart from language support, weighting schemes for types within the metric must be explored. Groups of identifiers can oversaturate a characteristic vector. Optimal weighting should give the greatest importance to types which influence overall structure.

Currently, single subtree pairs are matched. Comparison by adjacent subtrees can expand the size of detectable Type-3 clones In Deckard, a sliding window scans over adjacent subtrees. Additional characteristic vectors are created based on the adjacent combinations. While implementation of this window is straightforward, reporting and displaying of these clones suggest a greater rework.

CHAPTER 5

Appendix

5.1 GGPlot2 Clones

```
----------------------------------------------------------------------------------------
Similarity: 1.00  // LDMetric: 0.79  // AvgSize: 20   // LCS Pecentage: 0.837
facet-null-p.r (func: 4, id: 9, value: 0, seq: 2, assign: 2, binOp: 0, unOp: 0, if: 1, for: 0, while: 0, vec: 2)
facet-wrap-p.r (func: 4, id: 9, value: 0, seq: 2, assign: 2, binOp: 0, unOp: 0, if: 1, for: 0, while: 0, vec: 2)
----------------------------------------------------------------------------------------
if(themepanel.ontop) {
    panel_grobs <- c(geom_grobs, list(bg), list(fg))
}
else {
    panel_grobs <- c(list(bg), geom_grobs, list(fg))
}
----------------------------------------------------------------------------------------
if(themepanel.ontop) {
    panel_grobs <- c(geom_grobs, list(bg), list(fg))
}
else {
    panel_grobs <- c(list(bg), geom_grobs, list(fg))
}
```

```
----------------------------------------
Similarity: 1.00  // LDMetric: 0.94  // AvgSize: 49   // LCS Pecentage: 0.970
geom-contour-p.r    (func: 5, id: 15, value: 6, seq: 2, assign: 20, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
geom-density2d-p.r  (func: 5, id: 17, value: 5, seq: 2, assign: 20, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
----------------------------------------
{
geom_contour <- function(mapping=NULL, data=NULL, stat="contour", position="identity", ..., lineend="butt",
        linejoin="round", linemitre=1, na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
    layer(data=data, mapping=mapping, stat=stat, geom=GeomContour, position=position, show.legend=show.legend,
        inherit.aes=inherit.aes, params=list(lineend=lineend, linejoin=linejoin, linemitre=linemitre, na.rm=na.rm, ...))
}
GeomContour <- ggproto("GeomContour", GeomPath, default_aes=aes(weight=1, colour="#3366FF", size=0.5, linetype=1, alpha=NA))
}
----------------------------------------
{
geom_density_2d <- function(mapping=NULL, data=NULL, stat="density2d", position="identity", ..., lineend="butt",
        linejoin="round", linemitre=1, na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
    layer(data=data, mapping=mapping, stat=stat, geom=GeomDensity2d, position=position, show.legend=show.legend,
        inherit.aes=inherit.aes, params=list(lineend=lineend, linejoin=linejoin, linemitre=linemitre, na.rm=na.rm, ...))
}
geom_density2d <- geom_density_2d
GeomDensity2d <- ggproto("GeomDensity2d", GeomPath, default_aes=aes(colour="#3366FF", size=0.5, linetype=1, alpha=NA))
}
```

```
----------------------------------------
Similarity: 1.00  // LDMetric: 0.89  // AvgSize: 23   // LCS Pecentage: 0.939
geom-blank-p.r     (func: 3, id: 9, value: 0, seq: 1, assign: 9, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
geom-errorbarh-p.r (func: 3, id: 10, value: 0, seq: 1, assign: 10, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
----------------------------------------
geom_blank <- function(mapping=NULL, data=NULL, stat="identity", position="identity", ..., show.legend=NA, inherit.aes=TRUE) {
    layer(data=data, mapping=mapping, stat=stat, geom=GeomBlank, position=position, show.legend=show.legend,
    inherit.aes=inherit.aes, params=list(...))
}
----------------------------------------
geom_errorbarh <- function(mapping=NULL, data=NULL, stat="identity", position="identity", ..., na.rm=FALSE,
        show.legend=NA, inherit.aes=TRUE) {
    layer(data=data, mapping=mapping, stat=stat, geom=GeomErrorbarh, position=position, show.legend=show.legend,
    inherit.aes=inherit.aes, params=list(na.rm=na.rm, ...))
}
```

```
————————————————————————————————
Similarity: 1.00  // LDMetric: 0.73 // AvgSize: 23 // LCS Pecentage: 0.820
scale-linetype-p.r (func: 5, id: 6, value: 4, seq: 3, assign: 5, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
scale-shape-p.r    (func: 5, id: 6, value: 4, seq: 3, assign: 4, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
————————————————————————————————
{
    scale_linetype <- function(..., na.value="blank") {
        discrete_scale("linetype", "linetype_d", linetype_pal(), na.value=na.value, ...)
    }
    scale_linetype_continuous <- function(...) {
        stop("A continuous variable can not be mapped to linetype", call.=FALSE)
    }
    scale_linetype_discrete <- scale_linetype
}
————————————————————————————————
{
    scale_shape <- function(..., solid=TRUE) {
        discrete_scale("shape", "shape_d", shape_pal(solid), ...)
    }
    scale_shape_discrete <- scale_shape
    scale_shape_continuous <- function(...) {
        stop("A continuous variable can not be mapped to shape", call.=FALSE)
    }
}
```

```
————————————————————————————————
Similarity: 1.00 // LDMetric: 0.89 // AvgSize: 48  // LCS Pecentage: 0.934
geom-density2d-p.r (func: 5, id: 17, value: 5, seq: 2, assign: 20, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
geom-quantile-p.r  (func: 6, id: 16, value: 4, seq: 2, assign: 18, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
————————————————————————————————
{
    geom_density_2d <- function(mapping=NULL, data=NULL, stat="density2d", position="identity", ..., lineend="butt",
        linejoin="round", linemitre=1, na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
        layer(data=data, mapping=mapping, stat=stat, geom=GeomDensity2d, position=position, show.legend=show.legend,
                inherit.aes=inherit.aes, params=list(lineend=lineend, linejoin=linejoin, linemitre=linemitre, na.rm=na.rm, ...))
    }
    geom_density2d <- geom_density_2d
    GeomDensity2d <- ggproto("GeomDensity2d", GeomPath, default_aes=aes(colour="#3366FF", size=0.5, linetype=1, alpha=NA))
}
————————————————————————————————
{
    geom_quantile <- function(mapping=NULL, data=NULL, stat="quantile", position="identity", ..., lineend="butt",
        linejoin="round", linemitre=1, na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
        layer(data=data, mapping=mapping, stat=stat, geom=GeomQuantile, position=position, show.legend=show.legend,
                inherit.aes=inherit.aes, params=list(lineend=lineend, linejoin=linejoin, linemitre=linemitre, na.rm=na.rm, ...))
    }
    GeomQuantile <- ggproto("GeomQuantile", GeomPath,
        default_aes=defaults(aes(weight=1, colour="#3366FF", size=0.5), GeomPathdefault_aes))
}
```

```
------------------------------------------
Similarity: 1.00 // LDMetric: 0.62 // AvgSize: 78  // LCS Pecentage: 0.744
stat-ecdf-p.r      (func: 12, id: 29, value: 2, seq: 6, assign: 23, binOp: 0, unOp: 1, if: 2, for: 0, while: 0, vec: 2)
stat-function-p.r (func: 13, id: 30, value: 3, seq: 5, assign: 25, binOp: 0, unOp: 0, if: 1, for: 0, while: 0, vec: 1)
------------------------------------------
{
    stat_ecdf <- function(mapping=NULL, data=NULL, geom="step", position="identity", ..., n=NULL,
        pad=TRUE, na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
        layer(data=data, mapping=mapping, stat=StatEcdf, geom=geom, position=position, show.legend=show.legend,
                inherit.aes=inherit.aes, params=list(n=n, na.rm=na.rm, ...))
    }
    StatEcdf <- ggproto("StatEcdf", Stat, compute_group=function(data, scales, n=NULL, pad=TRUE) {
        if(is.null(n)) {
            x <- unique(datax)
        }
        else {
            x <- seq(min(datax), max(datax), length.out=n)
        }
        if(pad) {
            x <- c(-Inf, x, Inf)
        }
        y <-
        data.frame(x=x, y=y)
    }, default_aes=aes(y=..y..), required_aes=c("x"))
}
------------------------------------------
{
    stat_function <- function(mapping=NULL, data=NULL, geom="path", position="identity", ...,
        fun, xlim=NULL, n=101, args=list(), na.rm=FALSE, show.legend=NA, inherit.aes=TRUE) {
        layer(data=data, mapping=mapping, stat=StatFunction, geom=geom, position=position,
                show.legend=show.legend, inherit.aes=inherit.aes, params=list(fun=fun, n=n, args=args, na.rm=na.rm, xlim=xlim, ...))
    }
    StatFunction <- ggproto("StatFunction", Stat, default_aes=aes(y=..y..),
        compute_group=function(data, scales, fun, xlim=NULL, n=101, args=list()) {
        range <-
        xseq <- seq(range[1], range[2], length.out=n)
        if(scalesxis_discrete()) {
            x_trans <- xseq
        }
        else {
            x_trans <- scalesxtransinverse(xseq)
        }
        data.frame(x=xseq, y=do.call(fun, c(list(quote(x_trans)), args)))
    })
}
```

5.2 Broom Clones

```
--------------------------------------------
Similarity: 1.00 // LDMetric: 0.77 // AvgSize: 26 // LCS Pecentage: 0.852
kmeans_tidiers-p.r  (func: 10, id: 5, value: 1, seq: 5, assign: 4, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
mclust_tidiers-p.r  (func: 10, id: 6, value: 1, seq: 5, assign: 5, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
--------------------------------------------
augment.kmeans <- function(x, data, ...) {
    data <- fix_data_frame(data, newcol=".rownames")
    cbind(as.data.frame(data), .cluster=factor(xcluster))
}
--------------------------------------------
augment.Mclust <- function(x, data, ...) {
    data <- fix_data_frame(data, newcol=".rownames")
    cbind(as.data.frame(data), .class=factor(xclassification), .uncertainty=xuncertainty)
}
```

```
--------------------------------------------
Similarity: 1.00 // LDMetric: 1.00 // AvgSize: 34 // LCS Pecentage: 1.000
augment-p.r : []
glance-p.r : []
[7:1]  (func: 12, id: 4, value: 4, seq: 10, assign: 4, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
[23:1] (func: 12, id: 4, value: 4, seq: 10, assign: 4, binOp: 0, unOp: 0, if: 0, for: 0, while: 0, vec: 0)
--------------------------------------------
{
    augment <- function(x, ...) UseMethod("augment")
    augment.NULL <- function(x, ...) NULL
    augment.default <- function(x, ...) {
        stop("augment␣doesn't␣know␣how␣to␣deal␣with␣data␣of␣class␣", class(x), call.=FALSE)
    }
}
--------------------------------------------
{
    glance <- function(x, ...) UseMethod("glance")
    glance.NULL <- function(x, ...) NULL
    glance.default <- function(x, ...) {
        stop("glance␣doesn't␣know␣how␣to␣deal␣with␣data␣of␣class␣", class(x), call.=FALSE)
    }
}
```

5.3 Knitr Clones

```
———————————————————————————————————————
Similarity: 1.00 // LDMetric: 0.81 // AvgSize: 37 // LCS Pecentage: 0.855
hooks−html−p.r  (func: 14, id: 9, value: 3, seq: 5, assign: 2, binOp: 0, unOp: 0, if: 4, for: 0, while: 0, vec: 0)
hooks−latex−p.r (func: 14, id: 9, value: 3, seq: 5, assign: 2, binOp: 0, unOp: 0, if: 4, for: 0, while: 0, vec: 0)
———————————————————————————————————————
if(optionssplit) {
    name <− fig_path(".html", options, NULL)
    if(!file.exists(dirname(name))) dir.create(dirname(name))
    cat(x, file=name)
    sprintf("<iframe␣src="%s"␣class="knitr"␣width="100%%"></iframe>", name)
}
else x
———————————————————————————————————————
if(optionssplit) {
    name <− fig_path(".tex", options, NULL)
    if(!file.exists(dirname(name))) dir.create(dirname(name))
    cat(x, file=name)
    sprintf("\input{%s}", name)
}
else x
```

5.4 Broom/GGPlot2 Cross Project Clones

```
---------------------------------------------------------------
Similarity: 1.00 // LDMetric: 0.90 // AvgSize: 65 // LCS Pecentage: 0.924
map_tidiers-p.r     (func: 20, id: 17, value: 11, seq: 5, assign: 7, binOp: 4, unOp: 0, if: 0, for: 0, while: 0, vec: 1)
R/fortify-map-p.r   (func: 20, id: 17, value: 11, seq: 5, assign: 7, binOp: 4, unOp: 0, if: 0, for: 0, while: 0, vec: 1)
---------------------------------------------------------------
tidy.map <- function(x, ...) {
    df <- as.data.frame(x[c("x", "y")])
    names(df) <- c("long", "lat")
    dfgroup <- (cumsum((is.na(dflong)) & (is.na(dflat)))) + 1
    dforder <- 1 : (nrow(df))
    names <- do.call("rbind", lapply(strsplit(xnames, "[:,]"), "[", 1 : 2))
    dfregion <- names[dfgroup, 1]
    dfsubregion <- names[dfgroup, 2]
}
---------------------------------------------------------------
fortify.map <- function(model, data, ...) {
    df <- as.data.frame(model[c("x", "y")])
    names(df) <- c("long", "lat")
    dfgroup <- (cumsum((is.na(dflong)) & (is.na(dflat)))) + 1
    dforder <- 1 : (nrow(df))
    names <- do.call("rbind", lapply(strsplit(modelnames, "[:,]"), "[", 1 : 2))
    dfregion <- names[dfgroup, 1]
    dfsubregion <- names[dfgroup, 2]
}
```

REFERENCES

[Bak07] Brenda S. Baker. "Finding Clones with Dup: Analysis of an Experiment." IEEE Trans. Softw. Eng., **33**(9):608–621, September 2007.

[BYM98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Ann, and Lorraine Bier. "Clone Detection Using tract Syntax Trees." In Proceedings of the Inte8, pp. 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[JMS07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones." In Proceedings of the 29th International Conference on Software Engineering, ICSE '07, pp. 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. "Clone Detection Using Abstract Syntax Suffix Trees." In Proceedings of the 13th Working Conference on Reverse Engineering, WCRE '06, pp. 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code." IEEE Trans. Softw. Eng., **28**(7):654–670, July 2002.

[KMM14] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. "A Fast Abstract Syntax Tree Interpreter for R." SIGPLAN Not., **49**(7):89–102, March 2014.

[RCK09] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach." Sci. Comput. Program., **74**(7):470–495, 2009.

[SFZ10] G. M. K. Selim, K. C. Foo, and Y. Zou. "Enhancing Source-Based Clone Detection Using Intermediate Representation." In 2010 17th Working Conference on Reverse Engineering, pp. 227–236, Oct 2010.

[Yan91] Wuu Yang. "Identifying Syntactic Differences Between Two Programs." Softw. Pract. Exper., **21**(7):739–755, June 1991.