Clone Detection in Isabelle Theories

Dominik Vinan and Lars Hupel

Technische Universität München

Abstract. Duplicated code fragments within software projects complicate maintenance and require refactoring. Clone detection frameworks, such as ConQAT, offer well-engineered clone detection functionalities for a number of different programming languages. In this work, we developed a tool to search Isabelle theory sources for clones. This analysis takes the rich structure of Isabelle theories into account by extracting semantic information from document markup. After extraction, clone detection is performed using ConQAT's built-in facilities.

Keywords: Isabelle, Code clones

1 Introduction

Solving programming problems by copying, pasting and modifying existing code is a simple and effective method to achieve goals quickly. "Copy-and-paste" solutions may require less effort than re-inventing the wheel, but are harmful in the long term. Possible negative consequences caused by code clones include higher maintenance costs, faster system growth, and decay through propagation of errors.

While there are legitimate uses of code clones, e.g. in different branches of a source code repository, this work focuses on uncontrolled code cloning, i.e. copying, pasting and modifying code. Duplicating code is still a common practice in industry, instead of defining or using more abstract structures that facilitate reuse. The same problems also arise in theorem proving, where introducing another level of abstraction might make the formalization more difficult to handle.

Quality assurance frameworks like *ConQAT*[1] offer several analyzing and refactoring tools for a variety of programming languages. One of the well tested and developed building blocks of ConQAT is clone detection based on textual similarity. While code clone analysis is available for many programming languages, like Java, C# or C++, there is no support for Isabelle theory source code (which we will refer to as *Isabelle/Isar*). Processing Isabelle/Isar with external tooling poses the challenge that the language offers an extensible syntactic structure: both outer syntax (commands) and inner syntax (terms) may be extended by the user.

We have implemented a tool which extracts syntactic information from Isabelle theories via the Isabelle/Scala libraries and translates them into a format suitable for ConQAT. For ease of integration into the ConQAT framework, it is written in Java. Our tool works with both Isabelle2015 and Isabelle2016.

[1] available at `https://www.cqse.eu/en/products/conqat/overview/`

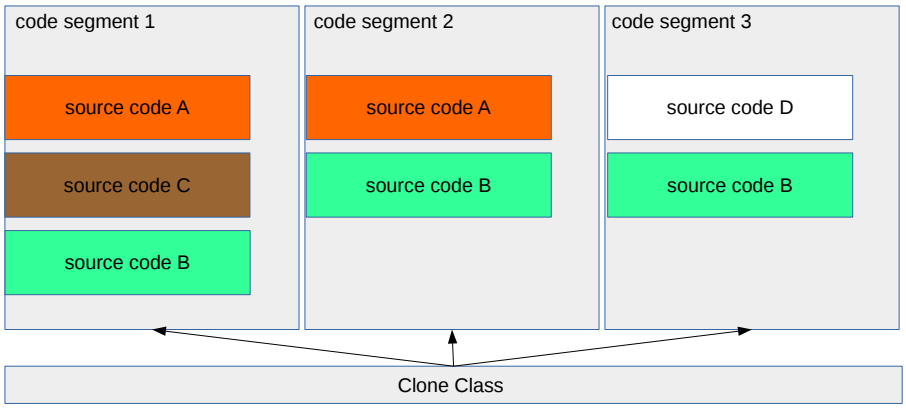

Fig. 1: A Clone Class and Clone Pairs (based on Roy & Cordy [2, §6])

Terminology According to Roy & Cordy [2, §6], equivalence relations can be used to describe the similarity relations between cloned code fragments.Two code fragments are in relation to each other if they are represented by the same *normalized* sequence. Those can be called *clone pairs*. An example is given in Figure 1. Two similar or identical pieces of code ("source code A") exist in the code segments 1 and 2, together forming a clone pair. A *clone class* is a maximal set of mutual clone pairs [2, §6]. In Figure 1, the three segments containing "source code B" form a clone class.

Clone types Code clones can be categorized into four types [2, §7.2]:

Type I "Identical code fragments except for variations in whitespace (may be also variations in layout) and comments."

Type II "Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments."

Type III "Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments."

Type IV "Two or more code fragments that perform the same computation but implemented through different syntactic variants."

All types are based on textual similarity, except for Type IV, which uses semantic similarity. The clone detection algorithm (provided by ConQAT) we used is based on textual similarity.

2 Syntactic and semantic analysis of Isabelle/Isar

Isabelle/Isar is the surface syntax for Isabelle theories. Because it is user-extensible, it is impossible to parse statically. Hence, we load theories into Isabelle using *libisabelle* [1] and observe the generated markup. In Isabelle/jEdit, this markup is used to e.g. annotate the source text with additional information, render the goal state, and display errors in proof processing [4]. But it also contains enough information (both syntactic and semantic) to produce a stream of tokens suitable for processing by other tools.

2.1 Document Markup

Markup is generated in an XML-style format. The following **datatype** declaration, taken from `~~/src/HOL/ex/Seq`:

```
datatype 'a seq = Empty | Seq 'a "'a seq"
```

produces hundreds of lines of markup, of which we reproduce an excerpt below (with some redundant information elided).

```
<keyword1 line='11' offset='120' end_offset='128' />
<tfree line='11' offset='129' end_offset='131' />
<delimiter line='11' offset='136' end_offset='137' />
<delimiter line='11' offset='144' end_offset='145' />
<tfree line='11' offset='150' end_offset='152' />
<string line='11' offset='153' end_offset='161' />
<entity line='11' offset='120' end_offset='128' ref='295711'
 def_line='1962' def_offset='85651' def_end_offset='85659'
 def_file='~~/src/HOL/Tools/BNF/bnf_lfp.ML' name='datatype'
 kind='command' />
<entity def='996462' line='11' offset='132' end_offset='135'
 name='Seq.seq' kind='type_name' />
<language offset='150' end_offset='152' name='type'
 symbols='true' antiquotes='false' delimited='false' />
<sorting line='11' offset='154' end_offset='156'>
 <language name='sort' symbols='true' antiquotes='false'
   delimited='false'>
  ...
 </language>
</sorting>
```

The markup contains syntactic information (e.g. what keywords are being used and where they are defined), but also semantic information (e.g. inferred types and sorts). For the purposes of clone detection, much of that information can be discarded. As the **<language>** element in the example indicates, Isabelle/Isar may contain many different languages with a varying levels of significance for clone detection. For example, the document language (following **text**, **txt**, ... commands) usually contains prose and can also be discarded.

2.2 Normalization

After filtering, the stream of source tokens is grouped into *statements*. A statement is an "atomic" unit of source text.

Statements are represented as strings, based on the kind of markup they contain. Entities (e.g. references to constants) are translated into their fully qualified name (e.g. `HOL.eq` for =). Literals, keywords, and delimiters (such as `|`) are copied verbatim. Free variables are represented by the constant string, suffixed with a fresh number.

The following example is taken from `~~/src/HOL/Library/FSet` and illustrates the normalized string representation of source text.

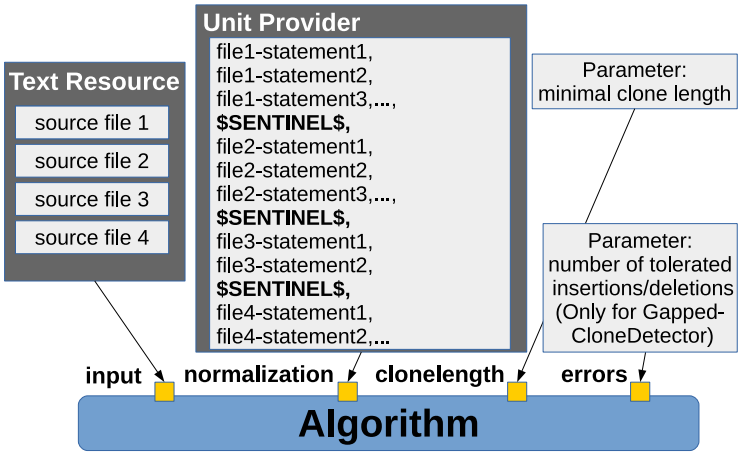

Fig. 2: Input Structure

```
definition (in term_syntax) [code_unfold]:
"valterm_femptyset = Code_Evaluation.valtermify ({|||} :: ('a :: typerep) fset)"
```

```
definition(inCode_Evaluation.term_syntax)[Code_Generator.code_unfold]:
id0HOL.eqCode_Evaluation.valtermify(FSet.fempty::(id1::Typerep.typerep)
FSet.fset)
```

The rules for statement grouping and string representation are currently defined in the source code of our tool. Support for new Isar commands, keywords or languages hence requires internal changes. For future versions, we are considering a facility for user-specified rules.

3 Clone Detection

Figure 2 shows an overview of the overall structure of our tool. ConQAT is a modular software quality framework [3], providing a variety of generic, configurable clone detection algorithms. It comes bundled with an Eclipse plugin which allows for graphical composition of *processors*, which act as functions transforming inputs to outputs. We have provided processors which produce inputs for ConQAT's `CloneDetector` and `GappedCloneDetector`. The former is used for Type I and II clones (identical code up to normalization), the latter for Type III clones (allowing edits or "gaps"). Both are parameterized by minimum clone length; the gapped detector is additonally parameterized by edit count. The default output mechanism is an HTML-formatted report.

4 Evaluation

In this section, we will show examples of detectable code clones in real-world Isabelle code, along with false positives and current limitations. Differences between code examples are highlighted in yellow.

4.1 Type I Clones

The following examples are taken from ongoing formalization work of one of the authors.

```
by pat_completeness auto
termination
apply (relation "measure (size \<circ> fst)") (* FIXME *)
apply auto
```

```
by pat_completeness auto
termination
apply (relation "measure (size \<circ> fst)")
apply auto
```

These fragments were detected as a clone pair in the same file on different locations and can be identified as Type I Clone, as they only differ in whitespace and comments.[2] In this situation (where the remaining part of the proofs are missing), the author should introduce appropriate simplifier rules such that termination proofs can be performed automatically.

4.2 Type II Clones

```
  assumes "pmatch t\<^sub>1 u = Some env\<^sub>1"
          "pmatch t\<^sub>2 u = Some env\<^sub>2"
  shows "t\<^sub>1 = t\<^sub>2"
using assms
```

```
  assumes "pmatch t\<^sub>1 u = Some env\<^sub>1"
          "pmatch t\<^sub>2 u = Some env\<^sub>2"
  shows "env\<^sub>1 = env\<^sub>2"
using assms
```

In this example, a Type II clone is identified because identifier names changed (t_1 vs. env_1). Although in both fragments the differing variables also have differing types, it is still considered a clone because typing information gets discarded.

[2] The presence of a "FIXME" in one fragment, but its absence in the other fragment serves to illustrate problems emerging from uncontrolled code cloning. Without clone detection, the author of the proof may fix the problem in the original fragment, but not in the cloned fragment.

4.3 Type III Clones

```
  unfolding \<open>irs = transform_irules irs'\<close>
            transform_irules_def
  by simp
then obtain cs where "(pats, cs) |\<in>| ?grp"
  using \<open>(pats, rhs) |\<in>| irs\<close> by force
```

```
  unfolding \<open>irs = transform_irules irs'\<close>
            transform_irules_def
  by simp
then obtain cs where "(pats, cs) |\<in>| ?grp" "rhs = Pabs cs"
  using \<open>(pats, rhs) |\<in>| irs\<close> by force
```

When choosing the gapped clone detection processor, Type III clones can be detected. The error rate (i.e. the number of changed statements) can be freely configured by the user. It is advisable to start with a low error rate, because a higher error rate naturally produces more false positives. However, high error rates can be used to identify strongly modified code, for example in a scenario where another developer copies code and modifies it according to their needs. In the above example, the error rate is 1. The cause for the clone is a missing lemma, which lead the author to copy some tedious, low-level reasoning, but also added some intermediate results along the way.

4.4 False Positives

The following fragments are taken from `~~/src/HOL/Word`.

```
    test_bit_bl word_size td_gal_lt_len [THEN iffD2, THEN nth_rcat_lem])
  done

lemma foldl_eq_foldr:
  "foldl op + x xs = foldr op + (x # xs) (0 :: 'a :: comm_monoid_add)"
  by (induct xs arbitrary: x) (auto simp add : add.assoc)
```

```
lemmas sdl = split_div_lemma [THEN iffD1, symmetric]

lemma given_quot: "f > (0 :: nat) ==> (f * l + (f - 1)) div f = l"
  by (rule sdl, assumption) (simp (no_asm))
```

The number of false positives can vary depending on the source text, and the configured minimum clone length and error rate. In the above example, we used a minimum clone length of 6 and an error rate of 6. Clearly, these fragments do not constitute clones. To avoid such false positives, the error rate should be set to a significantly lower value than the minimal clone length.

4.5 Languages of Isar

If analysed source text includes commands containing unsupported languages,[3] the whole command will be ignored by our tool. This avoids false positive caused by short and only partly-processed commands, but also means that we fail to detect certain clones.

4.6 Runtime Benchmarks

According to our measurements, most time during clone detection is spent by Isabelle to process theory sources and generate markup. For example, the formalization used above consists of 30000 lines (including dependencies), which takes about 90 seconds to load into Isabelle on our test machine. This is approximately the same time as opening the theories in Isabelle/jEdit. The actual analysis then only takes 15 seconds with a configured minimum clone length of 6.

5 Outlook

Our tool has been initially developed during a Bachelor's Thesis and is in experimental state. We outline future plans and potential improvements.

Analysis of ML fragments Currently, ML fragments are completely ignored. Although Isabelle produces markup for ML source code, further research needs to be done to devise a suitable filtering and normalization, given that ML is itself a full-blown language with rich syntax. In the future, Isar might gain support for even more languages, e.g. Scala, which poses the question about modular clone detection.

ConQAT analysis extension As the ConQAT framework offers a widespread collection of blocks and processors for software quality analysis, the existing bundle can be used as the basis for other Isabelle/Isar analyses.

Integration into Isabelle/jEdit As the clone detection output is both human-displayable HTML and machine-readable XML, an Isabelle/jEdit plugin could be developed, which would allow clone detection to be run with a push of a button. Since ConQAT's algorithms are quite fast, it may even be useful ot get live feedback in the spirit of the PIDE document model.

Plagiarism detection Clone detection algorithms might also be used to check for plagiarism, for example in university lectures where students are required to hand in Isabelle theories. We have ran some initial experiments on real-world homework submissions, but it is not clear yet how to configure the parameters to achieve a reasonable detection rate.

[3] At the moment, only non-ML commands defined in Pure and HOL are supported.

References

1. Hupel, L., Thomas, F.S., Archambault, A.: libisabelle: libisabelle 0.3.4 (Apr 2016), http://dx.doi.org/10.5281/zenodo.50347
2. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Tech. rep., Technical Report 541, Queen's University at Kingston (2007)
3. The ConQAT Team: Conqat user guide 2013.10 (Oct 2013), https://www.cqse.eu/download/conqat/conqat-book-2013.10.pdf
4. Wenzel, M.: Isabelle as Document-Oriented Proof Assistant, pp. 244–259. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-22673-1_17