Code Clone Detection in Rust Intermediate Representation

Davide Pizzolotto[1,a)] Makoto Matsushita[1,b)] Katsuro Inoue[2,c)]

Abstract: Code reuse is a common practice while developing software. While the detection of identical and nearly identical portions of code reached high accuracy, in the past years efforts shifted toward detecting seemingly different clones but with the same semantic value, anlysing almost always the Java language only. In newer languages, however, analysis is often complicated by the presence of syntactic sugar. In this paper we present an analysis of clones in the Rust language ecosystem at different compilation steps. The various stages in the Rust compilation, progressively flatten the code and remove unnecessary decorations until binary representation is reached. We analysed several Rust projects at source level and High-level IR, and compared the amount and types of clones for each refinement level.

1. Introduction

The practice of copying and pasting source code is frequently done by programmers, as this allows them to reuse the same source code multiple times. This leads to the creation of Code Clones: identical fragments of code scattered amongst a codebase. As harmless as these clones may seem, they instead present a burden for code maintainability and may become a huge technical debt. In fact, if a bug is found in a cloned fragment, it must be fixed in every instance of the clone, but this requires first the identification of every potential clone.

To alleviate this problem, during the years several code clone detection tools have been presented. These tools employ a variegate set of detection techniques: from token-based approaches of the *CCFinder* family [7] and *NiCad* [12], to the tree-based approach of *Deckard* [6], passing from graph based approaches [5] to the recent techniques employing deep learning [21]. As the detection performance in Type I clones (i.e. identical except for whitespaces and layout) and Type II clones (i.e. identical except for identifiers, literals and whitespaces) reached almost perfection, in recent years efforts shifted toward the detection of harder clones: Type III clones (i.e. identical up to a certain percentage) and Type IV clones (i.e. different code but with the same functionality).

In order to better detect this type of clones, sometimes also the binary code has been analysed, in particular by Ragkhitwetsagul et al. that applied a compilation and decompilation step to normalize differences between pieces of code with similar functionality [11]. Similar studies analyzed the possibility of applying normalization techniques directly to source code [10] and using LLVM Intermediate Representation to detect clones [2].

However, all these tools targets the same two languages: C/C++ and Java. Although some detectors provide a limited amount of additional languages, the study and the major focus is usually the Java language. This is done mainly due to the presence of a standardized clone benchmark, *BigCloneBench* [24] that contains clones in Java language only.

In this paper, we evaluate the detection of clones in a modern and new language: Rust. This particular choice is motivated by the fact that, unlike other programming languages, Rust guarantees the memory-safety, thread-safety and null-safety of its programs at the price of increased compilers checks and language restrictions. In order to validate statically these safety guarantees, several transformations are done during compilation. In this study, we analyze the clones evolution in this language, in the original source code and the intermediate pre-compilation step. We evaluate also the refactorability of each clone with respect to the additional language restrictions.

The paper is structured as follow: Section 2 explain some details about the Rust programming language in order to ease the understanding of this paper. After that, Section 3 explains our approach in performing the current study and Section 4 shows our research questions and the relative experimental results. Section 5 discuss about the limitations of our approach and Section 6 discusses related works in the field of clone detection. Finally, Section 7 closes the paper.

[1] Osaka University, Suita, Osaka 565–0871, Japan
[2] Nanzan University, Aichi, Nagoya 466–8673, Japan
[a)] davidepi@ist.osaka-u.ac.jp
[b)] matusita@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

2. Background

Before describing the approach we adopted to run our experiments, in this section we are going to explain the unique characteristics of the Rust language. The Rust language is designed around safety, in fact it is guaranteed by the compiler the prevention of data races, memory safety and type safety [15].

In order to satisfy these guarantees, the compiler relies on the ownership system [19]. This, in turn, contribute to two major differences from other languages:

Lifetimes
: When a reference to an owned variable is passed around the code, the compiler must ensure that the owned variable remains valid for the entire life the reference. This allows the compiler to safely drop a variable immediately as it goes out of scope without needing a garbage collector or a reference counter. Lifetimes **must** be explicitly assigned by the programmer if the compiler can not infer them.

Mutability
: Data races occurs when at least a pointer writes the data while another one can read the data, without any synchronization feature. In order to avoid this problem, Rust forbids the same reference to be hold mutably more than once, or to be hold mutably and immutably at the same time. The mutability for each variable **must** be declared by the programmer, otherwise the variable is considered immutable after the first assignment.

These two contraints may require additional keywords (e.g. `mut`, `'static`, ...) that might prevent the refactorability of otherwise identical snippets of code.

Moreover, in order to enforce these two constraints, the compiler employs a series of different Intermediate Representations (IRs) before emitting the final binary code. In order to understand this paper we are interested in particular in two types of IR: expanded code and High-level Intermediate Representation (HIR).

2.1 Macro expansion

The Rust programming language, unlike C or C++ does not have a preprocessor. However, the compiler still provides a macro engine in order to ensure better maintainability. In particular, several "common" implementation of methods (e.g. `clone`, `cmp`, `default`) can be created with builtin macros. These macros are then expanded into their respective code during the first phase of compilation [14].

Consider as an example the code in Figure 1. We can see that, despite the struct A and B having two completely different types, the implementation of the `clone` method performed by the macro `#[derive(Clone)]` is absolutely identical. In order to perform code clone detection in Intermediate Code is thus important to account for duplicated code generated by these kind of macros.

```rust
#[derive(Clone)]
struct A {
    inner_string: String
}
#[derive(Clone)]
struct B {
    inner_number: u32
}
```

(a) Original Rust code

```rust
struct A {
    inner_string: String,
}

#[automatically_derived]
#[allow(unused_qualifications)]
impl ::core::clone::Clone for A {
  #[inline]
  fn clone(&self) -> A {
    match *self {
      Self { inner_string: ref __self_0_0 } =>
      A {
        inner_string:
        ::core::clone::Clone::clone(&(*__self_0_0)),
      },
    }
  }
}

struct B {
    inner_number: u32,
}

#[automatically_derived]
#[allow(unused_qualifications)]
impl ::core::clone::Clone for B {
  #[inline]
  fn clone(&self) -> B {
    match *self {
      Self { inner_number: ref __self_0_0 } =>
      B {
        inner_number:
        ::core::clone::Clone::clone(&(*__self_0_0)),
      },
    }
  }
}
```

(b) Rust code after macro expansion

Fig. 1: *Automatic implementation of the* `clone` *method by the compiler, via a procedural macro.*

2.2 HIR and THIR

After creating the Abstract Syntax Tree (AST), the Rust compiler, converts this tree into a desugared version called High-level Intermediate Representation (HIR) [16]. The HIR differs from the normal AST for the following reasons:

- parenthesis are removed, as they are not necessary anymore with the AST structure.
- The `if let` syntax is normalized into the `match` syntax.
- The `for` loop syntax is normalized into the `loop` syntax.
- The `while` loop syntax is normalized into the `loop` syntax.
- Additional constraints are relaxed and converted into generics.

After these changes we expect the HIR to contains different clones than the original source due to normalizations performed by the compiler. It is the scope of this paper to

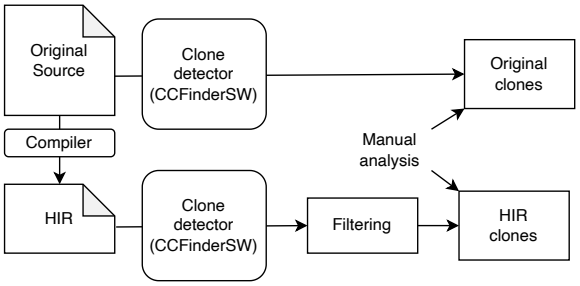

Fig. 2: Overview of the study

understand the evolution of original clones after the transformations in the HIR.

The HIR is then used to infer data types by the compiler, and transformed into Typed High-level Intermediate Representation (THIR) [18]. However, THIR does not add useful information for clone detection and as such we limit our study at the HIR level.

2.3 MIR and IR

Although not used in our study, the step after THIR is the Mid-level Intermediate Representation (MIR) [17]. The MIR code is more similar to a Control Flow Graph (CFG) rather than the original source code and is used mainly to checks the constraints of the ownership system [19]. If the ownership constraints are satisfied, the code can be lowered once again into a Codegen IR and the binary code finally generated. We plan to investigate the effects of MIR in code evolution as future works, as explained in Section 5.

3. Approach

An overview of the study is shown in Figure 2.

Firstly, for each case study, we gather each clone pair with the use of a Code Clone detector. Then, the original code is run through the Rust compiler and instructed to emit the HIR. At this point, we run the same code clone detector and collect the clone pairs for the HIR code. We then manually analyze each clone pair for every project in both the original code and HIR code, and report if the clone pair is really a clone or a false positive and compare it with the same clone in the HIR code. Note that we limit the analysis to Type I and Type II clones, given that most of the Type III and IV detectors target the C and Java language only.

3.1 Case Studies

We select 15 popular (more than 2M downloads) and decently sized (more than 2K Line of Code) Rust projects as our case study. These projects are listed in Table 1 along with the amount of Line of Code and their popularity. All these projects are publicly available in the Rust Package Registry[*1]. The amount of Line of Code are determined by the tool `cloc`[*2] and excludes comments and blank lines.

The scope of these project is greatly diverse: it ranges from byte manipulation (*bytemuck* and *bytes*), to data structures (*generic-array*, *smallvec*, *hashbrown*), including async computation (*crossbeam-channel*[*3] and *dashmap*).

Table 1: *Case studies evaluated, with version, number of lines of code excluding comments and blanks, and number of downloads*

Software	Version	SLOC	# of Downl. (10^6)
slab	0.4.6	1311	54
smallvec	1.8.0	2204	78
generic-array	0.14.5	2595	65
num-rational	0.4.1	3064	25
dashmap	5.3.4	2866	10
bytemuck	1.9.1	3026	8
bytes	1.1.0	4601	69
cgmath	0.18.0	8878	2
bstr	0.2.17	6227	26
hashbrown	0.12.1	10104	56
crossbeam-channel	0.8.1	15652	13
petgraph	0.6.2	20308	23
bitvec	1.0.0	26996	13
ndarray	0.15.4	29281	3

3.2 Compiler and Code Clone Detector

In our study we used an existing compiler to generate the HIR code and an existing Code Clone Detector to find the clone pairs. The compiler used to generate the HIR code is `rustc`, the compiler provided by the Rust project itself. This compiler enables emitting all the intermediate code such as HIR and MIR using the `-Z unpretty` flag. In particular, we used the option `-Z unpretty=expanded` to check the macro-expansion result, and `-Z unpretty=hir` to emit the HIR code. The compiler version we used is the 1.60.

Concerning the Code Clone detector, our options are pretty limited. Famous detectors like *CCFinderX* [7] and *NiCad* [12] do not support Rust. The modern *SourcererCC* tool [22], although not officially supporting Rust, can be easily extended, while an even newer tool called *MSCCD* [26] has builtin support. These two tools can support up to Type III clones. However they greatly lack in the clone reporting capabilities. For this reason, in this analysis we used the *CCFinderSW* tool [23] that, despite supporting only up to Type II clones, provides a more efficient reporting tool, allowing us to manually investigate all the clones in the 15 projects. The tool was run with a parameter `t`, the minimum number of tokens requires to signal a clone, equal to 65.

Note that it is not the scope of our paper to compare the difference between the various tools, rather than to analyze the code evolution in the Rust Intermediate Representation.

3.3 Filtering

As explained in Section 2.1, before generating the HIR, the compiler expands macros. These macros are designed to avoid clones in source code for repetitive tasks (i.e. implementing the `clone` method on a struct). However, given that the expansion is performed before the HIR generation, these macros will result in a lot of false positives: the macro

[*1] `https://crates.io`
[*2] `https://github.com/AlDanial/cloc`
[*3] *crossbeam-channel* is part of the *crossbeam* package

implementation can be found multiple times in the HIR but only once in the original source. In Figure 2 we can note a step called "Filtering" that refers exactly to this step: cleaning up the HIR code from obvious macro expansion.

The filtering step is actually performed in two different ways depending on the type of macro targeted. In case of `#[derive(...)]` macros, the one explained in Section 2.1, the expanded method will have the statement `#[automatically_derived]` prepended to it. This can be seen also in Figure 1 in the expanded code. The result of these macros can be easily removed by checking the AST, emitted by rustc, and by stripping away the entire block of code following the `#[automatically_derived]` keyword.

The second type of macros are the one defined by the `macro_rules!` syntax. The expansion of these macros is usually simpler than the one performed with the `#[derive(...)]` attribute, as they have a fixed set of parameters. We implemented a filter for the most common ones provided by the Rust language (e.g. `vec!`, `write!`, `println!`...) by using only regular expressions.

In both cases, however, our filter does not cover custom macros defined per-crate.

4. Evaluation

In order to determine the clone evolution in the Rust Intermediate Representation, we want to answer the following three Research Questions:

- **RQ1**: *type*. What type of clones can be usually found in a Rust project? Can the clones be easily refactored? RQ_{type} is a study on the type of clones that can be found in the Rust ecosystem. Rust is fundamentally different from other languages, as explained in Section 2, given its stricter compiler and limited usage of variables. In this research question we want to investigate if these differences implies additional clones that can not be refactored as one would do in a canonical language, without violating Rust's constraints of mutability and lifetimes.
- **RQ2**: *agreement*. How different are the clones between original code and HIR? What type of clones are detected only by one method?
  $RQ_{agreement}$ is meant to investigate the usefulness of the HIR representation in detecting clones. To answer this question we match all the clones reported in the original source code and all the clones reported in the HIR code and check for differences. We then report the principal causes of divergences between clones in the original code and clones in HIR code.
- **RQ3**: *accuracy*. How accurate is the clones detection in both original code and HIR? How many false positives are generated by the code? $RQ_{accuracy}$ is meant to investigate the accuracy of the clone detection in both original code and HIR code. This means reporting the amount of actual code clones and false positive code clones for each of the two code categories.

We ran this analysis on a Mac mini with M1 processor and 16GB of RAM, running macOS 12.3.1 and manually analyzed every single clone reported using the Gemini tool provided with *CCFinderSW*.

4.1 RQ1: type

To answer RQ1 we run the Code Clone detector on every project listed in Table 1 and obtain the results shown in Table 2.

Table 2: *Number of clones detected for each project in the original source code*

Software	#Clones
slab	11
smallvec	16
generic-array	N.A.
num-rational	14
dashmap	18
bytemuck	14
bytes	43
cgmath	328
bstr	179
hashbrown	159
crossbeam-channel	807
petgraph	N.A.
bitvec	561
ndarray	396

We then manually check these 3033 clone and categorise them. Among them, not a single clone is due to a limitation of Rust (e.g. the necessity of putting a variable `mut` instead of immutable). This is not surprising, given that declaring different lifetimes or different mutability requires additional keywords, that go beyond the Type II clones category. Note that the projects *generic-array* and *petgraph* do not report the number of clones: in fact for these projects *CCFinderSW* failed to generate a report, despite running for more than 24 hours. Moreover, these two projects are considerably smaller than others in our case study, and we can only assume the parser failed to correctly tokenize the codebase.

The highest amount of clones involves the manual implementation of "common" method such as `len` or `fmt` (debug print) that are copy pasted for each structure. This category is so ubiquitous that every project contains at least a clone of this type. In some cases, the definition varies only in the type of generics targeted, with the implementation copy-pasted several time. All these clones can be easily replaced by a macro, increasing the maintainability of the code.

Another common category, present in at least half the projects, is the variation of a method's parameters. Most methods can require a different number of parameters and most projects just copy-paste the method implementation. This problem is again, easily refactorable by having a generic method accepting all the possible parameters and specialized methods that calls the former after setting default values for the parameters.

Finally, a surprisingly high amount of clones can be found marked as *tests* or *benchmarks*. Rust allows mixing test code with implementation code by prepending the test function with a `#[test]` statement, and in our evaluation these tests mixed with implementation code have not been filtered

out. A notable example of this is the *crossbeam-channel*, where a greater majority of clones were reported amongst these tests mixed with the implementation code.

We can conclude RQ1 with the following statement:

> *No Rust-specific features can be found in the original Type II clones detected. Most clones involve manual implementation of common methods and traits or implementation of the same methods with variations in the number of parameters.*

4.2 RQ2: agreement

After determining the type of clones that can be found in the original Rust projects, we perform the same clone analysis on the HIR code. Table 3 reports the variation in SLOC for each project after running the macro expansion step and the HIR generation step. We can note how, in every project, the number of effective lines of code is decreased compared to the original project. This is due to the fact that, in order to obtain the macro expanded version or the HIR version, the compiler needs to be invoked. After invocation, all the boilerplate code is removed, the imports resolved and the eventual conditional compilations (e.g. tests) removed. This effectively reduces the amount of non-interesting benchmarks and tests clones reported in Table 2 and cited in Section 4.1.

Table 3: *Lines of code in the original source ($SLOC_{orig}$), after macro expansion ($SLOC_{exp}$), and after HIR generation ($SLOC_{HIR}$)*

Software	$SLOC_{orig}$	$SLOC_{exp}$	$SLOC_{HIR}$
slab	1311	671	838
smallvec	2204	1253	1783
generic-array	2595	2261	2970
num-rational	3064	2690	3680
dashmap	2866	1752	2189
bytemuck	3026	1206	1482
bytes	4601	3069	3837
cgmath	8878	17727	23843
bstr	6227	4453	5826
hashbrown	10104	4032	5043
crossbeam-channel	15652	4699	5480
petgraph	20308	15371	19348
bitvec	26996	15233	18772
ndarray	29281	19947	25886

After running the clone detection and collecting the results we obtain the number of clones show in Table 4, along with the results of the original code detection results.

The most impactful result we can notice is the immense increase in the number of clones for certain projects, in particular *cgmath* and *ndarray*. The gargantuan amount of clones is manually intractable, but we ensured to analyse at least a clone for each clone class reported by the detector. The reason of this immense increase in clones is the high amount of procedural code involved in these two projects. *cgmath*, a linear algebra library, requires the implementation of basic mathematical operations for different vectors of different size. This is done in the project by using macros that were not intercepted by our filtering process. This is evident also by the distribution of clones, as shown in Figure 3 and Figure 4.

Table 4: *Number of clones detected for each project in the original and HIR code*

Software	#Clones (original)	#Clones (HIR)
slab	12	53
smallvec	14	24
generic-array	N.A.	18
num-rational	16	215
dashmap	14	35
bytemuck	18	55
bytes	37	287
cgmath	328	165834
bstr	179	311
hashbrown	159	151
crossbeam-channel	807	351
petgraph	N.A.	2991
bitvec	561	3909
ndarray	396	30669

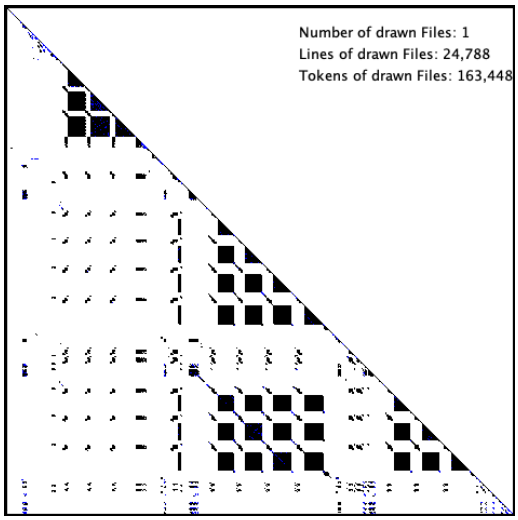

Fig. 3: *Clone scatterplot for* cgmath *HIR code*

We can see how most of the clones are clustered in big areas. Figure 3 shows clearly repeated structures in the clones.

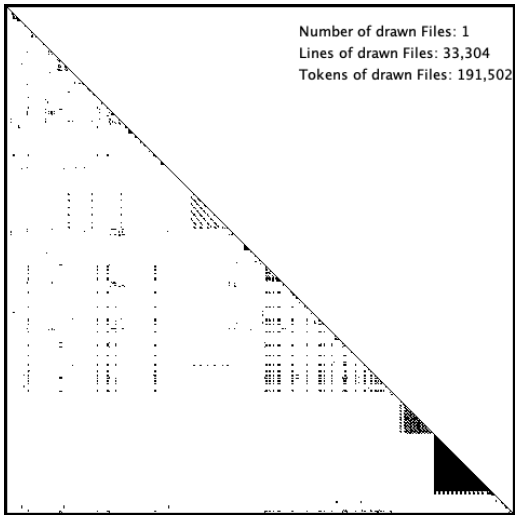

Fig. 4: *Clone scatterplot for* ndarray *HIR code*

Upon further inspection those cluster of clones are the implementations of identical mathematical operations for different dimensionality of linear algebra tools (e.g. `Vector2`, `Vector3`, `Matrix3`, `Matrix4`, ...), accepting different generic arguments. A similar reason can be given to the *ndarray* project, represented in Figure 4. Also in this case, we can note a big concentration of clones clustered together, due to the usage of procedural macros.

On the other side of the spectrum, instead, we can see how the clones for *crossbeam-channel* have been decimated, going from 807 to a meager 351. This result, however, is uninteresting, as we already discussed in Section 4.1 how these clones were mainly due to test code mixed with implementation code. After generating the HIR code, these tests are evicted by the compiler, and thus all the relative clones are not reported.

Smaller projects, on the other hand, turned out to be more interesting: on the *smallvec* project, for example, every additional clone is due to normalization into the `match` statement. However, these clones detected only in the HIR are usually present in a small amount.

We can summarize RQ2 as follow:

> *For big projects, highly dependent on procedural macro, analysing the HIR is detrimental as most of the clones are the result of macro invocation. In projects with less macro usage, however, HIR clones can highlights interesting similarities between methods due to code normalization.*

4.3 RQ3: accuracy

In the last question, we want to investigate the accuracy of the clone detection system and the usefulness of the HIR code for clone detection. After thoroughly analysing all the original source code clones and most of the HIR clones, we can assess that the Type II clones reported by *CCFinderSW* were all genuine. However, this does not mean they are useful.

```
let layout = layout_array::<A::Item>(new_cap)?;
```

(a) Original Rust code

```
let layout =
match #[lang = "branch"](layout_array::<A::Item>(new_cap))
{
#[lang = "Break"] { 0: residual } =>
#[allow(unreachable_code)]
  return #[lang = "from_residual"](residual),
#[lang = "Continue"] { 0: val } =>
#[allow(unreachable_code)]
  val,
};
```

(b) Rust code after macro expansion

Fig. 5: *Expansion of a simple statement into a more complex one triggering unrefactorable clones. This example is taken from the* smallvec *project, file* `src/lib.rs`

In fact, unlike the original clones, the HIR ones require a more careful detection threshold in order to provide useful insights. Consider for example the statement shown in Figure 5. The original code, line 905 in the file `src/lib.rs` from *smallvec*, is a one line invocation of a single function. However, when expanded, its length increases considerably, surpassing the threshold and being detected as a clone. We determined that, despite all clones being true positives, most of the HIR clones fall in this class: true positive but useless for refactoring purposes. Furthermore, only a handful of clones, usually a single digit number, can be efficiently refactored after being detected only in HIR code. In order to solve this problem, a different threshold must be used for HIR clones, given that most of the changed code in HIR increases considerably in size. This happens despite the HIR code having an overall less amount of line of code, as shown in Table 3.

We can thus conclude RQ3 as follow:

> *Despite the accuracy of the clone detection being 100%, most of the HIR clones are unusable due to the fact that the original code was already simple enough. Additional care must be taken in order to set a higher threshold for HIR clones.*

5. Threats to validity and Future Works

In this study we used 15 projects of diverse size and scope but this is only a small fraction of the ecosystem and may diverge substantially from the real distribution of clones in the Rust language. We assumed the correctness of the rustc compiler in emitting the HIR code and the correctness of the *CCFinderSW* code clone detector in detecting the clones for the Rust language, however, if one or both these fact do not hold, the entire analysis may not be valid.

Moreover, we focused and drew conclusion based only on Type II clones, where most of the Rust-specific features will likely require clones with minor difference in the used reserved words (i.e. Type III clones). For this reason we plan to conduct further studies on the ecosystem focusing on Type III clones and the transformations introduced by further lowering the code. This additional lowering targets the MIR that is more similar to a PDG style analysis [5] rather than a token based clone detector.

Finally, we did not have a comprehensive database of clones like in the *BigCloneBench* project [24], for this reason we can't now the real number of true positive and false negative clones present in our case studies. We can only count the number of false positives based on the performed manual evaluation.

6. Related Work

Several code clone detectors have been developed by the research community in the recent years. These spans different types of approaches. Some detectors converts the source code into a stream of tokens and perform analysis on these tokens. These detectors comprises *NiCad* [12],

CCFinder [7], *CP-Miner* [9], *iClones* [3], and many others [13]. More recently, additional tools have been developed to include more variety of languages, these includes *SourcererCC* [22], *CCFinderSW* [23] and *MSCCD* [26]. Several recent techniques also involves clone detector that does not operate on tokens or the AST: for example Amme et al. presented a clone detector for the Java language analysing the code dominator trees [1], while Saini et al. improved *SourcererCC* with deep learning capabilities in their *Oreo* clone detector [21].

Although a small amount of studies targeted specifically intermediate code generated by compilers [2] [10] [11], most studies targeting low level code focus on Java Bytecode. Recent works were performed by Yu et al. [25] that analysed fragments extracted from the bytecode and by Keivanloo et al. that used code fingerprint on the Java Bytecode [8].

Going even lower, some researcher even tried to find code clones directly on the binary layer. The most famous work is surely the one of Sæbjørnsen et al. [20] that analysed the similarity in assembly instructions. However, analysis at binary level is usually performed in order to retrieve information about software license violation, like in the work of Hemel et al. [4].

7. Conclusion

In this paper we investigated the code clone detection in a language different from the commonly targeted C or Java. We analysed 15 projects and all their reported Type II clones manually and determined the common causes of clones in the Rust language. We then compiled these clones, emitted their High-level Intermediate Representation (HIR), and ran again the code clone detection over this representation. Finally, we compared the original code and the Intermediate Representation (IR).

We determined that, although the code clone detector being capable of recognising genuine clones in both the original code and the HIR code, only a small amounts of code from the HIR code can actually be used. Most of them in fact are simple enough even in the original code and most of the complexity is added by the HIR generation step. For this reason the HIR code should be used as an extension of the original code, rather than completely replacing it.

8. Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 18H04094.

References

[1] Amme, W., Heinze, T. S. and Schäfer, A.: You look so different: Finding structural clones and subclones in java source code, *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 70–80 (2021).

[2] Caldeira, P. M., Sakamoto, K., Washizaki, H., Fukazawa, Y. and Shimada, T.: Improving syntactical clone detection methods through the use of an intermediate representation, *2020 IEEE 14th international workshop on software clones (IWSC)*, IEEE, pp. 8–14 (2020).

[3] Göde, N. and Koschke, R.: Incremental clone detection, *2009 13th European conference on software maintenance and reengineering*, IEEE, pp. 219–228 (2009).

[4] Hemel, A., Kalleberg, K. T., Vermaas, R. and Dolstra, E.: Finding software license violations through binary code clone detection, *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 63–72 (2011).

[5] Higo, Y., Yasushi, U., Nishino, M. and Kusumoto, S.: Incremental Code Clone Detection: A PDG-based Approach, *2011 18th Working Conference on Reverse Engineering*, pp. 3–12 (online), DOI: 10.1109/WCRE.2011.11 (2011).

[6] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones, *29th International Conference on Software Engineering (ICSE'07)*, IEEE, pp. 96–105 (2007).

[7] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).

[8] Keivanloo, I., Roy, C. K. and Rilling, J.: Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning, *2012 6th International Workshop on Software Clones (IWSC)*, IEEE, pp. 36–42 (2012).

[9] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code, *IEEE Transactions on software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).

[10] Pizzolotto, D. and Inoue, K.: Blanker: A Refactor-Oriented Cloned Source Code Normalizer, *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, IEEE, pp. 22–25 (2020).

[11] Ragkhitwetsagul, C. and Krinke, J.: Using compilation/decompilation to enhance clone detection, *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, IEEE, pp. 1–7 (2017).

[12] Roy, C. K. and Cordy, J. R.: NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, *2008 16th iEEE international conference on program comprehension*, IEEE, pp. 172–181 (2008).

[13] Roy, C. K., Cordy, J. R. and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of computer programming*, Vol. 74, No. 7, pp. 470–495 (2009).

[14] Rust Foundation: "Macro expansion (Accessed 2022-06).

[15] Rust Foundation: The Rustonomicon - Meet Safe and Unsafe (Accessed 2022-06).

[16] Rust Foundation: "The HIR (Accessed 2022-06).

[17] Rust Foundation: "The MIR (Mid-level IR)" (Accessed 2022-06).

[18] Rust Foundation: "The THIR (Accessed 2022-06).

[19] Rust Foundation: "What is ownership? (Accessed 2022-06).

[20] Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D. and Su, Z.: Detecting code clones in binary executables, *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 117–128 (2009).

[21] Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P. and Lopes, C. V.: Oreo: Detection of clones in the twilight zone, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 354–365 (2018).

[22] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K. and Lopes, C. V.: Sourcerercc: Scaling code clone detection to big-code, *Proceedings of the 38th International Conference on Software Engineering*, pp. 1157–1168 (2016).

[23] Semura, Y., Yoshida, N., Choi, E. and Inoue, K.: CCFinderSW: Clone detection tool with flexible multilingual tokenization, *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 654–659 (2017).

[24] Svajlenko, J. and Roy, C. K.: Evaluating clone detection tools with bigclonebench, *2015 IEEE international conference on software maintenance and evolution (ICSME)*, IEEE, pp. 131–140 (2015).

[25] Yu, D., Wang, J., Wu, Q., Yang, J., Wang, J., Yang, W. and Yan, W.: Detecting java code clones with multi-granularities based on bytecode, *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1, IEEE, pp. 317–326 (2017).

[26] Zhu, W., Yoshida, N., Kamiya, T., Choi, E. and Takada, H.: MSCCD: Grammar Pluggable Clone Detection Based on ANTLR Parser Generation, *arXiv preprint arXiv:2204.01028* (2022).