Oxidize: Open framework for idiomatic rule preservation
Work in Progress

Adrian K. Zborowski
Adrian.Zborowski@cwi.nl

University of Amsterdam
Centrum Wiskunde & Informatica

Abstract

This research investigates idioms in Rust programming language to improve both readability and code quality, by refactoring non-idiomatic code. This is done by enabling Rascal Metaprogramming Language to parse, analyse, and process the grammar of the Rust programming language.

1 Introduction

The research in progress, called Oxidize, looks into the refactoring of projects written in Rust[1] programming language. Rust is a general purpose, multi-paradigm, compiled programming language. Originally developed by Graydon Hoare, at Mozilla Research. Currently it is being developed by its open source community. Rust being a fairly new language, first stable version originating from 15th of May 2015, has gained interest and popularity among developers. The initial research questions for the development of Rust were "How do you do safe systems programming?", and, "How do you make concurrency painless?". Since then, the core design of Rust has changed into, and is known for thread and memory safety, its fast performance, segfaults prevention and compile-time warnings and errors.

The original concept of Oxidize is related to Corrode[2] project, which is an automatic semantics-preserving translation from C-to-Rust. Created and being developed by Jamey Sharp, it's intended for partial automation of migrating legacy code that was implemented in C. It reads a C source and outputs an equivalent Rust output. It is advised to manually correct the output afterwards for the use of Rust features and idioms where necessary.

The research is about refactoring of non-idiomatic Rust code statements, subsequently creating a Rust-to-Rust compiler. To define what idioms are and how they are related to this research, a paper by by Allamanis and Sutton, "Mining Idioms from Source Code"[3] is used. This publication states that an idiom is a syntactic fragment that recurs across projects and has a single semantic role. It can also contain metavariables within itself e.g. a body of a block statement.

For this research it is looked into the refactoring of projects written in Rust programming language, based upon output from Corrode project. Creating a Rust-to-Rust compiler. The main focus of the project lies within incorporating refactoring of idioms like Resource Acquisition Is Initialization (RAII), loop transformations, and also a static analysis of null-pointer checks.

The RAII idiom, is a class invariant which is a condition that can be relied upon within a scope or a lifetime. This enables the output code to use Rusts features, and mainly the ownership system. An example can be seen in listing 6 and 7. This transformation makes the code safer and shorter.

Rust just like most imperative languages possess over loop statements. In Rust there are three distinct cases of loop statements, being '`loop`', '`while`', and '`for`'. In some cases it is possible to interchange a statement into its more idiomatic state. This can be seen in listings 3, 4 and 5. The refactoring increases code understandability and readability.

The static analysis of the null-pointer checks, enables the program to make use of Rust safety features. It ensures that a pointer existing within a code block which is dominated by e.g. an '`if`' not '`null`' condition, can be wrapper with a '`NonZero`' '`Option`' denoting the object not being '`null`'. This informs the user and the compiler of the pointer being safe to use.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

This research contributes to the quality of Rust code readability, by refactoring the non-idiomatic code; extends the Corrode project functionality; adds Rust language grammar parsing, analysis, and processing functionality to Rascal Metaprogramming Language (MPL)[4]; and indexes the scattered Rust language grammar into more readable and functional Rascal grammar notation. Oxidize is being developed as a stand-alone framework for idiomatic code refactoring, but can also be used for educational purposes by new Rust developers.

2 Code Idiomacy

The programming idioms are syntactic fragments of code that reoccur in software projects, and are meant for a singular purpose[3]. In many cases can the following listings do exactly the same for the developer, but their purpose and meaning have a different focus. To demonstrate this, we can look at the refactoring of listing 1 into 2. Here we can see a simple refactoring of a non-idiomatic statement into an idiomatic one, decreasing code complexity and increasing readability.

```
1 while true {
2   // Statement(s)
3 }
```

Listing 1: An infinite loop in Rust programming language

```
1 loop{
2   // Statement(s)
3 }
```

Listing 2: An idiomatic infinite loop in Rust programming language

This being just a simple example of a possible interchangeability of loop statements within Rust, listings 3-5 show us a more complicated output from the Corrode project which can be transformed into its idiomatic state.

Beginning with the Corrode output code (listing 3), we can see an integer declaration on line 1 and its initialisation on line 2. This is then followed by a labeled loop called '`loop56`' with a block body containing a condition. This condition stating that if the variable '`i`' ever gets bigger than the '`argc`' the loop should be interrupted. The code also contains some arbitrary statements which do the actual work and should be preserved in the refactoring, followed by a counter '`i`' which is used in the earlier stated condition on lines 3-6.

```
1 let mut i : i32;
2 i = 0i32;
3 'loop56: loop {
4     if !(i < argc) {
5         break;
6     }
7     // Statement(s)
8     i = i + 1;
9 }
```

Listing 3: Output '`loop`' created by Corrode project from a C '`while`' statement

With the example from listing 3 we can see the following code structure being '`counter; loop{condition; statements; counter+;}`'. This is a similar structure to that of listing 4 being '`counter; while condition {statements; counter+;}`'. By detecting this similarity we can safely refactor listing 3 into listing 4.

```
1 let mut i : i32;
2 i = 0i32;
3 while i < argc {
4     // Statement(s)
5     i = i + 1;
6 }
```

Listing 4: Possible first transformation of listing 3 from a '`loop`' with a condition within, into a proper Rust '`while`' statement

In this specific case we can perform one more refactoring based on our previous refactoring having a linear growth condition ('`i+1`'). This enables us to make use of the typical linear growth of a condition within a '`for`' statement. Therefore eliminating the counter and its increasing state on lines 1,2 and 5 in listing 4. Leaving us with listing 5 being the idiomatic statement for this case.

```
1 for i in 0i32..argc {
2     // Statement(s)
3 }
```

Listing 5: A follow-up transformation based on the listing 4 transformation into its idiomatic state

Another examples of a possible refactoring case can be seen in listing 6 and 7. These two examples show us how an ownership system can be deployed. Listing 6 shows us how an array definition can be written with the use of a C binding library within Rust. The memory allocation keyword '`malloc`' and its freeing '`free`' can be refactored into Rusts ownership system, or otherwise called RAII. This can be done

by analysing the size and the type of the memory allocation, in this case its an array which can hold four integers, and transforming into the ownership system denoted by the keyword ‘`Box`‘ seen in listing 7.

```
fn create_malloc() {
 unsafe {
  // Allocate 4 long int
  let int_mem: *mut [i32;4] =
    libc :: malloc(
      mem::size_of::<[i32;4]>()
    ) as *mut [i32;4];

  // ‘int_mem‘ is freed
  libc :: free(
    int_mem as *mut libc::c_void);
 }
}
```

Listing 6: Allocating an int array of the size equal to four ints in C programming language (through Rust)

```
fn create_box() {
 // Allocate 4 long int array
 let int_mem: Box<[i32;4]>;

 // ‘int_mem‘ is automatically freed
}
```

Listing 7: Allocating an int array of the size equal to four ints in Rust programming language using the ownership property

3 Research Method

The implementation phase of the project consists of writing the currently scattered grammar of Rust into a single syntactic parser definition. An example of Rascals grammar definition can be seen in listing 8. Which show a partial representation of the Expression grammar present in Rust.

Listing 8 shows us how Rascal handles its grammar definitions with the ‘`syntax`‘ keyword, how associativity is defined with the ‘`right`‘ keyword, how optional keywords are stated with ‘`?`‘ (the questionmark), and how priority or in other words precedence is represented in the grammar by ‘>‘ (the greater-than) symbol.

Because of the architectural decision which have been taken with Rascal, we don’t have to write deterministic alternatives to grammar definitions which deviate from each other by one or more rules of difference. Example of this could be a statement which can have a body in the implementation but not in its interface representation. This would mean that one grammar definition would contain a body and the other would not. Rascal handles this situation within its syntax definition by disallowing stated labels from reaching a rule within the grammar. Example of this can also be seen in listing 8 in the rule that a ‘`box`‘ keyword with a ‘`parenExpr`‘ as label (the string present before a colon at the start of a rule) is not allowed to also have an ‘`Expression`‘. The current implementation of the grammar sums-down to around 1000 (one thousand) lines of code, in contrast to Bisons Rust grammar representation summing-up to 2000 (two thousand) lines of code, including comments and present documentation.

```
syntax Expression
 = right ( "−" Expression
         | "!" Expression
         | "∗" Expression
         | "&" "mut"? Expression
         | "&&" "mut"? Expression
         | "move"? Lambda_expression)
 | blockStmt: Block
 | blockExpr: Block_expression
 | Expression_qualified_path
 > right ( "box" "(" Expression? ")" Expression
         | "box" Expression!parenExpr)
 > parenExpr: Expression "(" (Expressions ","?)? ")"
 ;
```

Listing 8: A partial definition of Rusts Expression grammar implemented in Rascal (this grammar is still in development and could change)

4 Related Work

During a presentation of this project one of the questions was, what the difference is between *Oxidize* and a code linter. A definition of a linter by Darwin is a that of a particular program which checks/flags suspicious behaviour[5] or undesirable style in software, written in any language. The definition of *Oxidize* is that it is an open framework for idiomatic rule preservation and refactoring, which specialises in code analysis and code refactoring into its idiomatic form.

One of the relevant literatures for this project is the Allamanis *Mining Idioms from Source Code*[3] which presents an empirical proof of idiomatic code presence across project. Their project called *HAGGIS* has scanned projects across GitHub to prove this claim. Their definition of an idiom and method of using Abstract Syntax Trees (ASTs) to define a group of code as an idiom helps substantiate what our research is about. While their AST method not being fully usable because of our need to rebuild the files after refactor-

ing which needs Concrete Syntax Trees (CSTs) to be done correctly, helps better understand the matter at hand.

There exist projects which compile from a language to a language for idiomatic reasons. The implementation of *P#* is an example of this. Created by Cook and described in *Optimizing P#: Translating Prolog to More Idiomatic C#* where the implementation of *P#* already existed by the need was there to make it compile more human readable and idiomatic code[6]. This example while being similar, concentrates on the readability of the refactored code, while our implementation goes deeper into the implementation of idiomatic code with the effect of it enhancing the readability and usability. *Oxidize* also enables new developers with the knowledge of imperative languages to enhance their skills with the refactoring that can be provided with the project.

To start refactoring we need the general theory behind refactoring and its use-cases. This brings us to *Refactoring: Improving the Design of Existing Code* by Fowler et al. and the definitions of refactoring methods[7]. In this research we are not going to use the e.g. pull and push methods, but we are going to transform the used statements into their idiomatic for based on the context they are used in. This literature helps us better understand how we are deviating from the refactoring culture into the transformation culture.

For the actual refactoring and transformation work we are going to make use of the publication by Tip et al., called *Refactoring Using Type Constraints*[8]. The type constraints themselves, while being older and defined by Palsberg et al. in the 1993's, as a formalism for expressing subtype relationships between the types of declarations and expressions[9], have been revisited by Tip et al. for the refactoring method they enable.

The last literature we are going to mention in this section is that by Veerman from 2003 called *Revitalizing Modifiability of Legacy Assets*. Their use-case of legacy Cobol code while not being directly relevant to our case, stil substantiates our transformation method of language to language compilation[10].

5 Conclusion

The completion of this research project yields and extension to the Corrode projects functionality; extends the comprehensive language library of Rascal MPL; indexes the scattered grammar of Rust into an Extended BackusNaur form (EBNF) format; and establishes an extensible general-purpose Rust-to-Rust refactoring framework.
As a final contribution this research would output a thesis, which incorporates a master thesis containing the theory behind the project, elaboration on design and architectural choices, and the documentation of the project.

6 Acknowledgement

I thank Jurgen J. Vinju, and the Software Analysis and Transformation Group (SWAT) from Centrum Wiskunde & Informatica (CWI)[11] for assistance with the development of this project, and Clemens Grelck from University of Amsterdam (UvA)[12] for his feedback regarding this paper.

References

[1] "Rust `https://www.rust-lang.org`."

[2] J. Sharp, "Corrode `https://github.com/jameysharp/corrode`."

[3] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 472–483, ACM, 2014.

[4] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pp. 168–177, IEEE, 2009.

[5] I. F. Darwin, *Checking C Programs with lint.* " O'Reilly Media, Inc.", 1991.

[6] J. J. Cook, "Optimizing P#: Translating Prolog to more idiomatic C#," *Proceedings of CICLOPS 2004*, pp. 59–70, 2004.

[7] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[8] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 33, no. 3, p. 9, 2011.

[9]

[10] N. Veerman, "Revitalizing modifiability of legacy assets," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pp. 19–29, IEEE, 2003.

[11] Centrum Wiskunde & Informatica, "Centrum wiskunde & informatica `https://www.cwi.nl/`."

[12] University of Amsterdam, "University of amsterdam `https://www.cwi.nl/`."