Oxidize

Framework for Idiomatic Refactoring of Rust Programming Language Code

Adrian Zborowski
adrian.zborowski@student.uva.nl
ak.zborowski@gmail.com

October 17, 2017, 62 pages

Supervisors: Jurgen Vinju, jurgen.vinju@cwi.nl
Clemens Grelck, c.grelck@uva.nl
Host organisations: Centrum Wiskunde & Informatica, www.cwi.nl
Universiteit van Amsterdam, www.uva.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Table of Contents

Abstract 3

List of Acronyms 4

List of Figures 5

List of Listings 5

1 Introduction 7

2 The Bits of Rust 10
- 2.1 Rust Programming Language . . . 10
- 2.2 Relevant Constructs . . . 12
- 2.3 Rust Language Server . . . 14
- 2.4 Programming Idioms . . . 15

3 Foundation Background 18
- 3.1 Program Transformation . . . 18
- 3.2 Rascal Metaprogramming Language . . . 19
- 3.3 Corrode . . . 19
- 3.4 Constraints . . . 19

4 Oxidize: Foundation 22
- 4.1 Process Flow . . . 22
- 4.2 Parsing . . . 23

5 Oxidize: Structure 26
- 5.1 Overview . . . 26
- 5.2 Idiomatic Loop Transformations . . . 27
- 5.3 Ownership System Transformation . . . 31
- 5.4 NonZero Transformation . . . 38
- 5.5 Cleanup Transformations . . . 39

6 Evaluation 41
- 6.1 Introduction . . . 41
- 6.2 Method . . . 42
- 6.3 Application . . . 43
- 6.4 Code Contribution . . . 45
- 6.5 Threats to validity . . . 50

7 Related Work 52

8 Conclusion 54

9 Future Work 55

Bibliography **57**

Appendices **60**

A Usage **60**

Abstract

Rust is a systems programming language with a high-level of abstraction and a low-level control focusing on safety, speed and concurrency. Rust, as most programming and natural languages [1], contains statements with a single semantic purpose called idioms [2]. These idioms can reoccur across software projects and programming languages, helping programmers with understanding each other's code [2].

Rust is a fairly new programming language containing a steep learning curve for new and experienced programmers coming from other fields. The current Rust roadmap is planning on easing this curve [3], we want to contribute to this endeavour by researching refactoring resulting in idiomatic projects written in Rust.

For this research, we are interested in what needs to be considered in a transformation to generate idiomatic Rust code. Code transformation also brings the question of semantics preservation and validation of the generated code.

For this reason, we implemented in the Rascal *Metaprogramming Language* (MPL) a syntax definition for the Rust systems programming language, together with the Rust transformation framework, called Oxidize. This implementation enables us to create *Concrete Syntax Tree* (CST) of valid and compilable Rust code and transforms it into its idiomatic state specified by the transformation cases.

Our research focuses on three transformation cases, migration from the C style malloc memory management implementation in Rust to Rust's Ownership system implementation, idiomatic iterative statements transformations (‘`loop`‘, ‘`for`‘ and ‘`while`‘) and a NonZero construct implementation for compiler optimisation. To validate our results we have tested our solutions against the *Rust Language Server* (RLS) and have confirmed that no problems arise at the compile time.

List of Acronyms

RAII *Resource Acquisition Is Initialization*

MPL *Metaprogramming Language*

CWI *Centrum Wiskunde & Informatica*

UvA *University of Amsterdam*

EBNF *Extended Backus Naur form*

SWAT *Software Analysis and Transformation Group*

AST *Abstract Syntax Tree*

CST *Concrete Syntax Tree*

IDE *Integrated Development Environment*

CVS *Concurrent Versions System*

DSL *Domain Specific Language*

LALR *Look-Ahead Left-to-Right*

FLEX *Fast Lexical Analyzer*

JVM *Java Virtual Machine*

JRE *Java Runtime Environment*

CLI *Command Line Interface*

JDK *Java Development Kit*

RLS *Rust Language Server*

RFC *Request For Comment*

FFI *Foreign Function Interface*

TOML *Tom's Obvious, Minimal Language*

List of Figures

1.1 Overview of the transformation process . . . 8

2.1 Iteration statements available in Rust . . . 12
2.2 The C *malloc* construct as it can be created in Rust (on the left) and the Ownership system as introduced in Rust . . . 13
2.3 How and when a NonZero construct can be used . . . 14
2.4 Interchangeability of the'`loop`' and'`while`' statement . . . 15
2.5 Interchangeability of the'`while`' and'`for`' statement . . . 16
2.6 Interchangeability of the'`for`' and'`while`' statement . . . 17

3.1 Notation [32] . . . 20
3.2 Constraint variables [32] . . . 20
3.3 Type constraints [32] . . . 20
3.4 General Rust constraints applicable to our transformations . . . 21

4.1 The flowchart illustrating theOxidize project. . . . 22
4.2 An example of how a Rust function is represented in a CST tree structure . . . 23
4.3 Rust's single statement grammar definition as specified by Brian Leibig . . . 24
4.4 Rust's single statement grammar definition as specified by us in Rascal . . . 24
4.5 Listing and parsing source files of the Rust project . . . 25

5.1 The class-diagram of theOxidize framework . . . 26
5.2 Example of pre- and post-transformation code for visualisation of Figure 5.3 . . . 29
5.3 Visual representation of the'`loop`' to'`while`' transformation. Both flow-diagrams correspond to their code equivalents Figure 5.2 . . . 29
5.4 The Ownership system transformation activity flow . . . 32

6.1 Idiomatic transformation of the labeled'`loop`' construct into a'`while`' construct . . . 43
6.2 An example of the transformation performed by the ownership transformation. From the C-style malloc memory management to the ownership system . . . 44
6.3 The MBox and the NonZero transformation correction cases . . . 48

A.1 Output from Eclipse (on the left) and also *Command Line Interface* (CLI) (on the right) 62

List of Listings

2.1 Example of *Tom's Obvious, Minimal Language* (TOML) file . . . 11
2.2 The infinite'`loop`' iteration . . . 15
2.3 The finite'`while`' iteration . . . 16
2.4 The finite'`for`' iteration over ranges or collections . . . 16
2.5 The value ownership system . . . 17
5.1 Removing unused lifetime declaration from'`while`' statements (RASCAL implementation code) . . . 27
5.2 While statement as specified in the grammar (RASCAL grammar code) . . . 28
5.3 The'`used_lifetime`' function used to check if a given lifetime name is used in the given scope . . . 28
5.4 Transformation performing a'`while`' statement to'`loop`' statement refactoring . . . 30
5.5 The Ownership transformation from C memory allocation usage . . . 31
5.6 Unsafe function definition in Rust grammar . . . 32
5.7 The matching case used for the filtering step of the freeing statements . . . 33
5.8 The MBOX library specification and the use of MARRAY needed for the library to work with the code . . . 38
5.9 The NonZero transformation of values which cannot be zero (0) or None . . . 38
5.10 The NONZERO library specification and the use of NONZERO needed for the library to work with the code . . . 39
5.11 The clean up transformation for temporary variables left behind by the CORRODE project transformation . . . 39
5.12 Clean up for the empty'`if`' statements created by our Ownership system transformation. The if statements would normally contain the'`free`' statements for the MALLOC constructs freeing . . . 40
6.1 The transformation cases for the deletion of the unused labels . . . 45
6.2 Grammatical rules of the label cleaning transformations . . . 46
6.3 The case pattern for'`loop`' to'`while`' transformation . . . 46
6.4 The base Ownership transformation case . . . 47
6.5 The grammar notation used for the matching of f function with the unsafe modifier . . . 48
6.6 Temporary variables correction cases . . . 49
6.7 Empty'`if`' statements case . . . 49
6.8 The NonZero transformation . . . 50
9.1 Example of how a TOML looks like in the *Concurrent Versions System* (CVS) project . . . 55

Chapter 1

Introduction

> "With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic."
>
> Fowler and Beck [7]

Rust[4] is a systems programming language which lays its focus on safety, speed, and concurrency. The language design of Rust encompasses a high-level of abstraction and gives the developers fine-grained level of control over their performance and design. This low-level design with a high-level of abstraction makes Rust suitable for developers with a C or C++ background who are looking for a safer language alternative.

This high-level of abstraction is also suitable for developers using expressive languages like Python who are looking for a higher performance language alternative with as least as possible compromises compared to their language of choice. This mixed level of control and abstraction gives the developer a wide range of design choices, from optional type control on variables [5] to control over heap-allocation [6] life time. This brings us to the topic of idiomacy within a language.

A programming idiom is a syntactic fragment of code that reoccurs in software projects and is meant for a singular semantic purpose [2]. Idiomatic style writing plays a role in communication among the developers of a code base. By writing a statement, which has its own idiomatic meaning, we are informing not only the compiler about our context and requirements but also the other developers who are reading our code. The earlier mentioned semantic abstraction and control freedom can complicate the readability and the learning curve of a piece of code if not used in its idiomatic form.

Rust currently possesses a fairly steep learning curve which applies not only to novice programmers but also the expert programmers coming from other languages. In both cases, assuming the programmers know Rust's grammar, the programmers are able to write their code with their experience from other languages. This does not necessarily mean that their code is written in idiomatic Rust, which can increase the difficulty of understanding the code.

The current roadmap of Rust possesses over eight goals for the year 2017 and states that lowering the learning curve [3] is one of them. This is planned to be done by writing a book for a quicker startup and improving the documentation, the error messages, the language features, the *Integrated Development Environment* (IDE) support, and other tools.

To contribute to this endeavour, we have developed an idiomatic refactoring [7] tool for projects written in Rust. A code refactoring process restructures and/or transforms the existing body of code, by changing the decomposition without changing its external behaviour. Rust is a relatively young language that has only reached its stable 1.0 version on the 15th of May 2015 [8]. Its limited documentation and literature play an important role in our development and research.

For the development of our idiomatic transformation tool and the current state of the RUST's environment, we formulated the following research questions:

1. What needs to be considered in a transformation to generate idiomatic RUST code?
   1.1. What are the relevant idioms?
   1.2. What are the matching cases for non-idiomatic RUST?
2. What are the checks and actions that a transformation needs to perform to succeed?
   2.1. What are their pre-conditions and assumptions for correct application?
   2.2. How can we validate the correctness of the transformation?

Instead of using a synthetic benchmark code for verification of the refactorings, we make use of CORRODE [9] project created by Jamey Sharp [10]. It is a source-to-source translator for migrating legacy C code to RUST code. This translation focuses on the correctness of the target RUST code and does not feature many functions and constructs which RUST lets us make use of. This causes a disjunction between the functionality and the readability of the compiled code. Subsequently, this means that the output needs to be cleaned up and tweaked to use the native RUST features. The output of CORRODE provides us with code that could have been written by a developer, and at the same time code that has been produced by a code translation tool.

The CORRODE project is also the inspiration for our research. The CORRODE translation we focus on is the translation of the CVS project from the 1990's. This translation makes use of commonly used C and RUST constructs, and with that, it provides us with the possibility of widely applicable refactoring opportunities. By researching the CORRODE translated CVS code it is chosen to focus the idiomatic transformation research on three frequently occurring statement constructs. These are the ownership system, loop transformations and also a static analysis of null-pointer checks.

The outcome of the current research, OXIDIZE[1], is a framework for idiomatic refactoring. OXIDIZE makes use of CSTs generated with the RASCAL MPL [11], which has been created by *Centrum Wiskunde & Informatica* (CWI) in Amsterdam. We make use of CSTs instead of *Abstract Syntax Trees* (ASTs) because of their source code preserving goal. A CST is a representation of the grammar in a tree-like form and AST is a simplified representation of the source code [12]. This choice allows us to better understand the context of the source code and verify the validity of the target code without losing its context.

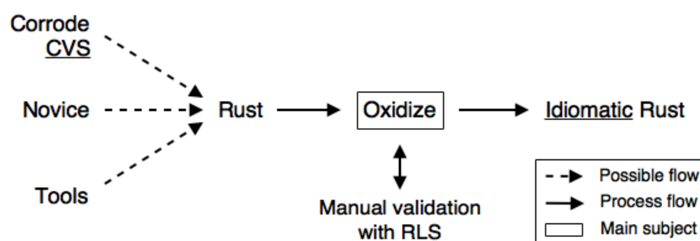


Figure 1.1: Overview of the transformation process

The creation of RUST's context-free grammar in RASCAL is also a part of our work. This enables us to perform code analysis and transformations on the CSTs without losing the context of the code. To verify our transformations we have formalised our transformation process with the type constraint notation to validate the pre- and post-transformations, and we make use of the RLS system to validate that our post-transformation code is compilable by the RUST compiler. A top-level overview of how OXIDIZE functions can be seen in Figure 1.1.

[1] https://github.com/zborowa/oxidize

The goal of our research is to create a refactoring framework for Rust code providing idiomacy related transformations. This goal is achieved through the creation of two main components which were made possible by Rascal.

The implementation of the Rust grammar in Rascal is the first and the longest component to create, and enables us to parse the source code into CSTs. Our grammar implementation accepts a superset of Rust's grammar to simplify the implementation and the resulting CSTs, making the grammar easier to understand and to use for processing of the trees. One of the prerequisites of running Oxidize is to have a valid and compilable Rust program which is already validated and compiled by the Rust compiler.

The second component is the implementation of the CST transformation in Rascal. The code transformations are the core of our research and the core complexity of Oxidize. They provide us the answers to our research questions and contain the logic for idiomatic code transformations. The transformations make use of the CSTs created by our grammar implementation and visit the trees for specified patterns to then analyse and transform them into idiomatic Rust code.

Subsequent chapters of this thesis further discuss the topic of Oxidize. The following Chapter 2 introduces the background to the Rust programming language and the consecutive Chapter 3 introduces the background of our research together with the context of the proejct. In Chapter 4 we present the process-flow, grammar and usage. The continuing Chapter 5 discusses the structure and the possible transformations of Oxidize.

Chapter 6 presents the final results of the research, together with the reflection on the research questions. In Chapter 7 we present related work to Oxidize and in Chapter 8 we conclude our research. The final Chapter 9 presents the future work which could benefit Oxidize with various functionality.

Chapter 2

The Bits of Rust

The following chapter presents the RUST programming language together with the important constructs in the language for our research. It also introduces the *Rust Language Server* (RLS) used for post-transformation code validation, the purpose of programming idioms in RUST, and, explanation of the type constraints and how we make use of them.

2.1 Rust Programming Language

The RUST programming language is a general-purpose, multi-paradigm, compiled programming language originally developed at Mozilla Research by Graydon Hoare. The language has reached its 1.0 version on the 15th of May 2015 and is currently stable in its 1.18 version since 8th of June 2017. The initial purpose of RUST was to solve two problematic questions [13]:

- How do you do safe systems programming?
- How do you make concurrency painless?

These problems concluded to be related to memory safety bugs and concurrency bugs that have to do with code accessing data when it should not. RUST's solution to this problem is its Ownership system. This is a discipline for access control that system programmers try to follow, but RUST's compiler checks it statically for them. The Ownership system enables the language to work without a garbage collector and without fear of segmentation faults.

The topic of safety talked by RUST and in our thesis mainly revolves about the following *unsafe* operations [14]:

- Dereferencing null or dangling pointers
- Reading uninitialized memory
- Breaking the pointer aliasing rules
- Producing invalid primitive values
- Unwinding into another language
- Causing a data race

Ownership system

The ownership is one of the key concepts of Rust. Each value possesses over an owner variable of the corresponding data, and there can only be one owner at a time. In case of a value moving to a new variable, the ownership is transferred to the new variable and the old variable is invalidated. We can also borrow values using references but Rust enables only one reference to be mutable at a time. This principle can also be seen in *Resource Acquisition Is Initialization* (RAII) which is closely associated with C++ [15].

The purpose of the Ownership system is managing the memory through a system of ownership combined with a set of rules that the compiler will check at compile time. The effect of this is that there is no run-time cost for any of the ownership features. The rules checking during the compilation time also enforces that a value can only have one owner. This particularly helps with memory safety (no dereference) and concurrency (no data races) [13].

Now that we understand the basic idea of the Ownership system, we can look deeper into its specifications. In Rust, we have names for concepts which are implicit in other languages. To better understand the Ownership system we need to understand a few of those normally implicit concepts. First is the *owner*, which introduces a new scope. This *owner* is in charge of the scope which from now on we will call the *lifetime*. The owner is now responsible for the safety and lifetime of a given value until it goes out of scope and is destroyed. The lifetime, which is introduced by the owner, is the range from where the owner is created until the end of the scope in which the owner resides. During this lifetime we can *borrow* the value from the owner and transfer the ownership of the value to a new owner. In this state can the original owner not be used and only the new owner can be addressed and modified (this case is only applicable if both or only one of the owners is mutable).

As stated before this is not a new concept and it is fairly similar to the RAII system in C++ [16]. The main difference between the Ownership system and the RAII system is that the Ownership system can be safer in some situations where the RAII system cannot be. In case of Rust and the Ownership system where the lifetime comes into consideration during the compilation time and in case of C++ and the RAII system where allocation happens during the run-time, we can expect of Rust to detect a dereferencing or moving of a pointer during its compilation while with C++ we could create a bug which could halt the program during run-time.

Cargo

The Rust environment possesses over a package management implementation called Cargo [17]. This package manager was created to formalise the canonical Rust workflow. It automates the standard tasks which can be associated with the distribution of software. This can be seen as standardising the structure of a new project, managing project dependencies and managing unit tests.

```
1 [package]
2 name = "cvsrs"
3 version = "0.0.1"
4 [lib]
5 name = "cvsrs"
6 path = "lib.rs"
```

Listing 2.1: Example of *Tom's Obvious, Minimal Language* (TOML) file

Cargo makes use of the *Tom's Obvious, Minimal Language* (TOML) file [18]. TOML objective is to be a minimal configuration file format that is easy to read and understand. It is designed to be mapped unambiguously to a hash table. TOMLs inspiration comes from the .INI file syntax, but aims for a more formal specification. This format uses a key/value pair (‘`key = value`‘) and a table (‘`[key]`‘) to match them into hash tables.

2.2 Relevant Constructs

To write a program in, or about, a language we need to know the relevant constructs for our research. In the case of Oxidize, we need to know about iteration statements, the Ownership system and the use of NonZero. In Rust there are three iterative statements, namely *loop*, *while* and *for*. We don't have an iteration statement like *do* in Rust.

Iteration statements

```
1 loop {
2   ...
3 }
```

```
1 while ... {
2   ...
3 }
```

```
1 for ... in ... {
2   ...
3 }
```

Figure 2.1: Iteration statements available in Rust

In Figure 2.1 we can see the three examples of iterative statements available in Rust. The first statement (from the left) is probably a new statement for many developers. This is a *loop* statement which does not contain any expression or iterator as its condition, like most iterative statements in other languages do. This statement will continue on iterating until it is broken with a '`break`' keyword. The second iterative statement is the *while* loop. This statement contains a condition which is an expression evaluating to any of the boolean values. The last iterative statement available in Rust is the *for* statement. This statement in contrast to the other two is optimized for a specific amount of iterations. This can be specified in its expression (right from the '`in`' keyword) in the form of a collection of objects and used from its variable (left from the '`in`' keyword) [19].

All three of the iteration statements can be assigned with a lifetime label. This lifetime label can then be used internally by the body of the same statement to stop the iterations. This is done by adding an identifier and a colon before the keyword of the statement. An example of this can be the following '`my_loop:loop { break my_loop;}`'. This example assigns '`my_loop`' as the identifier of the statement and then immediately targets it with the '`break`' keyword to stop the iteration.

Ownership system

```
1 fn create_malloc() {
2     unsafe {
3         // Allocate memory equal to the
4         // size of a pointer
5         let int_mem: *mut u8;
6
7         // Initialize the pointer with
8         // an integer value of 0
9         int_mem = libc::malloc(
10            mem::size_of::<i32>()
11        ) as (*mut u8);
12
13        // Show/Use the value
14        println!("{:?}", *int_mem);
15
16        // 'int_mem' is freed
17        libc::free(
18          int_mem as (*mut libc::c_void));
19    }
20 }
```

```
1 fn create_box() {
2
3     // Allocate memory equal to a
4     // signed 32bit integer
5     let int_mem: Box<i32>;
6
7     // Initialize a Box value
8     // with the integer 0
9     int_mem = Box::new(0);
10
11
12
13    // Show/Use the value
14    println!("{:?}", int_mem);
15
16    // 'int_mem' is freed automatically
17
18
19
20 }
```

Figure 2.2: The C *malloc* construct as it can be created in RUST (on the left) and the Ownership system as introduced in RUST

The following concept which needs explanation is the Ownership system. In this example, we are showing the difference between how a *malloc* is created in RUST by making use of the C library integrated into RUST. This has to do with that RUST introduces its own memory management called the Ownership system. This Ownership system enables RUST to make memory management checks and prevent segmentation faults at the time of the compilation by making use of an affine type system. An affine type system makes use of the affine logic where it is stated that a resource may only be used once [20]. This in contrast to C's *malloc* having to be manually checked and kept in mind during the runtime. The Ownership system also enables the RUST code to be cleaner and shorter in comparison to its C counterpart.

NonZero

```
1  if !p.is_null(){
2      let p = NonZero::new(p);
3      ...
4  }
5
6  ...
7
8  let x = *p;
9  let p = NonZero::new(p);
10
11 ...
12
13 let q = NonZero::new(&x as (*const _));
```

Figure 2.3: How and when a NonZero construct can be used

As the last construct, we have the *NonZero* for defining pointers which are explicitly checked for being not *null* and also not *zero* (the number 0). The NonZero has to do with the fact that we can further optimise our compiler usage and help it determine when a construct is safe or not to use. This construct can be used in combination with the *Option* construct. Option construct is commonly used in Rust for initial or return values and optional arguments. Every Option is either Some and contains a value, or None, and does not. In Rust if a construct is an enumeration of values (like Option with Some and None) the size of the enumeration is determined by the largest value in size. By using NonZero in combination with Option we can optimize the compiler to use the actual value of Option as its size. This removes the not needed overhead.

By firstly allocating the value in a NonZero wrapper and then passing it through to the *Option* construct, we only allocate the memory that is needed and no longer require the overhead. This is caused by the NonZero construct not allowing a *null* and *zero* value.

The first example (from the top) in Figure 2.3 shows us that the NonZero can be used after the pointer has been checked for not having a value, the second example showing that after dereferencing a pointer and after compiler determining that this case is valid and does not cause a segmentation fault we can wrap the pointer in a NonZero wrapper, and also when a pointer is by default not *null* we can also wrap it with the NonZero wrapper. The purpose of the NonZero wrapper is the memory size allocation of the resource for the compiled program.

2.3 Rust Language Server

As stated before, Rust performs its safety checks and memory management decisions during its compile time. In this way, the compiled program's runtime is not affected by those checks. This benefits Rust in that it can outperform other languages in some use cases like embedding Rust in other languages or creating device drivers for hardware [21].

This also benefits our research in a way that we can receive quick and precise feedback without manually having to compile and then check the output for problems. This can be done with the help of the RLS [22]. The RLS is a background server for providing quick Rust compiler information to the IDE in use. The information provided by RLS comes from the Rust compiler. In some cases where the required information can't come from the compiler (e.g. auto completion or compiling being slow) the information is provided by another project called Racer which is a Rust code completion utility [23].

During the development of Oxidize we have been making use of the Visual Studio Code and its Rust extension which supports the RLS. This gave us the ability to create code transformations to an existing Rust project which was open in the Visual Studio Code and scanned the transformed Rust code for safety checks and memory management decisions.

2.4 Programming Idioms

Creating programs and their code is a means of benefitting or providing functionality to a goal. This code is also a means of a finely detailed communication between current and future programmers. Hindle et al. states that source code can be perceived as natural, just like English is a natural language. Therefore, it is created by humans with accompanying constraints, limitations, and is likely to be repetitive and predictable [1]. By writing idiomatic code we can communicate with other developers in a way that they would consider natural.

According to Allamanis and Sutton an idiom is a "syntactic fragment that recurs across projects and has a single semantic role" [2]. This reoccurrence and single semantic role ease the learning curve and understanding of code for developers. An idiom does not have to be fully specified and can consist of meta variables which are an abstraction of identifiers and code blocks.

Particular examples of such idioms can be found in the first edition of the Rust language book, in the chapter about loops [24, ch3.6]. The examples are about the constructs discussed in Section 2.2 and can be seen in Figure 2.1. Here we are introduced to three similar, but yet different, examples of statements representing iteration activities. All three statements have their idiomatic use-case but still can be used interchangeably.

Loop

```
1 loop {
2     println!("Loop: keep refreshing state/UI/information/data/...");
3 }
```

Listing 2.2: The infinite `loop` iteration

The statement in Listing 2.2 is the `loop` iteration meant for infinite iteration of operations [19]. Syntactically it is the simplest loop statement in Rust. The syntax of this statement consists of only a keyword and statements in the body of the block, as follow: `loop { <Statements> }`. The semantics concerning the `loop` statement are that of an indefinite iteration with the goal of e.g. monitoring/refreshing of state/UI/information/data. It is possible to stop the iteration with the use of the `break` keyword.

```
1 loop {
2     println!("Hello, world!");
3 }
```

```
1 while true {
2     println!("Hello, world!");
3 }
```

Figure 2.4: Interchangeability of the `loop` and `while` statement

As stated before the loop statements can be interchanged with each other and in case of `loop` we could interchange it with the `while` statement, as shown in Figure 2.4. The pitfall of using a `while` statement as a `loop` statement is the compiler optimization with the `loop` statement for the infinite iteration. In this case it is clear that both statements are meant to loop infinitely but if the `true` value would be a variable which is initialised somewhere in the body of a method we would have to also search for it to know that this is the case.

While

```
1 while i < 10 {
2     println!("While: do something until ...");
3     i += 1;
4 }
```

Listing 2.3: The finite ‘`while`‘ iteration

The statement in Listing 2.3 is the ‘`while`‘ statement meant for a finite amount of iterations [19]. The syntax of this statement consists of a keyword, a condition in the form of an expression and a body of statements, as follow: ‘`while <Expression> { <Statements> }`‘. The semantics concerning the ‘`while`‘ statement are that of a finite iteration performing operations until a specific condition is fulfilled.

```
1 let mut i = 0;
2 while i != 3 {
3     println!("'i' is not yet 3!");
4     i += 1;
5 }
```

```
1 let mut i = 0;
2 for x in 1..4 {
3     println!("'i' is not yet 3!");
4     i = x;
5 }
```

Figure 2.5: Interchangeability of the ‘`while`‘ and ‘`for`‘ statement

The interchangeability example shown in Figure 2.5 visualises how interchanging one statement with another does now necessarily make it harder to understand. This is a more subtle difference of statement goal difference. In case of the ‘`loop`‘ statement the goal is present in its condition by denoting that the variable ‘`i`‘ needs to be of value ‘`3`‘ to stop the iteration. This is not the same case in the ‘`for`‘ loop. In the ‘`for`‘ loop we also know that the iteration will stop at value of ‘`3`‘ (because ‘`for`‘ makes use of exclusive ranges) but we don’t know why it needs to stop there. To know its goal we need to also read the body with the explicite message that the goal has to do with the value of ‘`3`‘. The pitfall of using the ‘`for`‘ statement in this way is that we need to know that ranges are exclusive and so could be error prone.

For

```
1 for x in 0..10 {
2     println!("For: do something as long as ...");
3 }
```

Listing 2.4: The finite ‘`for`‘ iteration over ranges or collections

The statement in Listing 2.4 is the ‘`for`‘ statement meant for finite amount of iterations over a range or collection of objects [19]. The syntax of this statement consists of a keyword, variable, expression and a body of statements, as follow: ‘`for <Var> in <Expression> { <Statements> }`‘.

```
for x in 0..10 {
    println!("The value is {}", x);
}
```

```
let mut i = 0;
while i < 10 {
    println!("The value is {}", i);
    i += 1;
}
```

Figure 2.6: Interchangeability of the `for` and `while` statement

The example present in Figure 2.6 is very similar to that of the example present in Figure 2.5. The difference being that in the previous example the variable `i` existed in the code for a reason not specified in our example (outside of the example). In this example the variable `i` is the outcome of the `while` statement needing it to complete its iteration. Both statements produce the exact same outcome but now we can see how the interchangeability can reduce the readability and introduce unnecessary variables. The pitfall of this example is the the reduced readability and being more error prone than its idiomatic version.

Ownership

```
let s1 = String::from("world"); // s1 is the owner of String value
let s2 = s1;                     // s2 is now the owner of the s1 value

println!("Hello, {}!", s1);     // This will error with "use of moved value"
println!("Hello, {}!", s2);     // This will print "Hello, world!"
```

Listing 2.5: The value ownership system

The last example of idiomatic use is that of the Ownership system. The basic principles of the Ownership system can be found in 2.2. The Ownership system is the idiomatic memory management way of handling the Stack and Heap allocation in Rust [25]. In Listing 2.5 we can see an example of the Ownership system usage. In this example we make use of the `String` object which should not be confused with its immutable counterpart of a string literal. The `String` object is mutable and does not posses over a `Copy` trait. This means that if one variable is assigned to a different variable it is moved and not copied. As the rules of the Ownership system state a value can only have one owner. In Listing 2.5 we can see the case of the `String` value switching owner from `s1` to `s2`.

This idiomatic memory management system makes the code safer and better readable for developers. In comparison to, for example, `C++` we don't have to free our memory and don't have to worry about dereferencing pointers. A different example of the Ownership system benefits could be working with network related code. In case of Rust we would not have to worry about closing sockets and leaving them open for potential data leaks [26].

By using the idiomatic form of a statement we can easier determine the context in which the statement resides, and should be used in. With the given four examples, we can see that idiomacy can be perceived as the context in which a statement should be used in. An idiomatic statement does not only fulfil its goal for the scope where it is written in but it also tells the most concise story of what it is meant to do to the reader.

Chapter 3

Foundation Background

The background to this research lies not only in the theory behind Rust but also in the theory of program transformations. This theory was applied by using two currently ongoing projects, namely Rascal MPL and Corrode project both are described in this chapter.

3.1 Program Transformation

The act of program transformation yields changing one programs source code into that of a different source code. Program transformation can be separated into two categories of translation and rephrasing. The translation category is meant for transformation from language A to language B like what the Corrode project does with C to Rust translation. The rephrasing category is a transformation happening from and to the same language. This is where Oxidize belongs because of its Rust to Rust transformation. To narrow the category even further we can say that Oxidize belongs in the rephrasing sub-category of program refactoring. Program refactoring aims at improving the design of the source code by restructuring for readability and preserving for functionality.

Program refactoring is a sub-category of Program transformation and Fowler and Beck state the following about refactoring [7]:

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimises the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.* – ([7, p. 9])

This is further on supported on Fowler's website by a clarification of the verb "refactoring":

> *Refactoring is not another word for cleaning up code - it specifically defines one technique for improving the health of a code-base. I use restructuring as a more general term for reorganising code that may incorporate other techniques.* – ([27])

A refactoring change is made to the internal structure of a program with a goal such as making the program easier to understand and to read. This kind of refactoring does not change the behaviour thus only changes programs structure. It would depend on the goal of the tool what kind of output it would create, but no matter the goal, the refactoring tools should still convey to the same process structure. By setting pre-conditions on what (sub)structure the refactoring could happen, transforming the structure into the desired structure, and as a final step, testing if the new structure satisfies a predetermined post-condition.

3.2 Rascal Metaprogramming Language

The RASCAL *Metaprogramming Language* (MPL) [11] is a *Domain Specific Language* (DSL) providing a high-level integration of source code analysis and manipulation [11]. Created by Paul Klint, Tijs van der Storm and Jurgen Vinju, together with the CWI and *University of Amsterdam* (UvA). RASCAL takes inspiration from many previous metaprogramming languages e.g. ASF+SDF [28], ANTLR [29] and CodeSurfer [30]. Grammar implementations within RASCAL are similar to that of the *Extended Backus Naur form* (EBNF) with Regular expression syntax, and defines grammar as non-deterministic and context-free.

An important part of *Oxidize* is its use of fully typed parse trees which are one of the RASCAL specialities [31, p.139]. The last category is the usability which focuses on the learnability, readability, debuggability, traceability, deployability and extensibility [11]. Applying all of those principles RASCAL takes the path of *least surprise* where no information is hidden and everything can be seen and accessed by the programmer.

3.3 Corrode

The CORRODE [9] project is an automatic semantics-preserving translator from C to RUST. It is a compiler written in the functional programming language HASKELL and is created by Jamey Sharp. This project was the starting point and the inspiration for Oxidize because of its aim to give deprecated and current C projects a second chance in a new environment. CORRODE is intended for the automation of migrating legacy C source code to RUST code. It is not a full automation of the translation process, and the output is as safe as the input code was. It is advised to clean up the output after the translation for the usage of idiomatic RUST and its available features.

The main focus of CORRODE is to preserve the original properties of the input source code into its target source code. This translation is meant to replace the originally used C implementation by the translated RUST code without any compromise of an intermediate step in a compiler toolchain. CORRODE aims to translate C code into RUST code with exactly the same behaviour.

3.4 Constraints

Constraint notation is a formalised way of denoting a type correct program. The notation provides us with rules which need to be satisfied for a program to be type correct by expressing subtype relationships between the types of program elements.

The actual constraints are generated from the abstract tree of a program. The type constraint notation makes use of the separation notation with a condition and a constraints ($\frac{condition}{constraints}$). An example of such notation could be a return statement of a method: ($\frac{\text{return } E \text{ in method } M}{[E] \leq [M]}$) [32]. This type constraint applies to an Expression which is returned in a method and a sub-type of the return type of the method.

$$M, M', ... \quad \text{methods} \tag{3.1}$$
$$F, F', ... \quad \text{fields} \tag{3.2}$$
$$T, T', ... \quad \text{types} \tag{3.3}$$
$$E, E', ... \quad \text{expression} \tag{3.4}$$
$$NumParams(M) \quad \text{the number of formal parameters of method M} \tag{3.5}$$
$$Param(M, i) \quad \text{the } i_{th} \text{ formal parameter of method M} \tag{3.6}$$

Figure 3.1: Notation [32]

$$\alpha ::= T \quad \text{a type constant} \tag{3.7}$$
$$[E] \quad \text{the type of E} \tag{3.8}$$
$$[M] \quad \text{the declared return type of M} \tag{3.9}$$
$$[F] \quad \text{the declared type of F} \tag{3.10}$$
$$Decl(M) \quad \text{the type in which M is declared} \tag{3.11}$$
$$Decl(F) \quad \text{the type in which F is declared} \tag{3.12}$$
$$CFG(E) \quad \text{Control Flow Graph of E} \tag{3.13}$$

Figure 3.2: Constraint variables [32]

$$\alpha = \alpha' \quad \text{type } \alpha \text{ is the same as type } \alpha' \tag{3.14}$$
$$\alpha < \alpha' \quad \text{type } \alpha \text{ is a proper subtype of type } \alpha' \tag{3.15}$$
$$\alpha \leq \alpha' \quad \text{type } \alpha \text{ is the same as, or a subtype of, type } \alpha' \tag{3.16}$$

Figure 3.3: Type constraints [32]

The type constraint inference rules are used to extract the exact conditions under which a program is (type) correct. These rules express specified relations between constraint variables (Figure 3.2). The constraint variables are the possible transformation candidates and may change during the refactoring. If the pre- and post-transformation programs satisfy the specified constraints, the transformation can be called *constraint-preserving* refactoring. By using the constraint notation we can also call transformations, *semantics-preserving* refactoring, if and only if they are also *constraint-preserving* refactoring and the transformation does not change anything else outside of the specified constraint.

We make use of type constraint notation to formally denote the pre- and post-conditions of a transformation in a concise and easy to read manner. In Figure 3.1 we can see the alphabetical letters which are used to denote the types of objects present in the formulas. In Figure 3.2 we can see the symbols which can be used in the combination with the letters present in Figure 3.1. In Figure 3.3 we can see the actual type constraint notation which can be used with both the Figure 3.1 and Figure 3.2.

$$\frac{assignment\ E_1 \ = \ E_2}{\begin{array}{c} E_1 \ owns\ E_2 \\ [E_2] \ \leq \ [E_1] \end{array}} \quad (3.17) \qquad \frac{\text{call } M_1}{\begin{array}{c} NumParams(M_1) \ = \ Decl(NumParams(M_1)) \\ [Param(M_1, i_1)] \ \leq \ Decl(Param(M_1, i_1)) \\ [M_1] \ \leq \ Decl(M_1) \end{array}} \quad (3.18)$$

$$\frac{Comparison(E_1, E_2)}{[E_2] \ \leq \ [E_1]} \quad (3.19) \qquad \frac{\begin{array}{c} E_1 \ owns\ E_2 \\ E_2 \ owns\ E_3 \end{array}}{E_1 \ owns\ E_3} \quad (3.20)$$

Figure 3.4: General Rust constraints applicable to our transformations

For our research we have compiled the following general constraints of Rust language shown in Figure 3.4. Each rule defines the type correct state of an element in the language. Rule 3.17 states that, for an assignment of an expression to an expression, we have the constraint of the left-hand side becoming the owner of the right-hand side of the assignment, and right-hand side is required to be a subtype of the left-hand side of the assignment. Without those constraints the assignment would not be type-correct.

Rule 3.18 shows us that a method/function call requires us to keep in mind three constraints. The first constraints has to do with the fact that Rust does not have the option of default values and that our function call has to include all the specified parameters in the declaration of the function, the second constraint specifies that any given parameter in the function call has to correspond to the subtype of the parameter definition, and the last constraint specifies that the return type of the function is a subtype of the function declaration.

Rule 3.19 specifies the constraints concerning the expressions comparison. This rules has only one constraint in the form right-hand side of an expression is required to be a subtype of the left-hand side of the comparison. Rule 3.20 specifies that an ownership of a resource if transitive in regard of what its corresponding resource owns. This means that if a resource is freed, because it went out of scope, we can be assured of that what the resource owned is also freed.

Not all of our transformations use the general Rust language constraints as their own constraints. This is because of our incremental transformation approach. Each of our transformations make changes to a specific part of the source code and does not complete the whole process at once. The rules specified in Figure 3.4 correspond to the begin state and the end state of each top level transformation specified in Oxidize(Section 5.2, 5.3, 5.4).

Chapter 4

Oxidize: Foundation

In this chapter, we will discuss all the steps needed for Oxidize to complete the analysis of the source code together with the transformation process. This also includes the written grammar for Rust in Rascal.

4.1 Process Flow

In the figure below we can see a visualisation of the process flow of Oxidize.

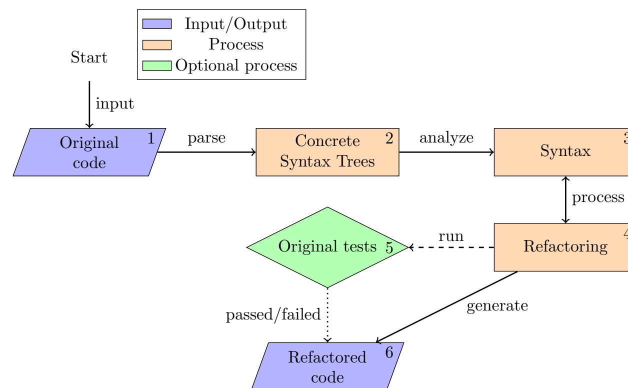


Figure 4.1: The flowchart illustrating the Oxidize project.

The steps to successfully complete the process of idiomatization by Oxidize are specified in the flowchart above and are elaborated on below (numbers are associated with the numbers in the flowchart):

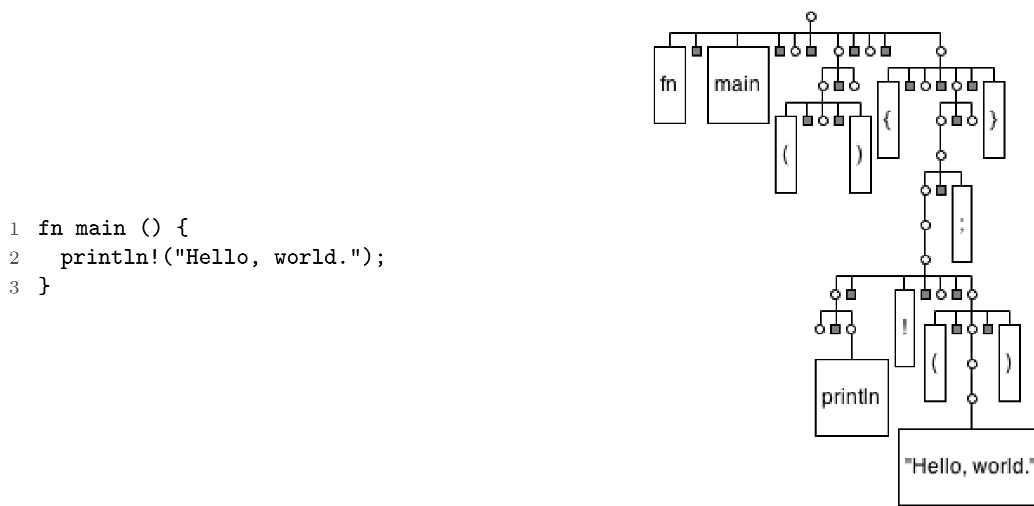
1. User specifies the location of the to transform source code in Oxidize and the project recursively scans through the source files (.rs)
2. The source code is parsed into CSTs for further analysis
3. The CSTs are traversed by Rascal for specified syntax cases
4. The CSTs are refactored by Rascal with the specified transformation patterns
5. (Optional) The (if present) original source code test cases are run the user will be informed of the output of the test cases
6. After completing all the parsing and transformations steps Oxidize will create a new neighbouring folder next to the folder of the original source code with the target code

Using this straightforward process and lifting as much complexity as possible from the user with the possibility of tweaking and modifying, we have achieved the final stage of OXIDIZE. With this process, we have managed to only require user involvement in the first step.

The following sections explain each step of the OXIDIZE process with the corresponding choices, claims and limitations.

4.2 Parsing

The first step in building the analysis and transformation of OXIDIZE was to enable RASCAL to read and parse source code into CSTs. By defining the grammar for RUST in RASCAL we can make use of RASCAL's special traits. For this section, the trait of most importance is the creation of the parse trees.




```
1 fn main () {
2   println!("Hello, world.");
3 }
```

Figure 4.2: An example of how a RUST function is represented in a CST tree structure

In Figure 4.2 we can see an example of a CST created by RASCAL. This example shows us how a visual representation of a parse tree looks like and how it preserves the scope level of indentation. By looking at the tree we can see that the string '`"Hello, world."`' resides within the parentheses of the macro function called '`println`'. This again resides within the body of a function called '`main`'. The indentation and the level at which a node within a parse tree is located can help us with debugging our grammar implementation by showing if the implemented association has been done correctly. A correctly indented tree is also needed for correct traversal of the structure.

Before we can make use of the CSTs we have to first have a grammar implementation of the RUST language. This implementation of the RUST grammar is a requirement for OXIDIZE to be able to parse, analyze and transform code. Our development language and environment, RASCAL, do not have such implementation of the RUST language. This makes the creation of the RUST grammar implementation a part of our research.

Creation of a grammar implementation of a language could be addressed with an official grammar specification created by the development team of a language. Such specification would normally include all information about the language syntax, from data typing to constructs. In case of the RUST language, there is unfortunately no complete official specification and the written grammar only exists in the bootstrapped implementation of the language.

For this reason, we have used the official and unofficial versions of the Rust language documentation to better understand the language. During this research for resources, we have found attempts of Rust language specification in the available documentation and unofficial projects. All of those resources were limited in some way, from missing to under-specified grammar. In those cases, we have contacted the Rust community for clarification of what was missing.

We have decided to use a custom implementation of the Rust grammar in a *Look-Ahead Left-to-Right* (LALR) parser by Brian Leibig [33] as the starting point of the development. This implementation is also not fully up to date, but is still maintained by the Rust community. This grammar resides within the official Rust repository [34]. This implementation while not one-on-one compatible with Rascal's syntax definition was possible to be rewritten into a compatible notation. The original LALR parser implementation was developed to be used by the GNU Bison parser generator together with the use of the *Fast Lexical Analyzer* (FLEX) lexer generator. Both implementations had to be used in order to translate the full specification of Brian Leibig.

```
stmt    ⇒  let
        |  stmt_item
        |  PUB stmt_item
        |  outer_attrs stmt_item
        |  outer_attrs PUB stmt_item
        |  full_block_expr
        |  block
        |  nonblock_expr ;
        |  ;
```

Figure 4.3: Rust's single statement grammar definition as specified by Brian Leibig

The grammar definition present in Figure 4.3 is an example of a Rust Bison grammar defined by Brian Leibig. This specific example represents Rust's statement grammar definition. From top to bottom of this example we can see that a Rust statement (‘`let`‘) can be a variable definition or initialization; (all the ‘`stmt_item`‘) a static, constant, alias, block item or a crate/library usage definition; (‘`full_block_expr`‘) a block expression, e.g. an ‘`if`‘ statement; (‘`block`‘) a code block defined with curly brackets (‘{}‘); (‘`nonblock_expr`‘) an expression which does not contain code blocks; (‘`;`‘) or a single semicolon. This example while working fine for Bison and also fine for Rascal (with a few minor tweaks) it can be modified to incorporate Rascal features. Those features can make the grammar more readable and also shorter, just like in the following example.

```
stmt    ⇒  let
        |  [outer_attrs]* [PUB] stmt_item
        |  full_block_expr
        |  block
        |  [nonblock_expr] ;
```

Figure 4.4: Rust's single statement grammar definition as specified by us in Rascal

The main difference between the original FLEX (Figure 4.3) and the new Rascal (Figure 4.4) grammar is the ability of combining grammar rule alternatives. This can be seen from the combined rule alternatives from Figure 4.3 on lines 2-5 into the single line 2 in Figure 4.4.

This kind of code changes has been applied to the originally used syntactical grammar. The total lines of code have been reduced from 1,945 total (including empty lines and comments) [35] to 968 total (including empty lines and comments). The biggest change to the grammar of Leibig was the combination of the four variations of the expression grammar. This grammatical rule had four different variations because of Rust's specific rules of expressions types. The variations were (1)

expressions without a block (‘{}‘) on the left-hand side of the expression, (2) general expressions with blocks and parenthesis (‘()‘), (3) expressions without parenthesis on both sides of the expression, (4) expressions without the ‘struct‘ construct.

The modifications to the originally used grammar by Leibig have made the grammar parsing broader than the original intention of the acceptable set. The acceptable set being the grammar set which is accepted by the Rust compiler. This modification while not being true to the language set, is a byproduct of creating a language grammar definition in the absence of the official specification. It also leads to a simpler grammar which is easier to understand and maintain. The lack of specification also causes missing of a few grammatical rules in our implementation, and as well in the grammar by Leibig. Our grammar implementation while not being complete is able to parse 85% of the Rust language implementation code. This percentage is based on the total amount of Rust source files present in the Rust language implementation (8,490) and the total amount of the source files being parsable by Oxidize (7,216).

As mentioned, the implementation of Oxidize did not achieve the full 100% of code coverage. This result comes from the absence of official specification and new features added in new versions of the language. This final result means that Oxidize is not able to parse all of the existing Rust code and thus can only be applied to code base which does not use the unsupported constructs. The addition of the missing/unsupported constructs is not hard, but is currently considered to be outside of the current thesis scope. This matter requires reverse engineering of the language features from the Rust compiler and interviews with the developers of the language. One of those constructs is the use of the ‘default‘ keyword before the function (‘fn‘) declaration. This construct is not supported because of its ambiguous use cases.

```
1  rascal>import util::Walk;
2  ok
3  rascal>import uril::Parse;
4  ok
5  rascal>source_locs = Walk(project_loc, extension);
6  list[loc]: [...]
7  rascal>Parse(source_locs, verbose=true);
8  Total files:  8499
9  Parsed:       7216
10 Failed:       1283
11 Amb:          768
12 list[Tree]: [...]
```

Figure 4.5: Listing and parsing source files of the Rust project

The figure above (Figure 4.5) shows the total amount of files present in Rust language implementation (8,499), a number of files which can be parsed with our implementation of the grammar (7,216), a number of files which cannot be (yet) parsed with our implementation (1,283), and a number of files containing ambiguity in their CST with our implementation(768).

Chapter 5

Oxidize: Structure

In this chapter, we are discussing our structural implementation of Oxidize. This includes a general overview of all the modules and their specification. Modules include the grammar, transformation and traversal implementation. The concrete explanation of the transformations paired with corresponding examples can be found in Chapter 6 - Evaluation.

5.1 Overview

The implementation of Oxidize exists out of nine essential modules visible in Figure 5.3. The main module of the grammar is the Oxidize module depending on almost all other modules. The transformation modules in the figure are the: Ownership, Idiomatic, NonZero, Correct and Cleanup which all depend on the Rust grammar module. We can also see the traversal modules: Walk and Parse which also depends on the Rust grammar.

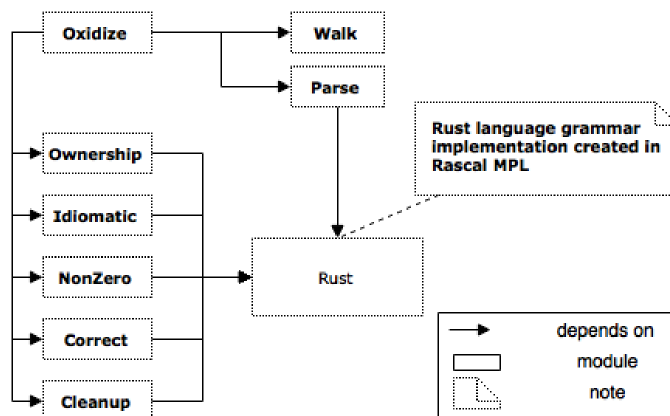


Figure 5.1: The class-diagram of the Oxidize framework

As first, we can see the main module of the framework, Oxidize. Containing the two possible usage functions which are through a CLI and the Eclipse IDE. This module is the main entry point of the program and contains the actions which can be performed by the user. It imports most of the other existing modules except for the Rust grammar it self.

We also have the five transformation modules, each dedicated to its own transformation. All of the transformation modules depend on the Rust grammar implementation for their usage of the matching patterns. The two last modules visible in the figure are the traversal modules for the analysis of the Rust files.

5.2 Idiomatic Loop Transformations

For the first four transformation performed by OXIDIZE we clean-up the unused lifetime labels created by the CORRODE translation transformation on the iteration statements. The CORRODE translation can add in some cases unused or clutter code. An example of this can be an iteration statement label which is unused by the statement and can confuse the reader into reasoning that it is used by the statement. These transformations are included in the idiomatic transformations. This is done because of its relevance to the idiomatic loop transformation after the clean-up.

Statement labels

The idiomatic loop transformation begins with a prerequisite cleanup step. This is done to cleanup the code of unnecessary *label* expressions which can be assigned to iterative statements. These labels can be used to escape out of an iterative sequence by targeting a specific iterative statement. By first verifying that an iterative statement does not break an iterative sequence we can later check if this statement qualifies for an idiomatic transformation.

```
1  case (Expression_while) ‘<Lifetime lt>: while <Expression cond> {
2                          '    <Statement* stmts> <Expression? expr>
3                          '}‘ =>
4      (Expression_while) ‘while <Expression cond> {
5                          '    <Statement* stmts> <Expression? expr>
6                          '}‘
7    when !used_lifetime(stmts, lt)
8
9  case (Expression_while_let) ‘<Lifetime lt>: while let <Pattern ptn> = <Expression cond> {
10                             '    <Statement* stmts> <Expression? expr>
11                             '}‘ =>
12     (Expression_while_let) ‘while let <Pattern ptn> = <Expression cond> {
13                             '    <Statement* stmts> <Expression? expr>
14                             '}‘
15   when !used_lifetime(stmts, lt)
```

Listing 5.1: Removing unused lifetime declaration from ‘`while`‘ statements
(RASCAL implementation code)

The first case transformations performed by OXIDIZE are the two cases present in Listing 5.1. The two cases of a ‘`while`‘ statement exist because of their dependency on the implementation of the grammar. The grammar distinguishes the two cases and so must the cases distinguish them too. Now that we have seen two examples of a transformation case we can begin explaining why they are created, how they are structured, why and what they can actually do. All the examples of transformations make use of the RASCAL's ‘`visit`‘ statement structure [36].

On the first line of Listing 5.1 we can see that the type of statement that we are looking for is the ‘`while`‘ statement because of its ‘`Expression_while`‘ typing specified in the grammar. This grammar typing is then followed by the actual code fragment that we are looking for. Which in this case is a ‘`Lifetime`‘ followed by a ‘`while`‘ statement with optional statements (zero-or-more statements denoted by ‘`*`‘) residing within its body and an optional single expression at the end of the body. In the Listing 5.2 we can see the grammatical rule for the ‘`while`‘ statement. This rule reflects our explanation of the ‘`while`‘ statement.

```
1 syntax Expression_while
2     = (Lifetime ":")? "while" Expression!pathStruct Block
3     ;
4
5 syntax Expression_while_let
6     = (Lifetime ":")? "while" "let" Pattern "=" Expression!pathStruct Block
7     ;
```

Listing 5.2: While statement as specified in the grammar
(Rascal grammar code)

In our transformation ‘`case`‘ we have chosen to fully write-out the contents of the ‘`while`‘ statements body for readability reasons. In both cases what happens is that the new ‘`Expression_while`‘ is inserted in place of the matched expression. Also in both cases, the statements are replaced by themselves with one distinct difference of not being assigned a ‘`Lifetime`‘.

In Listing 5.1 we can also see the usage of the keyword ‘`when`‘ which gives us the option of using an (assignable) expression which has to evaluate to a value, e.g. ‘`true`‘, to allow the transformation to succeed. If this value would evaluate to ‘`false`‘ the transformation would not happen and the next case would be tried. In all of the idiomacy cases, we make use of the ‘`used_lifetime`‘ function which is specified by us and checks for the use of the assigned lifetime name in the statements present in the body of the given block.

```
1 bool used_lifetime(Statement* stmts, Lifetime lt) = /lt := stmts;
```

Listing 5.3: The ‘`used_lifetime`‘ function used to check if a given lifetime name is used in the given scope

The function in Listing 5.3 is a boolean return value function taking in statements and a lifetime. In this function we are looking for the usage of the lifetime label specified in the source code. This is done by looking for the usage of the lifetime label in the body statements.

This ‘`used_lifetime`‘ check is necessary for our transformation to determine if the statement is safe to be transformed. If our transformation would just erase the lifetime label of the statement and this label would be used in the statement to escape it, we would introduce a logically incomplete program. This specific case applies to the following expression which could be present in the block: ‘`break 'my_loop;`‘. This break targets a specific loop in which we could not erase the original label from the program.

$$\frac{\text{label } L_1 \text{ defined on expression } E_1 \qquad L_1 \text{ not used in } E_1}{CFG(E_1)} \tag{5.1}$$

After completing the step of finding the lifetime and finding that a lifetime is not used in the body of the statement in question we can transform our found code into our target code without the lifetime. The exactly the same procedure applies to the other two types of loops: the ‘`loop`‘ and the ‘`for`‘ statements.

Loop transformation

Now that we have our first clean-up transformations done, we can continue with the following transformation to the structure of a statement. Our previous transformation has cleaned-up the unused lifetimes for us and has left the iterative statements untouched. This allows us to look for ‘loop‘ statements without a lifetime and reduced the number of cases that need to be implemented.

Each of the iterative statements can be used interchangeably with another iterative statement. In this specifics case we are trying to match a ‘loop‘ statement that is used as a ‘while‘ statement. This is done to then transform it into a ‘while‘ statement as it was already used as one. Such ‘loop‘ statement makes use of a inverted ‘if‘ statement condition within the statement which only contains a ‘break‘ statement.

```
1 loop {
2     if !(i < 10) {
3         break;
4     }
5     println!("Hello, world!");
6     i += 1;
7 }
```

```
1 while i < 10 {
2     println!("Hello, world!");
3     i += 1;
4 }
```

Figure 5.2: Example of pre- and post-transformation code for visualisation of Figure 5.3

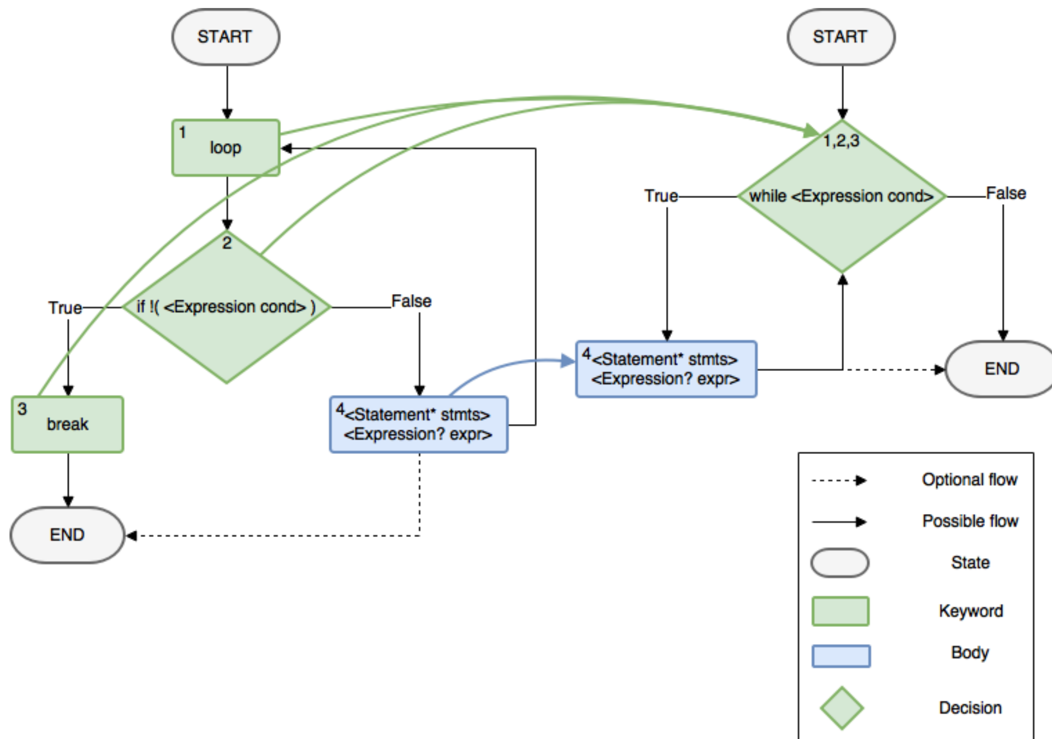


Figure 5.3: Visual representation of the ‘loop‘ to ‘while‘ transformation. Both flow-diagrams correspond to their code equivalents Figure 5.2

The structure of the ‘loop‘ to ‘while‘ transformation can be seen in Figure 5.3. The transformation changing the *green 1, 2 and 3* elements in the left flow diagram to the right *green 1,2,3* element. As well as, moving the blue (4) statements element in the body to the new body of the statement also denoted by the blue (4) element.

```
1 case (Block_expression) `loop {
2                         '    if !(<Expression cond>) {
3                         '        break;
4                         '    }
5                         '    <Statement* stmts> <Expression? expr>
6                         '}` =>
7      (Block_expression) `while <Expression cond> {
8                         '    <Statement* stmts> <Expression? expr>
9                         '}`
```

Listing 5.4: Transformation performing a `while` statement to `loop` statement refactoring

The transformation seen in Listing 5.4 is a transformation specifically designed for output produced by the Corrode project. This case is an example of the Corrode project generating a `loop` statement from a C `while` statement. This has been chosen for the practicality of the general loop statement transformation which can be used for all three loop statements present in Rust.

Oxidize can use this fact to transform the generally used `loop` of Corrode into its proper and idiomatic loop statement. In Listing 5.4 we can see an example of such transformation. Here we are looking for a `loop` statement containing an `if` statement as its first statement and containing only a `break` expression. From our previous transformation, we know that this `break` cannot be a break which is using a lifetime name.

By finding our `loop` statement without prerequisites we can then transform its contents into a `while` loop by moving the `if` condition into the `while` statements condition. The condition of a `if` statement works in an inverted way of a `while` statement condition. For this reason, we need to make use of the inverse of the `if` statement condition. By shifting from a `loop` to a `while` statement we can also delete the `break` expression. Listing 5.4 transformation does not require any further `when` checks because of our clean-up early on in the transformation process.

5.3 Ownership System Transformation

The previous few idiomacy transformations reduced the amount of code needed to fulfil the target codes goal and have also increased the readability of the target code by using commonly known constructs. The following transformation implemented in Oxidize is the Ownership system transformation from the C style memory allocation construct. This transformation set contributes to not only idiomacy and readability but also the overall memory safety of the target program.

```
1  Tree raii(Tree crate) = bottom-up visit(crate){
2      case (Block_item) `unsafe extern <String? st> fn <Identifier fn_id>
3                        '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
4                        '    <Inner_attribute* ia>
5                        '    <Statements stmts>
6                        '}` =>
7           (Block_item) `unsafe extern <String? st> fn <Identifier fn_id>
8                        '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
9                        '    <Inner_attribute* ia>
10                       '    <Statements otc>
11                       '}`
12       when fi  := find_Identifiers(stmts),
13            fdi := fi.def,
14            fii := fi.ini,
15            aid := fdi + fii,
16            fvf := find_variable_free(aid,stmts),
17            fdi := fdi & fvf,
18            fii := fii & fvf,
19            mt  := modify_type(fvf,stmts),
20            mdi := marray_definition_identifiers(fdi,mt),
21            mii := marray_initialization_identifiers(fii,mt),
22            mid := mdi + mii,
23            df  := delete_free(mdi,mt),
24            vtn := void_to_none(mid, df),
25            vac := value_assignment_correction(mid, vtn),
26            vpc := value_passing_correction(mid,vac),
27            vuc := value_usage_correction(mid,vpc),
28            otc := option_type_correction(vuc)
29 };
```

Listing 5.5: The Ownership transformation from C memory allocation usage

The transformation is specifically targeted at the code generated by Corrode. This compiled code does not feature certain Rust specific features as the Ownership system or idiomacy in iterative statements (as presented in Section 5.2). The basic motivation behind this transformation is that certain variables/fields pointing to a value in the memory (F points to value V) can become the owners of the value in Rust (F owns value V).

In Listing 5.5 we can see the Ownership system transformation specifying which steps need to be completed first before the actual transformation can be executed. This transformation depends on four functions performing various filtering tasks, seven functions performing in between transformations and four set mutations.

The first part of the Rascal's transformation construct present on lines 7-11 is the end transformation performed on the input source code. This transformation is performed after filtering and in between transformation have succeeded. The syntax construct used on lines 2-6 envelopes an `unsafe` function created by Corrode project to denote the use of `unsafe` language construct, e.g. the use of the C library package.

```
syntax Item_unsafe_fn
    = "unsafe" ("extern" String?)? "fn" Identifier identifier
        Generic_params? generic_params
        Fn_decl Where_clause? Inner_attributes_and_block
    ;

syntax Item_fn
    = "fn" Identifier identifier
        Generic_params? generic_params
        Fn_decl Where_clause? Inner_attributes_and_block
    ;
```

Listing 5.6: Unsafe function definition in Rust grammar

Two examples of the possible function declarations in RUST can be seen in Listing 5.6. The first one being for the unsafe function declaration and the second for a normal function. The difference is the use of the '`unsafe`' keyword and the '`extern`' keyword for *Foreign Function Interface* (FFI) [37] use in the unsafe function.

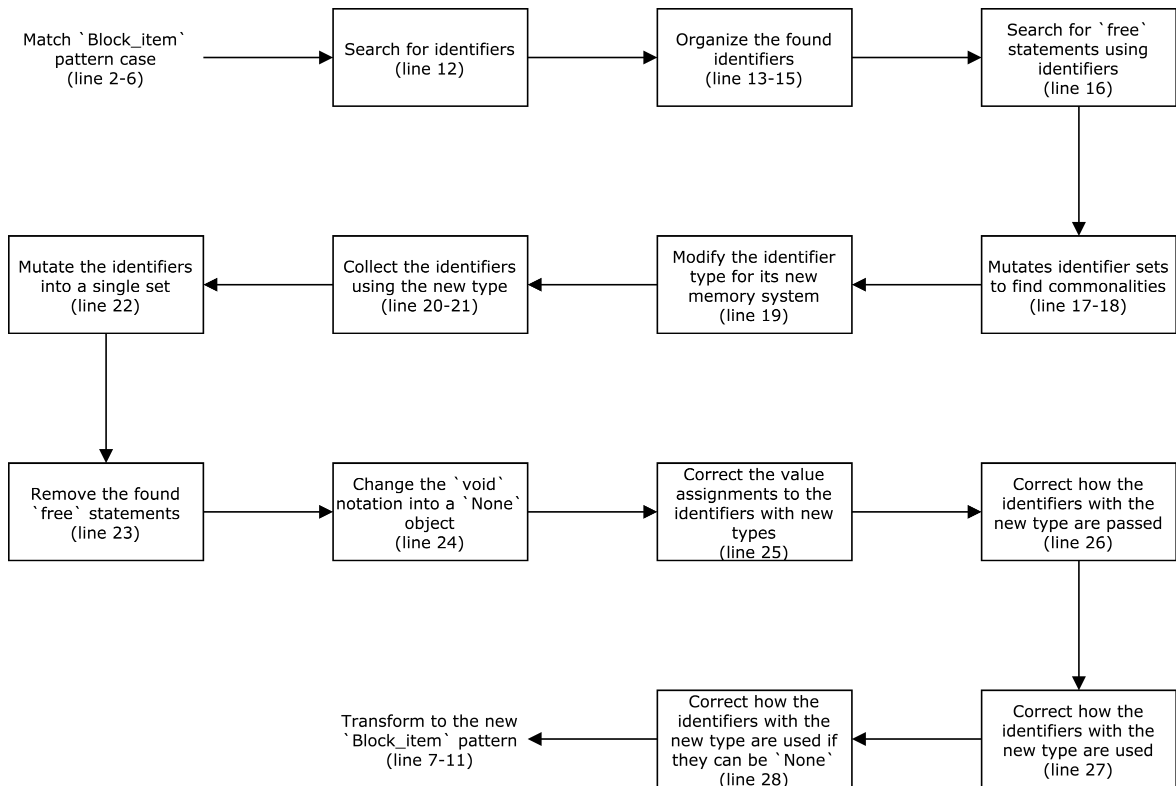


Figure 5.4: The Ownership system transformation activity flow

In Figure 5.4 we can see the activity flow of the Ownership transformation. In total we have developed 13 steps which need to be completed to transform a RUST program which uses the C library malloc memory management. By searching for a whole function in which we know a C library needs to be used to even consider this transformation, we narrow our search sample to again ensure that the transformation is only performed when and where it is needed. Looking at the target code (Listing 5.5) on lines 7-11 we can see that the only difference from the searching lines is that the statements present on line 10 are changed to the statements which are filtered and transformed at the end of the '`when`' pipeline on line 28 in Listing 5.5 (the '`otc`' assignment).

Variable identification

Let F be a variable. Define:

$$CreatedOnce(F) = \{[F] = [*mut\ u8]\ \text{, and}\ F \text{ is defined only once}\} \tag{5.2}$$

Our transformation begins (on line 12 of Listing 5.5)(Type Constraint 5.2) firstly with the identification of all the variable definitions and initialization in the function scope defined by lines 2-6. This identification is needed to determine if the variables present in the scope are shadowed by other scopes present in our found function scope.

The current state of the transformation code does not handle shadowing of variables with which it has been chosen to avoid this situation and only handle variables which are only defined and/or initializes only once. An improved scope handling and cross-file analysis have been planned for the future work. This variable classification also only handles variables with a specific type of '`*mut u8`' [38]. This type is the pointer data-type consisting of a raw mutable pointer ('`*mut`') of the unsigned integer type ('`u8`'). This is how pointers can be typed and are typed by the Corrode compiled code.

After the classification of both definition and initialization cases, the function returns a tuple of two sets, the first one containing a set of all variable definitions and the second one containing a set of all variable initializations. This separation of both sets has been chosen for the easier usability of the actions that follow in this transformation.

Local variables used by '`free`' statement

Let F be a variable. Define:

$$\begin{aligned} FreeUsed(F) = \{&free \text{ statement is used in scope, and} \\ &NumParams(free) = 1\ \text{, and} \\ &[Param(free, 1)] = [*mut\ c_void]\ \text{, and} \\ &Decl(Param(free, 1)) = [*mut\ u8]\} \end{aligned} \tag{5.3}$$

The step to follow is the function for identifying which variables are defined and/or initialized in the scope and are also used in the last step of the malloc usage. This step is the freeing of the allocated memory on the heap (on line 16)(Type Constraint 5.3). An example of such statement can be seen in Listing 5.7.

```
1   free(<Identifier id> as (*mut ::std::os::raw::c_void));
```

Listing 5.7: The matching case used for the filtering step of the freeing statements

The example in Listing 5.7 is how a C void type can be specified in Rust. By also filtering for the usage of the variables in the '`free`' statement we can determine which variables are of importance for us and the transformation. By filtering the variables by their local (in scope) definition and/or initialization, together with the filtering of the '`free`' statement usage on the filtered variables we have accumulated a new set of variables which qualify for the Ownership system transformation (on lines 17 and 18).

Pointer type modification

$$\frac{\begin{array}{c}\text{definition } E_1 \;=\; E_2 \\ [E_1] \;=\; [*mut\ u8] \\ CreatedOnce(E_1) \\ FreeUsed(E_1)\end{array}}{E_1\ owns\ E_2} \tag{5.4}$$

Now that we have identified our malloc variables and cleaned the unneeded ‘`free`‘ statement usage, we can start modifying the typing of the variables to fit our new structure of the Ownership system (on line 19)(Type Constraint 5.4). This begins by modifying the malloc variable type from the earlier defined ‘`*mut u8`‘, a raw pointer type, to ‘`MArray<u8>`‘, “a null-terminated array, which can be used to represent an e.g. array of C strings terminated by a null pointer” [39].

This type modification is needed to correctly and fully support the Ownership system. Usage of the MArray data type is a design choice because of its implementation wrapping a malloc into the Ownership Box. An example of this can be seen in Figure 2.2. The MArray is a part of a library called MBox. This library is based on malloc-based Box implementation in Rust [39]. This allows us to keep the data types of ‘`*mut u8`‘ if they are used as the function parameters. This has been chosen to preserve the behaviour of the original code as much as possible. A cross file analysis would be needed and recommended in the future to possibly refactor the MArray to a full Box implementation.

MArray type variable gathering

Let F be a variable. Define:

$$MarrayType(F) = \{F \text{ is of type } [MArray]\ \} \tag{5.5}$$

Now that we have performed a code transformation (type modification) and information gathering we perform an additional check for the variables which contain the ‘`MArray<u8>`‘ type. This is done to ensure the correctness of the transformations and to not transform code which has not qualified for the previous transformations. This is done with the help of two functions (on lines 20 and 21)(Type Constraint 5.5). Both functions return a new set containing identifiers qualifying for further transformations, one for the variable definitions and one for the variable initializations. This splitting of definition and initialization while not needed anymore is kept for code consistency and readability.

‘`free`‘ statement cleanup

$$\frac{\begin{array}{c}\text{call } free \text{ in method } M_1 \\ NumParams(free) = 1 \\ [Param(free,1)] \;=\; [*mut\ c_void] \\ Decl(Param(free,1)) \;=\; [MArray]\end{array}}{Decl(Param(free,1)) \text{ owns } Param(free,1)} \tag{5.6}$$

The consecutive step is a cleanup step to delete the usage of the ‘`free`‘ statement on the currently still malloc style of memory management (line 23)(Type Constraint 5.6). This is done by the same matching case as the ‘`free`‘ statement specified earlier (Type Constraint 5.3). The difference is the check for the used variable being of type ‘`MArray<u8>`‘.

Void to None

$$\frac{\begin{array}{c}\text{assignment } E_1 \; = \; E_2 \\ [E_1] \; = \; [\mathit{MArray}] \\ [E_2] \; = \; [\mathit{c_void}] \\ \mathit{MarrayType}(E_1)\end{array}}{E_1 \text{ owns } E_2} \tag{5.7}$$

The earlier specified void notation from C in Rust (Listing 5.7) can also be used to assign void to a variable and not only to specify its type. This C void pointer (‘`*mut ::std::os::raw::c_void`‘) can also be on the assignment and comparison side of a variable. This notation while looking slightly different, assigns or can be compared to a void value ‘`0i32 as (*mut ::std::os::raw::c_void) as (*mut u8)`‘. The assignment of this type is equal to the assignment of a ‘`None`‘ value in Rust and removing any value that was previously there or initializing a variable without an actual value (on line 24)(Type Constraint 5.7).

$$\frac{\begin{array}{c}\mathit{Comparison}(E_1, E_2) \\ [E_1] \; = \; [\mathit{MArray}] \\ [E_2] \; = \; [\mathit{c_void}] \\ \mathit{MarrayType}(E_1)\end{array}}{\mathit{Decl}(E_1) \text{ owns } E_1} \tag{5.8}$$

The comparison of this type checks if the type is null or not. Just like the previous example, this is equivalent to comparing a variable to a ‘`None`‘ and checking if the value is or is not null. By changing the values to a MArray type, we also have to modify the assignments and comparisons of the variables now specified as a MArray value instead of a raw pointer. This transformation (on line 24)(Type Constraint 5.8) has to do with the void value assignment/comparison to a Rust ‘`None`‘ type. This transformation consists of four cases applying to a variable initialization, variable definition, and variable equivalence check for equality and inequality.

Transformation corrections

In our approach to transformation, we have chosen to take every transformation step by step and not deal with any value assignment corrections at the same time as for example a type change. This can be already concluded from the previous paragraphs about the MArray type transformation and an additional transformation to the value assignment and comparison.

This approach lowered the complexity of our transformations with simple and readable code performing only the most needed actions. This decision also yields that code related to the transformed code needs to be “corrected”. The correction in this context does not necessarily mean that the transformations performed faulty actions but that the transformations did not handle all the relations to the, by this point, transformed code. Only after all transformations are performed, the code can be called semantically equivalent. The “corrections” are performed in the following ‘`_correction`‘ functions (lines 26-28 in Listing 5.5).

Assignment wrapping

$$\frac{\begin{array}{c}\text{assignment } E_1 \; = \; E_2 \\ [E_1] \; = \; [MArray] \\ [E_2] \; = \; [*mut\ u8] \\ MarrayType(E_1)\end{array}}{\begin{array}{c}E_1 \text{ owns } E_2 \\ [E_1] = [*mut\ u8]\end{array}} \tag{5.9}$$

The first is the correction for value assignment to the transformed variables (on line 25)(Type Constraint 5.9). The allowed assignment type of the MArray variables is the wrapper raw pointer assignment as specified by the previous implementation of the C malloc. This raw pointer has to be encapsulated/wrapped by the MArray wrapper. This transformation can be done by encapsulating the already assigned value with the from_raw function specified by MArray (‘`MArray::from_raw(<Expression expr>)`‘). This action can only be performed on raw pointers and so can't be done on the previously created ‘`None`‘ types for the void pointers. This transformation consists of two cases for the possible definition and initialization of the variable.

The correction transformations, just as the type transformations, are created with the principle of a single change to ease the complexity of transformation code. Our current transformation (Type Constraint 5.9) changes the type of a to be assigned raw value pointer. This is the first change that was required to be performed on the variables with changed types. The follow-up transformation handles the use of the value of the transformed variable.

MArray passing correction

$$\frac{\begin{array}{c}\text{call } M_1 \\ [Param(M_1, i)_1] \; = \; [MArray] \\ MarrayType(Param(M_1, i)_1)\end{array}}{\begin{array}{c}Decl(Param(M_1, i)_1) \; owns \; Param(M_1, i)_1 \\ NumParams(M_1) \; = \; Decl(NumParams(M_1)) \\ [Param(M_1, i)_1] \; = \; [*mut\ u8] \\ [M_1] \; \leq \; Decl(M_1)\end{array}} \tag{5.10}$$

$$\frac{\begin{array}{c}\text{assignment } E_1 \; = \; E_2 \\ Decl(E_2) \; = \; [MArray] \\ MarrayType(E_2)\end{array}}{\begin{array}{c}E_1 \text{ owns } E_2 \\ [E_2] \; = \; [*mut\ u8]\end{array}} \tag{5.11}$$

This next correction transformation deals with the transformed variables are passed as values to functions or assigned to other variables (on line 26)(Type Constraint 5.10-5.11). This transformation consists of five cases and one helper function transformation for the first case.

This first case is about variable passing to function, which starts with first detecting a value passing to a function. This subsequently follows with the helper function to find the earlier transformed variables to call a function within the MArray wrapper called ‘`as_mut_ptr`‘. This function call returns a mutable representation of a raw pointer within the MArray wrapper without moving the value out of the current scope.

This case is then followed with four other cases for the variables being assigned to other variables. All four cases handle a similar situation to the helper function in a way of modifying the assigned variable to make use of the MArray function ‘`as_mut_ptr`‘. The following correction transformation handles not the passing or the assignment but the usage of one of the modified values (on line 27). This transformation is very similar to the previous one with the exception of targeting a singular

expression making use of the modified values, e.g. a ‘is_null()‘ check on the variable needing to be modified to ‘as_mut_ptr().is_null()‘.

Option type correction

The last transformation for the Ownership system (on line 28) has the goal of correcting values which can possibly be or become null in the scope. This transformation exists out of a few subsequent transformations for the correctness of the possible null value. The following few transformations perform two actions per transformation instead of the normal one action per one transformation. This is to limit the number of visits on the tree and ensure that the correct identifiers are added to a set of identifiers for later use in the ‘when‘ pipeline checks for transformations.

$$\frac{\begin{array}{c}\text{assignment } E_1 = E_2 \\ [E_1] = [\mathit{MArray}] \\ E_2 = \mathit{None}\end{array}}{E_1 \text{ owns } E_2} \tag{5.12}$$

The first transformation (Type Constraint 5.12) handles the simple case of a variable declaration with the type of MArray and assigned the value of ‘None‘, e.g. ‘let mut <Identifier id> : MArray<u8> = None;‘. In this case, we save the identifier for later use in a set and modify the variable type to be encapsulated by the Option value, e.g. ‘let mut <Identifier id> : Option<MArray<u8>> = None;‘. This Option wrapper is either ‘Some‘ and contains a value or is a ‘None‘ and does not contain a value [40].

This construct comes in practice when the value of the Box pointer is of value None or is compared to None while not being able to be of type None and thus can cause a compilation halt if it is not handled correctly. The subsequent four cases handle the same situation as the previous one but with the difference of the variable being assigned or compared to None later in the code.

$$\frac{\begin{array}{c}\mathit{Comparison}(E_1, E_2) \\ [E_1] = [\mathit{MArray}] \\ E_2 = \mathit{None} \\ \mathit{MarrayType}(E_1)\end{array}}{\mathit{Decl}(E_1) \text{ owns } E_1} \tag{5.13}$$

The following two transformations perform the action of encapsulating the assignment of a value to the ‘Option<MArray<u8>>‘ data type (Type Constraint 5.13). This data type needs the wrapper Some to be the type of value that is assigned to it self. This means that every value assignment of type ‘MArray<u8>‘ which was transformed before from the ‘*mut u8‘ type needs to be transformed again to incorporate the data type change to the Option type and needs to be wrapped into the Some wrapper.

Unwrap pointer

The last transformation for the Ownership system is the correction of the ‘as_mut_ptr()‘ function to ‘as_mut().unwrap().as_mut_ptr()‘. This transformation is needed because of the previous case moving the actual pointer out of the wrapper and with that not borrowing the value to the passed function but giving the ownership away to the passed function. This causes the pointer to be deleted after the scope of the said function ends.

Crate Import

```
1 extern crate mbox;
2 use self::mbox::MArray;
```

Listing 5.8: The MBox library specification and the use of MArray needed for the library to work with the code

The final touch to the use of the Ownership system does not necessarily make part of the Ownership system transformation as its a part of its implementation. To make use of our MBox library for the MArray type we need to import the library into the file which is making use of this type. This is done by checking if this library is not already imported and if there is any use of the keyword MArray in the code. This way we can simply add the library to the file and specify the use of the required MArray. This transformation is to lift this action from the user and let Oxidize determine if this import is needed on file basis.

5.4 NonZero Transformation

```
1  Tree nonzero(Tree crate) = bottom-up visit(crate){
2      case (Block_item) `unsafe extern <String? st> fn <Identifier fn_id>
3                         '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
4                         '    <Inner_attribute* ia>
5                         '    <Statements stmts>
6                         '}` =>
7           (Block_item) `unsafe extern <String? st> fn <Identifier fn_id>
8                         '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
9                         '    <Inner_attribute* ia>
10                        '    <Statements inc>
11                        '}`
12       when nci := null_check_id(stmts),
13            iis := id_in_scope(nci,stmts),
14            inc := is_null_checked(iis,stmts)
15 };
```

Listing 5.9: The NonZero transformation of values which cannot be zero (0) or None

The following transformation in Oxidize is the NonZero compiler optimization. This is an experimental wrapper type for raw pointers and integers which cannot be and will not be null or zero (number 0) in the scope of its lifetime [41]. This wrapper gives us access to its two only methods which are new and get. This wrapper is in development by the Rust development community and is only partially implemented in Oxidize.

The NonZero wrapper is known under its #27730 [42] *Request For Comment* (RFC) number on GitHub since 2015. To make use of this wrapper we need to compile the project with the nightly compiler which is comparable to a bleeding edge version of a project. This is not an idiomatic construct and thus has not been fully realized in Oxidize.

This transformation of a wrapper construct requires handling of many possible cases in which the construct can be used and it is prone to change in future versions of NonZero implementation. The current documentation and resources available about the NonZero are just a few and do not show a real life application of the construct. This transformation has been partially implemented to present the possibilities of Oxidize and the use of not only stable constructs known in the language.

The current implementation only checks for three cases of applicability of the NonZero wrapper

to an existing variable construct. Those being the check if a variable is compared to a null in an '`if`' statement block (on line 12), a check if the variable is created in the scope (on line 13) and not passed by as a parameter and if the NonZero wrapper has not already been applied to the variable(on line 14).

This transformation does not take into account the use of the wrapped construct. In the case of CVS, this can be used in a '`free`' statement which was not qualified for transformation in our Ownership system or is used in a function where we currently cannot control the flow of actions.

```
1 extern crate core;
2 use self::core::nonzero::NonZero;
```

Listing 5.10: The NonZero library specification and the use of NonZero needed for the library to work with the code

The current implementation of the transformation adds the NonZero wrapper to the scope of the variable in question. An example of transformation can be seen in Figure 2.3 of Chapter 2 on line 2. Just like the Ownership system transformation, there is a correction transformation for the NonZero library usage import. This transformation adds the needed library usage shown in Listing 5.10.

5.5 Cleanup Transformations

The last set of transformations is focused on cleaning up the output code of the Corrode project and the output which our transformation can leave behind. In the case of Corrode we can see that the project can leave unneeded temporary variables behind which do not serve any purpose in the program and can clutter the code. This transformation can be seen in Listing 5.11.

```
1  case (Statements) `let mut tmp : *mut ::std::os::raw::c_void =
2                     '<Identifier _> as (*mut ::std::os::raw::c_void);
3                     '<Statement* stmts1>
4                     'free(tmp);
5                     '<Statement* stmts2>` =>
6       (Statements) `<Statement* stmts1>
7                    '<Statement* stmts2>`
8
9  case (Statements) `<Statement* stmts3>
10                    'let mut tmp : *mut ::std::os::raw::c_void =
11                    '<Identifier _> as (*mut ::std::os::raw::c_void);
12                    '<Statement* stmts4>
13                    'free(tmp);` =>
14      (Statements) `<Statement* stmts3>
15                    '<Statement* stmts4>`
16
17 case (Statements) `<Statement* stmts5>
18                    'let mut tmp : *mut ::std::os::raw::c_void =
19                    '<Identifier _> as (*mut ::std::os::raw::c_void);
20                    '<Statement* stmts6>
21                    'free(tmp);
22                    '<Statement* stmts7>` =>
23      (Statements) `<Statement* stmts5>
24                    '<Statement* stmts6>
25                    '<Statement* stmts7>`
```

Listing 5.11: The clean up transformation for temporary variables left behind by the Corrode project transformation

Because of a current bug in the implementation of Rascal we have to repeat our cases for specific transformation searches. All the three cases in Listing 5.11 are created for the same purpose of detecting a '`c_void`' type '`tmp`' variable initialization together with the freeing of this variable in the same scope. The origin of this construct is not clear and does not serve any purpose in the final output

of CORRODE and could be a remnant of one of the CORRODE transformation. The transformation of this construct has been created to remove the conflicts that it had with out Ownership system transformation.

```
1 case (Statement) ‘if <Expression _> {{}}‘ =>
2     (Statement) ‘{}‘
```

Listing 5.12: Clean up for the empty ‘`if`‘ statements created by our Ownership system transformation. The if statements would normally contain the ‘`free`‘ statements for the MALLOC constructs freeing

The other cleanup transformation implemented is to clean up the remnants of our own transformations from the Ownership system transformation. This case detects the existence of an ‘`if`‘ statement containing an empty block (two curly brackets) inside its own block and replaces this ‘`if`‘ statement by a single empty block. In the future, this transformation could be extended with a complete deletion of the empty block.

Chapter 6

Evaluation

In the first chapter, we have asked the two fundamental questions for our research. Those questions were about, what needs to be considered in a transformation to generate idiomatic Rust code, and what are the checks and actions that a transformation requires to be considered correct. In this chapter, we are discussing our research questions together with their evaluation.

6.1 Introduction

At the start of our research, we had two research questions which were of importance for the development of Oxidize. These questions can be seen in Chapter 1 and are repeated in this section.

1. What needs to be considered in a transformation to generate idiomatic Rust code?
   1.1. What are the relevant idioms?
   1.2. What are the matching cases for non-idiomatic Rust?
2. What are the checks and actions that a transformation needs to perform to succeed?
   2.1. What are their pre-conditions and assumptions for correct application?
   2.2. How can we validate the correctness of the transformation?

The research questions have helped us with the main focus of Oxidize. This focus is, of course, the idiomatic transformation of non-idiomatic Rust code. The first question applies to us at the time not knowing what to consider and what to focus on with a transformation. That is why we have developed the sub-questions of what idioms to focus on and what is their pattern that we could match.

The second questions were aimed at the transformation actions and how we should approach it. With the actions and checks of the transformation came the question of validation. The process of the program is designed to be as autonomous as possible and thus should deliver correct code transformation to the user. The second question with its sub-questions helped us focus on the transformation progression.

The two research questions were also applicable to the two phases in which Oxidize was created. The first phase was to acquaint ourselves with the Rust language and to develop the grammar. In this phase, we have decided on the idioms that we have focused on based on their applicability and occurrence in our main sample project, *Concurrent Versions System* (CVS).

The second development phase was dedicated to the development of the transformations for Oxidize. In this phase, we have decided on the cases to match and transform based on the results from the previous phase. Each transformation required a different approach to the process and evaluation.

In the subsequent sections, we are discussing both phases with their components and how they answer our research questions. In the next section, we are discussing how we have evaluated our results and how we can reconstruct it.

6.2 Method

Our evaluation chapter is the second phase of our research, the transformations and their evaluation. We begin with the second phase to demonstrate how we have determined the correctness of our solution to then in-depth explain the contributing components.

The Rust language is a compiled language which evaluates the majority of its safety checks at compile time. This enables the language to ensure that the compiled program is: type, memory and segmentation-fault safe. This has also enabled the language to inform and warn the developer of a program about incorrectness and problems.

This is where the *Rust Language Server* (RLS) is used to traverse, parse and analyze Rust code. The RLS was used to validate our transformed code and inform us about any incorrect code caused by our transformations. This includes safety, memory and typing checks all provided by the RLS.

An example of the output provided by the RLS can be the Problem list present after the RLS compilation of a Rust project. To validate our results we have validated this Problem output after each and single Oxidize transformation. The project used for the development transformations was the *Concurrent Versions System* (CVS) project translated by the Corrode project.

The source and target code of the CVS project transformed by Oxidize did not have any critical problems which would compromise the Rust compilation. The source and thus also the target code have minor problems reported by the RLS. Problems as style usage (snake case usage instead of the default camel case), variables which do not have to be mutable but are made mutable by Corrode and unused variables.

Our secondary sources of evaluation were applications created by us to test the transformations. These tests varied from minimally required code samples to trigger the pattern matching to applications filled with data to test if the transformations changed any crucial information to the validity of the code.

The general method of our transformation validation was to develop the smallest possible transformation part which would have an impact on the target code and then to validate it manually with the RLS. For this task, we have used the Visual Studio Code which implements the RLS and also Racer which is a Rust Code Completion utility.

1. Perform a transformation on source code
2. Generate the target code into a new directory
3. Open Visual Studio Code
4. Load the target code into the editor
5. Let RLS analyze the project
6. Act on the feedback

This manner of working has provided us with feedback to all of our transformation steps. It made us consider transformations and set mutations which helped us with the development of our solution. This process could be automated by the use of the RLS through the CLI to receive the feedback in Eclipse.

6.3 Application

In this section, we are dicussing our key results from the transformations performed by OXIDIZE. The code samples used in this section are that of the source code (on the left) and that of the target code (on the right). To reduce the amount of code clutter we are only showing the important excerpts of the source and target code. By only using the excerpts we can focus on what OXIDIZE analyzes and transforms.

The following figures demonstrate our idiomatic transformation of the loop statement.

```
1 'loop1: loop {
2     if !(i < wrap_count + wrap_tempcount) {
3         break;
4     }
5
6     i = i + 1;
7 }
```

```
1 while i < wrap_count + wrap_tempcount {
2
3
4
5
6     i = i + 1;
7 }
```

Figure 6.1: Idiomatic transformation of the labeled ‘`loop`‘ construct into a ‘`while`‘ construct

Figure 6.1 demonstrates how OXIDIZE transforms a labelled ‘`loop`‘ with a breaking ‘`if`‘ statement into a conventional ‘`while`‘ statement (also visible in Figure 5.3). In this transformation, we can have statements present anywhere in the statement except for the ‘`if`‘ statement. If that would be the case, this example would not be transformed and would only be stripped of its unneeded label.

This and similar source code often occurs in the generated code by CORRODE. The examples shown in this section are from the CORRODE generated *Concurrent Versions System* (CVS) project. During our transformation, we have found out that the label deletion transformation is performed **76** times and the transformation shown in Figure 6.1 **51** times.

This is an idiomatic transformation which makes use of the well-known construct of the while statement and also reduces the amount of code required for the statement. Our next example is the transformation of the C-style malloc memory allocation construct.

```rust
unsafe extern fn rlog_proc () -> i32 {
  let mut where_ : *mut u8;

  if is_rlog != 0 {
    where_ = Xstrdup(...);



    where_ = dir_append(...);


  } else {
    where_ = 0i32
      as (*mut ::std::os::raw::c_void)
        as (*mut u8);
  }
}

  err = start_recursion(
    where_ as (*const u8),
  );
```

```rust
unsafe extern fn rlog_proc () -> i32 {
  let mut where_ : Option<MArray<u8>>;

  if is_rlog != 0 {
    where_ = Some(
      MArray::from_raw(
        Xstrdup(...)));

    where_ = Some(
      MArray::from_raw(
        dir_append(...)));
  } else {
    where_ = None;


  }
}

  err = start_recursion(
    where_.as_mut().unwrap().as_mut_ptr()
      as (*const u8)
  );
```

Figure 6.2: An example of the transformation performed by the ownership transformation. From the C-style malloc memory management to the ownership system

In Figure 6.2 we can see how OXIDIZE transforms the construct of a C-style malloc into RUST's own Ownership system. This example also demonstrates OXIDIZE handling the case of the variable being possibly '`None`'. This involves the addition of the '`Option`' and '`Some`' constructs. In the case when the variable would never be None we would not see both of the constructs.

The CORRODE project generates the C-style malloc memory allocation as the default type of pointer variables. Our solution does not apply the Ownership transformation to all possible cases of this because of variable shadowing and scope conflicts. OXIDIZE is able to perform this transformation **141** times on the generated CVS project.

The presented code are just examples of their corresponding single transformation. These code examples were chosen because of their readability and inclusion of the majority of the present transformations. A different example could show that a block can be used in the condition of the while statement which would seem complex but after all, it's just one of the grammatical interpretations.

6.4 Code Contribution

In this section, we focus on the contribution of code to our research questions. Each transformation in this section is accompanied by its grammatical rule which is used in its pattern matching. The order of this section is written in the order of Oxidize execution and thus goes with the activity flow of the application transformation.

The three main non-idiomatic constructs which we have focused on are the loop constructs, memory management and static analysis of null pointer checks. Those three non-idiomatic constructs have been chosen because of their relevance to the output of a framework (memory management), relevance to a more wide spread application (loop transformations) and use of advanced Rust constructs (null pointer check analysis).

```
1   case (Expression_while) `<Lifetime lt>: while <Expression cond> {
2               '  <Statement* stmts> <Expression? expr>
3               '}` =>
4     (Expression_while) `while <Expression cond> {
5               '  <Statement* stmts> <Expression? expr>
6               '}`
7    when !used_lifetime(stmts, lt)
8
9   case (Expression_while_let) `<Lifetime lt>: while let <Pattern ptn> =
10                             '<Expression cond> {
11                ' <Statement* stmts> <Expression? expr>
12                '}` =>
13    (Expression_while_let) `while let <Pattern ptn> = <Expression cond> {
14                ' <Statement* stmts> <Expression? expr>
15                '}`
16   when !used_lifetime(stmts, lt)
17
18  case (Expression_loop) `<Lifetime lt>: loop {
19              '  <Statement* stmts> <Expression? expr>
20              '}` =>
21    (Expression_loop) `loop {
22              '  <Statement* stmts> <Expression? expr>
23              '}`
24   when !used_lifetime(stmts, lt)
25
26  case (Expression_for) `<Lifetime lt>: for <Pattern ptn> in <Expression cond> {
27             ' <Statement* stmts> <Expression? expr>
28             '}` =>
29    (Expression_for) `for <Pattern ptn> in <Expression cond> {
30             ' <Statement* stmts> <Expression? expr>
31             '}`
32   when !used_lifetime(stmts, lt)
```

Listing 6.1: The transformation cases for the deletion of the unused labels

The first code example presented in Listing 6.1 demonstrates how the approach to the transformation of the unused labels present in the use of the iterative statements. The pre-condition for this transformation is that the label is not needed and can be removed from the statement. This can only be true if the label is not used in the body of the statement. Which is why before transforming the code we are checking if the label is used in the function passed to the '`used_lifetime`' function.

The transformations present in Listing 6.1 contribute to the readability because of their cleanup function of unused code. It also ensures that the coming idiomatic '`loop`' transformation is executed on an iterative statement which ends when its condition is reached and is not stopped because it had to be escaped by the '`break`' keyword targeting a specific loop. Targeting a specific loop would imply

that the statement construct is more complicated than it needs to be and should be manually checked.

```
syntax Expression_while
  = (Lifetime ":")? "while" Expression!pathStruct Block
  ;

syntax Expression_while_let
  = (Lifetime ":")? "while" "let" Pattern "=" Expression!pathStruct Block
  ;

syntax Expression_loop
  = (Lifetime ":")? "loop" Block
  ;

syntax Expression_for
  = (Lifetime ":")? "for" Pattern "in" Expression!pathStruct Block
  ;
```

Listing 6.2: Grammatical rules of the label cleaning transformations

The grammatical rules of the label cleaning transformations are shown in Listing 6.2. These iterative statement rules show how simple our grammar can be and also shows the possible future optimizations to the grammar. Currently, we are distinguishing the two cases of the ‘`while`‘ statement from each other but in the future, we could possibly combine them if they can always be used in the same context.

The following transformation is the idiomatic transformation of the ‘`loop`‘ statement using an ‘`if`‘ statement as a condition.

```
case (Block_expression) `loop {
                        '    if !(<Expression cond>) {
                        '        break;
                        '    }
                        '    <Statement* stmts> <Expression? expr>
                        '}` =>
     (Block_expression) `while <Expression cond> {
                        '    <Statement* stmts> <Expression? expr>
                        '}`
```

Listing 6.3: The case pattern for ‘`loop`‘ to ‘`while`‘ transformation

Because of our previous transformation, we do not have any variables to check and the transformation is as simple as matching the case that we expect. This is done by matching a ‘`loop`‘ statement which uses an ‘`if`‘ statement in its body as seen in Listing 6.3. This is a CORRODE specific construct which can also be used by new programmers. This is valid code and can be used as an alternative to the ‘`while`‘ statement but misses its idiomatic state of an infinite loop.

To put it into contrast with our research question, this transformation contributes to both of our questions. It makes use of a relevant idiom construct for its transformation and ensures its correctness by having a strict matching pattern. This transformation could also be performed on iterative statements with a label but that would complicate the transformation with the addition of RASCAL’s ‘`when`‘ checks.

The following transformation is that of the Rust's Ownership system. This is our most complex transformation which needs 17 steps of code traversal, set mutations and code transformation to successfully perform the operation.

```
1  case (Block_item) ‘unsafe extern <String? st> fn <Identifier fn_id>
2                    '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
3                    '    <Inner_attribute* ia>
4                    '    <Statements stmts>
5                    '}‘ =>
6       (Block_item) ‘unsafe extern <String? st> fn <Identifier fn_id>
7                    '<Generic_params? gp> <Fn_decl params> <Where_clause? wc> {
8                    '    <Inner_attribute* ia>
9                    '    <Statements otc>
10                   '}‘
11    when fi := find_Identifiers(stmts),
12         fdi := fi.def,
13         fii := fi.ini,
14         aid := fdi + fii,
15         fvf := find_variable_free(aid,stmts),
16         df  := delete_free(fvf,stmts),
17         fdi := fdi & fvf,
18         fii := fii & fvf,
19         mt  := modify_type(fvf,df),
20         mdi := marray_definition_identifiers(fdi,mt),
21         mii := marray_initialization_identifiers(fii,mt),
22         mid := mdi + mii,
23         vtn := void_to_none(mid, mt),
24         vac := value_assignment_correction(mid, vtn),
25         vpc := value_passing_correction(mid,vac),
26         vuc := value_usage_correction(mid,vpc),
27         otc := option_type_correction(vuc)
```

Listing 6.4: The base Ownership transformation case

The Ownership transformation is the transformation with the largest scope of matching. The matching cases enable our solution to find the relevant constructs and analyse them for their potential transformation. In case of the Ownership transformation, the pattern to match is a whole Rust function with the ‘`unsafe`‘ modifier. This is to ensure that the whole scope of the variable in question is matched and analyzed.

To ensure the correctness and validity of our transformation we had to perform manual tests on the malloc allocation in the CVS project. As stated in the previous section the CVS project contains the malloc allocation because of its generation from the Corrode project.

The first two use cases are the application of the Option construct together with the MArray construct. The use of the MArray comes from the Ownership transformation using a boxed/scoped variable environment in its system. This means that when this variable goes out of scope it will be destroyed instead of in the case of a malloc it would still reside in the memory if left without the use of the ‘`free`‘ function.

The next four cases can be seen in the assignment of the ‘`where_`‘ variable. In these two cases, we can see that the variable is assigned with the ‘`Some`‘ function together with the ‘`MArray::from_raw`‘. Just like the previous two cases, this is needed because of the variable being of the Option type and containing the MArray object. The Some corresponding with the Option and the from_raw corresponding the the MArray.

In the generated CVS code, the void is represented by a long path to the C library present in Rust.

This can be seen in Figure 6.2 on the left side '`0i32 as (*mut ::std::os::raw::c_void) as (*mut u8)`'. When we transform our code to the Ownership system we can replace this annotation by the use of a NONE object. This is an important value to the Ownership system allocation because the MARRAY type can not be of type NONE or VOID. The use of the VOID lets us know that the value can be empty and thus needs to be of the type OPTION.

As for the last use case, we can see that the '`where_`' variable is passed to a different function and thus to a different scope. In normal terms of RUST this variable would be borrowed if it would use a type which extends the COPY trait. In case of the OPTION type we are able to make use of this trait but in the case of the MARRAY type we can not. This is why the value passing has to first make a call upon functions to allow the variable borrowing. This can be seen in Figure 6.2 on the right side '`where_.as_mut().unwrap().as_mut_ptr()`'.

This transformation requires us to perform analyses, checks and in-between transformation before it can be successfully completed. This being one of the main transformation also contributes to both of our research questions. It applies an idiom to the non-idiomatic C-style malloc memory allocation.

```
1 syntax Item_unsafe_fn
2   = item_unsafe_fn:"unsafe" ("extern" String?)? "fn" Identifier Generic_params?
3       Fn_decl Where_clause? Inner_attributes_and_block
4   ;
```

Listing 6.5: The grammar notation used for the matching of f function with the unsafe modifier

The Ownership system transformation consists of the function with the UNSAFE modified as seen in Listing 6.5. This modifier denotes when a memory unsafe operation is performed in the function. In normal RUST case this operation is the use of an external function and/or the use of the C library.

The following transformations are the correction transformations performed by OXIDIZE. These transformations have been developed to supplement the already performed transformations. OXIDIZE performs its transformation in small and simple steps which are separated and easy to read. A transformation which does not fit into the core of the main transformation has been separated and applied after the main transformations have been performed.

```
1 case (Crate) `<Shebang_line* sl>
2              '<Mod_item* mi>` =>
3     (Crate) `<Shebang_line* sl>
4              '
5              'extern crate mbox;
6              'use self::mbox::MArray;
7              '
8              '<Mod_item* mi>`
```

```
1 case (Crate) `<Shebang_line* sl>
2              '<Mod_item* mi>` =>
3     (Crate) `<Shebang_line* sl>
4              '
5              'extern crate core;
6              'use self::core::nonzero::NonZero;
7              '
8              '<Mod_item* mi>`
```

Figure 6.3: The MBOX and the NONZERO transformation correction cases

The first correction transformation is the external library import addition. This transformation contributes to the validity and correctness of the transformed code. These imports are required for the application to function and make use of its function calls.

In both cases of the transformation present in Figure 6.3, we can see that it lets a file import an external library. On the left-hand side we are allowing the transformation to import the MBOX library for the use of the MARRAY object. On the right-hand side we are allowing the transformation to import the CORE library to for the use of the NONZERO object.

The next transformation is dedicated to the temporary variable creation by CORRODE. In some

cases CORRODE creates temporary variables together with the pointer types. Those variables cause problems after our Ownership system transformation and need to the removed without causing any further problems.

```
case (Statements) ‘let mut tmp : *mut ::std::os::raw::c_void =
                  ’<Identifier _> as (*mut ::std::os::raw::c_void);
                  ’<Statement* stmts1>
                  ’free(tmp);
                  ’<Statement* stmts2>‘ =>
      (Statements) ‘<Statement* stmts1>
                  ’<Statement* stmts2>‘

case (Statements) ‘<Statement* stmts3>
                  ’let mut tmp : *mut ::std::os::raw::c_void =
                  ’<Identifier _> as (*mut ::std::os::raw::c_void);
                  ’<Statement* stmts4>
                  ’free(tmp);‘ =>
      (Statements) ‘<Statement* stmts3>
                  ’<Statement* stmts4>‘

case (Statements) ‘<Statement* stmts5>
                  ’let mut tmp : *mut ::std::os::raw::c_void =
                  ’<Identifier _> as (*mut ::std::os::raw::c_void);
                  ’<Statement* stmts6>
                  ’free(tmp);
                  ’<Statement* stmts7>‘ =>
      (Statements) ‘<Statement* stmts5>
                  ’<Statement* stmts6>
                  ’<Statement* stmts7>‘
```

Listing 6.6: Temporary variables correction cases

This transformation is a by-product of the CORRODE and OXIDIZE transformations. In its unchanged state, the code would cause problems with the compilation of the program and needs to be corrected to ensure the validity and correctness of the generated code.

Our last transformation is also a by-product of our own transformations. This is a by-product of the Ownership system transformation and its removal of the ‘`free`‘ statements.

```
case (Statement) ‘if <Expression _> {{}}‘ =>
     (Statement) ‘{}‘
```

Listing 6.7: Empty ‘`if`‘ statements case

The CORRODE project generates its ‘`free`‘ statements in some cases in a ‘`if`‘ statement checking for the value of the variable. This is to ensure that the object is empty and no longer needed. Our transformation changing this malloc way of working into the Ownership system changes the way the variables are freed and no longer requires the manual freeing.

6.5 Threats to validity

In this section, we discuss the threats to our validity by describing the internal and external validity together with the reliability of our solution.

Shadowing

During the development of OXIDIZE we have noticed that our code for scoping and shadowing of identifiers did not generate reliable results. This is of importance for our Ownership system transformations. This problem of incorrect transformations was present in our code for some time and was a threat to our internal validity. We could not develop a better solution for this problem and have regarded this as an uncertainty to the validity of our results. To cope with this setback we have decided to regard identifier shadowing in the same and child scopes as an uncertainty. An uncertainty in our research is seen as not a valid candidate for a transformation and is omitted from the analysis and thus also from the transformation. This results in that our transformation could transform more malloc constructs into the Ownership system, but because of the current shadowing problems omits those identifiers.

NonZero

One of our main transformations focused on the NONZERO construct. This construct requires many checks which we did not anticipate from the beginning. This caused us to have not enough time to finish the transformation completely. The transformation works in the case of the CORRODE output but does not work well in the combination with the Ownership transformation. The inconsistency is regarded as a threat to the internal validity of the target code. This transformation would require identifier use checks and value type checks to cooperate with the Ownership transformation. In the current state is the transformation disabled to ensure that no type and use incorrectness is introduced into the target code.

The following transformation developed in OXIDIZE is the NONZERO transformation. This transformation has not been finished by our research and requires to be further researched and developed in the future.

```
1  start[Crate] nonzero(start[Crate] crate) = bottom-up visit(crate){
2    case (Block_item) ‘unsafe extern <String? st> fn <Identifier fn_id>
3                     ’<Generic_params? gp> <Fn_decl params>
4                     ’<Where_clause? wc> {
5                     ’  <Inner_attribute* ia>
6                     ’  <Statements stmts>
7                     ’}‘ =>
8         (Block_item) ‘unsafe extern <String? st> fn <Identifier fn_id>
9                     ’<Generic_params? gp> <Fn_decl params>
10                    ’<Where_clause? wc> {
11                    ’  <Inner_attribute* ia>
12                    ’  <Statements inc>
13                    ’}‘
14   when nci := null_check_id(stmts),
15        iis := id_in_scope(nci,stmts),
16        inc := is_null_checked(iis,stmts)
17 };
```

Listing 6.8: The NonZero transformation

Validation variety

To test the validity of our transformation we have used the generated *Concurrent Versions System* (CVS) project by the Corrode project. This CVS project is a version control library which needs to be used by another project to function. This library project can be limited in the use of the user interaction constructs and can be a threat to our external validity. This could mean that our solution does not target all the possible constructs and should be tested with different types of programs.

An example of this could be other generated project by Corrode or different frameworks, a program written by Rust and non-Rust developers, a program which uses the Cargo framework and a program which does not use the Cargo framework. To tackle this problem we have targeted our checks and validations at specific cases present in the CVS library with which proved to use to be valid. In the case of the Ownership transformation, we focused on the identifier and its value, value assignments, value passing/borrowing and value going out of scope.

Wider validity

Our framework is not a complete framework for idiomatic rule refactoring. To refactor all of the common practices the framework would have to mature further on in the future. This is, of course, a threat to the reliability of our solution. At this stage of the project, we should not trust the generated code to the full extent. The output should still be manually checked by the developer to validate if the transformation generates the expected code. For this reason, we have targeted our transformation at a small group of common patterns present in the output of the Corrode project. This gave us the opportunity to focus on specific solutions to develop.

Testing suite

In the current state of our project, we do not have a thorough application testing suite process. At the beginning of our project, we wanted to test the source and the transformed application against its own testing suite. This testing suite would not be transformed by Oxidize and would be a vital feedback point to the user. This would be the last step of our process and its focus would be the user feedback about the changes that we have made. The best case scenario output would be that the test output would be exactly the same and in the worst that they would be different.

To counter this problem we have decided to focus on how the transformations are performed. The transformations target the usage of a construct and not its value. The Ownership system transformation is an example of this process. This transformation handles the usage of pointers and not their values. We change the pointer construct with the Ownership wrapper and from there on we unwrap it where it is needed. The pointer in its wrapped and unwrapped state is still the same pointer and the value does not change but its usage does. For this reason, we have created the correction steps which are performed during and after the Ownership transformation.

Chapter 7

Related Work

During our research we have used the literature by Darwin to determine the similarities and differences between Oxidize and code linters. The best way to explain the difference is by using the official definitions of a code linter and Oxidize. The definition provided by Darwin is that a code linter is a particular program which checks/flags suspicious behaviour or undesirable style in software, written in any language [43]. The definition of Oxidize is that it is a framework for idiomatic refactoring, which specialises in code analysis and code refactoring into its idiomatic form. The main difference is the detection, by a linter, and transformation, by Oxidize. This topic came up during the SATToSE[1] conference where we have presented Oxidize.

The following literature which has proved useful for our research was the empirical study of Allamanis and Sutton [2] about mining idioms from source code. This study presents an empirical proof of idiomatic code presence across projects. Their project called HAGGIS was used to scan projects across GitHub to prove this claim. Their definition of an idiom and method of using ASTs to define a group of code as an idiom helps substantiate what our research is about. Their AST method is useful for their idiom detection and analysis is not applicable to our field of research. By using an AST we would have to interpret a big part of the analysed source code by ourselves which would contradict our goal of preservation. This is caused by our need to rebuild the analyzed source code into target code for compilation and usage of the transformed code. To achieve this we need to make use of CSTs instead of ASTs to preserve as much code and developer style as it is possible around the transformations. This research while not be directly applicable to our case has helped us with better understanding of the topic.

During our research we have found a similar research to ours by Cook about the application of a project called P# [44]. The P# project is a language to language compiler or otherwise called a translator similar to the Corrode project. In this research, P# already existed but was not yet compiling the idiomatic and human-readable code. The main difference with our research is the implementation of idiomatic code with the effect of it enhancing the readability and usability. Our research also enables new developers with the knowledge of imperative languages to enhance their skills with the refactoring that can be provided with the project.

A different approach to this topic can be seen with the GoRefactor project by Pavkin Vladimir [45]. This project was a suggestion by one of the audience members during the Oxidize presentation at the CompSys[2] conference. This is a general-purpose refactoring framework for the Go language. The main difference between Oxidize and GoRefactor is the focus on idiomacy and Oxidize having a lower threshold of complexity.

Our refactoring research started with the research of the general theory behind refactoring and its use cases. This brought us to Fowler and Beck [7] and the definitions of refactoring methods. The

[1]http://sattose.org/2017
[2]http://www.asci.tudelft.nl/pages/events.php?event_id=21

refactoring methods explained by Fowler and Beck are e.g. pull and push methods, but in our research, we are transforming the used statements into their idiomatic state in the context they are used in. The methods described in their work did help us indirectly with the definition of our refactoring and gaining the understand the theory behind the refactoring.

For the refactoring and transformation work, we are making use of syntax similar to that of the work researched by Tip et al. [32, 46]. This syntax is not new and is actually from an older research by Palsberg and Schwartzbach [47] and is described as a formalism for expressing subtype relationships between the types of declarations and expressions. The difference between Palsberg and Schwartzbach and Tip et al. is that the later research has used the type constraints for refactoring and not only formalising of the subtype relationship. In our research we make use of a similar notation in RASCAL's visiting transformations. Research by Tip et al. and Palsberg and Schwartzbach has helped us with a better understanding of the type constraint transformations and how they can be similarly declared in RASCAL.

Chapter 8

Conclusion

In this research we have presented three idiomatic transformations together with a grammar implementation for Rust Systems Programming Language in Rascal *Metaprogramming Language*. The grammar is based on the Rust community supported grammar created by Brian Leibig for Bison and Flex. The transformations have focused on migration from the C malloc memory management implementation in Rust to Rust's Ownership system implementation, idiomatic iterative statements transformations (‘`loop`‘, ‘`for`‘ and ‘`while`‘) and a NonZero construct implementation for compiler optimisation. These transformations were chosen based on the output from the Corrode project by Jamey Sharp and improve on common construct presented by the project.

Given our first assumptions and existing limitations, we provide evidence of Oxidize working as it is expected. This is done by keeping the overall implementation strategy simple to understand and modify to requirements. During the development of Oxidize we have focused on readability and required functionality for the correctness of transformations. This was made possible by the Rascal *Metaprogramming Language* and its syntax definition capabilities combined with the streamlined syntax tree visiting functionality.

During this research, we have answered our research questions which are described in Chapter 1. To answer our first research question we have focused on the following topics.

We have determined our relevant idioms for our transformations to be the iterative statements, Ownership system transformation and the static analysis of the null pointer checks. These idioms are relevant because of our involvement with the Corrode project and the possible integration into its toolchain. Those are also the idioms commonly seen in the Corrode generated code and can also be applied to code written by a developer.

The matching cases for the non-idiomatic code are depended on the construct they are focused on. In case of the iterative statements, we can expect patterns which target a specific statement with a specific body construct as can be seen in Figures 6.1 and 6.3. The Ownership system transformation is different in that it targets a whole function scope to then analyze its body for the potential transformation cases.

Our second research question is about the validity of our transformations and correctness of our output. This research question is answered with the use of the *Rust Language Server* (RLS) system and our analysis and in-between transformations. The RLS system provides us with the evidence of code correctness and compilable code. While our checks are required to process and make our transformation valid.

What makes Oxidize so special compared to its comparable projects is its goal and state in-between a refactoring tool, a code beautifier and a compiler optimisation. While other projects focus on their specific use case, we would like to keep our functionality open to the requirements of the user.

Chapter 9

Future Work

In this chapter we list potential future research options for further development of our work.

Cross-file checks

The current implementation of OXIDIZE does not keep account for cross file dependencies. The usage of a flow graph and a CST would be of great addition to OXIDIZE. With both, OXIDIZE could determine concrete flow graphs of the data flow and apply transformations to a full flow instead of a per file basis analysis. This could be of benefit to the Ownership transformation by modifying the variables only related to the flow in which a C malloc system would be applied. A data flow could also help with visualisation of the transformation and simplification.

Toml grammar and transformation

A RUST project can be compiled by the RUST compiler (RUSTC) or by RUST's package manager called CARGO. The RUST compiler will compile the given project as is by the compiler and CARGO will also download and compile the required dependencies. This is done through the use of a TOML file. An example of such file can be seen in Listing 9.1. This TOML file is a configuration file for the CARGO package manager. It can list information as the owner information, package/project information, library information and dependencies information.

```
1 [package]
2 name = "cvsrs"
3 version = "0.0.1"
4 [lib]
5 name = "cvsrs"
6 path = "lib.rs"
7 [dependencies]
8 mbox = "0.4.0"
```

Listing 9.1: Example of how a TOML looks like in the CVS project

This TOML is used by CARGO to determine what information is needed to be kept in consideration during the compilation. By creating a new RASCAL grammar for the TOML syntax we can modify the compiler settings to our needs. The TOML grammar research would involve the grammar implementation in RASCAL syntax and its cross TOML and RUST file dependencies analysis.

Complete NonZero implementation

The current NonZero implementation is not complete and could be one of the first future transformation to finish. This transformation can benefit a program in many ways, e.g. halving of memory cost with combination of the Option wrapper.

Implementation of the automated test execution

The current state of Oxidize requires involvement of the user in the automation process. This process could be reduced by automating the verification process of the transformation validation tests. This would involve the research of the Rust test-suite use and validation of the test-suite results pre- and post-transformation. This research also depends on the implementation of the toml integration.

Completing the Rust grammar implementation

The existing Rust grammar implementation in Rascal is not complete and needs to be extended with the missing parts. This point is also about the ambiguities existing in the grammar which need to be solved to create a better grammar specification.

Rust developer interviews

We claim that our transformation are idiomatic and increase the readability of the code but this claim should be validated with active Rust and possibly non-Rust developers. Interviews with developers should shows us how well our transformations are performing in the eyes of other developers.

Bibliography

[1] Abram Hindle et al. "On the naturalness of software". In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE. 2012, pp. 837–847.

[2] Miltiadis Allamanis and Charles Sutton. "Mining idioms from source code". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 472–483.

[3] *The roadmap*. Accessed on: 18-07-2017. 2017. URL: https://blog.rust-lang.org/2017/02/06/roadmap.html.

[4] *Rust*. Accessed on: 18-07-2017. 2017. URL: https://www.rust-lang.org/.

[5] *Type Annotation*. Accessed on: 18-07-2017. 2017. URL: https://doc.rust-lang.org/book/first-edition/variable-bindings.html#type-annotations.

[6] *The Heap*. Accessed on: 18-07-2017. 2017. URL: https://doc.rust-lang.org/book/first-edition/the-stack-and-the-heap.html#the-heap.

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

[8] *Rust 1.0*. Accessed on: 19-07-2017. 2017. URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html.

[9] *Corrode*. Accessed on: 21-07-2017. 2017. URL: https://github.com/jameysharp/corrode.

[10] *Jamey Sharp*. Accessed on: 19-07-2017. 2017. URL: https://github.com/jameysharp.

[11] Paul Klint, Tijs Van Der Storm, and Jurgen 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*. IEEE. 2009, pp. 168–177.

[12] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading, 2007. URL: http://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/#id4.

[13] *Rust Fearless Concurrency*. Accessed on: 20-07-2017. 2015. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

[14] *Meet Safe and Unsafe*. Accessed on: 28-09-2017. 2017. URL: https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html.

[15] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011. URL: http://www.tomdalling.com/blog/software-design/resource-acquisition-is-initialisation-raii-explained/.

[16] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

[17] *Cargo*. Accessed on: 18-07-2017. 2017. URL: https://github.com/rust-lang/cargo.

[18] *Tom's Obvious, Minimal Language*. Accessed on: 28-09-2017. 2017. URL: https://github.com/toml-lang/toml.

[19] *Loops*. Accessed on: 11-08-2017. 2016. URL: https://doc.rust-lang.org/1.6.0/book/loops.html.

[20] Andrea Asperti. “Light affine logic”. In: *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on.* IEEE. 1998, pp. 300–308.

[21] *The Rust Programming Language*. Accessed on: 13-06-2017. 2017. URL: https://doc.rust-lang.org/book/second-edition/.

[22] *Rust Language Server (RLS)*. Accessed on: 7-08-2017. 2017. URL: https://github.com/rust-lang-nursery/rls.

[23] *Rust Code Completion utility (Racer)*. Accessed on: 7-08-2017. 2017. URL: https://github.com/racer-rust/racer.

[24] *The Rust Programming Language*. Accessed on: 13-06-2017. 2015. URL: https://doc.rust-lang.org/book/first-edition/.

[25] *What Is Ownership?* Accessed on: 28-09-2017. 2017. URL: https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html.

[26] *Rust Means Never Having to Close a Socket*. Accessed on: 28-09-2017. 2017. URL: http://blog.skylight.io/rust-means-never-having-to-close-a-socket/.

[27] Martin Fowler. *Refactoring*. Accessed on: 01-06-2017. 2017. URL: https://www.refactoring.com/.

[28] Mark GJ van den Brand et al. “The ASF+ SDF meta-environment: A component-based language development environment”. In: *Electronic Notes in Theoretical Computer Science* 44.2 (2001), pp. 3–8.

[29] Terence J. Parr and Russell W. Quong. “ANTLR: A predicated-LL (k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[30] Gogul Balakrishnan et al. “CodeSurfer/x86-A platform for analyzing x86 executables”. In: *Compiler Construction*. Springer. 2005, pp. 139–139.

[31] J.J. Vinju. “Analysis and Transformation of Source Code by Parsing and Rewriting”. PhD thesis. Universiteit van Amsterdam, Nov. 2005.

[32] Frank Tip et al. “Refactoring using type constraints”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.3 (2011), p. 9.

[33] *Brian Leibig*. Accessed on: 31-07-2017. 2017. URL: https://github.com/bleibig.

[34] *Rust*. Accessed on: 31-07-2017. 2016. URL: https://github.com/rust-lang/rust/tree/master/src/grammar.

[35] *Rust*. Accessed on: 2-08-2017. 2016. URL: https://github.com/rust-lang/rust/blob/6ebbe0ef50667af7a6ddb5ba674a6ab5e0827be1/src/grammar/parser-lalr.y.

[36] *Rascal Visit*. Accessed on: 23-06-2017. URL: http://tutor.rascal-mpl.org/Rascal/Expressions/Visit/Visit.html.

[37] *Rust FFI*. Accessed on: 23-08-2017. 2017. URL: https://doc.rust-lang.org/book/first-edition/ffi.html.

[38] *Primitive Pointer*. Accessed on: 14-08-2017. 2016. URL: https://doc.rust-lang.org/std/primitive.pointer.html.

[39] *mbox*. Accessed on: 14-08-2017. 2016. URL: https://docs.rs/mbox/0.4.0/mbox/.

[40] *Option*. Accessed on: 14-08-2017. 2016. URL: https://doc.rust-lang.org/std/option/.

[41] *NonZero*. Accessed on: 14-08-2017. 2016. URL: https://doc.rust-lang.org/core/nonzero/struct.NonZero.html.

[42] *NonZero RFC*. Accessed on: 14-08-2017. 2016. URL: https://github.com/rust-lang/rust/issues/27730.

[43] Ian F Darwin. *Checking C Programs with lint.* ” O’Reilly Media, Inc.”, 1991.

[44] Jonathan J Cook. “Optimizing P#: Translating Prolog to more idiomatic C#”. In: *Proceedings of CICLOPS 2004* (2004), pp. 59–70.

[45] *GoRefactor*. Accessed on: 31-07-2017. 2016. URL: https://github.com/vpavkin/GoRefactor.

[46] Frank Tip, Adam Kiezun, and Dirk Bäumer. "Refactoring for generalization using type constraints". In: *ACM SIGPLAN Notices*. Vol. 38. 11. ACM. 2003, pp. 13–26.

[47] Jens Palsberg and Michael I Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., 1994.

[48] *Rascal Start Manual*. Accessed on: 4-08-2017. 2017. URL: http://www.rascal-mpl.org/start/.

[49] *Oxidize on Github*. Accessed on: 4-08-2017. 2017. URL: https://github.com/zborowa/oxidize.

[50] *Rascal Unstable Build*. Accessed on: 4-08-2017. 2017. URL: https://update.rascal-mpl.org/console/rascal-shell-unstable.jar.

Appendix A

Usage

OXIDIZE has been fully created with RASCAL, which in turn makes use of the *Java Virtual Machine* (JVM) run-time based system. This enables us to make use of OXIDIZE in two distinct ways, namely with the help of its development environment, Eclipse, and also through a CLI with the help of *Java Runtime Environment* (JRE) and *Java Development Kit* (JDK).

To run OXIDIZE with the help of Eclipse we require the following software and their corresponding versions to run on the user's system:

- Eclipse for RCP Developers $\geq$ Neon.2
  - RASCAL $\geq$ 0.8.4.201706151132
    - JDK $\geq$ 1.8
    - JRE $\geq$ 1.8

The specific installation manual for the Rascal development environment in Eclipse can be found on the official Rascal website[48]. If OXIDIZE is integrated into a toolchain or used through a CLI it is required to run the following software on the user's system:

- RASCAL $\geq$ 0.8.4.201706151132
  - JDK $\geq$ 1.8
  - JRE $\geq$ 1.8

After verifying that the user's system features the pre-requisites we can start working with OXIDIZE. Using Eclipse to run OXIDIZE requires us first to import the project into our workspace. This can be done by doing the following:

1. Download the Oxidize project[49]
2. Start Eclipse
3. Click on *File*
   (a) Click on *Open Projects from File System...*
   (b) Import the project through the *Directory...*
   (c) Complete the steps through the wizard

Now that we have our project imported in the workspace of Eclipse we can start interacting with the project by following these steps:

1. Open the *Oxidize.rsc* through the *Rascal Navigator*
2. Right click in the editor and click on the *Start Console* button

3. Now that we have a new console in the *Terminal* tab we can import the *Demo* module
   - This can be done by typing in the console the following: '`import Demo;`' (this module extends all the required modules to run the project)
4. This gives use the ability of running the project like follow: '`Oxidize(|<location>|, [options])`';
   - Replace the '`<location>`' with the corresponding location of your to idiomatize project. Don't forget to have the vertical bars ('`|`') around your location. That is a location type within RASCAL, just like how a text enclosed by double quotes ('`"`') is a string.
   - We can also pass options to the function by adding them just like normal parameters of a function (at this time only one option exists):
     - '`verbose=true`': returns additional information in the terminal for the user to better understand what is happening
5. Hitting *Enter* on your keyboard will run the function and should not take long to complete

By following the steps above, Oxidize should create a new directory next to the directory in the given location with the addition of '`_idiom`' in the name. This directory will be on the same level as the directory given in the location parameter. This result is the same as when OXIDIZE would be used through a CLI. To interact with OXIDIZE through a CLI we need to follow these steps:

1. Download the Oxidize project[49]
2. Download the runnable .jar file from the RASCAL website[50]
   - This project makes use of the .jar file created by the unstable branch of RASCAL (this may change in the future)
3. Put the RASCAL .jar file into the root of the Oxidize project
   - Where the *Oxidize.rsc* file resides
4. Run the project like follow: '`java -Xmx1G -Xss32m -jar <rascal-version>.jar Oxidize.rsc [-v] <location>`'
   - The java '`Xmx`' (memory allocation pool) and '`Xss`' (stack size) can differ with each use-case
   - Replace the '`<rascal-version>`' with the corresponding name of the .jar file
   - Adding '`-v`' after the *Oxidize.rsc* will return additional information in the terminal for the user to better understand what is happening
   - Replace the '`<location>`' with the location of your to idiomatize project. In comparison to the Eclipse implementation, the location needs to be a string parameter (the string quotes are not needed)
5. Hitting *Enter* on your keyboard will run the function and should not take long to complete

Depended on the environment in which OXIDIZE is used, Eclipse or CLI, the usage of our research is different but the output is the same. In Figure A.1 on the left-hand side, we can see the output provided by the Eclipse terminal and also in Figure A.1 on the right-hand side, we can see the output provided by a CLI on a macOS. Both parsings are executed with the '`verbose`' parameter and are run on a translated version of the CVS project provided by the CORRODE project.

The differences between both methods lie in the execution type of RASCAL. The Eclipse implementation of RASCAL is executed with the help of the Rascal compiler while the CLI version makes use of the Rascal interpreter. This while not differentiating in the functionality differs slightly in the output. We can see the differences in the command execution, the interpreter also returning the version of itself, the execution time is quicker with the compiler, and also compiler returning an '`ok`' confirmation at the end of the command execution.

```
1  rascal>Oxidize(|file://.../cvs-rs/|,
2                 verbose=true);
3  Example of 7 ambiguous files:
4  [
5    |file://.../cvs-rs/commit.rs|,
6    |file://.../cvs-rs/entries.rs|,
7    |file://.../cvs-rs/fileattr.rs|,
8    |file://.../cvs-rs/hash.rs|,
9    |file://.../cwi/cvs-rs/log.rs|,
10   |file://.../cwi/cvs-rs/vers_ts.rs|,
11   |file://.../cwi/cvs-rs/wrapper.rs|
12 ]
13 Total files:   14
14 Parsed:        14
15 Failed:        0
16 Amb:           7
17 Oxidize completed after:
18                18s 541ms
19 ok
```

```
1  $ java -Xmx1G -Xss32m -jar rascal.jar
2    Oxidize.rsc -v .../cwi/cvs-rs
3  Version: unknown
4  Example of 7 ambiguous files:
5  [
6    |file://.../cvs-rs/commit.rs|,
7    |file://.../cvs-rs/entries.rs|,
8    |file://.../cvs-rs/fileattr.rs|,
9    |file://.../cvs-rs/hash.rs|,
10   |file://.../cwi/cvs-rs/log.rs|,
11   |file://.../cwi/cvs-rs/vers_ts.rs|,
12   |file://.../cwi/cvs-rs/wrapper.rs|
13 ]
14 Total files:   14
15 Parsed:        14
16 Failed:        0
17 Amb:           7
18 Oxidize completed after:
19     19s 742ms
```

Figure A.1: Output from Eclipse (on the left) and also CLI (on the right)

The remaining output is the actual information to the user about the progress of the parsing process. The first information that is provided is a list of files which contain ambiguity. This enables the user to research the files in question and look into the ambiguity of the files grammar. The following information shows the user how many Rust files (.rs) exist within the provided location, how many files have been parsed by Oxidize, how many files have failed to parse by Oxidize and how many files contain ambiguity.