Using Rust as a Complement to C for Embedded Systems Software Development

(A Study Performed Porting a Linux Daemon)

Karl Rikte
karl.rikte@gmail.com

January 30, 2018

Master's thesis work carried out at Axis Communications AB.

Supervisors:
Christoffer Cronström, christoffer.cronstrom@axis.com
Mathias Bruce, mathias.bruce@axis.com
Gustav Cedersjö, gustav.cedersjo@cs.lth.se

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

Rust aims to bring safety to low-level programming by using zero-cost abstractions. These provide, among other things, guaranteed memory safety and threading without data races.

Garbage collected languages have become popular to guarantee safety, but in performance critical, memory limited or real time applications, it is not an ideal solution. Rust is safe and still has manual memory management, with strict rules.

This report presents a case study of using the Rust language and associated tooling such as debuggers and IDEs in practise. The study was carried out by porting 5000+ lines of an embedded Linux daemon to Rust.

Rust upholds the safety and zero-cost claims. Using Rust has been found to aid in achieving an improved, shorter, more expressive architecture. The learning curve is a bit steep, but productivity has been found to be high once learned. Tooling support is mature, but IDEs are not yet full featured.

Keywords: Rust, language review, memory safety, smart pointers, productivity, cross-compilation

Acknowledgements

First and foremost, this case study owes its existence to Axis Communications AB. Axis is willing to spend time and resources on investigating potential benefits and drawbacks of modern languages, for possible use and deployment in the long term.

Special thanks go to the supervisors at Axis, Christoffer Cronström and Mathias Bruce, who have provided countless hours of assistance and advice, primarily with Axis-specific hardware, software and build systems, but also with valuable input and feedback about methodology and report writing.

At LTH, supervisor Gustav Cedersjö helped with formalizing goals and methodology to get the work started and has helped a lot with finalizing the report, and for this a warm thank you is in order.

Finally, the CSN (Swedish Central Studies Authority, freely translated), receives a honourable mention for funding Swedish students during their time in university.

Contents

1 Introduction **9**
1.1 Outline . . . 10
1.2 Background . . . 10
1.2.1 Reviewing a programming language . . . 11
1.2.2 The Rust Programming Language . . . 12
1.3 Evaluation goals . . . 15
1.4 Problem definition . . . 16
1.4.1 The Rust Programming Language . . . 16
1.4.2 Building and Compiling . . . 16
1.4.3 Porting . . . 16
1.4.4 Development Tools . . . 17
1.4.5 Quality factors . . . 17
1.5 Previous research . . . 18
1.6 Distribution of work . . . 18

2 Approach **19**
2.1 The Rust Programming Language . . . 19
2.1.1 Basic Rust syntax . . . 19
2.1.2 Memory safety . . . 22
2.1.3 Slices . . . 29
2.1.4 Strings . . . 29
2.1.5 Static lifetimes and elision . . . 30
2.1.6 Basic struct and enum . . . 31
2.1.7 Tuples . . . 31
2.1.8 Instance methods . . . 32
2.1.9 Optional data and error handling . . . 33
2.1.10 Error handling . . . 35
2.1.11 Resource management . . . 35
2.1.12 Iterators, closures and functional programming . . . 35
2.1.13 Maps and Sets . . . 36

2.1.14 A longer Rust example 36
2.1.15 Advanced concepts 36
2.1.16 Concurrency and multi-threading 38
2.1.17 Dependency management and code structuring 40
2.1.18 Language development and backwards compatibility 40
2.1.19 Legal 41
2.1.20 Performance and Memory utilization 41
2.1.21 Popularity, adoption and the future 42
2.2 Building and Compiling 43
2.2.1 Supported platforms 43
2.2.2 Adding support for a platform 44
2.2.3 Stability 44
2.2.4 Adaptation of Makefile based build system 44
2.2.5 Cross compilation 44
2.2.6 Building 45
2.2.7 C integration 45
2.3 Development Tools 45
2.3.1 Testing 46
2.3.2 Benchmarking 46
2.3.3 Profiling 46
2.3.4 Finding memory leaks 46
2.3.5 Debugging 47
2.3.6 IDEs 47
2.4 Porting C code to Rust 50
2.4.1 Physical Access Control System (PACS) 50
2.4.2 PACS software stack 51
2.4.3 Choosing parts to port 51
2.4.4 Understanding the code 52
2.4.5 Porting approaches 53
2.4.6 Testing 54
2.4.7 Definitions of C functions and structs 54
2.4.8 Callbacks 55
2.4.9 Glib 55
2.4.10 Wrapping C structs 56
2.4.11 D-Bus 57
2.4.12 Code size reduction 57
2.4.13 Binary size 57
2.4.14 Memory consumption 57
2.4.15 Performance 57

3 Discussion **61**
3.1 The Rust Programming Language 61
3.1.1 Type system 61
3.1.2 Data structures 61
3.1.3 Optionals 61
3.1.4 Error handling 62

3.1.5 Object oriented programming 63
3.1.6 Safety 63
3.1.7 String types 63
3.1.8 The deref trait 64
3.1.9 The borrow checker 64
3.1.10 Documentation 66
3.2 Quality factors 66
3.2.1 Learning Rust 66
3.2.2 Productivity 67
3.3 Porting 68
3.3.1 Porting methods 68
3.3.2 Global state mutex and callbacks 69
3.3.3 Expiry timers 69
3.3.4 Replacing loops with iterator functions 70
3.3.5 Separation of operations 70
3.3.6 Separation of input, output and state 70
3.3.7 Set operations 71
3.3.8 Replacing null pointers with enums 71
3.3.9 Resolving segmentation faults 71
3.3.10 Resolving deadlocks 72
3.4 Building and Compiling 72
3.4.1 Continued support for custom target 72
3.4.2 Using the compiler 72
3.4.3 Linking with a Makefile based build system 72

4 Conclusions **73**
4.1 Summary 73
4.2 Future research 75

Appendix A A longer Rust example **85**

Appendix B Changes made to Rust compiler to support target platform **89**
B.1 Target definition Rust code 90

Appendix C Makefile integration **91**

Appendix D List of Changes **93**

Chapter 1
Introduction

In the last few years many major corporations have started releasing new programming languages. These include Google's Go [23], Apple's Swift [3] and the community developed Rust [51] [1], sponsored by Mozilla.

Why do we need all these new languages? To understand, we have to look back in time. The C language [5] was intended as a procedural language for programming operating systems, and was designed with designed with the following goals in mind: direct access to memory, a small library, efficient execution and reasonable portability (far better than assembly).

The C [2] language was never intended to be perfectly portable or safe, and C++ which extends C, still allows potentially dangerous C code. It is possible to write very short and cryptic code in C and C++. There are even such competitions [27]. The combination of short cryptic and potentially *unsafe* makes for a dangerous cocktail if used incorrectly.

Some features are just hard or impossible to add, remove or change without starting from scratch. Examples of what the emerging languages are trying to provide include: safer memory management, fragmentation reduction, easier dependency management, cross platform concurrency and cross platform types.

Many studies have shown that between 5 and 30 percent of the defects can be caught early on in the development cycle using static analysis [26]. The Rust compiler does very strong static analysis to guarantee type safety [51]. Many white papers and articles have concluded that the later bugs are fixed, the more expensive it gets, and the cost rises super-linearly [38]. This alone makes Rust worth considering.

In terms of bug severity, here is a quote [37] from a Mozilla report:

> In Gecko, roughly 50% of the security critical bugs are memory use after free, array out of range access, or related to integer overflow, all mistakes

[1] The Rust programming language with associated tools may henceforth be referred to as simply *Rust*.

[2] Before standardization, the C language used is commonly referred to as K&R C. The first standard was C89, followed by ANSI-C99, and currently C11 (at the time of writing).

commonly made by even experienced C++ programmers with access to the best static analysis tools available.

Thus, there is a real potential for Rust to reduce the bug count, and at an early stage in development. Since security-related bugs can harm company reputations, especially in the security sector, Rust seems like a logical choice when both security and low-level are requirements.

1.1 Outline

First, some background and an introduction to Rust will be given, secondly the evaluation goals will be stated in section 1.3, finally the problem formulation in section 1.4 will break the main effort/benefit question down into sub-questions.

Rust design decisions and syntax, and the approach used for the rest of the evaluation will be discussed in chapter 2. The findings from the approach will be discussed in chapter 3, presenting answers to the questions. Finally the conclusions will be summarized, in chapter 4.

1.2 Background

At Axis Communications, the primary language used in embedded products is C, because of the speed and the small runtime environment. However, fixing memory and threading related bugs is sometimes considered to cost too much. The code is thoroughly tested for memory and threading related issues, which takes time and costs money. Therefore, languages that promise to reduce the number of memory and threading related bugs have been considered for adoption. However, no tested candidate language has yet promised to provide enough benefits compared to the cost and complexity of a switch.

The Go programming language was evaluated in 2015, but cross compiling was a problem, a segmented stack of the language made it hard to debug, and Go has mandatory garbage collection (which implies it can be more memory intensive, and sometimes might pause). Calling Go to/from C was not trivial either [16].

Rust on the other hand is much closer to C in terms of being very low-level, but limits the allowed pointer operations and thread data sharing to a subset of what C allows. This is similar to MISRA C[3], the difference is that Rust enforces the rules out of the box. Rust does this through the use of language constructs that guarantee safe memory management without the use of a garbage collector, and freedom from data races. Rust also provides an extensive standard library with lots of convenient functionality such as iterators, collections, I/O and easy functional programming among other things.

The real question is whether it is beneficial and worth the effort to rewrite existing code in Rust or perhaps code new projects in Rust.

To try to answer this question, a Linux daemon for an embedded product was mostly re-implemented in Rust in an incremental fashion. The daemon chosen consisted of more

[3] Strict rules for writing safe C code for critical systems, used in automobile industry.

than ten thousand lines of C code, of which more than half was ported. Not only the language itself was reviewed, but rather the entire development process including building a cross compiler, using the compiler, build system integration, linking, package management, debugging, IDEs, etc. This report shares the findings which resulted from this case study.

1.2.1 Reviewing a programming language

When reviewing a programming language, one should keep in mind that it will always be possible to write bad programs. The following quote by Lawrence Flon from 1975 [32] illustrates this:

> There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.

This report will focus on how Rust can aid developers in using good programming practises. *unsafe* blocks in Rust are allowed to do anything, proving that the point made by Lawrence Flon holds true also for Rust. However, there is another saying, with unknown origin that has been used by many throughout history:

> With great power comes great responsibility.

This could be considered to apply to the *unsafe*-keyword. It should not be used for optimization unless absolutely necessary. Most of the time the standard library will provide ample alternatives, leaving the safety responsibility to the Rust developers. Another famous quote by Donald E. Knuth is in order [14]:

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

That said, in some sense every programming language is functionally equivalent to any other language in terms of being Turing compatible. This way of reasoning is commonly referred to as the Turing tarpit [1]:

> All computing languages or computers can compute anything in theory but nothing of practical interest is easy.

What can be read between the lines is that while any language can be used to compute anything, it can be incredibly difficult to do so, depending on how the language is designed. The Brainfuck [62] language illustrates this by implementing a bare minimum functionality to qualify as Turing compatible, and doing so with a very cryptic look.

The way out of the Turing tarpit is to choose a suitable language, or to design a language which is suitable to a particular task [41].

1.2.2 The Rust Programming Language

Here, the relevant key decisions made by the Rust developers when designing the Rust programming language will be discussed. This is necessary in order to fully understand the implications it has as to how code written in Rust is structured. Later, in chapter 2, some key concepts such as ownership, mutability, borrowing and lifetimes will also be introduced. Their implications will be discussed in section 3.

Design goals

The design goals of the Rust programming language are stated as follows, on the Rust FAQ [49]:

> To design and implement a safe, concurrent, practical systems language.
>
> Rust exists because other languages at this level of abstraction and efficiency are unsatisfactory. In particular:
>
> 1. There is too little attention paid to safety.
> 2. They have poor concurrency support.
> 3. There is a lack of practical affordances.
> 4. They offer limited control over resources.
>
> Rust exists as an alternative that provides both efficient code and a comfortable level of abstraction, while improving on all four of these points.

With these goals in mind, the Rust language was designed. Rust is a safe systems programming language, where the code compiles to predictable assembly. Rust does not make use of a garbage collector. Memory is managed manually, and the compiler guarantees that there are no memory related errors such as use after free, double free, etc. Compiling to predictable assembly and not having garbage collection are important properties of any systems programming language. The reason for that is, in languages that do have garbage collection, or that do not map closely to assembly, a simple statement without any function call could lead to unbounded amounts of work being performed. This is not ideal in, for example, operating system functions, interrupt handlers and real time applications.

C is the most common systems programming language in use today. One may argue that the main design goal behind C was to be a portable assembler, because C deals directly with addresses, words, halfwords etc. Safety was never a design goal of C, and a lot of guarantees are left up to the programmer. The vast majority of the code of the Linux kernel is written in C [33]. Unfortunately, safety is a key property that is desirable in a kernel.

Today we see that a lot of security critical bugs are due to buffer overflows, double free, unfortunate race conditions etc [37]. What do all these errors have in common? They are all related to memory or race conditions where the same memory is accessed concurrently with improper locking causing nondeterministic behavior. They are also caused by people overlooking rare corner cases, and not the fault of the language. People are notoriously bad at considering all corner cases, and languages that do cover corner cases tend to be higher

level languages. Rust was created to remedy this, and aims to marry safe and low-level, and do so obeying the zero overhead principle.

Bjarne Stroustrup explains the zero overhead principle in his paper on C++ as follows [6]:

> In general, C++ implementations obey the zero-overhead principle:
> What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better

Undefined behavior

A program written in C and/or C++ may exhibit what is known as *undefined behavior*. It is the programmers responsibility to make sure this cannot occur. A program that does not under any circumstances produce undefined behavior is called *well defined*. A Rust program is always well defined unless the *unsafe* construct is used, more on this later. In C99 (latest version is C99 + TC1 + TC2 [29] + TC3), undefined behaviour is defined as:

> 3.4.3
> undefined behavior
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

In practice this could mean anything, including but not limited to:

- Have no effect
- Work as expected by the programmer
- Do anything
- Crash immediately
- Continue and crash at a later time (any time)
- Run and produce incorrect result
- Overwrite a variable that should not have changed value
- Change the meaning of later well defined constructs such as return location
- Do different things on different compiler/optimization level/OS/hardware etc.
- Varying behavior depending on input data

Unfortunately, that last item is a major cause for concern, as it opens up a wide array of possibilities for exploiting code exhibiting undefined behavior, as an attacker could carefully design input data to have the program do what he or she wishes.

Type safety

Type safety is a property that a programming language has if everything that can be expressed using the language is well defined. Type safety guarantees the inability to misinterpret the type of a variable. Type safety includes memory safety, which is the inability to copy bytes from a memory location with one type to a memory location with a different type, etc.

Most programming languages are type safe, for example Python, Java and JavaScript. The only languages that are not type safe which are in wide use today are C and C++. The reason why type safety (including memory safety) is not a property of the C family of languages is that type safety in many languages comes at a cost. Python, Java and JavaScript all have garbage collection. So does C# and many other type safe languages.

Garbage collection can frequently be a source of unpredictable pausing during execution, which makes garbage collected languages unsuitable for use in low-level code.

Another reason why those languages are not in wide use in kernels and drivers is that to control hardware, writing to arbitrary registers and memory locations as well as running certain instructions may be required.

Rust is a systems programming language that *is* type safe during normal use, but also provides *unsafe* blocks to allow the programming of operating systems etc. Code inside *unsafe* blocks is allowed to do anything, including writing to arbitrary memory locations or run inline assembly code. This can lead to undefined behavior, if done incorrectly. Restrictive usage of *unsafe* blocks can limit the amount of code that needs careful manual analysis to a minimum, and all other code will be portable.

Memory management

Rust enforces Resource Acquisition Is Initialization [11] (RAII) also known as Scope-Bound Resource Management (SBRM). The reason why there are two names, is that the former name has been criticized by Bjarne himself.

The concept basically means to release all acquired resources in the opposite order of their acquisition, done automatically when a function or method returns. This should happen regardless of if any errors occurred.

C++ has provided this for quite some time, but it requires the resources to be stack allocated and have appropriate destructors [7].

Keywords

The Rust language has 36 keywords [50], however, among those are ”true”, ”false” and ”loop” (short for ”while(true)”), so it is perhaps more fair to regard it as 33, the reader will be the judge. C89 [10] in comparison has 32 (and does not include those mentioned previously).

It should also be mentioned that as of this writing, the keyword ”box” is unstable (but usable).

There are also 16 reserved words, that are either old keywords from before the first stable version of Rust, or reserved for potential future features.

Platform definitions

A *host platform* is a platform used to run a compiler. A *build platform* is a platform used to build the compiler itself. The platform which programs produced by the compiler will run on is called the *target platform*.

1.3 Evaluation goals

The overall goal of this project is to evaluate Rust in the context of Axis' primary use case: programming Linux daemons running on MIPS hardware.

Evaluating a programming language from every possible angle is no easy task, and this report is by no means a full evaluation of the Rust language. General performance and memory footprint evaluations of Rust have been conducted previously [12]. This evaluation focuses on using Rust in practise, integrating Rust with existing C code and build systems.

The primary reasons to evaluate Rust in the first place are the security and freedom from data races promised by the language. Another reason is the large standard library, and closely integrated build system. However, there could be downsides to using the language as well.

There are lots of potentially interesting topics to address. Perhaps the most interesting is language design and implications such as which code patterns and paradigms translate well into Rust. Lots of possible approaches exist to porting code. Some ways may be more suitable than others. Possibly, conclusions could be drawn with regards to porting methodology.

Linking and build systems integration is another area of interest. If a language presents issues with linking and/or build systems, using it in production may result in release delays etc. Axis makes extensive use of Glib [20] and D-Bus [18], and therefore interoperability with those is especially important.

The learning curve of a new language is important to understand in order to assess the cost associated with starting using the language.

Even if Rust makes promises about memory safety, the entire software stack will likely never be written in Rust. Therefore memory issues may arise despite using Rust, like when linking other libraries or using *unsafe* blocks. Debugging will always remain important, both for analysing memory issues and finding logic errors. Debugging Rust, as well as a mixed C and Rust code base is therefore an area of interest.

Running on an embedded system imposes requirements on resource usage, both in terms of binary size, memory footprint and CPU usage. These areas are naturally of interest. While measuring this in detail is not a goal of this report, some basic measurements will be performed.

Unit testing, compilation time, documentation and IDE support are also areas of interest, and will be covered briefly.

Finally, there is always the unknown factor to consider. Perhaps some unforeseen problem may occur, or some benefit other than the expected memory and thread safety may be achieved.

1.4 Problem definition

To fulfil the goals of investigating the previously mentioned areas of interest, the goal questions in the following subsections were formulated. Some questions could potentially fit in several categories. Some questions are also closely related.

1.4.1 The Rust Programming Language

- Is the language well structured, and do the language features interact in a logical way?
- Does the language structure aid in writing correct programs? Does it prevent common errors made when using other languages?
- Is the standard library full featured, mature and suitable for use in a production environment? Is it well documented, and is the documentation easy to navigate and understand?
- How is the language maintained? Who has control over the language and is it possible to influence the future development of the language?
- About the legal aspect, does Rust come with a permissive license?

1.4.2 Building and Compiling

- Does the compiler provide helpful error messages?
- Is the Rust build system, Cargo, easy to use?
- Is it difficult to add support for a target platform (supported by LLVM) to the Rust compiler?
- Is cross compiling difficult?
- Will adapting a Makefile based build system to include Rust pose any problems?
- Does adding a Rust compilation and linking step to the build process slow it down significantly?
- Is it easy calling Rust from C and C from Rust? Does it require writing glue code? Are callbacks handled well between C and Rust?

1.4.3 Porting

- Is incremental translation of C code to Rust feasible?
- How easy is it to interface with sockets, D-Bus and Glib?

- Are there paradigms used in the reference C code, that are hard to translate into Rust, without major restructuring?
- Can Rust aid in reducing copying, when Rust provides the guarantees for pointer safety?

1.4.4 Development Tools

- How does linking C with Rust affect the debugging of code with GDB?
- Will Valgrind [63] produce false positives when debugging Rust code?
- Are there any easy to use profiling tools for Rust?
- What is the state of IDEs for Rust?

1.4.5 Quality factors

ISO/IEC 25010:2011 [28] defines eight major categories of quality factors: Functional sustainability, Reliability, Performance efficiency, Operability, Security, Compatibility, Maintainability & Transferability. There are a total of 31 subcategories. This thesis does not aim to be a complete evaluation according to ISO/IEC 25010:2011. Quality factors are of a naturally subjective nature and thus a large survey with many participants would be required to be able to give statistically significant answers. This is out of the scope of this master's thesis. However, based on the experience attained by using the language, the author will attempt to answer the following questions as fairly as possible. Some of these questions may be answered by referring to other sources, possibly sources using other architectures.

- Is Rust code maintainable? Will functionality implemented in one version of Rust still work with future versions?
- How are errors/problems/exceptions handled in Rust? Is it possible to recover from a fault? Are unforeseen crashes handled well?
- How does the performance of Rust compare to C/C++? Does Rust use significantly more resources than C/C++? How is binary size affected by implementing parts of a C program in Rust?
- Is code written in idiomatic Rust secure?
- Is Rust compatible with libraries written in C/C++?
- Is code written in Rust portable?
- Is Rust code easily testable? Does the language provide a standard way to test code?
- Does the language have a steep learning curve? How is productivity affected once learned? Coming from C or other high level languages, is it easy to understand Rust code?

1.5 Previous research

The Go evaluation, performed before this study, stated as a goal to be reference for how to perform programming language evaluations in the future [16]. In order for this report to be easily comparable to the Go report, it follows a somewhat similar structure. While Go certainly showed promise, it turned out to not be suitable for Axis' use case, as described in section 1.2 and detailed in [16].

This report will be somewhat more oriented around porting, and how the language affects the software-architectural changes imposed. Naturally, since one of the main selling points of Rust is memory safety, this will also be covered in greater detail.

1.6 Distribution of work

The evaluation has been conducted by the author of this report.

Chapter 2
Approach

This chapter will describe the approach used to evaluate the different subsections of the problem definition. In the case of the Rust Programming Language, an introduction to Rust will be presented. Concerning building and compiling, the approach to building a Rust cross compiler for the specific target will be described, and the build system integration approach will also be presented. In the case of porting, the porting approaches tested will be presented. Regarding tooling, the tooling evaluation approach will be detailed.

2.1 The Rust Programming Language

A subset of the Rust programming language (the most central aspects) is presented in this section. It is somewhat simplified, and by no means a full language specification. It serves as a quick introduction intended for programmers familiar with the C language. Full understanding of all the concepts and examples is not required to understand the discussion chapter. The examples are given so that the reader can form his or her own opinion about the language syntax. Rust has support for object oriented programming, but it is not mandatory to write object oriented programs. Some of the examples will be somewhat object oriented.

2.1.1 Basic Rust syntax

The Rust language is designed to look familiar to programmers with experience from today's most popular languages. A minimal Rust program may look like the following:

```
fn main() {
  let x : i32 = 42;
  println!("The answer to the Ultimate Question is: {}", x);
}
```

Some things to note:

- Functions are defined using the *fn* keyword.
- Variables are introduced with the *let* keyword.
- Types of variables are defined using the *: T* syntax, where *T* is a type, in this case signed 32 bit integer.
- To print, the *println!* macro is invoked, which works similarly to *printf* in C. Note the *!* which differentiates between macro invocation and function call. The *{}* will be replaced by the value of x. The *println!* macro will generate efficient code for each invocation as format string processing can be done at compile time. Another reason for printing to use a macro is that in Rust, macros can be variadic while functions cannot.[1]

To introduce function calls and control structures, consider the following simple example:

```
fn main() {
  let a = 42;
  let b = 13;
  let m = max(a, b);
  assert!(m==a);
}
fn max(x: i32, y: i32) -> i32 {
  if (x > y) {
    return x;
  }
  else {
    return y;
  }
}
```

This code should seem familiar to any programmer. The following is worth noting:

- The max function does not need to be declared before use.
- The types for the let bindings of a, b and m do not need explicit types, as the Rust compiler can infer the types unambiguously from the fact they are used to call the max function.
- Functions must have parameter and return types explicitly declared.

In Rust, any code block is an expression. The max function could have been written as:

[1] Yet another reason is to avoid function calls taking ownership of printed value. Ownership will be discussed later.

```
fn max(x: i32, y: i32) -> i32 {
  if x > y {
    x
  }
  else {
    y
  }
}
```

Rust considers the above as better style than the first implementation of max. The simplified code above is legal in Rust because:

- The last expression of the function code block is an if expression, thus the (return) value of the function is the value of the if expression.
- The value of the if expression will always have type i32 regardless of whether the condition evaluates to true or false.
- Conditions do not need parentheses.

Do note the difference between y and z in the following code:

```
let x : i32 = 42;
let y = {
  x
};
let z = {
  x;
};
```

After running the above code, y will have type i32, and z will have type () meaning nothing, similar to the void type in C, because of the semicolon after x. After the semicolon is an empty expression, which is the last expression of the block. This is a common pitfall coming from other languages where each line ends with semicolon.

Rust provides the for and while loop control structures. The syntax is as follows:

```
for x in 0..10 {
  // statements
}
let mut n = 0;
while n < 10 {
  /* statements */
  n++;
}
```

The above code illustrates the following:

- Comments use familiar syntax.
- In the range of the for loop, the lower bound is inclusive and the upper bound is exclusive.

- The loops are equivalent except for the fact that n remains in scope after the while loop.
- The range in the for loop is an iterator. Iterators will be discussed later.
- The types of the variables need not be explicitly stated. The type will depend on how they are used, for example if they are passed to a function.
- The *mut* keyword indicates that we will need to reassign the value of n. Rust has immutability by default.

These are the very basics of the Rust language and can be used to write some simple programs. Before introducing more advanced concepts such as strings, data structures, etc. the memory management aspects of the language must be introduced, as data structures and non-static strings use heap-allocated data.

2.1.2 Memory safety

Rust accomplishes memory safety by using the concepts of *ownership*, *mutability*, *borrowing* and *lifetimes*. This section will attempt to explain what they mean without going into implementation details as far as possible. These rules apply mostly to types which allocate data on the heap [2] and thus cannot be trivially passed around without thinking about memory concerns.

Mutability

As we have already seen, Rust has immutability by default. This is good because it prevents accidentally overwriting previously declared and assigned variables by mistake. The following code is erroneous:

```
fn main() {
  let x = 4;
  x = 5;  //expected error here
}
```

Attempting to compile it will result in the following output:

```
error[E0384]: re-assignment of immutable variable `x`
 --> src/main.rs:3:3
  |
2 |   let x = 4;
  |       - first assignment to `x`
3 |   x = 5;  //expected error here
  |   ^^^^^ re-assignment of immutable variable
```

To fix it, x needs to be explicitly marked as mutable:

```
let mut x = 4;
```

As mentioned, immutability can prevent many classes of errors. Immutability also makes sharing of data which does not need to be modified between threads a lot easier in Rust. Immutability is closely related to const in C. More details will be given in subsection 2.1.2.

[2] Used to allocate memory at runtime when sizes cannot be predicted.

heap-allocated data and ownership

Ownership is best demonstrated using heap-allocated data. In Rust, allocating memory manually is not recommended, so a vector will be used in the example. Types will be explicitly annotated for the purpose of clarity. Consider the following code:

```
fn main() {
  let mut numbers : Vec<i32> = Vec::new();
  numbers.push(1);
  numbers.push(2);
}
```

- The static function new on the struct Vec is used to create a vector.
- Rust supports generics.
- The instance method push is called on the vector instance called `numbers`, adding 1 and 2 to the vector.

The same could be written with the *vec!* macro.

```
let numbers : Vec<i32> = vec![1, 2];
```

The most interesting point in the program is when the main function ends. The variable `numbers` owns the vector instance, meaning it has *ownership* of the vector. The vector `numbers` is a stack allocated vector, which has a pointer to the heap, containing the vector elements. When `numbers` goes out of scope, the drop method of the vector will be run deallocating the heap data, similarly to a destructor in C++ (which runs if the vector numbers is not itself a pointer, but exists on the stack) [11].

If the vector would be reassigned to a new variable, the language must define what happens with the heap-allocated data.

```
let mut numbers : Vec<i32> = vec![1, 2];
let newnumbers = numbers;
```

Solutions to this problem could be:

- Copy the entire vector, including allocating new space for the elements and copying the elements as well. This could lead to unbounded amounts of work caused by a single assignment, and there are many cases where it could be unnecessary. This is the approach taken by C++, calling a copy constructor, and if a copy is not desired one should use a pointer [61].
- Make both variables point to the same data. This is the approach taken by Java among others. The problem with this approach is that the data must not be deallocated until the last variable goes out of scope, which is monitored by the garbage collector.

- Invalidate the original variable, making it unusable after the ownership has been *moved* to the new variable. If a copy is needed, it can be done explicitly. If copying is too expensive, pointers with strict guarantees can be used. This is the approach taken by the Rust language. [3]

Attempting to use the original variable after ownership has been moved as follows:

```
fn main(){
  let mut numbers : Vec<i32> = vec![1, 2];
  let newnumbers = numbers;
  numbers.push(3);
}
```

causes the following error:

```
error[E0382]: use of moved value: `numbers`
 --> src/main.rs:4:3
  |
3 |   let newnumbers = numbers;
  |       ---------- value moved here
4 |   numbers.push(3);
  |   ^^^^^^^ value used here after move
  |
  = note: move occurs because `numbers` has type `std::vec::Vec<i32>`,
  ↪  which does not implement the `Copy` trait
```

The same happens if we pass a vector to a function, and then attempt to use it:

```
fn main(){
  let mut numbers = vec![1, 2];
  push3(numbers);   //now the type is known to be Vec<i32>
  numbers.push(4); //error here. ownership moved to v.
}
fn push3(mut v: Vec<i32>) {
  v.push(3);
}
```

What happens is that the vector is moved to the parameter v which takes ownership. The vector is deallocated when push3 finishes and v goes out of scope. This could be fixed by returning the vector back to the caller as such:

```
fn main(){
  let mut numbers = vec![1, 2];
  numbers = push3(numbers); //type inferred as Vec<i32>
  numbers.push(4);          //ok, we have ownership again
}
fn push3(mut v: Vec<i32>) -> Vec<i32> {
  v.push(3);
  return v;
}
```

[3] Loops in Rust work on iterators, and iterators may either consume the values form a collection, allowing moving values out of the collection, or iterate references. This depends on the type of iterator created.

At this point the reader might feel that this seems cumbersome, and it is, and one would normally use pointers for such code which will be introduced in the next section. However, it illustrates well how ownership is moved back and forth.

References and borrowing

Pointers exist in Rust, and work much like in C/C++ with one important distinction, and therefore they are instead referred to as *references*. The difference is that when a pointer exists, the owning variable must not be modified at the same time. Rust uses the & operator for ”address of” and the * operator for ”value pointed by”, just like C. The following is valid Rust code:

```
let x = 4;
let xp = &x; //references x
let y = *xp; //the value referenced by xp (4)
```

A regular reference must not be modified. Modification requires a mutable reference, and a mutable reference can only be created for mutable data:

```
let mut x = 4;    //x may be mutated
let xp = &mut x; //mutable reference to x
*xp = 5;          //set the value pointed by xp (x) to 5
```

Note the important difference between the following (assuming mutable x and mutable y exist):

```
let mut p1 = &x; // p1 can be reassigned to point elsewhere
// Note: p1 cannot be used to modify the value pointed
p1 = &y; //ok
*p1 = 5; //not ok
let p2 = &mut x; // what is pointed by p2 can be modified
// Note: p2 cannot be reassigned to point elsewhere
p2 = &y; //not ok
*p2 = 5; //ok
```

Borrowing can be used to pass a reference to a function but to keep the ownership of a vector:

```
fn main(){
  let mut numbers = vec![1, 2];
  push3(&mut numbers); //send temp mut reference
  numbers.push(4);     //ok, no references exist anymore
}
fn push3(v: &mut Vec<i32>) {
  *v.push(3); //push 3 to vector referred to by v
}
```

A mutable reference to numbers is created and sent to the function push3. At the end of push3 the pointer v goes out of scope. No pointers to numbers exist any more. Therefore it is ok to push 4 to numbers.

Rust borrowing rules

Now that mutability and borrowing has been introduced, there are three rules that govern how references may be used. Brace for impact, this is known to be the most difficult part of learning Rust as written in multiple sources [35, 31], and the author can identify with those claims. These rules are what makes memory safety and absence of data races possible:

1. Every value (such as a vector), has a single owning variable at any given time. Whenever the owning variable goes out of scope, the value is no longer accessible and any heap data associated will be freed. Releasing other resources such as file handles and sockets works in the same fashion.

2. References must never be able to outlive the data they refer to.

3. A value may only be modified if there is exclusive access to it.

At first glance, the above rules may seem intuitive, but being able to fully grasp the implications of the above rules for a specific application may sometimes be non trivial. As this is known to be the most difficult part of Rust, and these rules are central to many of the later made claims, these rules will be reiterated, interleaved with some facts and implications:

- Any value (such as a vector) has exactly one owning variable at any given time.

- Ownership can be moved leaving the previous owner uninitialized.

- When an owning variable goes out of scope, the owned value is no longer accessible and any heap data associated with the value will be freed.

- References can be created allowing a function to borrow a value without the variable used in the call giving up the ownership.

- References must never be able to outlive the owning variable.

- Modification of the owning variable can only occur if it is declared mutable and there are no references to the owned value.

- A mutable reference can only be created if the owning variable is mutable and there are no other references to the value.

- An immutable reference can only be created if there is no mutable reference.

- Multiple immutable references are allowed, and immutable references can be copied.

- Creating a mutable reference will lock the owning variable until the mutable reference goes out of scope.

- Creating any reference will prevent modifying the owning variable (even if it is mutable) until the reference goes out of scope.

The rules may seem restrictive, and they have to be, in order to make guarantees about memory safety, and freedom from data races, in all legal cases.

However, in some cases the rules can be too restrictive to implement some specific functionality (such as graph traversal and modification algorithms). Rust solves this with smart pointers, as will be presented later. Some smart pointers allow concurrent modification and others can verify that no concurrent modification takes place during runtime.

This is exactly what many languages, Java for example, do. Java defines a ConcurrentModificationException [42] which will be thrown if attempting to modify a data structure while reading it with an iterator. Rust prevents these types of faults at compile time in almost all cases (including modification while iterating) with zero run-time cost. When it cannot be done, the programmer has to explicitly use a suitable smart pointer, so that when reading the code, it will be clear that there is some (tiny) extra expense.

For performance critical applications, there is also the possibility of using raw C style pointers in an *unsafe* block, and the programmer then commits to take over all the memory safety guarantees inside the *unsafe* block.

Lifetimes

Rust's borrowing rule number 2 dictates that a reference must never outlive the data it refers to. This necessitates the concept of a lifetime. The lifetime of a variable starts when it is created and ends when it goes out of scope. The following is legal, because *xp* does not outlive *x*.

```
fn main(){
  let mut x = 4; //lifetime of x starts here
  {
    let xp = &x; //lifetime of xp starts here
    afunction(xp);
  } //lifetime of xp ends here
  x = 5; //no references to the data owned by x exist
} //lifetime of x ends here
```

The following is also ok, as a temporary reference only exists until the statement finishes:

```
fn main(){
  let mut x = 4; //lifetime of x starts here
  afunction(&x); //temporary reference expires after statement
  x = 5; //no references to the data owned by x exist
} //lifetime of x ends here
```

If lifetimes are not limited with blocks, the borrowing rules are violated and errors such as the following are produced:

Attempt to modify borrowed value:

```rust
fn main(){
  let mut x = 3;
  let xp1 = &x;
  x = 5;
}
```

```
error[E0506]: cannot assign to `x` because
↪  it is borrowed
 --> src/main.rs:4:3
  |
3 |   let xp1 = &x;
  |             - borrow of `x` occurs
↪  here
4 |   x = 5;
  |   ^^^^^ assignment to borrowed `x`
↪  occurs here
```

Attempt to use mutably borrowed value:

```rust
fn main(){
  let mut x = 3;
  let xp1 = &mut x;
  let y = x;
}
```

```
error[E0503]: cannot use `x` because it
↪  was mutably borrowed
 --> src/main.rs:4:7
  |
3 |   let xp1 = &mut x;
  |                 - borrow of `x`
↪  occurs here
4 |   let y = x;
  |       ^ use of borrowed `x`
```

Attempt to mutably borrow already borrowed value:

```rust
fn main(){
  let mut x = 3;
  let xp1 = &x;
  let xp2 = &mut x;
}
```

```
error[E0502]: cannot borrow `x` as mutable
↪  because it is also borrowed as
↪  immutable
 --> src/main.rs:4:18
  |
3 |   let xp1 = &x;
  |             - immutable borrow occurs
↪  here
4 |   let xp2 = &mut x;
  |                 ^ mutable borrow
↪  occurs here
5 | }
  | - immutable borrow ends here
```

Attempt to borrow already mutably borrowed value:

```rust
fn main(){
  let mut x = 3;
  let xp1 = &mut x;
  let xp2 = &x;
}
```

```
error[E0502]: cannot borrow `x` as
↪  immutable because it is also borrowed
↪  as mutable
 --> src/main.rs:4:14
  |
3 |   let xp1 = &mut x;
  |                 - mutable borrow
↪  occurs here
4 |   let xp2 = &x;
  |             ^ immutable borrow occurs
↪  here
5 | }
  | - mutable borrow ends here
```

As can be seen, the error messages are quite readable and explicit, given one has fully understood the borrowing rules. Note that to conserve space, error message output has been line wrapped. The error messages are color coded if viewed in a compatible shell. One should keep in mind that these are toy examples. New Rust programmers are advised to compile often, to not make a borrowing mistake produce a lot of errors. Although, compiling often is hardly a Rust specific recommendation.

2.1.3 Slices

Slices are special references, also known as fat pointers. In the following example, five immutable slices are created:

```
let numbers : Vec<i32> = vec![1, 2, 3];
let x : &[i32] = &numbers; //all elements
let y : &[i32] = &numbers[..]; //all elements
let a : &[i32] = &numbers[1..]; //exclude index 0
let b : &[i32] = &numbers[..1]; //exclude inices 1 and 2
let c : &[i32] = &numbers[1..2]; //only index 1
```

They are represented as a pointer and a length. Any consecutive elements of the same type are, or can be treated as slices. References to vectors or arrays can be treated as slices. References to parts of vectors or arrays are slices. String literals in Rust are slices. References to heap-allocated strings can be treated as slices. Substrings of any string type are slices.

2.1.4 Strings

Rust has several variants of strings. The simplest string is a static constant string.

```
fn main(){
  let s = "this is a static string constant";
  print_it(s);
}
fn print_it(s: &str){
  println!("{}", s);
}
```

As can be seen from the called function, it has type immutable reference to str, which is slice of consecutive characters. To modify a string, a heap-allocated string is required.

```
fn main(){
  let mut s = String::from("this is a ");
  s.push_str("dynamically allocated string");
  print_it(&s);
}
fn print_it(s: &str){
  println!("{}", s);
}
```

The way to pass the heap-allocated string to the print_it function is *&s*. Thanks to the *Deref trait* [4], *&String* can always be used as *&str*, and all addition and printing of strings (heap-allocated or static) will work intuitively. Since the type *&str* is actually what is known as a *fat pointer* or a *slice* which are represented as a pointer and a length, in Rust, string length is a zero-cost operation.

[4] *traits* are similar to iterfaces in other languages.

Strings are encoded as UTF-8, and when indexing them, one refers to bytes rather than characters, so with uncommon symbols one must refer to the start of a UTF-8 code point (the first byte of a multi byte character) [47]. This guarantees that indexing a string is O(1) even though it is encoded in UTF-8.

2.1.5 Static lifetimes and elision

To return a string constant, one must explicitly state that the returned string reference has a lifetime as long as the execution of the program.

```
fn sign_str(a: i32) -> &'static str{
  if a > 0 {
    "positive"
  }
  else {
    "negative"
  }
}

fn main(){
  let a = 4;
  println!("a is {}", sign_str(a));
}
```

This is due to how Rust reasons about lifetimes, and since the string reference is not constructed from any borrowed input value, the lifetime must be explicitly stated. The following would work just fine:

```
fn first3chars(name: &str) -> &str {
  &name[0..3]
}
fn main(){
  println!("{}", first3chars("Rust"));
}
```

This works because Rust can infer that the lifetime of the returned reference is the same as the lifetime of the input reference. This is known as lifetime elision, and works if the function takes exactly one reference parameter. In the case of passing several references as input to a function and returning a reference, it would be unclear from which input variable the output is taken. In this case explicit lifetimes must be used:

```
fn fun<'lt>(a: &'lt str, b: &'lt str) -> &'lt str {
  println!("{}", b);
  a
}
fn main(){
  let name = String::from("The Rust Language");
  println!("{}", fun(&name, "Rust"));
}
```

This means that *'lt* is the minimum of the lifetimes of a and b. And the output will have that lifetime. It is possible to give the values different lifetimes and explicitly tell which is returned. The reason for requiring explicit lifetimes in this case is that full inference of lifetimes and types across method calls was deemed infeasible in order to provide readable errors and for compilation time performance.

2.1.6 Basic struct and enum

Rust provides many ways to organize data. C-style structs and enums are supported, but the Rust variants can optionally provide extended functionality. Basic enums and structs can be created and used as follows:

```
#[derive(Debug)] //auto implement Debug printing
struct Person {
  name: String, // owned String
  age: u32, // unsigned 32 bit integer
  gender: Gender, // some variant of Gender
}
#[derive(Debug)] //auto implement Debug printing
enum Gender {
  Male,
  Female,
}
fn main(){
  let p1 = Person {
    name: String::from("Malerie"),
    age: 42,
    gender: Gender::Female,
  };
  println!("{:?}", p1); //print using Debug implementation
}
```

Running the program generates the following output:

```
Person { name: "Malerie", age: 42, gender: Female }
```

2.1.7 Tuples

If some data needs to be bundled together and only used in a few places, creating a struct may seem like too much work. Tuples exist for this very purpose. They allow for multiple return values, and assignment of multiple variables on a single line. They are also great for key-value-pairs and iterating data structures, among many other uses. For example tuples can be used to return multiple values:

```
fn main() {
  let (a, b) = (5, 2);
  let (quotient, remainder) = int_div(a, b);
```

```
}

fn int_div(dividend: u32, divisor: u32) -> (u32, u32) {
  (dividend/divisor, dividend%divisor)
}
```

Worth noting is that *(u32, u32)* forms a type and thus tuples can be used like any other type.

2.1.8 Instance methods

Any type in Rust can have instance methods associated with it, even basic types like *u32* (unsigned 32 bit integer). For example, let us implement a couple of methods for *Person* as defined in 2.1.6:

```
impl Person {
  fn say_hi(&self){
    println!("Hi my name is {}", self.name);
  }
  fn change_name(&mut self, &str new_name){
    self.name.clear();
    self.name.push_str(new_name);
  }
  fn say_bye_drop(self){ // takes ownership
    println!("Bye {}! Deallocated!", self.name);
  } // drops self and deallocates
}
```

From the main function we could call these instance methods using:

```
p1.say_hi();
p1.change_name("Alice"); //requires p1 declared mutable
p1.say_hi();
p1.say_bye_drop();
```

Those familiar with Python should feel right at home with explicitly stating a self parameter. Basically, the object instance a method is called on, goes into the self parameter of the method, and is converted automatically to reference or mutable reference. Normal borrowing rules apply also to the case of self parameters. The *say_bye_drop* method should say bye and drop the person from memory. Therefore it takes self by value, transferring ownership, and lets the person go out of scope. The *change_name* method should not drop the data, but the data needs to be mutable, thus a mutable reference is chosen, leaving the ownership in *p1*. The *say_hi* method only needs to read the name, and thus an immutable reference will suffice. Observe that the self parameters do not need declared types. The type of the enclosing impl is assumed. This is only allowed for self parameters. What the short hand self parameter versions mean can be seen in table 2.1.

In addition to the above, there are two rarely used versions where both the pointer and the pointed data may be mutated.

Table 2.1: Short version of self parameters for implementation of Person

short version	full version	meaning
self	self: Person	take ownership (drop unless retuned)
mut self	mut self: Person	take ownership (drop unless retuned, may mutate)
&self	self: &Person	borrow immutably
&mut self	self: &mut Person	borrow mutably

2.1.9 Optional data and error handling

We have already seen basic structs and enums, but they can optionally provide additional features. Rust's take on struct, enum and tuples is of vital importance to understand how the language is built, such as how the concept of null is not part of the language but instead there is a safe version of null called *None*. It is the foundation of a lot of the standard library.

To understand how Rust does not need null, and yet has the same flexibility as having null, we need to introduce Rust style enums. [5] What makes enums in Rust special is that variants may contain values:

```
/// returned by access control system on access request
enum AccessDecision {
  Denied,
  GrantedToDoorToken(String),
}
```

The first line is a documentation comment. Because it starts with ///, it will be used to describe the enum in generated HTML documentation. The most important thing to understand is that an AccessDecision is either Denied, which does not contain any String, or it is GrantedToDoorToken which contains String, indicating which door. A reference to the contained String can only be obtained by checking which enum variant is contained in a variable. This is commonly done using a match statement:

```
fn main(){
  let access_decision = AccessDecision::Denied;
    match access_decision{
      AccessDecision::Denied => {
      println!("Access denied!");
    },
    GrantedToDoorToken(a_door_token) => {
      println!("Access granted to door: {}", a_door_token);
    }
  }
}
```

In C, for example, this would probably be implemented with two variables, one being a decision enum and the other opt_door_token char pointer with a comment such as ”this pointer will be null if decision==Denied”.

[5] Parallels can be drawn to Haskell's Maybe monad and Scala's Option.

An enum in Rust is basically a C enum combined with a C union, with one data variant for each enum variant, with forced checking of using the corresponding union data for each enum variant. There is no union type in Rust that allows reinterpreting memory data as another type, this requires the use of the unsafe function *transmute*.

Rust solves this situation more elegantly, by ensuring that it is only possible to use the internal data of GrantedToDoorToken, after having checked that it is actually GrantedToDoorToken. This eliminates the need for a null value. However, oftentimes in C, null can be used to represent an optional value. Rust has a built in enum for this use case called Option.

```
pub enum Option<T> {
  None,
  Some(T),
}
```

Here generics are used to allow the Option enum to be used with any contained type. The option either contains some value of that type, or none. The implementation for Option provides a method called unwrap(), which is basically equivalent to asserting not null in other languages, allowing to use the inner value without matching every Option for None first.

```
fn main(){
  let opt_name = Option::Some(String::from("Rust"));

  // check wich option variant.
  match opt_name {
    Some(name) => {
      // name has type &String
      println!("Hi from {}", name);
    },
    None => {
      //this never happens
    }
  }

  // bind name to the inner string, but only if exists.
  if let Some(name) = opt_name {
    // name has type &String
    println!("Hi again from {}", name);
  }

  // assume exists, or crash with nice error (like assert)
  // takes opt_name by value taking ownership and returns
  // inner string by value, so name owns inner string.
  let name = opt_name.unwrap();
  println!("Hi again from {}", name);
}
```

The above code may seem useless, but illustrates the different ways one may gain access to the inner Optional data. The main point is, for data that should always exist, Option should not be used at all. A plain String should have been used in the example above. This is often the case in C, but not-null asserts can be added just to be safe. For actual optional data, one should use match or if let constructs. In rare cases, unwrapping could be considered, but unwrapping too often has a bad smell.

In the general case, enums with different variants containing different data will occupy the space of the largest variant. If some variants are considerably larger than others, it is possible to store a reference to the contained type in the Option instead. This actually leads to the Option type compiling to nullable pointers! There is therefore literally zero-cost associated with having the Option type, but it forces checking for None during compile time! The Option::unwrap() method actually does check for None, and causes a panic with stack trace with file names, function names and line numbers if it occurs.

2.1.10 Error handling

Just as with the Option type, for replacing null pointers for optional data, there is also a special enum to replace null for error pointers. The Result type is an enum, that either contains the expected result in Ok or contains an error in Err.

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

Errors are passed back from all standard library functions using this type. In order to use the result, the user is forced to check if it is an error, either with a match statement or by other means.

2.1.11 Resource management

When a variable owns some heap-allocated memory, and it goes out of scope, the memory is freed. The same applies to other types of resources, such as file handles and sockets. When a variable owning any resource goes out of scope, the resource is released. This means that there is no need to call any close/flush/destroy functions, as it will be handled automatically.

2.1.12 Iterators, closures and functional programming

Rust has iterators and allows functional programming such as map, fold, etc. Standard library data structures such as vectors and maps have iterator implementations.

Iterators can either iterate with mutable references to the elements or immutable references.

Functional programming is well supported by Rust, but Rust does allows side effects (when the behaviour does not only depend on parameters to a function call, such as working with globals).

Here follow some iterator and functional programming examples:

```
let strings = vec!["Rust", "supports", "functional"];

match strings.iter().find(|s| s == "Rust") {
  Some(_) => println!("it concerns Rust"),
  None => println!("not Rust related")
}

let num_characters = strings.iter()
  .map(|s| s.len())
  .fold(0, |acc, len| acc + len);
```

2.1.13 Maps and Sets

Rust has both hash and binary tree implementations of maps and sets. Maps can be used as follows:

```
let map = HashMap::new();
map.insert("name", "Rust");

match map.get("name") {
  Some(name) => println!("name is set to {}", name),
  None => println!("name is not set")
}

for (k, v) in map.iter() {
  println!("key {} has value {}", k, v);
}
```

2.1.14 A longer Rust example

After having introduced parts of the Rust language, a longer Rust example may be of interest. This is available in appendix A.

2.1.15 Advanced concepts

The subset of the Rust programming language presented thus far is sufficient to be able to follow the reasoning in this report, however, the concepts of smart pointers, concurrency and multi-threading will be covered here in short. These concepts are all implemented using *unsafe* in a respectful manner under the hood of the Rust standard library.

Smart pointers

When the borrowing rules are too restrictive to implement some functionality, there are two options. Implement it in some different way, and rely on compiler optimization, or use smart pointers and containers. In Rust, there is no single smart pointer but rather there are multiple different ones depending on what guarantees are needed.

Cell: Unchecked multiple modification

A cell allows multiple mutable references to the same data. Cell does this by using *unsafe* code in the get and set functions to mutate the internal data without a mutable reference. Cell does not point to data, but rather it actually contains the data.

```
let a = Cell::new(59);
let b = &a; //create two immutable references to the Cell
let c = &a;
// normaly mutating the Cell through b and c would not
// work, but set uses unsafe internally, and allows it
b.set(42);
c.set(13);
```

RefCell: Runtime checked multiple modification

RefCell behaves just like ordinary references, but the borrowing rules are checked at runtime. It contains the data it points to, it is not a heap allocation. If multiple mutable references exist at the same time, the program will panic with a nice error message. In some complex cases, the compiler just cannot make static guarantees that the borrowing rules are fulfilled. The RefCell then allows the code to compile anyway, and the borrowing rules can be checked at runtime. The code below would compile, but not run:

```
let a = RefCell::new(42);
let b = a.borrow_mut();
let c = a.borrow();
```

Rc: Read-only Dependency cycles

Rc has the internal data heap-allocated. Rc uses reference counting to allow multiple pointers to same data, and when the last pointer goes out of scope, the heap data will be deallocated. This is useful when implementing graph traversal algorithms. [6] A node may store several connections to other nodes, and the connections could be bidirectional:

```
struct Node { // a node may have parent and child
  parent: Option<Rc<Node>>,
  child: Option<Rc<Node>>
}
```

[6] Other possible ways are arena allocation libraries, or token to node maps (using tokens as references to other nodes).

```
// create unconnected nodes
let mut n1 : Rc<Node> = Rc::new(Node {
  parent: None,
  child: None
});
let mut n2 : Rc<Node> = Rc::new(Node {
  parent: None,
  child: None
});
// connect the nodes
n1.parent = Some(n2.clone()); // clone increases refcount.
n2.child = Some(n1.clone()); // Does not clone inner Node.
```

The Deref *trait* makes it possible to use *&Rc<Node>* in method calls as if it were *&Node*. Usually, if bidirectional relationships are necessary, Rc is a good choice, as it cannot be achieved using references. If there is a circular dependency, it is not clear which node owns which, and references may not be possible to use. Rc allows creation of cycles, and can actually leak memory. Rust does not make guarantees about not leaking memory. The programmer must not let Rc go out of scope if they have cyclic dependencies, the cycle must be manually broken first!

Cycles of mutable references, composing types

There is no single type which allows both dependency cycles needed for graphs, and modification. Such a type must be composed:

```
Option<Rc<RefCell<Node>>>
```

That would be an optional, reference-counted, pointer to a heap-allocated Node that at run-time allows at most one mutable reference. This provides similar functionality to how objects behave in Java, or other garbage collected languages. One can always fall back to this type if one has problems implementing some functionality, which one would be able to implement in a garbage collected language. The type is long to write, but that is no problem, it is possible to alias types:

```
type OptNodePtr = Option<Rc<RefCell<Node>>>;
```

2.1.16 Concurrency and multi-threading

Many tricky errors can occur from improper locking and synchronization of resources. Rust provides a toolbox of ways to synchronize data between threads. In order to pass a type to another thread, the type must implement the *Send trait*. To be able to synchronize data through a type, it must implement the *Sync trait*. This guarantees that it is not possible to cause concurrency related data races (unless using *unsafe*). Data can be shared between threads in two manners: atomic smart pointers and channels.

Atomic smart pointers

There are concurrency-safe versions of the smart pointers mentioned previously. A table mapping from non-atomic to their atomic versions is found below in table 2.2:

Table 2.2: Mapping from non atomic to atomic wrapper types

Non-atomic	Atomic	Purpose
Rc<T>	Arc<T>	Multiple references to heap-allocated data, with reference counted auto deallocation.
RefCell<T>	Mutex<T>	Write through read only reference, at runtime make sure only one at a time.

```
fn main(){
  // create atomic reference counted mutexed empty vector
  let m = Arc::new(Mutex::new(vec![]));
  // spawn 10 threads, each pushes their number
  for i in 0..10 {
    let mref = m.clone(); // increase refcount
    let icopy = i;
    thread::spawn(move || thread_fun(mref, icopy));
  }
}
fn thread_fun(mref: Arc<Mutex<Vec<i32>>>, i: i32){
  // Arc<Mutex> implements Deref to Mutex
  // call Mutex::lock, explicit type for clarity
  let vecref : MutexGuard<Vec<i32>> = mref.lock();
  // MutexGuard<Vec> implements Deref to Vec
  // call Vec::push
  vecref.push(i);
} // MutexGuard goes out of scope, unlocks Mutex
```

For now, please ignore the `move |<params>|` syntax. [7] Importantly, is it is impossible to access the inner vector of the mutex without locking the mutex first. It is also impossible to forget unlocking the mutex, because it automatically unlocks when the mutex guard, returned by lock, goes out of scope. A Rust mutex protects its internal *data*, not the *code* between lock and unlock such as in C, or the *code* in a synchronized block, as in Java.

The Unix implementation or Mutex uses pthread_mutex_t, and can be assumed to perform similarly.

Channels

A channel transfers a stream of objects. The main thread could create the channel, resulting in an input and an output. The input could be cloned to 4 instances, these 4 inputs could

[7] thread::spawn expects a function with no parameters. The `move |<params>|` syntax is the definition of a closure (like a lambda function in C++), which is allowed to access variables in the outer scope, transferring ownership.

passed to one worker thread each, and the workers can write results to the channel inputs, and the main thread can read the results through the channel output.

Join handles

Spawning threads return join handles, which can be used to wait until a worker thread has finished. The mutex example above should have pushed the join handles to a vector, and then joined them all, if printing the vector before exit was needed.

2.1.17 Dependency management and code structuring

Code can be divided into modules. Modules are defined using the *mod* keyword. They can either be defined in-line or refer to a file. To use functionality from a module, it must be imported.

When a module has been imported, it is possible to refer to the contents of the module with a full qualifying path. It is possible to import some content from a module into the local scope, which allows referring to it without using the full path. It is also possible to import everything from a module. This however can slightly reduce readability, as if everything is imported from several modules, it is not clear where something referred to is defined. This is however convenient when prototyping and re-factoring, before reaching a final design, as functions can be effortlessly moved between files.

Modules not defined by the current project can be imported using the *extern crate* syntax. They must then also be listed in the dependencies file *Cargo.toml*. When compiling they will be automatically downloaded and compiled, along with their dependencies recursively, if any.

The module system in Rust is very easy to use. It is not possible to create import loops, or accidentally import something several times, as is possible in C/C++ and Python.

2.1.18 Language development and backwards compatibility

Rust is developed on GitHub, by the community, where an RFC process is conducted before any changes to the language are made. Rust is sponsored by Mozilla, but it is not controlled by Mozilla.

Rust has changed a lot over time. The language used to have green threading [64] among other things which have since been removed [59]. All these changes happened before the official 1.0 stable release of the language. Any change to the stable version of Rust will retain backwards compatibility. No functionality will be removed or changed, but possibly deprecated.

2.1.19 Legal

The following quote regarding the legal aspects of using the language is taken from the FAQ [49]:

> Rust's code is primarily distributed under the terms of both the MIT license and the Apache License (Version 2.0), with portions covered by various BSD-like licenses.

All the mentioned licenses have no issues regarding usage in commercial software, and thus there should be no legal problems associated with using Rust in commercial products.

2.1.20 Performance and Memory utilization

The goal of this thesis is not a performance evaluation. However, this subsection contains measurements collected from benchmarksgame of Debian [12]. The performance was measured by solving the same problems in several programming languages. The results presented are those for the fastest known implementation. Comprisons with C and C++ are provided, both in terms of execution speed and memory consumption. The results are summarized in tables 2.4 and 2.3.

Table 2.3: Performance Rust vs C++ measured using rustc 1.23.0 and g++ 7.2.0 2018-01-09 [12]

	C++		Rust	
Test	**Time (s)**	**RAM (B)**	**Time (s)**	**RAM (B)**
reverse-complement	0.64	247,800	0.37	250,784
k-nucleotide	7.75	165,180	5.06	137,864
pidigits	1.88	4,276	1.74	4,612
fasta	1.49	4,372	1.47	2,988
fannkuch-redux	10.61	2,044	10.59	1,776
spectral-norm	2.02	1,332	2.27	2,624
mandelbrot	1.51	25,684	1.97	13,340
binary-trees	2.55	135,568	4.02	167,776
n-body	8.23	1,856	13.57	1,768
regex-redux	1.61	203,572	3.05	188,644

The results are mixed, both with regards to memory usage and to execution time. In many cases, both execution time and RAM usage are very close, but some results differ very much, which might indicate very different implementations. It is clear that no compiler or language is universally superior to the others. Rust aims to have comparable execution speed to C++, and it appears that this is also the case. Rust does not set as a goal to be faster, or use less memory than C.

It is worth noting that there are multiple C compilers and only one viable Rust compiler at the moment. If the only compiler has bugs, then they affect all code, and it is not possible to verify compilers against each other. On the other hand, one could expect less problems with code using features available in one compiler which are not yet available in another. The same goes for compiler specific language extensions.

Table 2.4: Performance Rust vs C measured using rustc 1.23.0 and gcc 7.2.0 2018-01-09 [12]

	C		**Rust**	
Test	**Time (s)**	**RAM (B)**	**Time (s)**	**RAM (B)**
k-nucleotide	6.67	130,160	5.06	137,864
reverse-complement	0.48	200,492	0.37	250,784
pidigits	1.74	2,716	1.74	4,612
fasta	1.32	2,912	1.47	2,988
spectral-norm	2.00	1,300	2.27	2,624
mandelbrot	1.64	29,424	1.97	13,340
fannkuch-redux	8.66	980	10.59	1,776
n-body	9.12	1,176	13.57	1,768
binary-trees	2.44	133,956	4.02	167,776
regex-redux	1.48	152,352	3.05	188,644

These results were obtained on one machine and a specific sets of compiler versions and flags. One should be careful to not draw general conclusions. It appears that (with these constraints) C generally uses less RAM and is slightly faster than Rust, and that Rust is generally faster than C++.

What makes a programming language fast, is being strict enough to allow optimization, performing optimization, and generating machine code that fully utilizes the target hardware. It is safe to say that Rust performance is at least comparable. Work on improving the mid level intermediate representation of the Rust language in the compiler is ongoing. One goal of this undertaking is that some Rust specific optimizations could be performed before hitting LLVM. This could potentially result in even better performance in the future.

2.1.21 Popularity, adoption and the future

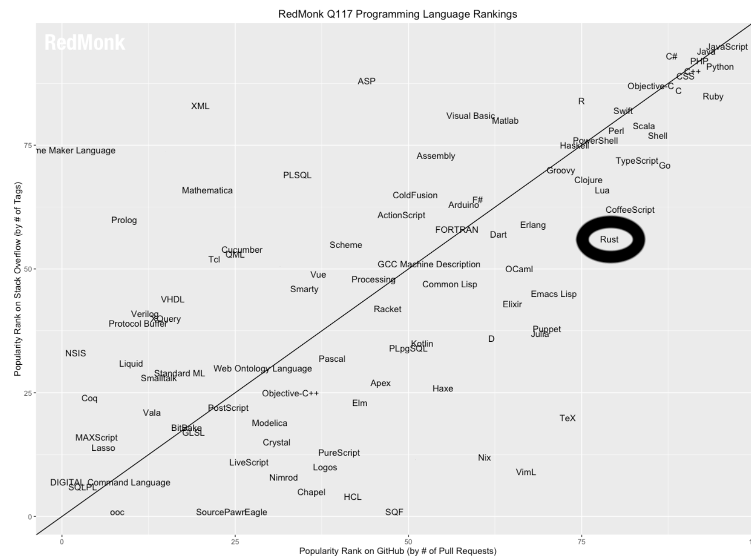
Rust is gaining popularity at a rapid pace, a recent study found Rust to be the fastest growing language based on data from GitHub and Stack Overflow [57]:

> One of the biggest overall gainers of any of the measured languages, Rust leaped from 47 on our board to 26 ... What a difference a few months can make. By our metrics, Rust went from the 46th most popular language on GitHub to the 18th. Some of that is potentially a result of the new process, of course, but no other language grew faster.

A graphical representation of the current popularity rankings according to this study can be found in figure 2.1.

Rust is shipped in official Firefox builds. The reason was stated as "the advantage of using Rust is too great" by maintainer Ted Mielczarek [55]. Rust is also gaining traction in the GNOME camp, with functionality in librsvg [21] being implemented in Rust. This was done to achieve better memory safety, use nicer built-in abstractions and for easier unit testing [36].

Figure 2.1: Rust popularity ranking on Stack Overflow and GitHub [57].



Rust may be gaining popularity now, but many languages have lost popularity throughout the years. It could potentially happen to Rust too. It is difficult to compete with C, because C will always continue being developed. A lot of high quality tested code is written in C, and even if C is not perfect, time has proven it is not flawed enough to be replaced, at least not in a long time. To switch from C, you would want to know first that you are betting on the right horse. I cant make any promises, but for what it is worth, I think Rust got a lot of things right, and the popularity gains suggest that others seem to agree. C is really good when you need detailed control, and it should be buildable on any system, which always comes with a C compiler. Maybe one day, systems will come with a Rust compiler pre-installed? Maybe not, but at least, I think Rust will be an inspiration for how to build safe languages on the level of C in the future.

2.2 Building and Compiling

2.2.1 Supported platforms

Rust runs on many platforms, which are divided into tiers. Cross compilers are available for many combinations of host and target platforms. Tier 1 platforms are ”guaranteed to build and work”. Tier 2 platforms are ”guaranteed to build” and finally there are Tier 3 platforms with no guarantees.

The Tier 1 platforms are all common desktop operating systems: Windows (GNU and MSVC ABI), Linux and OS X on x86 and x86_64.

The Tier 2 platforms are ”guaranteed to build”. These include, but are not limited to, architectures such as ARM and MIPS variants on Linux, Android and iOS.

For the Tier 1 platforms, all that is needed is to download a pre-built statically linked (no dependencies) rustc binary from the Rust website. For the Tier 2 platforms, pre-built binaries are available through the same page but hidden under the archives.

There are also musl [44] targets available allowing the creation of completely statically linked Linux binaries, with no external dependencies, not even libc. This allows the binary to run on virtually any Linux platform of the targeted architecture. All of the standard library is then implemented on top of syscalls.

2.2.2 Adding support for a platform

When programming for embedded systems, there are very many variations of many architectures. In the case with the embedded system being used for testing, the combination of mips32r2 and little endian is not supported officially by Rust, but it is supported by LLVM, which is used to generate the binary. To add support for a platform, a target definition has to be written and integrated into the compiler source code. Then the compiler has to be built. Relevant patches and build instructions can be found in appendix B.

2.2.3 Stability

Rust is a relatively young language which is evolving at a steady pace. It has a large stable set-in-stone standard library. These APIs are guaranteed to still work in future versions of Rust.

However, many additional features are available through the use of unstable APIs, which are not guaranteed to stay compatible with future versions of the Rust language.

There are three channels, with varying levels of reliability and stability: stable, beta and nightly. The beta channel contains a preview of the upcoming stable release. The nightly channel is the only channel to contain unstable APIs, which may or may not become part of a future stable release.

2.2.4 Adaptation of Makefile based build system

The Makefile was modified to no longer compile some of the C files, representing more than half the C code. Instead it was configured to call the Rust build system, Cargo, if any Rust related file had been modified, with the target flag set appropriately, to produce a static library. This static library was then linked by the Makefile to the C binary, replacing the original C implementation of the majority of the service with the Rust port.

A dependency list of all Rust related files was obtained by a executing a shell command from the Makefile, so that the Makefile would not have to be modified each time a Rust file was added. This approach proved to be very simple and effective. It was done the first week and did not change considerably for the duration of the project. There was just no need to modify it, it just worked. Also, the Makefile based build systems at Axis are being phased out in favour of a more modern build system, so investing time in it did not seem worthwhile. The makefile can be found in appendix C.

2.2.5 Cross compilation

A tool called rustup [52] is available, which allows installing cross compilers with a simple command on the command line. As stated previously, the target platform is not officially

supported so instead a compiler had to be built, see appendix B for details.

Once a suitable cross compiler is installed, all that is needed is to call rustc (or the Cargo build system) with the –target flag set to the appropriate target platform identifier string.

2.2.6 Building

The Cargo [45] build system is closely integrated with the Rust language. It provides convenient ways to refer to dependencies similarly to the pip and npm package managers for Python and JavaScript respectively.

Cargo can be used to download and compile software written in Rust automatically.

Cargo is able to pull in dependencies from crates.io (a place to publish Rust libraries), or from git. Version tagging of dependencies is supported, and Cargo can compile a large project with no configuration but is also capable of executing build scripts and lots of other advanced functionality.

Cargo will be used to compile all Rust code written during this evaluation.

2.2.7 C integration

Rust provides a foreign function interface allowing to call C functions easily. Only providing an equivalent function definition in Rust is required, along with the extern keyword, which tells the compiler to use the GNU C ABI.

Support for handling C structs is also present. To achieve this, an equivalent Rust definition of the C struct is needed, with the #[repr(C)] directive, which represents the struct in the same way as C.

Built in support exists for conversion between optional references and nullable C pointers.

Rust code can link C code by specifying a link directive. It is also possible to build a static and/or dynamic Rust library, and link that using C.

Rust contains built in support for converting between Rust strings and C strings using the ffi::CStr and ffi::CString types. The former represents a string coming from C in the form of a pointer, while the latter is a heap-allocated C string created by Rust, which can be passed to C as a pointer.

2.3 Development Tools

Unit testing, profiling and debugging were evaluated on x86_64. The reason for this was that recent versions of GDB have improved official support for Rust, and building a recent version of GDB for the target architecture was not trivial, as some MIPS system header file definitions were not the versions required to build GDB. As mentioned earlier, building the most recent official GDB for a specific target is not a Rust specific problem, but is needed for a good Rust debugging experience. Spending time on resolving such issues was deemed infeasible due to the time constraints. It is indisputable that C/C++ has better support form tools than Rust has. Old versions of GDB will work just fine debugging C.

2.3.1 Testing

Unit testing in Rust is very easy and supported out of the box without the need for any additional libraries. All that is needed is to create test functions, and to annotate those with *#[test]*. It is then possible to call *cargo test* on the command line, which will compile the project for testing and run the test functions, producing a test result log.

Ways to do test coverage analysis on Rust executables exist. It is possible to use a recent version of *kcov* [56] to get line coverage results.

Once a recent version of kcov has been compiled, all that is needed, is to invoke Cargo to produce a test executable, without running it, and then invoke kcov on the produced executable.

```
$ cargo test --no-run
$ kcov target/cov target/debug/myprogram
```

This will produce a nice detailed HTML coverage report, detailing covered and not covered lines. One caveat is that kcov only collects line coverage, which is only one of many possible metrics for code coverage, and not very detailed. Another caveat is that code that should never run, such as some error handling code, will not be covered, unless an error actually occurs [54]. After all, Rust is a young compiled language, and having some sort of code coverage analysis is very positive.

2.3.2 Benchmarking

Rust has built in support for benchmarking which works very similarly to testing. Benchmark functions are annotated with *#[bench]*, and take a *Bencher* parameter, which can be used to perform some work a number of times. The number of times Rust executes the workload will depend on how long it takes. Fast workloads will execute many times and will have higher precision than slow workloads.

Benchmarking is invoked by running *cargo bench*. A log detailing how long every iteration takes for each bench function will then be printed.

2.3.3 Profiling

While no profiling tools are bundled with a Rust installation, Rust executables can be profiled using standard profiling tools. The easiest way is to install Valgrind [63], and then install cargo-profiler by invoking *cargo install cargo-profiler* [58]. Rust programs can then be profiled using *cargo profiler callgrind* and *cargo profiler cachegrind –release*. It is possible to find out how much time is spent in different functions, and how many cache misses occur. The release flag is recommended for cachegrind as information about cache misses on an unoptimized debug build would probably not be of much use. Profiling was tested on toy examples on x86_64, and all worked without problems.

2.3.4 Finding memory leaks

A program was designed to leak memory by boxing some data in a heap allocation, and calling *Box::into_raw*, which causes Rust to not deallocate the data. This is supposed

to be used for example when registering a callback. When receiving the callback, one should get back the pointer and deallocate the data. This was purposefully overlooked. The program was then run with Valgrind, which found the leak. Running a Rust web server with Valgrind, written by the author in 100% safe Rust, does not generate any false positives.

2.3.5 Debugging

Rust installs bundle both *rust-gdb* and *rust-lldb*. Both require GDB [17] and LLDB [34] to be installed, respectively. Calling *rust-gdb* will load GDB with Rust pretty printers. For the optimal debugging experience, a recent version of GDB should be used, as it has official support for Rust [19]. As mentioned earlier, building the latest GDB for the target architecture posed some header file issues, and as this is not really related to Rust, and debugging was not really very necessary during the porting evaluation because of the existence of a large functional test suite, debugging was tested on x86_64.

Setting breakpoints, stepping into, stepping over, stepping out etc. all worked as expected. The Rust pretty printers are very good, and they allow showing Rust vectors and Maps without showing internal pointers or data structure implementation details. Printing of structs is very convenient, and works even if they contain strings and data structures.

There were no problems changing the values of variables during runtime. It is even possible to call Rust functions from GDB, with some limitations such as when using generics. Instantiating Rust structs and adding them to data structures during runtime was tested. No problems were found, and the experience was very positive. It should be noted, that data allocated with GDB will not be freed.

A Rust program which should segfault was created by casting a number to a pointer, and dereferencing it in an *unsafe* block. The program was then run, and analysing the crash dump worked without problems.

2.3.6 IDEs

To test how well Rust is supported by IDEs, two popular, open source IDEs with Rust plugins were chosen for evaluation: *Eclipse RustDT* [8] and *IntelliJ Rust* [2]. Eclipse RustDT is a plugin for Eclipse which uses racer [43], rainicorn [9] parse_describe and rustfmt [40] to implement the functionality it offers. These tools are used by numerous plugins for other editors as well. The IntelliJ Rust team takes a different approach and aim to implement all functionality on the IntelliJ platform, including parsing, file structure, auto completion etc. keeping external tools dependency to a minimum.

First the IDEs were installed and set up, then the author worked using each editor for two days, writing down experiences gathered during this period. The findings will be summarized in the following paragraphs.

IntelliJ Rust is easier to install than Eclipse RustDT, due to the fact that it needs considerably less external tools. It is also easier to configure, as it detects paths automatically and it suggests to install required Rust standard library source code, which can be done with a single click. The run configurations for Eclipse RustDT seem outdated after installation, and have to be manually configured, and the names cannot be changed. IntelliJ

Rust provides a single run configuration, and adding more, e.g. for testing and benchmarking, is easy and intuitive. Getting up and running with Eclipse took the author around 45 minutes, while for IntelliJ it took around 5 minutes.

Type inference in both IDEs works well for simple cases, but not so well when generics or macros are involved. In both editors, to take advantage of type dependent features such as auto completion, return types from macros or generic functions have to have explicitly annotated. Otherwise the editors lack sufficient type information.

Syntax highlighting is good in both IDEs but arguably slightly better in IntelliJ, as instance methods are colored yellow if the editor has type information.

Auto formatting works very well in both editors, but it is slightly better in IntelliJ in the case where several coding styles are allowed, such as when breaking long statements into several lines. IntelliJ's auto formatter will leave correctly formatted code alone, and clearly shows how many lines were affected and what changed, while Eclipse RustDT tends to prefer one line constructs and does not format statements containing comments.

As for auto completion (showing valid instance methods and fields available on a variable), in both editors it works only if the type can be determined, which as stated previously is problematic if generics or macros are involved. If the type is known to the editor, the experience is slightly better in IntelliJ Rust, due to Eclipse RustDT showing duplicate results and having a minor scrolling annoyance in the case of long methods.

Both editors can provide error highlighting on parsing errors, but neither manages to highlight undeclared symbols. IntelliJ does however underline variables for which it can find the definition, so if there is no underlining, one can assume that the variable is misspelled. Some errors in addition to parsing errors are highlighted in IntelliJ, such as return type not matching, assigning to immutable variable or ignoring a result which could be an error. Many possible cases such as passing an immutable reference to a function requiring a mutable one, or sending a variable with the wrong type as parameter generate no error highlighting in either editor.

Both editors have good file structure browsing capabilities, showing functions, structs, fields, members, enums, variants, instance methods and test functions. Both editors provide some basic code templates for typing common code quickly. However, Eclipse RustDT has predefined templates for matching which IntelliJ Rust lacks, but they are easy to add. No editor provides an automatic template or completion for matching all possible variants of an enum, which would be a welcome improvement.

IntelliJ RustDT feels a bit snappier when it comes to reacting to keyboard shortcuts such as opening auto completion among other activities, the editor also starts faster than Eclipse RustDT.

A big plus for Eclipse is that plugins for C and C++ are free, while some official plugins for those languages in IntelliJ are not. This matters mostly if developing applications linking Rust and C/C++ code. IntelliJ does have some free C/C++ plugins for basics such as syntax highlighting, and it is possible to configure build scripts etc.

When it comes to development activity, it seems that there are more frequent commits to the IntelliJ Rust code base than to Eclipse RustDT code base lately, however, note that since Eclipse RustDT makes use of external tools to implement some functionality, the development activity for those tools could be taken into consideration.

As for in editor debugging, the free version of IntelliJ does not come with GDB/LLDB support. To use this functionality, a paid CLion license is required. As this evaluation

targets open source software, it has not been evaluated, but it is worth mentioning that there is a project planning to use this infrastructure for Rust debugging in IntelliJ. Setting up debugging in Eclipse is covered by the Eclipse RustDT user guide, and should work after installing Rust pretty printers and a recent version of GDB, according to the user guide. Debugging has been analyzed in the debugging section.

Both editors provide functionality to jump to a function definition, variable declaration and type declaration. IntelliJ Rust parses the output of Cargo to be able to go directly to a source line in the affected file for all types of errors or test failures. In cases where the error says something like ”declared immutable in location x, but assignment occurs in location y” it is possible to go to either location by clicking. This functionality is lacking in Eclipse RustDT.

Table 2.5: Goal quiestion metrics grading for experience using Eclipse RustDT and IntelliJ Rust. The grading scale ranges from 0 to 5 where 0 represents lacking, 1 represents something and 5 represents excellent.

short version	Eclipse RustDT	IntelliJ Rust
ease of installation	3	5
ease of configuration	3	5
run configurations	2	5
type inference	3	3
syntax hilighting	4	5
auto formatting	4	5
auto completion	2	3
error highlighting	1	2
file structure	5	5
code templates	4	3
editor performance	3	4
free C code support	5	3
debugging using GDB	4	1
development activity	3	4
sum	**46**	**53**

Table 2.6: Implemented features in Eclipse RustDT and IntelliJ Rust. This table lists features that either exist or not.

short version	Eclipse RustDT	IntelliJ Rust
go to error location	no	yes
go to test failure	no	yes
go to variable declaration	yes	yes
go to type definition	yes	yes

Based on these experiences, a very rough assessment of the state of Rust IDEs has been summarized in tables 2.5 and 2.6. It is worth noting, that type inference is central to many of the other features, and once type inference handles generics and macros (perhaps

only for *try!*, *vec!*, *Option* and *Result*) the grading of many other features would improve significantly. [8]

A project called Rust Language Server (RLS) is being developed and is available in alpha state which will be running parts of the compiler as a background service, communicating with editors via IPC. The author took the time to quickly try out RLS with Visual Studio Code which is what the developers use to test it. The RLS shows some very impressive suggestions (including the cases where current IDEs are lacking) but it is not yet mature enough for production use, as stated by the project developers.

The State of Rust Survey 2016 [31] found that IDE support is important to attract new users to the language:

> Of non-Rust users, 1 in 4 responded that they aren't currently using Rust because of the lack of strong IDE support.

A lot has happened since then, looking at the change log of IntelliJ Rust.

2.4 Porting C code to Rust

The porting study was conducted by porting parts of the pacsd daemon of the PACS [4]. The PACS and the pacsd daemon, as well as why they were chosen for the evaluation, will be presented in the following subsections.

2.4.1 Physical Access Control System (PACS)

The Physical Access Control System, henceforth referred to as PACS, is a product which controls PIN code terminals, RFID card readers and electric door locks. It also features door monitor capability, to check if the door is closed. It is compatible with various protocols used by card readers, and works with several different kinds of cards. It is compatible with door locks using both high and low-level signals. Each PACS can operate two doors with associated PIN/Card terminals and door/lock monitors.

The PACS is capable of working without a central control computer. This reduces the single point of failure problem associated with such architectures. All of the hardware, authorization data (PIN, card number etc.), and access policies (who can access what and when), can be configured through HTTP API calls (several formats), or via a web interface.

Several PACS systems are able to form a swarm. The swarm, after being configured, will synchronize data across all the participating PACS systems. Modifications to one PACS will be propagated throughout the swarm. If the connections from one PACS to the rest of the swarm are broken, it is capable of continued operation. And all data will synchronized once the connection can be re-established, and an algorithm will resolve any inconsistencies.

The PACS has event logging capabilities. Logs can be filtered and viewed in a browser and by other means.

[8] As of 2017-03-20, type inference for generics has been added in IntelliJ Rust, along with various other fixes and improvements. However, it still does not work with macros, if let ..., for ... in ... or match statements. This was stumbled upon by accident. Unfortunately, there is not enough time to re-evaluate Eclipse RustDT and therefore the score has not been updated to reflect this, as it could be considered unfair.

2.4.2 PACS software stack

Each PACS runs Linux, and the functionality is implemented in several micro services, communicating primarily over D-Bus. One service handles I/O, on top of that two services, one for doors control and one for id points control, are implemented. An id point is commonly a pin code terminal with integrated card reader.

The goal of the design is that each daemon could restart without severely impacting the rest of the system.

At the core of the PACS is the pacsd daemon. It communicates with the id point daemon and the door control daemon, keeps a database, and is responsible for session tracking, making access decisions, opening doors, event logging, and giving user feedback through blinking different coloured LEDs on pin code terminals.

This central daemon was chosen as a good target for the evaluation. It was chosen mostly because it interfaces a lot with different services and uses many libraries. Another reason to choose this particular daemon was that the previously mentioned Go evaluation project attempted to port parts of the same daemon, but that goal could not be realized because of a platform update and time constraints.

2.4.3 Choosing parts to port

As the amount of code in pacsd is vast, the scope for the porting evaluation had to be limited. The C code was analysed and large parts of the code was excluded from porting, for the following reasons:

- Short functions — API calls with GLib types, called from other libraries, doing a simple operation such as database data fetch and returning it were — were deemed not worthwhile porting. The method signatures could not be modified. A Rust implementation would have to deal directly with GLib types, and thus be marked *unsafe* anyway. There were a lot of these functions adding up to thousands of lines of code. Some of this code dealt with generated code. Porting partially generated code seems like a waste of time. The generator could instead have Rust code generation implemented.

- Database management code made heavy use of C APIs and C strings for SQL queries. It would have to be implemented with a lot of type conversion and ffi (foreign function interface). This also consisted of thousands of lines of code. Porting this would likely be a good idea for performance reasons, as it could build Rust types instead of GLib types, eliminating conversion, but time was limited.

- Process initialization. The daemon used traditional double forking to drop privileges. The initialization does a lot of C function calls, initializing different libraries, and would be largely *unsafe* if written in Rust anyway. It would also start Glib main loop on a non main thread. Using systemd was considered, but fully understanding library initialization code proved to be too much work given the available time.

While Rust could have been used to implement the excluded code, it would not yield much benefit, and it would require a lot of extra work. The code would be more or less the

same. If the Rust and C implementations look the same, it does not make Rust bad, it just makes it not worth the effort.

As has previously been stated, while any language could be used to implement any functionality, one should choose a programming language which is suitable for the task at hand. Therefore this evaluation does mostly target code where it is reasonable to think that Rust would do well. C is suitable for the excluded code, Rust is not, as most of that functionality would require *unsafe* blocks. The optimal solution would be to port the libraries, and have them export both a C and a Rust API. That way, the *unsafe* blocks would not be necessary. However, that is out of scope for this project.

What remained after excluding the above was the session management, id point state management, decision making logic, and D-Bus communication with id point and door control. This code represented more than half of the ten thousand lines. It was chosen because it appeared that Rust could have things to contribute to this code:

- Data structures were used heavily, which would showcase the Rust standard library.
- Global state and mutexes were used, and Rust is good at concurrency guarantees.
- Very many C functions would need to be called, but still few compared to a full port.
- GLib types would only be required for input and output. Internally, the code could use Rust types.
- A lot of copying and asserting not null was taking place. Rust solves these situations elegantly, with safety guarantees.
- Picking this code allows evaluating interfacing with Glib [20], D-Bus [18] and callbacks, in accordance with the project goals.

2.4.4 Understanding the code

Some knowledge about the overall structure of the code is required before starting porting, in order to make informed decisions. The following methods and tools were used to gain an understanding about what the code was actually doing:

Header files analysis

The functions exported by header files can be a good indicator of what functionality is called by the rest of the program if functions are exported restrictively.

Structs definition, creation and free

Structs in the C code files were analysed and the code relating to creation and destruction was studied. Tracing the life of the struct through the program gave valuable insight into whether pointers to the struct were stored in data structures. Unfortunately, Glib uses void pointers, so this was the only way to find out.

Call graph and dependency graph analysis

Doxygen [13] was used to generate call graphs from C code.

2.4.5 Porting approaches

After getting familiar with the overall architecture, a few possible porting approaches were considered, and tested:

1. Leaf first / bottom up approach
2. Branch first / top down approach
3. Data flow analysis approach

Bottom up approach

To start with, a bottom up approach was used to port single function to Rust. This was done by first analysing a simple C function, porting and exporting it with the same parameter and return types, removing it from the C code, and linking the static library produced by Rust to the C binary, as described in appendix C.

This process would continue until a number of functions were ported. At a certain point, a function would never be called by C, but from Rust to Rust only. When this occurred, the parameters and return value could be re-factored to use Rust types instead.

After getting some experience in porting in this simple fashion, small steps at a time, eventually a full C file would be ported. The entire file could then use Rust types internally, and only use C types for the exported interface defined by the C header file. The entire C file could then be cut from the Makefile build process.

Top down approach

Another porting approach tested was the top down approach. The interface exported by a C header file would be ported to Rust and exported. Then the entire C source code implementing it would be analysed carefully, by using tools such as Doxygen to generate call graphs.

The code of each function would be studied in detail, and comments would be added stating the purpose of the code. Some loops would be commented as “find something” other loops would be commented as “remove duplicates” etc. While existing comments were very useful, they were of a higher level nature.

Then a Rust architecture achieving the same purpose would be prototyped, and implemented, all in one go. The entire C file could then be cut from the build process all at once.

Data flow analysis approach

In some cases files were huge, consisting of several thousand lines of complex code. In these cases, a data flow oriented approach was used. A struct would be selected, and the

creation, lifetime and point of freeing would be analysed. Then all operations performed on the struct would be analysed.

This struct would then be implemented in Rust, with the operations performed on it being implemented as instance methods where possible. A function to heap allocate the Rust struct and return a pointer would be written in Rust, and a free function would also be implemented in Rust. Rust wrapper functions would be created to allow calling the instance methods from C.

The C code would then be modified to use the Rust version instead. The C version of the struct and all related functionality could then be deleted from the C file, reducing the amount of C code.

This process would then continue with then next struct, until the entire C file was ported. Then major re-factoring in Rust would take place.

Hybrid approach

The above methods have been simplified, and different methods were employed at different points in time and in different situations. They all have their upsides and downsides. The findings will be discussed in section 3.3.1.

2.4.6 Testing

The porting of code made the original unit tests incompatible. This could not be avoided, because they would call C source file internal functions (by defining STATIC to be static except when testing). To preserve unit testing capability would mean to use C types in the entire Rust code, and to not be able to modify the architecture to suit Rust. Porting the unit test suite would have been to much work.

A large functional test suite existed, to test the functionality by issuing API calls over HTTP. This was the method used for continuous testing when porting the code base.

2.4.7 Definitions of C functions and structs

As stated previously, in order to use functions and structures defined in C, one must provide equivalent definitions in Rust. In the beginning this was done by hand. It is not very demanding, intellectually, but it is easy to make mistakes.

A very handy library called bindgen [15] is available. It can be easily installed with *cargo install bindgen*. After doing this, bindgen can be executed on the command line, providing a C header file as input. It will then generate equivalent Rust definitions of all structs, enums and functions. It will recurse into other files, but a command line parameter is available to filter the output. This was used to prevent all of Glib from being exported in every single header file.

A simple batch script iterating over all header files, with appropriate output filters was created. Unfortunately, a lot of include directories had to be located to get a hold of all the needed header files. The output was not always immediately usable, and some definitions such as Glib types, and references to types defined in other headers had to be imported by manually editing the output (because of the usage of the output filters). The time saved by

bindgen, and all the manual translation errors that were prevented because of it, are both likely large quantities.

One small issue with bindgen (or rather the original C code) was that it did not always use const pointers for parameters, despite not modifying the data. This resulted in being translated into mutable pointer parameters in Rust. Mutable pointers cannot be created easily in Rust, as other pointers are forbidden to exist at the same time by the borrowing rules. This required manual editing of the pointer parameters, and analysis of the C code to understand if it did mutate the data or not.

2.4.8 Callbacks

C code using callbacks is very easy to deal with in Rust. Callback functions just have to be defined with the *extern* keyword. This function can then be registered as a callback by calling some C function, and passing the name of the callback function.

In the case where the callback has raw C pointer parameters, one should null check those, and in the case of void pointers, cast them into pointers to appropriate types, depending on the data type passed when the callback was registered/requested/subscribed.

In the case of C strings, those could be converted to Rust strings using the *ffi::CStr* type, from which, one can obtain a Rust &str slice or a heap-allocated String.

If the callback has a data void pointer, where the user can decide the type of this data when registering the callback, Rust has a Box type, which can heap allocate anything, and then there is an *Box::into_raw* function, allowing Rust to "forget" about deallocating the heap data when it goes out of scope. This approach is excellent for registering callbacks.

Then when the callback is received, there is a *Box::from_raw* function, allowing to get the heap allocation back. The callback data would then be deallocated after the callback finishes.

If the callback is supposed to happen many times, one could simply skip the *Box::from_raw* call. This should then instead be done when unsubscribing from the callback.

2.4.9 Glib

A library called glib-sys [24] exists, which contains Rust definitions of Glib types and functions. This library was a great aid in handling Glib types. The wrapper adds very little on top of Glib. Using the functionality is very similar to how it is done from C. Instead of nullable pointers in some places, optionals are used instead. This does mean that there is no easier way of using Glib data structures than the C way.

All operations on GPtrArray and GHashTable use void pointers, just like in C. Therefore, dealing with Glib types in Rust requires *unsafe* every time they are touched.

A wrapper Rust module was created, providing generic functions for easy conversion between:

- `GPtrArray` ⟷ `Vec<T>` and
- `GHashTable` ⟷ `HashMap<S, T>`.

These functions were implemented to copy the data inside the vectors and maps in both directions. This allows the ownership system of Rust to work the way it was designed to.

Special cases were implemented when the contained void pointers were C strings, allowing the following conversions:

- `GPtrArray` ⟷ `Vec<String>`,
- `GHashTable` ⟷ `HashMap<String, String>` and
- `GHashTable` ⟷ `HashMap<String, Vec<String>>`.

Several other approaches were possible, like wrapping the Glib types to provide similar functionality as the Rust counterparts, but this would not feel like idiomatic Rust. This approach was chosen because it allows nearly all code to be safe, except the conversion code in a single module. Generics allow converting any Glib type, by specifying the type when making the conversion.

2.4.10 Wrapping C structs

Lots of database structs were defined in C. These structs would contain C strings and pointers to other structs. To easily be able to interface with the database, Rust versions of all these structs were defined. The Rust variants would contain Rust String and Rust sub-structs instead of pointers. Any database operation would convert the entire result to Rust types by copying, and immediately free the C versions afterwards. This allowed working with only owned Rust types in the entire implementation, and limited much of the *unsafe* code to a single place.

The following code is an example of how this could look (simplified):

```
impl DbWrapper {
  fn get_users() -> Vec<DbUser> {
    unsafe {
      //call C database
      let c_users : GPtrArray = db_get_users();
      //convert to convenient Vec, inform the compiler
      //that the type of elements is *const db_user_t
      let users : Vec<*const db_user_t> =
      GlibWrapper::g_ptr_array_to_vec(c_users);
      //map the pointers to Rust DbUser structs
      //the from_ptr function converts each db_user_t
      let r = users.map(|p| DbUser::from_ptr(p)).collect();
      //free the c data
      db_free_users(c_users);
      r //return the result
    }
  }
}
```

`GlibWrapper::g_ptr_array_to_vec` and `DbUser::from_ptr` both perform null checks. To the outside world, the database is now very easy to use, and does not require *unsafe* to call. The *unsafe* is entirely contained in the DbWrapper and GlibWrapper.

2.4.11 D-Bus

The D-Bus code basically only performed bus acquisition, registration and synchronous calls. The code was implemented using a now deprecated D-Bus library. Because of this, the D-Bus code in the Rust pacsd port was basically just modified to use Rust types, and made direct C ffi calls to the same deprecated D-Bus functions used in the original version. Some wrapper code converting the Rust types to the needed GVariants was implemented. This simple approach worked well, and there was little reason to build a better wrapper on top of a deprecated API [22].

Some Rust D-Bus wrapper APIs are available, but pulling in extra dependencies and libraries when only such rudimentary functionality was used was not motivated.

2.4.12 Code size reduction

Of the C code that was translated, the number of lines of code could be reduced from around 5600 lines of C to around 3200 lines of Rust. This number was gathered by manually looking through both the C code and Rust code. Imports were not counted. The code contained the approximate same amount of comments and whitespace. The numbers do not include auto generated Rust definitions of C headers obtained using bindgen [15]. Rust lines were of about the same length as the C lines.

2.4.13 Binary size

Binary size tests were performed, by compiling statically linked optimized executables with different options. It turns out that auto implementation of JSON-like debug printing of all structs, and debug symbols for the entire standard library accounts for the vast majority of the binary size. Looking at the smallest binaries for each language, Rust produces a 36% larger (223kB larger) binary. The reason why some functionality is removed from the Rust language for a binary size comparison is that C does not do stack unwinding or debug printouts. C also uses the system allocator. The findings are presented in table 2.7. Note that LTO (Link Time Optimization) was enabled.

2.4.14 Memory consumption

In terms of maximum memory consumption (RSS [9]), the Rust port used 7% (3MB) more memory, as seen in table 2.8. It should be noted that the Rust code copies data structures which may account for parts of the difference.

2.4.15 Performance

While performance evaluation was not an outset goal, some basic performance measurements certainly could not hurt. The performance measurements were conducted by run-

[9]Resident set size (RSS) is the portion of memory occupied by a process that is held in main memory (RAM) [66].

Table 2.7: Binary sizes for various compilation options. Strip refers to running the strip command on the produced executable. Abort refers to aborting execution on thread panic instead of stack unwinding with trace printing. Allocsys refers to the use of the system allocator instead of jemalloc. No impl debug refers to not using auto implementation of structs JSON-like debug printing.

Options	**Size**
C	1,779,138 bytes
C+strip	633,228 bytes
Rust	13,959,408 bytes
Rust+strip	1,461,320 bytes
Rust+allocsys	2,563,822 bytes
Rust+allocsys+strip	1,461,304 bytes
Rust+allocsys+abort	2,419,081 bytes
Rust+allocsys+abort+strip	1,324,792 bytes
Rust+allocsys+abort+strip+no impl debug	861,316 bytes

Table 2.8: Maximum memory usage of smallest executable for each language. The memory usage was measured by running ”time -v pacsd” and running the test suite.

Language	**Maximum resident set size (kbytes)**
C	43056
Rust	46112

ning the functional test suite, which performs API calls, resulting in calls to the state tracking and decision logic. The performance testing was done by implementing a stopwatch in both the C and the Rust versions of pacsd, starting the stopwatch when entering the decision logic code, and stopping the stopwatch when returning from it.

Measuring the time spent in the ported code for each language, Rust was 14% slower. If excluding the worst case time from both languages, it was 9% faster. It should be noted that when analyzing the decision logic, a minor bug was uncovered. It is fixed in the Rust port, but has the side effect of more expensive computation in the worst case. This is due to computing unused results and due to copying large data structures. This case is unrealistic and could be optimized to perform the expensive computation only if access is denied, but this has not been done.

Table 2.9: Time spent in decision making logic compared to C when running the test suite.

Test cases timed	**Rust port time spent**
All	114%
All except worst case	91%

Probably the performance figures are not of importance, as it is highly unlikely that real world interaction with the system could be performed very quickly. Still, it somewhat

interesting to see that a largely unoptimized port using a different architecture has about the same performance. Performance figures of a manually optimized Rust port would be interesting, but comparing it to an unoptimized C version could be considered unfair.

Chapter 3
Discussion

Here the findings from the evaluation section will be discussed, and then the conclusions will be presented in chapter 4.

3.1 The Rust Programming Language

3.1.1 Type system

The type system in Rust is very well designed. Struct, tuples and enums cover every use case in elegant ways. Allowing implementation of instance methods on any type is very convenient. *trait*s offer a good way of implementing interface-like functionality.

3.1.2 Data structures

Data structures are very convenient to use and iterator functionality allows great expressiveness. Cross platform threading, networking and I/O is easy to use. Closures are very convenient, especially when used in conjunction with iterator methods for functional programming.

3.1.3 Optionals

The introduction of the concept of null pointers has been referred to by its inventor as ”my billion-dollar mistake” [25]. It was implemented because it could be useful, and it was easy to add.

The concept of a pointer means that it should point to some valid instance in memory. This is the path taken by Rust. The option type is very useful and very intuitive to use. It is used throughout the standard library in all cases where a return value may or may not

exist, such as when getting the value of a key from a map, or when getting an the value at an index from a vector.

While optionals are nothing new in the context of programming languages, Rust is perhaps the first truely low-level language to make use of optionals throughout its standard library.

3.1.4 Error handling

Error handling in Rust is handled by returning the result type, which as has been seen previously, is an enum containing either the expected return type or an error type. The result has to be matched and checked for errors before being usable.

A minor issue with this system of error handling is that different functions may return different error types. When a function needs to return potentially multiple error types, this must be solved either by conversion to something like a string representation of errors, or a custom error enum has to be defined, which can contain either the one error type or the other. Defining such error types can be tedious.

There are libraries [67] that provide convenient functionality for cases where results may have to contain different error types, but these libraries must be pulled in as dependencies, they are not part of the standard library.

In future versions of Rust the need for returning multiple error types may be solvable by a new syntax, which would allow writing that a function may return any type which implements some custom error *trait* defined by the user. This *trait* could then be implemented for all the error types which need to be returned.

Error handling is handled in different ways in different languages, each of these ways have their upsides and downsides. In C, commonly some special value such as a negative integer value or a null pointer may be returned to indicate that an error occurred. The programmer could forget to check for errors, and proceed to use this pointer as if it were a success return value. To handle errors appropriately, oftentimes, one has to check if the returned value matches the criteria for if an error occurred. If an error occurred, to find out details about the error, one often has to call a special function to get an error description [60]. In the light of this, the way errors are handled in Rust must be considered strictly superior to how errors are handled in C.

In C++, errors are often handled by exceptions. How exceptions are implemented is not defined by the C++ language standard. It is implementation defined [7], meaning that error handling may incur different costs depending on the compiler used. Thus the error code path should not be used unless strictly necessary. This causes some implementations to use the C method for handling errors.

Exceptions however do allow calling functions which may potentially throw errors without checking. An error would be propagated back by popping the stack until a try/catch block is found. Rust has a macro called *try!* which is used to provide similar functionality. The try macro can be used to call a function which may potentially return an error without matching on the return value. The try macro basically generates code which inserts ”if error then return it, else unwrap the result”. The added burden of having to write try for every call to functions which may fail does not outweigh the benefits of code readability, according to the author.

However, the try macro is unable to work if the calling function returns a result with

a different error type (such as an enum which may contain different error variants). This is also handled in part by error composition libraries available for Rust, which provide similar macros for error handling.

In summary, error handling in Rust is well designed, as it is very explicit, yet does not require a lot of boilerplate, and error handling cannot be forgotten. The situation of returning multiple error types is handled well by the current stable version of Rust, but requires a bit of added effort of having to define a custom error enum with multiple error variants. If this burden is considered too big in some case, error handling libraries with convenience macros are available. Future versions of Rust will remain backwards compatible, and will provide error handling requiring even less work. Performance wise, the error code path in Rust should not incur added run-time cost compared to the success code path.

3.1.5 Object oriented programming

For those who prefer object oriented programming to imperative/procedural programming common in lower level languages, this section compares the features of the Rust language with the concept of classes, found in other languages.

A struct with implementation is more or less equivalent to a class without inheritance, except it forces separation of variables and code, which in many languages is considered best practice anyway. There are also traits, which are more or less equivalent to interfaces in Java.

However, there is no extending structs, which was a conscious design choice. One might just as well enclose the type one wishes to extend in a new struct, and make it public, providing all the methods of the enclosed type.

It is not possible to override instance methods in Rust. If Rust had the possibility to override methods and extend structs, it would conceal the pointer following and dynamic dispatch that would have to take place to make it work, which conflicts with Rust's philosophy of generating predictable machine code.

3.1.6 Safety

Rust puts a strong emphasis on safety. This is reflected throughout the entire language and standard library. The technique is well thought out and forces all code to be safe unless explicitly giving up the safety guarantees by using *unsafe* constructs.

While C++ does provide some safety, if stack allocation and destructor are used properly, it is nowhere near what Rust has to offer. The important distinction is that C++ offers some opt in security, which the programmer must choose to make use of. Rust has very strict security by default, which the programmer can choose to give up.

3.1.7 String types

Rust has many string types: heap-allocated strings, string slices, owned C strings, and borrowed C strings. Conversion between all these types can be tedious. heap-allocated

Rust strings are the most flexible. Cloning them on assignment provides a hassle free experience.

Rust has the explicit goal of being low-level, and to allow optimization. That is why all these string types are needed. For example Java only has heap-allocated strings, and they can not be modified. When adding a character to a Java string, the string gets cloned.

Conversion to/from C strings needs several error checks for various reasons, and thus it is advised to not work with C strings except on ffi boundaries.

3.1.8 The deref trait

As explained previously, the `Deref` *trait* allows automatic conversion of some types into other types. Notably dynamically allocated strings can be treated as string references. This allows string literals and dynamically allocated strings to be used interchangeably in many cases, without having to explicitly convert strings.

The Deref *trait* is also very useful when it comes to container types and smart pointers, such as *Rc*. An *Rc* can be used as a standard reference in many cases. The same applies to *MutexGuard* etc. However, in some cases it may be required to first dereference and then reference in order to pass these types as function parameters which require references. Failing to do so can produce error messages that may look scary to beginners, however after getting familiar with the errors, it is clear what is wrong and why.

3.1.9 The borrow checker

The borrow checker does definitely prevent the programmer from writing potentially *unsafe* or incorrect programs. It is not possible to make any type of memory, error/corner case handling or race condition mistake. Preventing writing incorrect programs and helping writing correct ones are not equivalent things though. The errors from the borrow checker are in most cases quite clear, but avoiding them in the first place is easier. A few notes may help prevent usage of patterns that the borrow checker does not accept. Avoiding the following pitfalls will allow the borrow checker to focus its efforts on issues that the programmer may have not thought about, and help in writing correct programs.

Never write functions in Rust that take a reference to more data than they actually need. The author has found this to be the most common reason for borrow checker errors. One such example is taking an entire struct reference as a function parameter, and only operating on certain fields of the struct. This is the wrong design in Rust in almost all cases. Holding a reference to the entire struct will prevent any modification to any part of it. Instead, consider taking a reference to only the required field as a parameter.

Another possible source of borrow checker errors, is holding a reference to some element of a vector, and trying to modify some other element. The reference to one vector element was obtained by using a reference to the vector. Thus the entire vector is locked from modification. It is now not possible to modify another element of the vector at the same time with this method. To be able to solve this use case, a *split_at_mut* method exists, allowing to split a mutable reference to a vector into two mutable references at an index. It is now possible to modify one part of the vector based on a value in the other part.

A couple of cases exist where the author finds the borrow checker to be a bit picky. One such case is:

```
let mut v = vec![1,2,3];
v.push(v.len());
```

Attempting to compile it results in:

```
error[E0502]: cannot borrow `v` as immutable because it is
 ↪  also borrowed as mutable
 --> src/main.rs:3:8
  |
3 | v.push(v.len());
  | -      ^      - mutable borrow ends here
  | |      |
  | |      immutable borrow occurs here
  | mutable borrow occurs here
```

Intuitively, according to the author, *v.len*, should need to immutably borrow the vector, and then return a number. Then the immutable borrow should not live any more. After that, *v.push* should need a mutable borrow to the vector, which should not be a problem, because the immutable borrow should be dead. This is however not how the borrowing system works. The mutable borrow v.push is considered to live until the statement finishes, and within the statement, we attempt to immutably borrow v, which is not allowed.

Another example is to attempt to insert a key into a map if it does not exist, as follows:

```
let mut m = HashMap::new();
m.insert("a", "exists");

match m.get("b") {
  Some(text) => {},
  None => {
    m.insert("b", "did not exist. inserted.");
  },
}
```

Attempting to compile this example results in:

```
error[E0502]: cannot borrow `m` as mutable because it is
 ↪  also borrowed as immutable
  --> src/main.rs:9:5
   |
6  | match m.get("b") {
   |        - immutable borrow occurs here
...
9  |     m.insert("b", "did not exist. inserted.");
   |     ^ mutable borrow occurs here
10 |   },
11 | }
   | - immutable borrow ends here
```

Here we see the same type of issue. After checking if the key b existed or not, one might expect the borrow to be ”no longer needed”. The problem is that according to the borrowing rules, the borrow is still alive until the end of the match statement, thus this is not allowed.

The examples are constructed. One-liners exist in libstd to insert something into a map iff it does not exist. However, the same patterns occur quite often in self written code as well. The solution is always to assign a copy or a boolean to a temporary variable, let the borrow die, and then use the variable. While this works, the author feels that there is sometimes a mismatch between intuition and the borrowing rules. The concept of Non-Lexical Lifetimes (NLL), will likely solve many cases. It will basically introduce the possibility of pausing a lifetime in a conditional branch (such as the mach arm for None in the map example) [39].

3.1.10 Documentation

The Rust standard library is very well documented, and the documentation is available in HTML format, allowing easy navigation. It is possible to click on types anywhere in the documentation, and there are lots of usage examples present in throughout the docs.

Documentation can be auto generated from source code, and documentation generation can easily be invoked by calling *cargo docs*. The generated documentation is also searchable offline, which is very handy.

The documentation aspect of Rust leaves very little to be desired. It is very good in every regard in the eyes of the author.

3.2 Quality factors

As has previously been stated, many quality factors are of a subjective nature. The author aims to be as objective as possible.

3.2.1 Learning Rust

Rust itself is not very difficult to learn. The difficult part of learning Rust is fully understanding the implications of the borrowing and mutability rules [35, 31]. When learning Rust, the author would recommend to use cloning and copying as much as possible, and avoid the use of references. Remember the quote about premature optimization being the root of all evil. To learn Rust, it is also recommended to learn by heart which types deref into which, as it helps a lot when reading the documentation.

Very good learning materials exist, in different forms depending on preference. An official online book exists [53], where the user can complete tasks in the browser, and the code can be executed on the server, checking the answers. For those that prefer to learn by example, there are great materials oriented in this way as well [48].

For those that like to learn a new programming language by implementing data structures, a word of caution is in order. Implementing data structures in Rust with manual allocation and deallocation needs to be done using *unsafe* code. Writing this kind of code

correctly is not a Rust beginner friendly task. The standard library contains great data structures, and it exists for the very purpose of allowing the use of heap-allocated data in a completely safe manner, and to not have to re-invent the wheel (or rather the vector and map). The very well thought out, user friendly and well documented data structures also help to reduce fragmentation by implementing mostly every functionality one could have use for.

The author did have some limited experience using Rust before going into this project. This experience was attained by reading the formal grammar oriented The Rust Book, and the Learn Rust by example materials. Then the author proceeded to write programs keeping the usage of references to a bare minimum. This was done by cloning heap-allocated data and copying stack allocated data. When references had to be used, the author would make sure they live for as short amount of time as possible. The author was then able to write a web server implementing the websocket protocol in a day.

Having some experience with C++ and using constructors and destructors to manage heap allocation was of great help in understanding the memory management in Rust. Programmers with experience in C will likely not have any problems understanding how allocation and deallocation is managed in Rust, but it may be frustrating to deal with compiler errors related to borrowing rules. This is perhaps especially true when the programmer knows that the program is correct, but the compiler does not understand that. The fact that it is not allowed to do whatever one wants with pointers can take some time getting used to. However, many programmers report having less issues with borrowing over time [31, 46].

Being used to a high level language like Java or Python, the data structures will likely seem intuitive and easy to use. When borrowing can be avoided, in favor of copying or cloning, programmers with Java or Python backgrounds will likely not have any major problems learning Rust. The problem is that borrowing cannot be completely avoided. Reference counted smart pointers could likely help programmers with backgrounds from higher level languages, as using smart pointers in Rust feels similar to using a garbage collected language.

Experience with functional programming would likely help in learning Rust, as a lot of the iterator functionality is based around a functional approach. The *Option* and *Result* types are actually types of what is known as *Monads* in functional languages [65].

3.2.2 Productivity

Using borrowing and referencing does save some memory, and it does optimize performance, but it also does hurt productivity in the sense that re-factoring code is made more difficult because the borrowing system in Rust is so strict.

Therefore, having a master plan what data structures should own data, and where references should be used is a good starting point, but the author recommends just copying everything as far as possible, until performance becomes an issue, at which point references can be used to reduce copying.

Using this strategy allows re-factoring code without being hindered by ownership and borrowing issues, which may only affect an intermediate architecture as a re-factoring step. When the final architecture has been reached, the copying could be replaced with references, which may require further re-factoring to be legal.

The author believes that this strategy is in line with premature optimization being the root of all evil. If one adheres to this, productivity in Rust is definitely high.

That said, Rust offers absolutely phenomenal productivity tools. Especially the iterator implementation in Rust is just beautifully made. The combination of iterators and closures give a very good ability to formulate algorithms with very little numbers and symbols, focusing on what is important. During the porting project, the iterator *trait* is definitely the main reason why thousands of lines of code could be removed from the C implementation. Algorithms became much shorter, reformulated as sub-steps, and significantly easier to understand due to using *find*, *filter* and *map* instead of loops in loops with continue and break. Rust has very good support for functional programming, allowing mapping and folding and other operations to be trivially performed on iterators.

3.3 Porting

This section will provide some insight into areas of particular interest related to the ported code.

3.3.1 Porting methods

Different porting methods were used as detailed in section 2.4.5. The bottom up approach was used in the beginning, before full understanding of the architecture had been attained.

The main advantage of this method was that it could be done in very small steps, and was doable without full understanding of the code. Another advantage of this approach was that it allowed continuous testing.

The main disadvantage was lots of casting and type conversion code used in intermediate steps. This would result in incrementally deleting half of the work when parameter and return types were changed to use Rust versions instead. Using this approach, there was little possibility to change the architecture to better fit another language.

The author advises to not use this approach for the most part, but it was useful as a starting point.

The top down approach was used when it could be determined more or less exactly what the purpose of the code was, and there existed an obvious way to implement the same functionality in Rust, fundamentally different from the way it was done in C.

The advantage of this approach is that few intermediate steps were required. It also allows an idiomatic Rust architecture. In most cases, the code written using this porting method would stay mostly the same throughout the project.

The main drawbacks are that it requires the understanding of large chunks of code at a time, potentially thousands of lines of hash tables mapping void pointers to void pointers with different free functions. This was not always easy. It did result in difficult to locate runtime errors, because of very large increments at a time.

The top down approach was very efficient when it would be used. In most cases though, the code implementing some functionality would just be way to long to fully understand all at once. In these cases, the data flow analysis approach was used.

The best thing about this approach, of tracing the creation, use and freeing of structs, was that it allowed understanding which data structures stored instances of a particular

struct. Tracing the pointers of an instance through the code closely relates to the Rust concept of lifetimes, greatly aiding in avoiding borrowing problems. This approach would also get rid of allocation and deallocation code from the C code, so that the remaining code would be mostly logic oriented, gradually making porting easier.

The issues with this approach were largely in between those of the top down and bottom up approaches. Medium sized chunks of code were ported at a time, making it somewhat hard to locate errors because of preventing testing for some time. Problems would then arise when implementing several structs which reference each other in Rust. In these cases refactoring had to be employed in order to resolve any borrowing issues.

The top down approach was the most effective when it could be employed. When the amount of code was too vast to use the top down approach, the data flow oriented approach was used, which was also very effective. It turns out that refactoring from intermediate Rust using raw pointers to safe Rust is a lot easier than from C to Rust directly, so any re-factoring would not take too long to perform. This same observation has been made by others in the past, and there is even a project translating C to *unsafe* Rust to aid in this way [30]. This tool was not used, because a fully Rust idiomatic design was desired, and because of fear that hints about purpose might get lost in the translation process. The bottom up approach was really only useful as a starting point, because too much re-factoring would be required.

3.3.2 Global state mutex and callbacks

The original design made use of a global state mutex. This mutex would be locked each time anything touching the state was done. The Rust design also uses a global state mutex.

A minor issue occurs when using both mutex and callbacks with void pointers. The original design would receive a callback, and the data pointer to the callback would point to a session, which requires locking the mutex. This is incompatible with how Rust mutexes work. It is not possible (in safe Rust code) to have a pointer to data protected by a mutex without locking it.

Registering this callback, passing this pointer pointing inside the mutex, is thus impossible because it would outlive the MutexGuard. Some other method has to be used to resolve this conflict. Several options are possible. One possibility (the one chosen), is for the mutex to contain a map from token to session. It is then possible to pass a pointer to a string token as the callback data. The callback then proceeds to lock the mutex and access the session through the mutexed map using the token received in the callback data parameter.

3.3.3 Expiry timers

The original code kept sessions in memory longer than needed in many cases. The sessions were most likely being kept, because it was difficult to analyse if callbacks could refer to them after freeing them. A safety approach was being employed, letting the sessions expire through a timer, guaranteeing that no pointers to the sessions would be dereferenced any more. This illustrates that safety is top priority, no risks were taken.

The Rust design has remedied this, contributing to a cleaner design, seemingly without any side effects when running the test suite or swiping cards in reality. The Rust ownership

system, forbidding passing of pointers to mutex protected sessions guarantees that it will be safe.

This change has the great added benefit of simplifying data structures protected by the global state mutex, as only one session per id point (card reader and pin code terminal) needs to be kept.

3.3.4 Replacing loops with iterator functions

A lot of code in the original implementation consisted of long loops, incrementing the index in several places, sometimes containing inner loops, etc. Decoding what was actually the point of such loops, and formulating it in terms of "find this, remove that, modify the others, and map to something else", allowed the use of Rust iterators and a functional programming approach.

The original loops were very clever in many ways, but understanding that took some time in some cases. Expressing it in terms of iterator functions greatly increased readability, both in terms of being much shorter, and more expressive. Some minor bugs in the original complex code were found, but none were security critical, or memory related.

3.3.5 Separation of operations

Original loops would sometimes perform several operations at once, such as potentially deleting some elements and modifying others from a list.

This is not compatible with how the standard library is structured. An iterator iterating a vector with a mutable reference to the current element is locking the vector from modification (the vector is borrowed mutably by the iterator), preventing deletion of elements.

Such algorithms would have to be formulated as first, modify the elements which match some predicate, and then delete those that match some other predicate. This does aid in readability, as it separates concerns.

A quick test was carried out to find out weather this would incur a performance penalty. A method for deleting elements from a vector while iterating it was implemented. A benchmark was constructed, and run on the two implementations compiled in debug mode. There was a large performance difference. The compilation was then repeated in release mode (with optimization turned on). With optimizations turned on, there was no difference in performance, and both implementations were considerably faster than their unoptimized versions.

3.3.6 Separation of input, output and state

The original C code would put both input data pointers, output data pointers, and state pointers in the same struct. Most likely this was done because it was easier to pass around.

This is incompatible with how Rust is designed. If the output should be set based on the input, a reference to the input would lock the struct containing all of input, output and state from modification. Presence of an immutable reference forbids modification.

Instead the input had to be passed as parameters, the output had to be returned, and the state was the only data remaining in the struct. This is without a doubt a more logical

architecture.

This pattern occurred in several places. In one case dangling pointers were left behind to deallocated input. They were never dereferenced, so no safety issues were present. This illustrates that Rust can aid in correctness in this regard. The improved readability thus comes at no cost.

3.3.7 Set operations

Rust provides sets and set operations such as checking for subsets etc. Some algorithms were heavy on duplicates checking, and a lot of code related to this could also be removed from the original code by using set types. The PACS has a concept called further access possible, meaning that one door could open when swiping a card, but another door could potentially also open, if a pin code is entered. This could be expressed as if opened doors is a subset of doors that could open, then further access is possible. This rewrite removed hundreds of lines of code.

3.3.8 Replacing null pointers with enums

The decision making logic would return a variant of a C enum. This enum had very many possible variants. The call would also return (through the use of pointer pointer parameters), different data depending on which enum variant was returned. This additional data could be to which doors access was granted, which doors could also be granted with a pin code, reasons for denying the request, or lists of id points to show feedback on, to name a few.

Rust's enum type made it possible to bundle the appropriate data in each variant, largely cutting out the need for optional data. The call now takes a single parameter (the id data for a person requesting access), and returns an enum. This call used to have 8 pointer pointer parameters, which would be passed on by more function calls. This change increased the readability of the code by a very significant amount by removing all these pointer pointer parameters. Performing this change also removed countless assert not null checks. Both for pointers, and for the pointers pointed by pointers.

3.3.9 Resolving segmentation faults

The author is not perfect, unfortunately. On some occasions the author called the wrong free function in *unsafe* code. Other times the author assumed the wrong types of void pointers, also in *unsafe* blocks.

When segmentation faults occurred, the region where the error was likely to be could be identified easily by looking at the last logged information. Because of a practise of continuously compiling and testing and committing to git, what had recently been changed could also be used as an indicator of the location of the error.

Because of very restrictive use of *unsafe* code, due to converting almost all types as early as possible, very little code needed to be analysed to find the problem. There was never really any need to fire up a debugger, the error could always be located very quickly.

3.3.10 Resolving deadlocks

The C code of the database implementation would occasionally lock the state mutex. This resulted in deadlocks. The Rust code had to lock the mutex, in order to refer to the data contained, and then the C database functionality would attempt to lock the same mutex again.

After some analysis, the locking performed by the database could be entirely removed, as in all instances, it could be concluded that in order to execute that database code, Rust would have already locked the mutex in all possible code paths.

Rust does not promise the absence of deadlocks. Making sure deadlocks cannot occur is the responsibility of the programmer. This design choice was made as preventing deadlocks was deemed to make the language too difficult to use. Deadlocks can in many cases be avoided by locking mutexes in the same order in all places where multiple mutexes are used, and holding them for the shortest possible amount of time. Since the Mutex type in Rust is implemented using pthread mutex, it should be no problem to use helgrind to find the causes of deadlocks.

3.4 Building and Compiling

3.4.1 Continued support for custom target

The build system used to build the Rust compiler has recently changed, from a Makefile based system to a Python based system. The instructions given in appendix B apply to the new build system. At the time of the change, there was little information, and figuring out how to repair the compiler build system for the custom target did take a workday. The Rust developers have no responsibility for making sure that the compiler builds effortlessly for unsupported targets. Even without documentation, it was not very hard or time consuming to have it all back and working again after the massive change. The working solution was very simple, only digging for the information took some time. The change of build systems was clearly motivated, as the new build system is noticeably faster than the old one. Once this new build system reaches the stable channel, official instructions are likely to surface.

3.4.2 Using the compiler

Using this unsupported target posed no problems at all. Every binary produced by the custom cross compiler worked as intended. The compiler did not crash or misbehave once.

3.4.3 Linking with a Makefile based build system

Two different methods of linking Rust and C code were tested. Specifying the needed C libraries in Rust with link directives worked effortlessly.

However, the complex initialization code and forking done by the C program was deemed not worthwhile to port to Rust, it ended up being easier to build a Rust static library and simply link it to the C binary. No problems were encountered.

Chapter 4
Conclusions

4.1 Summary

All in all, using Rust for a larger project has been a very pleasant experience.

The language offers a great standard library, allowing fast prototyping and functional programming. Very good implementations of data structures are provided. The Rust versions of enum (allowing variants to contain values), struct (allowing easy initialization and instance methods), traits (similar to interfaces in other languages) and tuples (allowing multiple return values) are very easy to work with. These factors have all been found to be associated with great productivity.

The documentation is great: searchable, easy to navigate, with many examples and detailed explanations.

The language, and the standard library are safe to the core. This safety is implemented by means of zero-cost abstractions. The type system, borrowing system and the standard library interact in a very well designed way. Rust definitely aids programmers in writing correct programs.

Rust uses the borrowing system to guarantee safety, and is thus very strict about how references (pointers) are used. The learning curve associated with using borrowing correctly has been found to be a bit steep. One out of four who replied to the State of Rust Survey 2016 [31] commented on a steep learning curve. This could potentially be remedied by education. Having a Rust expert at hand to ask would likely improve the learning curve associated with the borrowing system considerably.

Very good learning materials are available, and the author learned the language very well in a week. This was done by avoiding borrowing as much as possible, and instead copying and cloning, as the author knew that borrowing is considered difficult. Borrowing could then be introduced and learned through optimization. Learning Rust in this way is recommended by the author. If one does make mistakes with borrowing rules, the Rust compiler gives very user friendly error messages.

Building a compiler from source, and adding an unofficial target was not very difficult. Cross compiling has been completely problem free. The compiler has been rock solid. It never crashed or misbehaved.

Integrating Rust with a Makefile based build system has worked without issues. Building with Rust takes a few seconds longer, but the Rust compiler needs to perform safety analysis.

Interfacing with C libraries has been found to be easy, especially when bindings are available, as is the case for Glib. When bingings are not available, such as for proprietary libraries, bindgen [15] has been of great help translating C header files to Rust. There is very little effort required to call C libraries. Interfacing with callbacks, Glib, D-Bus and proprietary libraries has been found to be easy, especially after designing wrappers, limiting *unsafe* code to an absolute minimum, and concentrating it to few places.

Dealing with C strings is not technically difficult, but it is cumbersome, and requires *unsafe* code. Lots of error checks have to be performed, and thus, converting C strings to Rust strings as early as possible is recommended. Implementing a wrapper is advised if dealing with a lot of C strings.

Porting pointer-heavy C code to Rust is a challenge, as the borrowing rules can be restrictive. Therefore, having the overall architecture thought out beforehand, and using copying and cloning when porting is advised. This copying and cloning can then be re-factored to instead use references where possible. A top down approach is recommended when documentation is available, or the purpose of the code can be easily determined. When this is not the case, call graphs and tracing pointers through the code to find out where they are stored is of great aid. The approach of porting a struct along with related functionality at a time, and then using the Rust struct from C, worked well.

Porting the major part of pacsd was a success story. The Rust language contributed to significantly shortening the code. Formulating algorithms in terms of iterators and functional programming provided great benefits in terms of readability. Similar benefits could have been attained with other languages, but Rust really shines in interoperability with C. The Rust data structures such as vecors, maps and sets, and the enum type were of great aid in achieving a logical architecture. The borrowing system allowed performing re-factoring, without fear of accidentally making pointer related mistakes.

Benchmarking the Rust binary, performance decreased by 14% and maximum memory usage increased by 7% (3MB) respectively. This includes a worst case which takes a considerable hit from type conversion and computing unused results. This could have been optimized, but improved performance was not an outset goal. Comparing the performance, excluding the worst case, the Rust implementation was 9% faster. The binary size went up 36% (223kB), and has no added runtime dependencies. Rust performance and memory usage has been found by the Debian benchmarksgame [12] to be competitive to that of C++, but not quite as good as C on x86_64. The results of the performed port on MIPS seem to be in line with this.

As for tooling, Cargo provides tools for testing, benchmarking, dependency management and building, which are excellent and closely integrated and bundled with Rust.

Wrapper scripts for easy interfacing with GDB and LLDB are also bundled, but require recent versions of the respective debuggers to be installed. The debugging experience with rust-gdb is excellent on x86_64, complete with pretty printers for Rust data structures, and ability to run a subset of Rust directly in GDB. Multi threaded debugging worked very

well.

Tools for basic profiling and test line coverage analysis are available.

As for IDEs, IntelliJ Rust currently provides a better experience than Eclipse RustDT according to the author. The IDEs offer nowhere near the functionality offered for say, Java, such as advanced re-factoring capabilities, but they still provide a much richer experience than a text editor. Development activity in this department is high, and the experience is continuously improving.

The tools feel mature enough for production use, even though, the IDEs may not yet be full featured.

Rust is open source, community developed, and sponsored by Mozilla. Rust comes with a permissive license allowing proprietary use without mandatory compensation. Rust also makes assurances about backwards compatibility and portability.

4.2 Future research

The author possesses extensive experience with using Python and Java. Rust provides similar high level functionality. The author also has quite some experience using C and C++, and the memory model in Rust resembles RAII in C++. Parallels would be drawn to all of these languages when learning Rust.

It is therefore difficult to draw conclusions about how experience with a particular language affects the learning curve when studying Rust.

The State of Rust Survey 2016 [31] contained the question ”What programming languages are you most comfortable with”. The results were (most common answer first): Python, C, C++, Java and JavaScript with similar popularity, followed by other languages far behind. Reading this source, it is a bit unclear if the people who answered were using Rust or not. Also, this result does not reveal how many aborted learning at an early stage from each language.

A learning study would be very interesting to see, where programmers with different backgrounds would learn Rust, measuring the progress for different groups.

It could also be very interesting to see how different learning approaches, such as courses, grammar studies, learning by example and learning by trial and error would affect the learning curve.

Because this thesis was written by a single author, and the code porting was conducted by the author only, no conclusions can be drawn with regards to team-working in Rust. It would be of interest to see a future study on how Rust affects the ability to work as a team.

Because of issues building a recent version of GDB for the target architecture (which has nothing to do with Rust), the author would like to know if debugging on the target architecture would pose any problems, or if it would be as easy as on x86_64.

As Rust with tooling is evolving and improving at a steady pace, similar future studies could also be performed. Preferably, case studies porting other code bases, as this study ported parts of a specific Linux daemon.

Bibliography

[1] Alan Perlis. Epigrams on Programming. 1982.

[2] Aleksey Kladov, Alexey Kudinkin. IntelliJ Rust. https://intellij-rust.github.io/docs/.

[3] Apple. Swift. https://developer.apple.com/swift/.

[4] Axis Communications. Physical Access Control. http://www.axis.com/global/en/products/access-control.

[5] B. Kernighan and D. Ritchie. The C Programming Language. Prentice Hall, 1988.

[6] Bjarne Stroustrup. Foundations of C++. 2012 In: Seidl H. (eds) Programming Languages and Systems. ESOP 2012. Lecture Notes in Computer Science, vol 7211. Springer, Berlin, Heidelberg.

[7] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[8] Bruno Medeiros. Eclipse RustDT. https://github.com/RustDT/RustDT.

[9] Bruno Medeiros. Rainicorn. https://github.com/RustDT/Rainicorn.

[10] Cppreference. C keywords. http://en.cppreference.com/w/c/keyword.

[11] Cppreference. RAII. http://en.cppreference.com/w/cpp/language/raii.

[12] Debian. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/.

[13] Dimitri van Heesch. Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[14] Donald Knuth. Structured Programming with Goto Statements. Computing Surveys 6:4 (December 1974), pp. 261–301, §1, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6084.

[15] Emilio Cobos Álvarez, Jyun-Yan You and Nick Fitzgerald. bindgen. https://github.com/servo/rust-bindgen.

[16] Fredrik Pettersson and Erik Westrup. Using the Go Programming Language in Practice. Master's Thesis, Lund University, 2014-06-05, ISSN 1650-2884.

[17] Free Software Foundation. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[18] freedesktop. D-Bus. https://www.freedesktop.org/wiki/Software/dbus/.

[19] Freedesktop. GDB Change log. https://dbus.freedesktop.org/doc/dbus-glib/.

[20] GNOME. GLib Reference Manual. https://developer.gnome.org/glib/.

[21] GNOME. Librsvgl. https://github.com/GNOME/librsvg.

[22] GNU. D-Bus GLib bindings - Reference Manual. https://www.gnu.org/software/gdb/news/.

[23] Google. The Go Programming Language. https://golang.org/.

[24] gtk-rs. glib-sys. https://github.com/gtk-rs/sys.

[25] Hoare, Tony (25 August 2009). Null References: The Billion Dollar Mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

[26] IBM. Minimizing code defects to improve software quality and lower development costs. ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf.

[27] IOCCC. The International Obfuscated C Code Contest. http://www.ioccc.org/.

[28] ISO. ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. https://www.iso.org/standard/35733.html.

[29] ISO. ISO/IEC 9899:TC2. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[30] Jamey Sharp. Corrode: Automatic semantics-preserving translation from C to Rust. https://github.com/jameysharp/corrode.

[31] Jonathan Turner. State of Rust Survey 2016. https://blog.rust-lang.org/2016/06/30/State-of-Rust-Survey-2016.html.

[32] L. Flon. On research in structured programming. ACM SIGPLAN Notices, 10(10):16-17, Oct. 1975.

[33] Linus Thorvalds et al. The Linux Kernel Archives. https://www.kernel.org/.

[34] LLVM Developers. The LLDB Debugger. https://lldb.llvm.org/.

[35] Mathieu De Coster. Fighting the Borrow Checker. https://m-decoster.github.io/2017/01/16/fighting-borrowchk/.

[36] Michael Larabel. GNOME's SVG Rendering Library Migrating To Rust. https://www.phoronix.com/scan.php?page=news_item&px=librsvg-2.41-Rust.

[37] Mozilla. Preprint: Engineering the Servo Web Browser Engine using Rust. https://raw.githubusercontent.com/larsbergstrom/papers/master/icse16-servo-preprint.pdf.

[38] NASA. Error Cost Escalation Through the Project Life Cycle. http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf.

[39] Nicholas D. Matsakis. Non-lexical lifetimes: adding the outlives relation. http://smallcultfollowing.com/babysteps/blog/2016/05/09/non-lexical-lifetimes-adding-the-outlives-relation/.

[40] Nick Cameron, Marcus Klaas de Vries. Rustfmt. https://github.com/rust-lang-nursery/rustfmt.

[41] Olga Antropova, Lisa Hollermann, Hon-Chi Nga and Praveen Sharma. Comparing Programming Languages. http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/comparing/hw0_2.html.

[42] Oracle. ConcurrentModificationException. https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html.

[43] Phil Dawes. Racer. https://github.com/phildawes/racer.

[44] Rich Felker. musl C standard library. https://www.musl-libc.org/faq.html.

[45] Rust Community. Cargo Guide. http://doc.crates.io/guide.html.

[46] Rust Community. Rust Book - Ownership. https://doc.rust-lang.org/book/ownership.html.

[47] Rust Community. Rust Book - Strings. https://doc.rust-lang.org/book/strings.html.

[48] Rust Community. Rust by Example. http://rustbyexample.com/.

[49] Rust Community. Rust FAQ. https://www.rust-lang.org/en-US/.

[50] Rust Community. Rust keywords. https://github.com/rust-lang/rust/blob/master/src/libsyntax/symbol.rs.

[51] Rust Community. Rust website. https://www.rust-lang.org/en-US/.

[52] Rust Community. rustup. https://www.rustup.rs/.

[53] Rust Community. The Rust Programming Language. https://doc.rust-lang.org/book/.

[54] Rust Community. Tutorial: How to collect test coverages for Rust project. https://users.rust-lang.org/t/tutorial-how-to-collect-test-coverages-for-rust-project/650.

[55] Serdar Yegulalp. Mozilla binds Firefox's fate to the Rust language. http://www.infoworld.com/article/3165424/web-browsers/mozilla-binds-firefoxs-fate-to-the-rust-language.html.

[56] Simon Kagstrom. kcov. https://github.com/SimonKagstrom/kcov.

[57] Stephen O'Grady. The RedMonk Programming Language Rankings: January 2017. https://redmonk.com/sogrady/2017/03/17/language-rankings-1-17/.

[58] Suchin Gururangan. cargo-profiler. https://github.com/pegasos1/cargo-profiler.

[59] The Rust Core Team. Announcing Rust 1.0 Alpha. https://blog.rust-lang.org/2015/01/09/Rust-1.0-alpha.html.

[60] Tutorialspoint. C - Error Handling. https://www.tutorialspoint.com/cprogramming/c_error_handling.htm.

[61] Tutorialspoint. C++ Copy Constructor. https://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm.

[62] Urban Müller. Brainfuck. http://www.muppetlabs.com/~breadbox/bf/.

[63] Valgrind developers. Valgrind. http://valgrind.org/.

[64] Wikipedia. Green Threads. https://en.wikipedia.org/wiki/Green_threads.

[65] Wikipedia. Monad. https://en.wikipedia.org/wiki/Monad_(functional_programming).

[66] Wikipedia. Resident set size. https://en.wikipedia.org/wiki/Resident_set_size.

[67] Yamakaky, Brian Anderson. error-chain - Consistent error handling for Rust. https://docs.rs/error-chain/0.10.0/error_chain/.

All on-line references accessed and verified still valid as of 2017/05/11.

Appendices

Appendix A

A longer Rust example

The following example code implements a multi threaded, zero copy web server, with request line and headers parsing, in less than 100 lines including comments. It is hard coded to reply ”It works!” if requesting ”/” or otherwise send HTTP 404 Not Found. The code avoids error handling to mainly illustrate memory management. It also shows off some iterator functionality and closures (like lambda functions in C++).

```
use std::io::*;
use std::net::*;
use std::collections::*;
use std::thread;

/// main entry point for lazy multithreaded no copy webserver.
fn main() {
    let listener: TcpListener =
        TcpListener::bind("0.0.0.0:8080").expect("could not listen on port 8080.
↪   crashing.");
    println!("awainting connections on port 8080");
    loop {
        let (socket, addr): (TcpStream, SocketAddr) =
            listener.accept().expect("could not accept connection. crashing.");
        println!("connection from {}", addr);
        // spawn expects a closure (like a lambda function in C++). move indicates
↪   moving
        // ownership of any referenced variables to the closure, which will run on a new
↪   thread.
        thread::spawn(move || serve(socket)); // Quit the thread after this call.
    }
}

/// Expects to receive a valid http request of max 1kB and replies / => "it works" or
↪   not found.
/// If anything goes wrong it causes a panic. Run on new thread to not crash main loop.
fn serve(mut socket: TcpStream) {
    let mut buffer = [0; 1024]; // we assume that request will fit in 1kB of zeroed stack
↪   memory
    let (_read, header_len) = HttpRequest::read(&mut socket, &mut buffer);
    let http_request = HttpRequest::parse(&buffer[0..header_len]);
    println!("processing request: {:?}", http_request);
    // we should now read the rest of the body if method is POST, or do Websocket
↪   handshake
    if http_request.path == "/" {
```

```
        socket.write(b"HTTP/1.0 200 OK\r\n\r\nIt works!").expect("write error");
    } else {
        socket.write(b"HTTP/1.0 404 Not Found\r\n\r\n").expect("write error");
    }
} // Drop socket, causing it to close the tcp connection gracefully.

/// A HttpRequest can be parsed from a buffer without copying anything.
/// All fields will reference the data in the buffer which was used to parse it.
/// Thus the lifetime, 'a will be the lifetime of the buffer passed to the parse
↪ function.
#[derive(Debug)]
struct HttpRequest<'a> {
    method: &'a str,
    path: &'a str,
    version: &'a str,
    headers: HashMap<&'a str, &'a str>
}

impl<'a> HttpRequest<'a> {
    /// Reads a request from a stream to a buffer. This guarantees to read request line
↪   and headers,
    /// and may also read part of the body. This is a static function, not an instance
↪   method.
    /// Returns bytes read and the index of the first byte following the header.
    /// The buffer is a slice, represented as a pointer and a length, and cannot overflow.
    fn read(socket: &mut TcpStream, buffer: &mut [u8]) -> (usize, usize) {
        let mut read = 0;
        loop {
            // each iteration reads more from socket and stops if \r\n\r\n found.
            let from = if read >= 4 { read - 4 } else { 0 }; // look for \r\n\r\n from
↪   this index.
            let num_new = socket.read(&mut buffer[read..]).expect("read fail"); // from
↪   read to end
            if num_new == 0 {
                panic!("Read EOF before end of http headers or buffer is full.");
            }
            read += num_new;
            // create sliding window of 4 bytes into buffer, include 4 bytes of old data
↪   if any.
            let found_end = &buffer[from..read].windows(4).position(|t| t ==
↪   b"\r\n\r\n");
            match *found_end {
                Some(position) => return (read, position),
                None => {}
            }
        } // read some more
    }

    /// Parses Http request from request_bytes (pointer and length). The data should
↪   contain
    /// request line and headers but should NOT contain final \r\n\r\n.
    /// The returned HttpRequest references data in the buffer, and thus locks the buffer
↪   from
    /// modification until the returned HttpRequest is dropped.
    fn parse(request_bytes: &[u8]) -> HttpRequest {
        let mut lines = request_bytes.split(|e| *e == b'\n');
        let mut request_line = lines.next().expect("no request line").split(|e| *e == b'
↪   ');
        let mut headers = HashMap::new();
        for line in lines {
            // the rest of the lines are headers
            let pos = line.iter().position(|e| *e == b':').expect("no colon");
            let key = std::str::from_utf8(&line[0..pos]).expect("utf8 err");
            let val = std::str::from_utf8(&line[pos + 2..line.len() - 1]).expect("utf8
↪   err");
            headers.insert(key, val);
        }
        let mut r = HttpRequest {
            method: std::str::from_utf8(request_line.next().expect("no
↪   method")).expect("utf8 err"),
```

```
            path: std::str::from_utf8(request_line.next().expect("no
↪   path")).expect("utf8 err"),
            version: std::str::from_utf8(request_line.next().expect("no
↪   ver")).expect("utf8 err"),
            headers: headers
        };
        r.version = &r.version[0..r.version.len() - 1];// remove \r at end of request
↪   line
        r // return value. only early returns need return keyword
    }
}
```

Running the program and connecting with Firefox produces the following output:

```
awainting connections on port 8080
connection from 127.0.0.1:51587
processing request: HttpRequest { method: "GET", path: "/", version: "HTTP/1.1",
↪   headers: {"Accept-Language": "en-US,en;q=0.5", "User-Agent": "Mozilla/5.0 (Windows
↪   NT 10.0; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0", "Cache-Control": "max-age=",
↪   "Accept-Encoding": "gzip, deflate", "Host": "127.0.0.1:8080", "Accept":
↪   "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
↪   "Upgrade-Insecure-Requests": "1", "Connection": "keep-alive"} }
```

Appendix B

Changes made to Rust compiler to support target platform

To build a cross compiler to support for the custom target platform, follow the following instructions:

1. Install build dependencies (for Debian Jessie, released in 2015, only a more recent cmake had to be compiled from source). Make sure needed linker is available in PATH. Set proxy environment variables if needed for git.

2. Get rustc source code from git

   ```
   $ git clone https://github.com/rust-lang/rust
   ```

3. Add a file defining the custom target *mipsisa32r2el-axis-linux-gnu*, as detailed in appendix B.1 and add the target string to a list of supported targets.

   ```
   $ git am --3 add_axis_target_for_1.14.0.patch
   ```

4. Configure the build for the added target platform:

   ```
   $ ./configure --target=mipsisa32r2el-axis-linux-gnu --disable-docs --enable-ccache
   ```

5. Define the appropriate CC environment variable to point to the needed linker for the target architecture and invoke the new Python based build system:

   ```
   $ CC_mipsisa32r2el_axis_linux_gnu=mipsisa32r2el-axis-linux-gnu-gcc ./x.py build
   ↪  --target=mipsisa32r2el-axis-linux-gnu -j8 ./x.py dist
   ```

6. Install the new compiler in `/usr/local/bin/` by issuing:

   ```
   $ sudo make install
   ```

These steps will build both the compiler and the standard library for the target platform. If an x86_64 targeting compiler and standard library are also desired, they can be built in the same way, just by not passing the target flag.

The the build step outputs binary packages, which can be used to install the compiler and standard library on several computers. The installation step just basically extracts the binary packages.

The Rust compiler itself is written in Rust and is entirely self-hosting. It builds using the latest version of Rust, for which it downloads a binary distribution during the build process. The file defining the custom target contains only standard Rust code.

B.1 Target definition Rust code

To add ruport for a the Axis Rust target, the file
src/librustc_back/target/mipsisa32r2el_axis_linux_gnu.rs
should be created, with the contents:

```
use target::{Target, TargetOptions, TargetResult};

pub fn target() -> TargetResult {
    Ok(Target {
        llvm_target: "mipsel-unknown-linux-gnu".to_string(),
        target_endian: "little".to_string(),
        target_pointer_width: "32".to_string(),
        data_layout: "e-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64".to_string(),
        arch: "mips".to_string(),
        target_os: "linux".to_string(),
        target_env: "gnu".to_string(),
        target_vendor: "axis".to_string(),

        options: TargetOptions {
            cpu: "mips32r2".to_string(),
            features: "+mips32r2".to_string(),
            max_atomic_width: Some(32),
            ar: "mipsisa32r2el-axis-linux-gnu-ar".to_string(),
            linker: "mipsisa32r2el-axis-linux-gnu-gcc".to_string(),
            ..super::linux_base::opts()
        },
    })
}
```

and then the target needs to be added to the list of supported targets in
src/librustc_back/target/mod.rs
by adding the line marked with +:

```
    supported_targets! {
        ("x86_64-unknown-linux-gnu", x86_64_unknown_linux_gnu),
+       ("mipsisa32r2el-axis-linux-gnu", mipsisa32r2el_axis_linux_gnu),
```

Appendix C
Makefile integration

The following Makefile rules were used to automatically compile and link Rust code. *libpacsdrs.a* is the output from compiling the *pacdrs* Cargo project folder.

```
libpacsdrs.a: $(shell find pacsdrs/ -type f -name '*.rs') pacsdrs/Cargo.toml
	rm -f libpacsdrs.a
	cd pacsdrs && cargo build --target=mipsisa32r2el-axis-linux-gnu --release
	cp pacsdrs/target/mipsisa32r2el-axis-linux-gnu/release/libpacsdrs.a libpacsdrs.a

pacsd: $(OBJS) libpacsdrs.a
```

First, a search is done for all Rust source code files, which end with *.rs*. Then *libpacsdrs.a* is made depend on all of those, and the Cargo configuration file *Cargo.toml*.

Appendix D
List of Changes

Since 2017/03/10

- Initial draft.
- Combined several documents.
- Made sure information is up to date.
- Reduced repetition, and implementation details.
- Moved references from in-line to bib index.

Since 2017/03/25

- Inserted performance results.
- Inserted memory results.
- Inserted binary size results.

Since 2017/04/21

- Approved by Axis for handing over to LTH.
- Made changes and corrections proposed by Axis supervisors.

Since 2017/05/11

- Looked up authors of references.
- Spelling and grammar corrections.

- Moved details to appendices.

Since 2017/05/19

- Added reference and paragraphs about non lexical lifetimes planned improvement.
- Updated compiler build instructions in appendix with details.
- Updated an example to reflect newer version of Rust.
- Inserted missing code into existing appendices.
- Removed paragraph about ”small performance gains”, that should have been removed, but was forgotten. The ”standard use case” sees peformance gains, but there exists a worst case which is slower.
- Minor spelling corrections.
- Referenced billion dollar mistake.

Since 2017/06/13

- Referenced various mentioned open source projects
- Corrected some references (versions, etc)
- Lots of small typo corrections
- Clarifications
- Updated info in tables