TÉCNICO LISBOA

Automatic Translation of C++ to Rust

Henrique Maria Almeida Rocha Preto

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Nuno Lopes
Eng. Dmytro Hrybenko

Examination Committee

Chairperson: Prof. Andreas Miroslaus Wichert
Supervisor: Prof. Nuno Lopes
Member of the Committee: Prof. José Faustino Fragoso Femenin dos Santos

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my thesis supervisors, who were always available and provided valuable guidance. My family, for their unconditional support and motivation, giving me all the conditions to complete my dissertation. Finally, my friends, without whom it would have been much harder.

Abstract

Memory safety bugs are a critical class of software security vulnerabilities, with 70% of security vulnerabilities in major Microsoft and Google projects stemming from these bugs. C and C++, while not memory safe, are efficient languages commonly used to build complex and critical systems software. Rust is a memory-safe programming language that offers comparable performance to C and C++. Other memory-safe languages such as Java and Go can also achieve similar performance, but these languages use garbage collection (GC) for memory management, which can lead to issues such as low memory efficiency and unpredictable delays. Rust is more efficient as it uses an ownership system, rather than GC. Rust is becoming more popular as a programming language for implementing software systems such as web browsers, operating systems, and garbage collectors. However, manually rewriting older software systems in Rust is not practical since these kind of efforts are expensive and do not scale well. One approach is to automatically translate some or all of the C++ code to Rust, which would make the software safer while preserving performance and efficiency.

In this work we implement an automatic source-to-source compiler that converts C++ code into safe Rust. A specific subset of C++ is selected for automatic conversion, extensive enough to fully support the automatic translation of a chosen real C++ codebase (WOFF2 library). Since C++ is not memory-safe as Rust is, we design various translation models. We evaluate their performance and the quality of the generated code. More specifically, we develop a translation model based on reference counting for the direct conversion of C++ into safe Rust. Finally, a translation pipeline is proposed, including refactoring passes to optimize and enhance the initially generated Rust code.

Keywords

C++; Rust; Automatic Translation; Memory-Safety.

Resumo

Os *bugs* relacionados com a segurança de memória constituem uma classe crítica de vulnerabilidades em segurança de software, sendo 70% das vulnerabilidades de segurança encontradas em projetos da Microsoft e do Google relacionadas com memória. O C e o C++, embora não ofereçam segurança de memória, são linguagens de programação eficientes frequentemente usadas para desenvolver software de sistemas complexos e críticos. Rust é uma linguagem de programação com segurança de memória, oferecendo desempenho comparável ao de C e C++. Existem outras linguagens que oferecem segurança de memória, como Java e Go, que também podem atingir desempenho semelhante, mas essas linguagens utilizam *garbage collection* (GC) como mecanismo de gestão automática de memória, o que pode resultar em problemas como utilização ineficiente de memória e atrasos na execução imprevisíveis. Rust é mais eficiente, pois utiliza *ownership* como mecanismo de gestão automática de memória em vez de GC. No entanto, reescrever manualmente na totalidade sistemas de software antigos em Rust não é prático, uma vez que este tipo de esforço é dispendioso e não escalável.

Neste trabalho, implementamos um transpilador automático para converter código C++ em Rust seguro. Um subconjunto específico de C++ é selecionado para a conversão automática, abrangendo o suficiente para suportar integralmente a tradução automática de uma biblioteca real de C++ (biblioteca WOFF2). Como o C++ não oferece segurança de memória como o Rust, vários modelos de tradução são apresentados, e o seu desempenho bem como a qualidade do código gerado são avaliados. Mais precisamente, um modelo de tradução baseado em *reference counting* é desenvolvido para a conversão direta de C++ para Rust seguro.

Palavras Chave

C++; Rust; Tradução Automática; Segurança de Memória.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

Contents

Nowadays, the use of digital technology is increasing and is part of everyday life for many people. Software plays a major role in critical sectors such as in transport, energy, health and finance. Businesses increasingly rely on digital technology to deliver services, making software even more ubiquitous.

However, the increased reliance on software also leads to vulnerabilities such as security threats and inefficiency. The ongoing software revolution, although offers solutions to the various challenges facing the world, it also exposes the society to cyberattacks. In 2021, ransomware attacks alone cost an estimated $623.7 million[1] and hit critical national infrastructure.[2] Additionally, in 2021 nearly 20,000 new software security vulnerabilities were identified, more than double than in 2016.[3]

The advent of technologies like cloud computing and the Internet of Things (IoT), while making society more connected and open than ever before and allowing various industries unprecedented growth and progress, it also opened up vast opportunities for cybercriminals, coming up with more and more

[1] https://www.emsisoft.com/en/blog/41518/ransomware-cost-us-local-governments-623-million-in-2021-but-fewer-incidents-in-2022/

[2] https://www.cisa.gov/uscert/ncas/alerts/aa22-040a

[3] https://www.cve.org/About/Metrics

sophisticated cyberattacks. Therefore, certain industries, specifically in critical infrastructure where software is becoming increasingly complex, acknowledge the importance of taking comprehensive security measures and make huge investments to protect themselves from major disasters.

Even so, as cyberattacks evolve and become more complex, software security teams in many organizations struggle to keep up and develop proper strategies to prevent them. For example, the Heartbleed Bug, a memory-related security vulnerability discovered in 2014 in OpenSSL (a popular cryptographic software library), granted anyone access to the systems memory, compromising confidential data such as secret keys, usernames and passwords.[4]

Memory safety bugs are a specific and critical class of software security vulnerabilities. About 70% of security vulnerabilities present in major Microsoft [2] and Google [3] projects stem from software memory safety bugs. And, the National Security Agency (NSA) has recently published a report about poor memory management issues which can dangerously be exploited by cybercriminals, for example, to gain unauthorized access to confidential information or execute malicious code [4].

C and C++ are programming languages that are not memory safe. However, these languages are capable of producing highly efficient code. Consequently, C and C++ have been extensively utilized in the development of critical systems software, including operating systems and web browsers, that require complex and low-level programming. Nevertheless, it is undeniable that many efforts have been made to make systems software implemented in C and C++ safer, such as the use of software engineering best practices, like Secure by design[5], or the integration of software analysis tools to automatically detect security vulnerabilities in programs, like the Scudo hardened allocator[6] or AddressSanitizer[7].

However, this is not enough since there are still many cyberattacks coming from exploiting critical memory-related bugs in programming languages like C and C++. Therefore, there is an increasing need to design software tools that truly and effectively eliminate memory-related bugs in systems software [5]. Rust [6] is one such tool: a memory safe programming language that offers code as fast and as efficient as C++ [7].

Indeed, there other are memory safe programming languages, such as Java and Go, which are also able to reach performance levels similar to those of Rust, C and C++ [8]. However, these programming languages use garbage collection (GC) in order to have an automatic and safe memory management mechanism, but which unfortunately introduces multiple problems like low memory efficiency and high nondeterministic delays in programs. In contrast, Rust uses an ownership-based memory management mechanism that has no overhead compared to manual memory management.

For these reasons, these languages are not as efficient as Rust. And, in some domains they cannot

[4] https://heartbleed.com/

[5] https://joinup.ec.europa.eu/collection/common-assessment-method-standards-and-specifications-camss/solution/elap/security-design

[6] https://llvm.org/docs/ScudoHardenedAllocator.html

[7] https://clang.llvm.org/docs/AddressSanitizer.html

be used, such as in embedded systems, which demands very fast and very consistent response times and therefore cannot rely on the nondeterministic delays caused by garbage collection [9].

Rust was designed for bridging the gap between security and efficiency, targeting the development of both large and complex low-level systems software and critical embedded systems, that demand a high level of reliability and performance. Specially, Rust aims to replace unsafe languages such as C and C++ that have weaker memory safety guarantees, providing a safer language, without incurring performance costs unlike memory safe languages with GC.

Rust enforces memory safety using static analysis. Static analysis is a method of analysing code without executing it and can be used to check for the presence or absence of memory safety bugs in a program, such as buffer overflows and use-after-free. That is, the ownership system is enforced through the use of the borrow checker, a static analysis tool, that analyses the code to ensure that the ownership rules are being followed before the Rust program is executed.

Its focus on safety and performance and its unique technical features quickly gained attention from software developers after the release of its first version in 2010. Over the following years, Rust has been recognized as the "most loved" programming language in multiple software developer surveys, such as Stack Overflow, where it has been earning this recognition since 2015.[8]

Also, throughout these years, Rust has also been constantly updated, and has grown to become a mature programming language with a rich ecosystem of libraries and built-in tools available for Rust development, including a package manager (Cargo), a central repository for Rust libraries (Crates.io), and many other tools that are constantly being developed by the large and active Rust community of developers.

Google, Amazon Web Services[9] and Microsoft[10] are also adopting Rust as an alternative to C/C++, and together with Mozilla and Huawei, formed the Rust Foundation, a non-profit organization that aims to maintain and promote the development of the Rust programming language and its community, in an open-source ecosystem.

These days, software projects are being implemented from scratch in Rust, such as web browsers [10], operating systems [11] and garbage collectors [12]. Rust is also being integrated into large existing C/C++ codebases, such as the Mozilla's Quantum project [13], Linux [14] and Android [15].

However, existing software systems cannot be fully and manually re-implemented or rewritten in Rust, since these kind of efforts are expensive and do not scale well. For that reason, one way to integrate Rust into existing codebases is to entirely or partially migrate that existing software into Rust, automatically making that software safer.

There is a huge amount of code written in unsafe C and C++ that could greatly benefit from being

[8] https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

[9] https://github.com/firecracker-microvm/firecracker

[10] https://www.theregister.com/2022/09/20/rust_microsoft_c/

automatically translated into Rust, which would eliminate a large class of potential security vulnerabilities, maintaining the same efficiency and performance. Indeed, Google recently reported that between 2019 and 2022, the percentage of memory safety vulnerabilities out of Android's total vulnerabilities decreased from 76% to 35% and memory safety bugs are no longer the majority of Android's vulnerabilities. The reason is simple: Google started to write software in Rust, a memory safe programming language [16].

1.1 Contributions

In this work we implement an automatic source-to-source compiler that translates code from C++ into safe Rust. As C++ is a large programming language, a subset of C++ that can be automatically translated to Rust is selected. This subset is designed to be sufficient for converting the **WOFF2** library, which is a real C++ codebase.

A translation pipeline, divided into two main steps, is proposed. In the first step, the transpiler translates C++ code to Rust using a certain translation model. Depending on the translation model used, the generated Rust code exhibits specific characteristics, such as safety, execution time efficiency, and memory usage efficiency, among others. Finally, the modeled Rust program can be refactored and optimized through a series of passes to enhance the final translated code.

However, this work focuses on the first step: the automatic translation of C++ to Rust using a particular translation model, with a specific emphasis on the automatic translation of C++ to safe Rust, as opposed to a more elementary conversion to unsafe Rust.

Unlike Rust, C++ is not memory-safe. Therefore, a translation model based on a run-time memory management mechanism provided by the Rust Standard Library is designed, implemented, and used to automatically perform the translation of C++ to safe Rust. On the other hand, translating C++ to unsafe Rust is a much more direct approach.

Additionally, this work also involves the study of the automatic conversion of advanced C++ features to Rust, including classes, function overloading, templates, and more. These features are used in the original **WOFF2** C++ library.

Ultimately, after the initial generation of Rust code and considering the specific translation model in use, we examine potential issues. For example, related with readability and optimization of the generated code. These problems could be addressed using static analysis during the refactoring phase, laying the groundwork for future enhancements in automatically verifying memory safety properties of C++ programs within the generated Rust code. However, static analysis algorithms have theoretical limitations. In practice, this means that static analysis might not always yield enough information for the source-to-source compiler to enhance these translated Rust programs.

2

Background

Contents

2.1 Type, Memory and Thread Safety

Rust is a type, memory and thread safe compiled programming language. Type-safety denotes the ability of a programming language to prevent type errors, which occur when trying to perform operations on values that do not have the expected data type. Type safe languages can either be static or dynamic, which means that variable types are checked either at compile-time (variables have a fixed type) or at run-time (variables may have different types), respectively.

Static analysis is the inspection of a computer program performed before running it, i.e. at compile-time. This type of analysis is based on the set of possible values that each variable can be assigned, depending on its type (i.e., the domain or range). In contrast, dynamic analysis is performed on a

computer program while it is being executed, i.e. at run-time. And, the analysis is based on the values currently assigned to the variables instead. Rust is statically-typed and its powerful type system allows memory and thread safety to be checked at compile-time.

Regarding memory management, Rust is a memory-safe programming language. Memory-safety denotes the ability of a programming language to prevent or detect bugs that may occur when a program manipulates memory and that could lead to security vulnerabilities, undefined behavior or crashes. Examples of common memory-related bugs include: dereferencing a null pointer (a pointer is null and its value is accessed), accessing an array out of bounds (an index that is outside of the bounds of an array is used to access the array), dangling pointers (a pointer is used to access deallocated memory, also known as use-after-free) and buffer overflows (writing beyond the boundary of the memory buffer and over adjacent memory locations, exceeding its storage capacity and granting a malicious user the ability to modify memory that does not belong to the program in execution).

There are a variety of mechanisms which memory-safe programming languages use to ensure memory safety. As with type-safety, memory safe languages can either be static or dynamic, which means that memory safety is enforced at compile-time or at run-time. Some examples of static memory-safety mechanisms, which Rust makes use of, include static typing, ownership and borrowing. On the other hand, there are techniques that ensure memory-safety with run-time checks as well, such as dynamic type checking, garbage collection and bounds checking and are more frequently used, in dynamic programming languages.

Garbage collection (GC) is an automatic memory management mechanism that automatically and periodically reclaims and frees memory that is no longer needed by the program, making the memory available for allocation again.

Bounds checking is a method of checking whether an index used to access an element of an array is within its range of valid indices (its bounds) before access happens, detecting out-of-bounds errors and thus preventing more serious consequences. Usually bounds checking is implemented using assertions. Nevertheless, while some programming languages fall back on the compiler or run-time system to automatically perform bounds checking, as Rust does (e.g., in safe Rust, indexing a `Vec` performs bounds checks at run-time), others call for the programmer to manually take on that responsibility, such as C.

Rust relies on the static typing, ownership model, reference borrowing and lifetimes to ensure memory safety, eliminating all aforementioned memory-related bugs, at compile-time. The ownership model is what makes Rust different from most safe system programming languages: it guarantees memory safety without resorting to a garbage collector, thereby speeding up programs and avoiding run-time costs. Generally, memory safety is important as it prevents common memory-related bugs, improving the reliability and security of programs.

With respect to concurrency, Rust is a thread-safe programming language. Thread-safety denotes the ability of a computer program to be safely executed by multiple threads concurrently. Overall, thread-safety is important because it enables multiple threads to safely read and modify shared data, which can speed up an application. Rust uses ownership and borrowing mechanisms to guarantee that data races cannot occur.

2.2 Ownership and C++

Ownership[1] is the memory management system that Rust programs use to manage memory in the heap. The ownership system keeps track of all mutable and immutable references to variables and, in addition to achieving memory safety, also influences system design [17]. This ownership system allows the Rust compiler to determine when to free allocated memory, ensuring memory safety, and to eliminate data races.

The Rust's ownership-based type system is related to the region inference algorithm developed in the MLKit project, which is a compiler from Standard ML to assembly language [18]. The Region Inference is an algorithm for analyzing a program and inferring the lifetimes of values at compile-time (i.e., it is a static program analysis). The Cyclone programming language also implements a memory management system based on regions [19].

In Rust, the ownership consists of a set of rules checked by the compiler. It is implemented by an additional static analyzer called borrow checker which enforces the ownership management and determines when memory deallocation takes place. The ownership system does not add any run-time overhead as these rules are checked at compile-time. A Rust program compiles if none of the following ownership rules are violated:

1. Every value in Rust is owned by some variable.
2. At any given time, for each value there is exactly a single owner variable.
3. Once the owner goes out of scope (it is no longer in use), its value is automatically deleted.

The scope of a variable defines the region of the program where the variable is valid. A variable belongs to the scope where it is declared (e.g., in Listing 1 in line 2) and remains valid until the end of that scope, at which point it is automatically dropped (e.g., in Listing 1 in line 3). The program in Listing 1 demonstrates the variable scope and when the ownership system frees the variable's allocated memory.

The memory that is allocated on the heap at run-time by the `Vec` type is managed by itself, a mechanism that works like in other programming languages (e.g., `std::unique_ptr` in C++). Rust differs from

[1] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

```rust
{
  let v1: Vec<i32> = Vec::new();
}
```

```cpp
{
  const std::vector<int> v1;
}
```

Listing 1: Scopes, constructors and destructors in Rust and C++

garbage collection and manual memory management mechanisms in order to deallocate the memory when the `Vec` is no longer needed: once the owner of a value, in this case the variable `v1` that has data on the heap, goes out of scope, its allocated memory is automatically freed. Therefore, the programmer does not have to manually release the memory used by `v1`. In order to implement this, when the scope is over, Rust automatically calls a special function for us called `drop`, which is where the code to clean up the `Vec`'s allocated memory is.

The `drop` function in Rust is equivalent to the destructor in C++. In C++, there is the **Resource Acquisition Is Initialization** (RAII), which is a pattern for deallocating resources at the end of a variable's lifetime. The variable `v1` is initialized in line 2 and it is automatically freed in line 3, at the end of the scope.

In Rust, the assignments between variables are handled differently depending on where the data types involved in the operation are stored in memory. If the variable has memory allocated on the heap,[2] as `Vec` and `String` types have, then the variable is moved. Otherwise, if the variable is stored entirely on the stack,[3] as integers, floating-point and boolean types are, then the variable does not move but rather it is just trivially copied because contrary to the last case, there is no difference between doing a deep or shallow copying.

```rust
let v1 = vec![1, 2, 3];
let v2 = v1; // move
println!("{}", v2[0]);
println!("{}", v1[0]); // error
```

```cpp
const std::vector<int> v1({1, 2, 3});
auto v2 = v1; // copy
std::cout << v2[0] << std::endl;
std::cout << v1[0] << std::endl;
```

Listing 2: Comparing the default move and copy semantics in Rust and C++

In Listing 2, the variable `v1` of `Vec` type created in line 1 has data on the heap that contains its elements. Thus, the assignment of `v1` to `v2` in line 2 does not copy but rather moves the variable `v1` to `v2`, which means that `v2` is now the owner of the values of `v1`. First, the move from `v1` to `v2` copies the data on the stack of `v1` (like a shallow copy), without copying the heap data that its pointers might refer to (like a deep copy). This makes any move an inexpensive operation in terms of run-time performance. Finally, the move also invalidates the variable that is being moved, which in this case corresponds to Rust considering `v1` as no longer valid after assignment to `v2`.

If Rust did a shallow copy instead of a move, a double free error would occur because `v1` and `v2` would have pointers pointing to the same memory location and, as soon as these variables went out of

[2]Memory available to programs where data is allocated with a dynamic or unknown size at compile time.
[3]Memory available to programs where data is pushed with a known and fixed size at compile time.

scope, Rust would free the same memory twice. In order to guarantee memory safety, Rust moves the variable `v1` to `v2`, ignoring the invalid variable `v1` and freeing the heap memory of the valid variable `v2` exactly once, when it goes out of scope. Therefore, the Rust compiler rejects the code above, giving a compile-time error. It detects that in line 4 there is an access to the invalidated reference `v1`, because the ownership of the value `vec![1, 2, 3]` was previously moved from `v1` to `v2`.

Although the Rust and C++ programs in Listing 2 are syntactically the same, they are semantically different. Contrary to Rust, in C++, assignments between variables copy by default. In line 2 of this last C++ program, the vector `v2` gets a deep copy of `v1` and therefore, `v1` remains valid to be accessed in line 4. Thus, everything works fine and no compile or run time error is thrown by the C++ compiler.

Noteworthy, in Rust, passing a variable to a function works just like an assignment, moving the value by default. Since the ownership is transferred, the value will be deleted after the end of the function's scope and the original variable will be marked as invalid after the function call.

On the other hand, in C++, to transfer ownership of a variable, `std::move` must be explicitly called. Unlike Rust, C++ does not invalidate access to a previously moved value, but instead, this access is considered unspecified behavior, as the moved variable remains in a valid but unspecified state.

2.3 References, Borrowing and C++

The value of a reference is the address of some other owner variable. Moreover, a reference can be used to indirectly access the value stored at that address, which is owned by the other variable. The borrow checker also has a set of rules to ensure memory safety when dealing with references, as it does with owner variables.[4] Similarly, a Rust program compiles if none of the following referencing rules are violated:

1. A value can only have a mutable reference if there is no other reference to the value.

2. There can be no references that live longer than the values they point to.

Therefore, memory and thread safety are achieved by checking the ownership and referencing rules at compile time. For example, rule number one and rule number two statically address data races and use-after-free bugs, respectively.

After a function call, there are two options available in order to keep the ownership of values passed as arguments: returning the value back or borrowing.

In line 3 of Listing 3, the ownership of the value of the variable `v1` is moved into the function but it is also explicitly handed back to the caller. This way, we can still use the vector value after the function call.

[4] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

```rust
fn main() {
  let mut v1 = vec![1, 2, 3];
  v1 = return_ownership(v1);
  println!("{}", v1[0]);
  borrow_ownership(&v1);
  println!("{}", v1[0]);
}

fn return_ownership(
  v2: Vec<i32>
) -> Vec<i32> {
  v2
}

fn borrow_ownership(v2: &Vec<i32>) {
  // v2 is not the new owner
}
```

```cpp
int main() {
  std::vector<int> v1({1, 2, 3});
  v1 = return_ownership(std::move(v1));
  std::cout << v1[0] << std::endl;
  borrow_ownership(v1);
  std::cout << v1[0] << std::endl;
}

vector<int> return_ownership(
  std::vector<int> v2
) {
  return v2;
}

void borrow_ownership(std::vector<int> &v2) {
  // v2 is not the new owner
}
```

Listing 3: Comparing move, copy and borrowing with references in Rust and C++

Alternatively, Rust provides references that allow you to use a value without taking its ownership (i.e., without invalidating the former owner), avoiding the need to return the ownership back. Borrowing refers to the concept of creating a reference. Therefore, a reference is a variable that borrows the value of another variable. However, there are some ownership systems that have been implemented without being able to temporarily borrow owned values, such as the Singularity operating system [20].

Although references and pointers similarly have addresses as their values, the ownership system only ensures that references point to a valid memory location for their entire lifetime. In line 5 the ownership of the value is not taken. Instead, a reference that points to the value of `v1` is passed into the function. This way, when the reference `v2` goes out of scope, the value it refers to will not be dropped.

Regarding the C++ code of Listing 3, in line 3 `v1` is moved into `v2`, rather than copied, just like in Rust. Therefore, the variable `v1` is only guaranteed to be valid in line 4 due to the value returned in line 3. References in C++ are similar to references in Rust, so `v2` works as an alias for `v1` on the second function call, avoiding making copies in line 5. Since `v1` was passed by reference, rather than moved, it is still valid after the second function call, in line 6. In the end, this example works very similarly in both languages.

There are two kinds of references: immutable and mutable references. Setting a reference as mutable or immutable defines whether it is possible to modify the value it points to or not, respectively. Rust throws a compile-time error if the program attempts to modify an immutable borrowed value. Moreover, in Rust variables and references are immutable by default which allows for controlled modification of owned values.

Rust code also does not compile when there is a mutable reference and other references to the same value within the same scope, regardless of their kind. This prevents data races at compile time,

for example by not letting multiple mutable references to mutably borrow the same value at the same time.

Rust recently added a **Non-Lexical Lifetimes** (NLL) algorithm to the borrow checker, in order to decrease the number of programs that fail to compile but do not violate the ownership-based type system. With NLL, the scope of a reference begins where it is declared and ends the last time it is used (the same is not true for actual values, as NLL does not change the dropping point).

Unlike Rust, in C++ variables and references are mutable by default. In order to make them immutable, the developer must annotate the type with the `const` qualifier. Additionally, in C++ the compiler does not prevent to have multiple mutable references to the same data.

Finally, dangling references are a very common bug in languages with pointers and references. A dangling reference is a reference that outlives the variable which it refers to. This happens when a reference to a variable remains accessible after the variable is freed, allowing an invalid read to a deallocated memory location, through that reference. Contrary to other system programming languages, the Rust compiler ensures that dangling references will never occur.

```rust
let mut v1: Vec<i32> = Vec::new();
v1.push(1);
let r1 = &v1[0];
v1.clear();
println!("{}", r1);
```

```cpp
std::vector<int> v1;
v1.push_back(1);
auto &r1 = v1[0];
v1.clear();
std::cout << r1 << std::endl;
```

Listing 4: Dangling references in Rust and C++

The code in Listing 4 is a common scenario prone to creating dangling references: referencing a specific element owned by a data structure that is deallocated before the reference goes out of scope. In line 3, an immutable reference is created, that points to the first element of the vector. In the following line, the vector frees all of its allocated memory, including the first element. So the reference `r1` becomes a dangling reference at that time. In the last line, Rust throws a compile-time error, successfully preventing the access to an invalid memory location, through the dangling reference `r1`. On the other hand, the C++ compiler accepts this program but results in undefined behavior at run-time.

2.4 Lifetimes and C++

In Rust, there is an intrinsic lifetime[5] associated with every reference. A lifetime determines the time span (i.e., the scope) during which a reference remains valid. Thus, in order to ensure that references will always be valid, the Rust compiler uses this concept of lifetimes, which allows elimination of dangling references, at compile-time.

[5] https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

The Rust compiler uses the borrow checker to statically analyze these lifetimes, determining whether or not the code is invalid in terms of lifetimes. This static analysis verifies whether all borrows are valid by comparing scopes. Therefore, in Rust, type checking can be divided into two parts: first the traditional type checker and afterwards the borrow checker [21].

```rust
fn main() { // 'a
  let s1 = String::from("a string");
  let mut s2 = &s1;
  { // 'b
    let s3 = String::from("a dangle");
    s2 = &s3;
    println!("{}", s2);
  } // 'b
  println!("{}", s2);
} // 'a
```

```cpp
int main() { // 'a
  std::string s1("a string");
  auto &s2 = s1;
  { // 'b
    std::string s3("a dangle");
    s2 = s3;
    std::cout << s2 << std::endl;
  } // 'b
  std::cout << s2 << std::endl;
} // 'a
```

Listing 5: Lifetimes after rebinding a reference in Rust and C++

The code in Listing 5, visually shows that the variable `s2`, that is part of the outer scope, lives longer (i.e., has a larger scope) than the variable `s3`, that is part of the inner scope. The annotations `'a` and `'b` demonstrate the existing lifetimes of the variables in the program. Since the variable `s2` and `s3` are within the outer and inner block, respectively, the annotations of the lifetimes of `s2` and `s3` are `'a` and `'b`, respectively.

In line 9 of Listing 5, Rust throws a compile-time error saying that the borrowed value of `s3` does not live long enough compared to the reference variable `s2`, set in line 6. This violates a referencing rule that states that a reference cannot outlive the memory it points to. Again, the Rust compiler prevents the use of a dangling reference. The problem with the code in Listing 5 is that the variable `s3` is dropped in line 8 (i.e., it goes out of the inner scope) but it is still being borrowed in line 9 (i.e., the reference `s2` is still valid in the outer scope). So, the reference `s2` ends up pointing to an invalid value.

From the compiler's point of view, the borrow checker checks that `s2` has a lifetime of `'a` and `s3` has a lifetime of `'b`. The compiler rejects the program because the lifetime of `'a` is larger than the lifetime of `'b`.

Again, although the Rust and C++ programs in Listing 5 look very similar, they have semantic differences. In contrast with C++, Rust allows the rebinding of references without modifying the original value, as shown in line 3 and 6 of Listing 5. This means that the assignment of line 6 in Rust changes the value to which the reference `s2` points to, but does not change the original value of `s1`. On the other hand, the same assignment in C++ changes the original value of `s1`, but does not change the value that the reference `s2` refers to. After the assignment in the inner scope `'b`, in the outer scope `'a`, the reference `s2` still points to the variable `s1` which has the lifetime `'a`. Therefore, no dangling reference is created and the C++ program is memory safe.

There are very simple lifetime elision rules in Rust (e.g, the code in Listing 5 compiles without ex-

```
1  fn main() { // 'a1
2    let s1 = String::from("a string");
3    let s2 = String::from("a new string");
4    let mut result = compare_and_swap(&s1, "a string", &s2);
5    println!("The result is '{}'.", result);
6    { // 'a2
7      let s3 = String::from("other new string");
8      result = compare_and_swap(&s1, "other string", &s3);
9    } // 'a2
10   println!("The result is '{}'.", result);
11 } // 'a1
12
13 fn compare_and_swap<'a>(cur: &'a str, old: &str, new: &'a str) -> &'a str {
14   if cur == old {
15     new
16   } else {
17     cur
18   }
19 }
```

Listing 6: Lifetime annotation syntax in Rust

plicit lifetime annotations). However, the Rust compiler demands the programmer to manually annotate lifetimes when there are several possible ways to describe the relationships between the lifetimes of references, a paradigm not very common in most programming languages. Like this, there is a trade-off between giving the compiler more information about lifetimes through special annotations and using garbage collection.

In Listing 6, a generic lifetime parameter named `'a` is annotated in the function signature, in line 13. Also, `cur` and `new` are the arguments of this function that are immutable references with explicit lifetimes, as demonstrated in line 13. The same happens with the return type. With these lifetime annotations, the Rust compiler now exactly knows how the lifetimes of the reference parameters and of the return value should relate to each other: the returned reference must remain valid as long as both the parameters are valid. This means that all of these references must have the same lifetime `'a` (i.e., the parameters `cur` and `new` and the return value will live at least as long as lifetime `'a`). The borrow checker rejects code that calls the function with arguments that have lifetimes that do not follow this constraint.

To implement this constraint, the borrow checker sets the lifetime of the returned reference to the smallest of the lifetimes passed in `cur` and `new` function arguments, because they all are annotated with the same lifetime parameter `'a` (i.e., the lifetime `'a` is the lifetime that results from the intersection of the lifetimes of variables cur and new). From this point on, the Rust compiler can statically analyze that the returned reference remains valid as long as the lifetime `'a`.

In Listing 6, the string `s1` and `s2` are valid until the end of the main scope, so their lifetimes are both equal to the lifetime `'a1`. Thus, the variable result, assigned in line 4, has a lifetime equal to the intersection of the lifetimes of the variables `s1` and `s2` passed as concrete parameters to the function.

Since they are equal, the result from the intersection is the lifetime `'a1`. Here, everything works fine and the borrow checker accepts this first part of the code.

Regarding the scope `'a2` in Listing 6, while the string `s1` has the lifetime `'a1`, the string `s3` has the lifetime `'a2`. The reference in variable `result`, assigned in line 8, has a lifetime equal to the smallest lifetime between `'a1` and `'a2`, which in this case is the lifetime `'a2`. Hence, the reference `result` points to a variable that has a lifetime `'a2`, which is smaller than the lifetime `'a1`. Moreover, the variable `s3` is dropped at the end of the scope of `'a2` and, the program attempts to use the reference `result` after the scope of `'a2`, more precisely in the scope of `'a1`, in line 10. A compile time error is thrown, saying that the variable `s3` does not live long enough.

Noteworthy, the Rust compiler does not demand the programmer to annotate the parameter old in the function signature with a specific lifetime, because there is no lifetime relationship required to be described with it.

Recently, the Clang Frontend community announced that they are implementing lifetime annotations for C++, following a Rust-inspired scheme[6]. These lifetime annotations would allow for better C++/Rust interoperability and the prevention of many common memory-related bugs such as dangling references.

[6] https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

3

Related Work

Contents

3.1 C2Rust

C2Rust [22], developed by Galois and Immunant, is one of several industry-made source-to-source compilers that translate C programs to Rust. Other examples of experimental C to Rust transpilers include Citrus [23] and Corrode [24]. C2Rust claims to be a Rust migration tool with the primary goal of automatically (or with minimal manual effort) translating C99-compliant code into unsafe Rust code, making the integration of Rust a much easier and scalable process.

In this manner, C2Rust is a source-to-source compiler (or transpiler) that takes a C program as input and outputs a Rust program [25]. C2Rust is also available online for quick demos.[1] C2Rust helps with

[1] https://c2rust.com/

this software migration problem by offering a set of different tools: a translator, a refactoring tool and tools to cross-check execution of code. Before running this toolset to perform the translation, C2Rust uses Clang to parse and type check the C source code.

Overall, the C2Rust workflow goes through these last three tools. First, the transpiler is used to translate C to Rust. Second, there is the possibility to take advantage of the refactoring tool in order to improve the quality of the produced Rust code. Finally, cross-checking tools are also available to test the execution of the C code against the execution of the new Rust code in order to ensure that no bugs were introduced into the output program during refactoring.

Regarding the translator, the C program it receives as input is unsafe by definition. At this stage, the main goal of the translator is to generate unsafe Rust code that is syntactically similar and semantically equivalent to the input code. Therefore, the transpiler produces unsafe Rust code from the C code. Rust allows unsafe code through the use of the keyword 'unsafe', which unlocks access to additional language features (e.g., dereferencing a raw pointer) that can be misused and cause undefined behavior. However, ownership rules are still enforced in unsafe blocks by the borrow checker.

The various refactoring tools available allow you to make the output Rust code safer or more idiomatic. This is necessary because the translator produces a Rust program that is syntactically similar to the C program and therefore not idiomatic nor safe. For example, while Rust code translated from C uses while loops and indexing to access element in arrays, iterators would be a better choice to make the Rust code more idiomatic. Then, the suggested approach by C2Rust is to incrementally and iteratively rewrite and improve the translated Rust code using specific refactoring tools.

In the end, the refactoring tools do not cover all cases. Therefore, errors can be introduced, derived from the manual refactoring that must be done to handle these cases. In order to automatically detect these errors (i.e., to check if the translated program has the same behavior as the original C program), C2Rust proposes the use of the cross-checking tool that after running the original C, unsafe and refactored Rust programs with identical inputs, check if there is any divergence in the their outputs.

The comparison of the outputs can be done during the programs execution (i.e., online) using the ReMon MVEE[2] or after the programs execution (i.e., offline) using log files. The implementation of the C2Rust cross-checking tool consists of modifying the source code of each of these programs at compile-time (i.e., inserting instrumentation code at the beginning and end of a function and in its returned values and called arguments).

As an example, to translate the **bzip2** project written in C to Rust, clone the library from https://github.com/libarchive/bzip2.git, move to its main directory and enter the following commands on the terminal:

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 .
```

[2] https://github.com/ReMon-MVEE/ReMon

```
c2rust transpile compile_commands.json
```

In addition to C2Rust being able to produce unsafe Rust code from entire codebases written in C, it solves other transpilation issues remarkably well, such as loop recovery.

```c
preswitch:
  switch (s->mode) {
  case BZ_M_IDLE:
    return BZ_SEQUENCE_ERROR;
  case BZ_M_RUNNING:
    if (action == BZ_RUN) {
      progress = handle_compress ( strm );
      return progress ?
        BZ_RUN_OK : BZ_PARAM_ERROR;
    }
    else if (action == BZ_FLUSH) {
      s->avail_in_expect = strm->avail_in;
      s->mode = BZ_M_FLUSHING;
      goto preswitch;
    }
    else if (action == BZ_FINISH) {
      s->avail_in_expect = strm->avail_in;
      s->mode = BZ_M_FINISHING;
      goto preswitch;
    }
    else
      return BZ_PARAM_ERROR;
```

```rust
loop {
  match (*s).mode {
    1 => return -(1 as libc::c_int),
    2 => {
      if action == 0 as libc::c_int {
        progress = handle_compress(strm);
        return if progress as libc::c_int != 0 {
          1 as libc::c_int
        } else {
          -(2 as libc::c_int)
        };
      } else if action == 1 as libc::c_int {
        (*s).avail_in_expect = (*strm).avail_in;
        (*s).mode = 3 as libc::c_int;
      } else if action == 2 as libc::c_int {
        (*s).avail_in_expect = (*strm).avail_in;
        (*s).mode = 4 as libc::c_int;
      } else {
        return -(2 as libc::c_int)
      }
    }
```

Listing 7: Loop recovery after C2Rust translation of `BZ2_bzCompress` function from **bzip2** library

```c
size_t strlen(const char *s) {
  goto a;
  b:
  s++;
  goto d;
  a:;
  // int count = 0;
  unsigned count = 0;
  goto d;
  c:
  return count;
  d:
  if(*s == '\0')
  goto c;
  count++;
  goto b;
}
```

```rust
pub unsafe extern "C" fn strlen(
  mut s: *const libc::c_char
) -> size_t {
  let mut count = 0
    as libc::c_int
    as libc::c_uint;
  while !(*s as libc::c_int == '\0' as i32) {
    // count += 1;
    count = count.wrapping_add(1);
    s = s.offset(1);
  }
  return count as size_t;
}
```

Listing 8: Loop recovery after C2Rust translation of an adapted `strlen` function from the standard C library

The C2Rust examples in Listing 7 and 8 demonstrate the ability of C2Rust to recover loops from unstructured control-flow, turning the translated Rust much more idiomatic and readable. In line 2 of

Listing 7, C2Rust replaces the `switch` statement of C with the pattern matching statement of Rust. Furthermore, C2Rust also infers a loop through the label and `gotos` present in lines 1, 14 and 19 of C in Listing 7, respectively, translating them to an idiomatic `loop` statement in Rust. Even so, C2Rust manages to translate more complex blocks of `goto` statements, as can be seen in Listing 8: the 5 `gotos` and the single `if` statement in C are successfully translated into a simple `while` statement in Rust.

Regarding overflow problems, when C2Rust finds an arithmetic operation on an unsigned type (e.g., `unsigned` in C), it translates it to a safe wrapped arithmetic operation,[3] in order to preserve semantics, as shown in Listing 8, in lines 15 and 9 of the C and Rust examples, respectively (i.e., Rust does not use modular arithmetic on operations on unsigned types to disallow overflows by default).

On the other hand, if the arithmetic operation was on a signed type (e.g., `int` in C), it translates to a regular arithmetic operation with no wrapping, as shown in Listing 8 in comment lines 7 and 8 of the C and Rust examples, respectively.

This is due to the fact that the C99 standard states that unsigned integer overflow is defined behavior, in contrast to signed integer overflow, which is undefined behavior [26].

However, there are some issues with the translated Rust code. First, there are an overwhelming number of unnecessary casts, as shown in both translated examples. Cast operations in Rust are performed with the `as` keyword (e.g., in Listing 7 in line 7 of the translated Rust code example). In order to make the Rust code more idiomatic, the solution would be to reduce the number of casts by choosing the best types in the variable definitions.

Second, it is sometimes guaranteed that unsigned integer overflow will not occur even without calling a wrapping method. Therefore, it would be better to prove at compile time that overflow is impossible, allowing the removal of unnecessary wrapped arithmetic operations, which would improve the performance and readability of the translated Rust code. Finally, `#define` macros are replaced by numbers used directly in the source code, losing readability.

Another feature of C2Rust is its ability to translate fall through cases of switch statements in C into Rust. The equivalent of `switch-case` statements of C in Rust is the `match` statement. However, the fall through cases make translation more difficult because once the `match` statement argument matches one of its arms, it never falls down the arm below,[4] as in C `switch-case statements`. C2Rust's solution is to generate a sort of state machine. First, each arm in the first Rust `match` statement executes code that corresponds to the code executed uniquely by each case, in the `switch` statement of C. Second, the Rust `match` statement below executes code that corresponds to the code executed by multiple cases, i.e. after the fall through, in the `switch` statement of C.

This is exactly what happens in the Listing 9. Case `TK_ID` in C corresponds to arm `59` in Rust. This case block in C executes some code, represented by the comment line with `block A` in line 4, and so

[3] https://doc.rust-lang.org/std/intrinsics/fn.wrapping_add.html
[4] https://doc.rust-lang.org/book/ch06-02-match.html

```
1  switch( pExpr->op ){
2    // ...
3    case TK_ID:
4        /* block A */
5    case TK_COLUMN:
6    case TK_AGG_FUNCTION:
7    case TK_AGG_COLUMN:
8        /* block B */
9    // ...
10   case TK_VARIABLE:
11       /* block C */
12   default:
13       /* block D */
14 }
```

```
1  let mut current_block_39: u64;
2  match (*pExpr).op as libc::c_int {
3    // ...
4    59 => {
5      /* block A */
6      current_block_39 = 15652330335145281839;
7    }
8    167 | 168 | 169 => {
9      current_block_39 = 15652330335145281839;
10   } // ...
11   156 => {
12     /* block C */
13     current_block_39 = 7828949454673616476;
14   } // ...
15 }
16 match current_block_39 {
17   15652330335145281839 => { /* block B */ }
18   7828949454673616476 =>    /* block D */,
```

Listing 9: State machine after C2Rust translation of `exprNodeIsConstant` function from **sqlite3** library

the corresponding arm in Rust also executes it, represented by the comment line with `block A` in line 5. Also, the fall through of the `TK_COLUMN`, `TK_AGG_FUNCTION` and `TK_AGG_COLUMN` cases, which starts at line 5 and ends at line 7, executes `block B` in line 8 of the C code.

But in addition to the case `TK_ID` in C executing `block A`, it also falls through and executes `block B`. Therefore, C2Rust uses the second `match` statement to execute this common `block B`. C2Rust starts by setting the value of the variable `current_block_39`, declared in line 1, to `15652330335145281839` in both arms (i.e., in arms `59` and `167`, `168` or `169`, in lines 6 and 9, respectively).

Finally, in the second `match`, since the value of the variable `current_block_39` matches the arm `15652330335145281839`, `block B` is executed, in line 17 of the translated Rust code. The same approach is followed in the `switch-cases` `TK_VARIABLE` and `default`, executing blocks `C` and `D` in the first and second `match` statement, respectively, and setting the variable `current_block_39` to `7828949454673616476`.

In the end, the translated code is neither idiomatic nor readable Rust. Here, a possible solution would be: for each arm in the second `match` statement, move the code inside it into a function and call that function on the corresponding arms in the first `match` statement. In addition to these shown problems, there are more limitations of C2Rust.[5]

In summary, although C2Rust is capable of translating most of the C sources into unsafe and semantically equivalent Rust code, there is still room for improvement.

[5] https://github.com/immunant/c2rust/blob/master/docs/known-limitations.md

Benchmark	Application Domain	C LoC	Rust LoC	# Functions	# `unsafe` Functions
libcsv	Text I/O	1,035	951	23	23
urlparser	Parsing	440	1,114	22	21
RFK	Video games	838	1,415	18	17
genann	Neural nets	642	2,119	32	27
lil	Interpreters	3,555	5,367	160	159
json-c	Parsing	6,933	8,430	178	178
libzahl	Big integers	5,743	10,896	230	230
bzip2	Compression	5,831	14,011	128	120
TI	Time series analysis	4,643	19,910	234	229
tinycc	Compilers	46,878	62,569	662	625
optipng	Image processing	87,768	93,194	576	572
libxml2	Parsing	201,695	430,243	3,029	3,009
Total	—	366,001	650,219	5,292	5,210

Figure 3.1: Number of unsafe functions in translated Rust [1]

3.2 Unsafe Rust

The major problem with C2Rust is that its output is an unsafe Rust program. Therefore, the Rust compiler cannot guarantee memory-safety of these programs, which is the main feature of Rust. The C2Rust translation is limited to translating to Rust in a syntactically similar and semantically identical way to the input C code.

For example, in C the use of raw pointers is too common and in Rust can only be used in unsafe code. Moreover, references are never created from unsafe raw pointers by C2Rust and references are memory-safe in Rust. Thus, the creation of safe references from these raw pointers, would improve the memory-safety of the translated Rust programs. And, as a consequence, make the original C programs safer.

C2Rust outputs a Rust program with almost all functions being marked unsafe. This is because all functions in the translated code that were directly translated from the input C program are conservatively labeled as unsafe by C2Rust. The only functions that are safe are the auxiliary functions injected during the translation process. However, the translation unnecessarily marks some of the functions that are translated directly from C code as unsafe (i.e., the `unsafe` keyword can be dropped in some of these functions) [1]. As shown in Figure 3.1, the number of translated functions (i.e., column named '# Functions') is practically the same as the number of unsafe functions (i.e., column named '#unsafe Functions').

There are multiple sources of unsafety, i.e. the behaviors that cause a function or a block to necessarily be marked unsafe [6]. The `unsafe` keyword is necessary when: a raw pointer is dereferenced; a function or method that is labeled unsafe is called; a variable that is mutable static or external is accessed; a trait[6] that is labeled unsafe is implemented; fields of unions are accessed.

It is questionable whether Rust programs that were automatically translated from C by C2Rust have

[6] Similar concept to an interface in object oriented programming

Benchmark	Union	Global	InlineAsm	Extern	RawDeref	Cast	Alloc
libcsv	0	2	0	35	174	4	0
urlparser	0	1	0	122	60	43	55
RFK	0	127	0	86	24	0	2
genann	0	164	0	183	339	3	5
lil	0	10	0	149	1,668	11	62
json-c	101	93	0	208	1,843	17	30
libzahl	0	430	29	63	2,457	0	43
bzip2	0	700	0	422	3,764	1	14
TI	0	108	0	352	1,847	84	9
tinycc	613	2,552	0	464	5,632	31	2
optipng	82	1,361	0	812	6,062	37	43
libxml2	499	3,571	0	4,593	52,546	15	15
Total	1,295	9,119	29	7,489	76,416	246	280

Figure 3.2: Number of times each source of unsafety statically appears in translated Rust [1]

the same sources of unsafety as those originally written by developers. Here, the sources of unsafety come directly from the original C program, making the translated Rust program unsafe. Therefore, a study was carried out to discover the main causes of unsafety in Rust programs translated by C2Rust and the impact of fixing each one of them was measured [1].

The implementation of this study consisted of first converting the translated Rust programs to the **Rust High-level Intermediate Representation** (HIR), which is an AST-based representation, and second, counting the nodes with an unsafety feature in the HIR (e.g., counting the number of raw pointer dereference nodes present in the AST). In the end, the conclusion of the benchmark was that raw pointer dereferencing is the most common source of unsafety (i.e., row Total and column RawDeref in Figure 3.2), followed by access to global variables (i.e., raw Total and column Global in Figure 3.2) and calls to external functions (i.e., raw Total and column Extern in Figure 3.2). Therefore, in order to make the maximum number of functions safe, resolving raw pointer dereferences is the best strategy to follow.

Regarding the raw pointer dereferencing problem: if the way a raw pointer is used is categorized, different solutions can be implemented to solve each of them (i.e., different techniques for converting the different types of raw pointers into safe references in Rust). Raw pointers can be categorized as follows [1]: lifetime raw pointer (obtained by calling `malloc` or a public API function); `void` raw pointer (obtained by casting a raw pointer to or from a `void*`, which is the idiomatic approach to implement polymorphism in C), extern raw pointer (obtained by passing to or returning from an external function call), arithmetic raw pointer (obtained when using pointer arithmetic). Although there are specific problems and solutions for each of these categories, the main raw pointers problem is common to all categories: to make raw pointers safe, their lifetime information must be inferred.

First, to convert lifetime raw pointers into safe references, each `malloc` call would need to be replaced by a Rust safe memory allocation mechanism (e.g., `Box::new`) and subsequently the respective lifetime information would need to be computed. Second, as generics[7] or traits are used to implement

[7] https://doc.rust-lang.org/book/ch10-01-syntax.html

polymorphism in Rust, they can be used to convert `void` raw pointers to safe references. Third, the only way to convert extern raw pointers into safe references is to replace that unsafe external call with a safe function call (e.g., replace `realloc` and `memcpy` by `resize_with` and `copy_from_slice`, respectively). Finally, unlike raw pointers, safe references in Rust do not have arithmetic, thus converting arithmetic raw pointers to safe references involves first replacing the pointer arithmetic with safe array slices, and secondly replacing the pointer with indexes.

A taint analysis can be used in order to discover the way pointers are used in the code and to categorize each of them [1]. This analysis was based on Steensgaard's pointer analysis [27] and Rust's type system. Essentially, a pointer is classified into a specific category (i.e., lifetime, void, extern or arithmetic raw pointer) based on whether the pointer contains a value that comes from a source or that flows into a sink related to that specific category (e.g., the value passed as argument into or the value returned from an external function call, respectively).

Rudra [28] is another study of sources of unsafety in Rust. Rudra is an open source[8] static program analyzer that finds and reports possible memory safety bugs in unsafe code present in large Rust codebases. It only takes a single bug in some unsafe code to break Rust's memory safety guarantees and thus Rudra's static analysis is very important to detect those bugs. Because of this, the official Rust linter[9] included the Rudra's algorithm. Related static analysis of unsafe code was made in order to turn safer the unsafe C# code [29].

Miri [30] is an alternative to Rudra, that identifies specific classes of undefined behavior by interpreting Rust's mid-level intermediate representation (MIR). In contrast to Rudra, Miri interprets potentially unsafe code by executing it, and when it encounters an error, it reports it.

3.3 Translating C to Safer Rust

There is a novel translation technique that automatically turns raw pointers into safe references built on top of the C2Rust [1]. The process takes a C program as input and uses C2Rust to translate it to an unsafe Rust program. Subsequently, the translation technique is applied on that unsafe Rust program, making it safer (i.e., safer may not be completely safe Rust).

The goal is to convert a subset of the raw pointers into safe references. The selected subset is the lifetime raw pointer category because it appears most frequently in C programs.

This translation technique, rather than implementing a static analysis for the original C program or the translated unsafe Rust program, it uses the Rust compiler as an oracle to infer the information necessary to determine which lifetime raw pointers can be converted into safe references. First, the unsafe Rust program is optimistically rewritten to convert lifetime raw pointers into safe references (i.e., the translation

[8] https://github.com/sslab-gatech/Rudra
[9] https://nnethercote.github.io/perf-book/linting.html

technique makes optimistic assumptions about lifetimes and mutability). If this version does not compile, errors produced by the compiler are iteratively used in a loop to continually refine the program, until the compiler accepts it.

A raw pointer can be converted into a safe reference if the Rust compiler can first prove that the translated reference points to a value with a single owner and, second, can infer the proper lifetime for that referenced value.

```
1  node_t* find(
2    int value,
3    node_t* node)
4  {
5    if (value < node->value
6        && node->left) {
7      return find(value, node->left);
8    } else if (value > node->value
9        && node->right) {
10     return find(value, node->right);
11   } else if (value == node->value) {
12     return node;
13   }
14   return NULL;
15 }
```

```
1  pub unsafe fn find(
2    value: c_int,
3    node: *mut node_t
4  ) -> *mut node_t {
5    if value < (*node).value
6        && !(*node).left.is_null() {
7      return find(value, (*node).left)
8    } else if value > (*node).value
9        && !(*node).right.is_null() {
10     return find(value, (*node).right)
11   } else if value == (*node).value {
12     return node
13   }
14   return 0 as *mut node_t;
15 }
```

Listing 10: The original C program and the translated Rust program after first step [1]

The translation technique rewrites the translated Rust program into an initial, optimistic, and safer version. This rewritten program will probably not compile and all unsafe markers due to Lifetime raw pointers are dropped. As shown in Listing 11, the function signature no longer contains the `unsafe` marker, in contrast to the program demonstrated in Listing 10, which showcases the initial translation of the function `find` performed by c2rust.

First, the technique converts the raw pointer declarations in data structure fields and function parameters into safe reference declarations, rewriting the raw pointers with optional references to handle null pointer values (an optional can be `Some` with a value or `None` without one). Finally, the most optimistic assumptions about the lifetime information for the new references is provided in a way that limits declarations to a minimum. This means that in this step each type in a data structure field with a reference gets a different lifetime variable and, function signatures are rewritten accordingly, as shown in lines 3 and 4 in Listing 11.

Afterwards, the function bodies are rewritten. For instance, dereferencing a raw pointer is replaced with unwrapping the optional reference by calling a Rust standard library method called `unwrap` (e.g., in Listing 11 from lines 5 to 11). However, unwrapping the option moves the ownership from the original `Option` object. Thus, it is necessary to call the method `as_ref()` (another Rust standard library method) to avoid taking ownership.

It is worth mentioning that all null pointers and the null pointer check `p.is_null()` are converted into `None` (e.g., in Listing 11 in line 14) and `p.is_none()` (e.g., in Listing 11 in line 6), respectively. Therefore, unwrapping a `None` in Rust is equivalent to dereferencing a null pointer in C, which panics in Rust and results in undefined behavior in C.

```
1  pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(
2    mut value: c_int,
3    mut node: Option<&'a1 mut node_t<'a2, 'a3>>
4  ) -> Option<&'a4 mut node_t<'a5, 'a6>> {
5    if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap()
6      && !(**node.as_ref().unwrap()).left.is_none() {
7        return find(value, borrow_mut(&mut (*node.unwrap()).left))
8    } else if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap()
9      && !(**node.as_ref().unwrap()).right.is_none() {
10       return find(value, borrow_mut(&mut (*node.unwrap()).right))
11   } else if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() {
12       return node
13   }
14   return None
15 }
```

Listing 11: The translated Rust program after second step [1]

In the final step, after rewriting the initial version, this translation technique iteratively uses Rust compiler errors in a loop to continually rewrite the program until it compiles. The compiler fails with the program in Listing 11 because there are lifetime constraints that are not satisfied. For instance, from the return statement in Listing 11 in line 12, the compiler knows that the returned reference cannot outlive the function argument. To fix this error, this technique automatically includes lifetime constraints in the function signature, as shown in Listing 12 (i.e., `'a1: 'a4` means that the lifetime variable `'a1` must be as long as the lifetime variable `'a4`, and so forth).

```
1  where 'a1: 'a4, 'a5: 'a2, 'a6: 'a3, 'a3: 'a6, 'a2: 'a5
```

Listing 12: Lifetime constraints in the `find` function signature after the final step

3.4 Our Approach

This work proposes a two-step translation pipeline, akin to c2rust. However, this work extends c2rust by proposing multiple translation models. In the initial step, the compiler generates Rust code based on a selected translation model. Subsequently, refactoring and optimization passes can be applied to improve the translated Rust code, depending on the selected translation model. Nevertheless, the primary focus is on the automatic translation of C++ into both unsafe and safe Rust code during the first step. Additionally, this work extends the scope of c2rust by translating advanced C++ features.

4

C++ to Rust Translation Models

Contents

In this chapter, we explore in detail various methodologies for modeling C++ code in Rust, as the first step in the translation pipeline. We begin with an overview of the several translation models before delving into a comprehensive study of each one. Then, we compare the translation models across multiple criteria, summarizing their advantages and drawbacks. We proceed to implementation details and finally evaluate the translation models using micro-benchmarks.

4.1 Overview

There are several factors motivating the exploration of multiple models. Currently, it is well-understood that C++ and Rust exhibit significantly different semantics in several fundamental operations. For in-

stance, consider the move operation, which invalidates access to the moved object in Rust but not in C++ (i.e., the moved object might be left in a valid, but unspecified state), as well as dereferencing a reference, which is safe in Rust but unsafe in C++, to name a few examples.

However, it is essential to acknowledge that preserving the semantics of C++ when translating to Rust might come at a higher cost compared to producing an improved Rust version that optimistically does not strictly preserve semantics. In some cases, experienced Rust software engineers can readily address compile-time errors introduced during this process. In other words, if C++ code was directly translated into safe Rust, how long would it take for an experienced Rust software engineer to manually fix lifetime annotations and multiple mutable borrow errors (assuming there is no static analysis to help the translation)?

Therefore, the compiler could offer multiple translation models, tailored to the user's needs and assumptions about the input C++ code. This forms the central focus of this chapter. These models differ based on the following criteria:

Manual Effort: The amount of manual effort required to compile the translated Rust code, reflecting the quantity of compile-time errors the model may introduce in the translation process. These errors can include missing lifetime annotations, multiple mutable borrow errors, and memory safety problems.

Memory Efficiency: The memory consumed by the translated Rust code in comparison to the original C++ code.

Time Efficiency: The execution time taken by the translated Rust code in comparison to the original C++ code.

Idiomatic: The model's ability to generate idiomatic Rust code.

Safety: The model's ability to generate safe Rust code.

Interoperability: How closely the translated code resembles the original C++ code in terms of syntax and semantics.

These models encompass three primary types in C++: a value of data type **T**, which includes built-in types (e.g., int, double, char), constant arrays, and user-defined types like classes and structures; a pointer to a value of type T (**T***); and a reference to a data type T (**T&**). To provide an overview, Listing 13 shows a simple C++ program, along with its translation into Rust using various models.

The input C++ code, as referenced in item (a), is a simple function that takes a reference and a pointer to an integer as inputs and returns their sum. The first translation model to be described is the **Safe** model, which **optimistically** converts references and pointers to their safe versions in Rust, as shown in lines 1 and 6 of item (b). However, it is important to note that this model does not preserve the original semantics of C++ reference and pointer types. In Rust, references and pointers have been made safe, and pointers are now adapted to exclusively point to a single variable, with the important

```cpp
int add(int &rx1, int *px2) {
  return rx1 + *px2;
}
int x1 = 1;
int x2 = x1 + 1;
int x3 = add(x1, &x2);
```

(a) Input C++ code

```rust
pub fn add(rx1: &mut i32, mut px2: Option<&mut i32>) -> i32 {
  return *rx1 + **px2.as_ref().unwrap();
}
let mut x1: i32 = 1_i32;
let mut x2: i32 = x1 + 1_i32;
let mut x3: i32 = add(&mut x1, Some(&mut x2));
```

(b) Translation into Rust Using the **Safe** Model

```rust
pub unsafe fn add(
  rx1: &mut i32, mut px2: *mut i32
) -> i32 {
  return *rx1 + *px2;
}
let mut x1: i32 = 1_i32;
let mut x2: i32 = x1 + 1_i32;
let mut x3: i32 = add(&mut x1, &mut x2);
```

(c) Translation into Rust Using the **Naive** Model

```rust
pub unsafe fn add(
  rx1: *mut i32, mut px2: *mut i32
) -> i32 {
  return *rx1 + *px2;
}
let mut x1: i32 = 1_i32;
let mut x2: i32 = x1 + 1_i32;
let mut x3: i32 = add(&mut x1, &mut x2);
```

(d) Translation into Rust Using the **Unsafe** Model

```rust
pub fn add(rx1: Weak<RefCell<i32>>, px2: Option<Weak<RefCell<i32>>>) -> Rc<RefCell<i32>> {
  return Rc::new(RefCell::new(
    *rx1.upgrade().expect("ub: dangling reference").borrow() +
    *px2.as_ref().unwrap().upgrade().expect("ub: dangling pointer").borrow()));
}
let x1: Rc<RefCell<i32>> = Rc::new(RefCell::new(1_i32));
let x2: Rc<RefCell<i32>> = Rc::new(RefCell::new(*x1.borrow() + 1_i32));
let x3: Rc<RefCell<i32>> = Rc::new(RefCell::new(
    *add(Rc::downgrade(&x1), Some(Rc::downgrade(&x2))).borrow()));
```

(e) Translation into Rust Using the **Reference Counting** Model

Listing 13: A simple C++ program and its translation to Rust using several models

note that pointer arithmetic is not supported.

The second described translation model is the **Naive** model, which downgrades pointers from a safe `Option<&mut T>` into unsafe raw pointers `*mut T`, as illustrated in lines 1 and 8 of item (c). Since dereferencing a raw pointer is considered unsafe in Rust, the keyword `unsafe` is added to the function signature in line 1.

The downgrading process culminates with the **Unsafe** model, which semantically preserves the translation by ultimately representing reference from C++ as an unsafe raw pointer `*mut T` in Rust, as shown in lines 1 and 8 of item (d). It preserves the semantics because a reference in C++ is unsafe. It is noteworthy that in all the aforementioned models, data types are modeled consistently - that is, a type `T` in C++ is modeled as type `T` in Rust.

Finally, we transition to a distinct model: the **Reference Counting** model, which is referenced in (e). This model **pessimistically** converts unsafe C++ code into safe Rust without requiring additional manual effort. In this model, data types, references and pointers are represented using Rust's reference counting

mechanisms (i.e., `Rc` and `Weak` types) and shareable mutable containers (i.e., `RefCell` type).

It is crucial to highlight that, theoretically, only the **Unsafe** and the **Reference Counting** models are capable of performing a fully automatic translation of C++ to Rust. The main difference is that the first model generates totally unsafe Rust code, while the latter generates entirely safe Rust code.

The upcoming sections will provide a detailed description of each of these models, with special emphasis on the reference counting model.

4.2 Compile-Time Models

The translation models described in these sections all share a common attribute: they do not incur additional run-time costs compared to the original C++ code, except for Rust's default bounds checks. However, they differ from each other semantically by offering various safety guarantees and, consequently, requiring different levels of manual effort to compile the translated Rust code. In other words, the more compile-time safety is assumed in the C++ code, the more errors might be introduced during the translation process.

We start with the optimistic models, which make more assumptions about the input C++ code and support fewer features, and then transition to the pessimistic models, which make fewer assumptions about the input C++ code and support more features.

4.2.1 Safe Model

The transformations performed by the Safe model are presented in Table 4.1. This table demonstrates that data types are semantically preserved in the translation from C++ to Rust, in contrast to the handling of references and pointers. Notably, the mutability of a data type is determined by whether the corresponding type in C++ was qualified with the `const` specifier.

Furthermore, the pointer type is converted into a safe optional reference, representing a nullable safe reference in Rust. It is important to note that the presence or absence of the `const` qualifier is applied to the type `T` itself, not to the pointer. This means that the pointer can be reassigned to point to another memory location, resulting in a `mut` qualifier in both cases.

Finally, an unsafe reference in C++ is naively converted into a safe reference in Rust. In C++, references are not rebound (i.e., once a reference is initialized to refer to an object, it cannot be altered to point to a different object), and as a result, they are qualified as immutable variables, in contrast to pointers.

The program in Listing 14 demonstrates two fundamental pointer operations: taking the address of a variable in line 11, and dereferencing a pointer in lines 4, 5, and 7. Regarding the address-of operation, it is worth noting that this operation essentially represents an idiomatic immutable borrow wrapped around

C++ Variable Declaration	Rust Variable Declaration
`Type v`	`let mut v: Type`
`const Type v`	`let v: Type`
`Type *p`	`let mut p: Option<&mut Type>`
`const Type *p`	`let mut p: Option<&Type>`
`Type &r`	`let r: &mut Type`
`const Type &r`	`let r: &Type`

Table 4.1: Translation of C++ to Rust using the Safe Model

```cpp
int max(
  const int *a, const int *b
) {
  if (*a > *b) {
    return *a;
  }
  return *b;
}
int a = 1;
int b = 2;
int c = max(&a, &b);
```

```rust
pub fn max(
  mut a: Option<&i32>, mut b: Option<&i32>
) -> i32 {
  if **a.as_ref().unwrap() > **b.as_ref().unwrap() {
    return **a.as_ref().unwrap();
  }
  return **b.as_ref().unwrap();
}
let mut a: i32 = 1_i32;
let mut b: i32 = 2_i32;
let mut c: i32 = max(Some(&a), Some(&b));
```

Listing 14: A C++ `max` function and its translation to Rust using the Safe model.

an option in Rust. Additionally, as previously explained, safely dereferencing an option requires first calling `as_ref()` to avoid moving the inner object. Only after that can we safely call `unwrap()` to access the value being pointed to.

```cpp
const int *pa = &a;
const int *pb = &b;
if (!pb) {
  pa = pb;
}
```

```rust
let mut pa: Option<&i32> = Some(&a);
let mut pb: Option<&i32> = Some(&b);
if !pb.is_none() {
  pa = pb.as_ref().map(|x| &**x);
}
```

Listing 15: A C++ program with pointer reassignment and nullability check, translated to Rust using the Safe model.

More fundamental pointer operations are present in the program in Listing 15: making a nullability check in line 3, and reassigning a pointer to another memory location in line 4.

In the context of the nullability check, it is important to note that in Rust, `NULL` and `nullptr` from C++ are represented by the empty optional value `None`. This is why we use the `is_none()` method. Regarding reassigning a pointer, one must also avoid moving the inner value. To achieve this, we again call `as_ref()` first, and then call `map()` to clone the immutable reference referring to the inner value.

Additionally, one limitation of this model, as demonstrated by this example, is that line 4 of this Rust program would not compile if the references were mutable because it would result in two mutable references to the same value.

```
const int &max(
  const int *a, const int *b
) {
  if (*a > *b) {
    return *a;
  }
  return *b;
}
```

```
pub fn max<'a, 'b, 'c>(
  mut a: Option<&'a i32>, mut b: Option<&'b i32>
) -> &'c i32 where 'a: 'c, 'b: 'c {
  if **a.as_ref().unwrap() > **b.as_ref().unwrap() {
    return &**a.as_ref().unwrap();
  }
  return &**b.as_ref().unwrap();
}
```

Listing 16: Changing the return type of C++ `max` function and its consequences in translated Rust

If the return type of the `max` function is changed to return a reference instead of a value, manual effort will be required to compile the translated Rust code, as shown in Listing 16. Since there is a relationship between the function's arguments and the return reference value, explicit lifetime annotations are necessary. In this case, different lifetime parameters were manually added for each argument and the return type (`'a`, `'b` and `'c`, respectively in line 1 of Rust code). A restriction is imposed using a `where` clause, explicitly stating that the lifetimes of the variables with lifetimes `'a` and `'b` must be as long as the lifetime of the return value with lifetime `'c`, as shown in line 3 of Rust code.

There are other problems with this translation model, such as its lack of support for pointer arithmetic, multiple mutable borrow errors, missing lifetime annotations, and more. Later in this work, we translate the **WOFF2** library, a real codebase that heavily relies on pointer arithmetic and other low-level pointer operations.[1] Consequently, the model would be inadequate for automatically translating the entire C++ codebase to Rust.

4.2.2 Naive Model

The Naive model's transformations are shown in Table 4.2. The primary distinction between this model and the previous one lies in the conversion of C++ pointer types, which are now translated into unsafe raw pointers in Rust, qualified with `const` or `mut` depending on the original C++ pointee type. We named it 'Naive' because it directly maps C++ pointers to Rust pointers and C++ references to Rust references.

The program presented in Listing 17 demonstrates the previously mentioned C++ `max` function translated to Rust using the Naive model. The notable change is the inclusion of the `unsafe` keyword, which is evident in both the function signature (line 1) and the function call (line 8). This safety alteration arises from the dereferencing of raw pointers within the function, occurring in lines 3 and 4. Moreover, the address-of operation no longer needs to be wrapped around an optional value, as shown in the arguments passed into the function (line 8).

In the previous model, having two mutable pointers was impossible because they were translated into two optional mutable references, which the borrow checker restricts. However, in the current model,

[1] https://github.com/google/woff2/blob/master/src/buffer.h

C++ Variable Declaration	Rust Variable Declaration
`Type v`	`let mut v: Type`
`const Type v`	`let v: Type`
`Type *p`	`let mut p: *mut Type`
`const Type *p`	`let mut p: *const Type`
`Type &r`	`let r: &mut Type`
`const Type &r`	`let r: &Type`

Table 4.2: Translation of C++ to Rust using the Naive Model

```
1 pub unsafe fn max(mut a: *const i32,
2                   mut b: *const i32) -> i32 {
3   if *a > *b { return *a; }
4   return *b;
5 }
6 let mut a: i32 = 1_i32;
7 let mut b: i32 = 2_i32;
8 let mut c: i32 = unsafe { max(&a, &b) };
```

```
1 let mut pa: *mut i32 = &mut a;
2 let mut pb: *mut i32 = &mut b;
3 if !pb.is_null() {
4   pa = pb;
5 }
```

Listing 17: The previous C++ programs translated to Rust using the Naive model.

where pointer types are translated into unsafe raw pointers, the program on the right side of the Listing 17 passes the borrow checker because pointer types have no guarantees about aliasing or mutability, except that direct mutation is not allowed through a `*const T`. Also, the pointer comparison to `nullptr` is translated into the `is_null()` method call (line 3), and the pointer reassignment expression remains the same as in C++ (line 4). Thus, this model demonstrates greater interoperability with the original C++ code.

Although the lifetime annotations are not applied to the unsafe raw pointers present in the arguments of the function in Listing 18, a lifetime annotation is still required in the return type, as indicated in line 1, since the function returns a value by reference. Consequently, some manual work would still be necessary, albeit less than in the previous model, to pass the borrow checker and successfully compile this code.

References are frequently utilized in the **WOFF2** library.[2] However, their usage does not seem to introduce significant complexity, as they are primarily passed by reference as function arguments, and functions never return values by reference. Therefore, while this model may require some additional manual adjustments for translating the **WOFF2** library to Rust, the amount of work needed is likely to be minimal and easily manageable, making the Naive model suitable for the entire codebase translation process.

One advantage of the Naive model compared to the previous one is its increased interoperability

[2] https://github.com/google/woff2/blob/master/src/glyph.h

```rust
pub unsafe fn max<'a>(mut a: *const i32, mut b: *const i32) -> &'a i32 {
    if *a > *b { return &*a; }
    return &*b;
}
```

Listing 18: Translation of the C++ `max` function that returns value by reference into Naive Rust

with C++ regarding pointer offsets and pointer arithmetic, as shown in Listing 19. The function is again marked as unsafe (lines 1 and 9) because pointer offsets (line 4) and dereferencing a raw pointer (line 3) are considered unsafe operations in Rust and are used within the recursive function `strlen`. In the Reference Counting model, we will demonstrate how to translate this unsafe C++ function into safe Rust without requiring any additional manual work.

It is worth noting that a `char` in C++ is translated into an `i8` in Rust. This is because the `char` type in Rust represents a 'Unicode scalar value' and occupies 4 bytes, in contrast to C++ where it occupies only 1 byte. As a result, the casting of `char` to `i8` in the array initialization in lines 7 and 8 is necessary.

```rust
int strlen(const char *s, int n) {
  return *s
    ? strlen(s + 1, n + 1)
    : n;
}
char s[] = { 's', 't', 'r', '\0' };
auto len = strlen(s, 0);
```

```rust
pub unsafe fn strlen(mut s: *const i8,
                     mut n: i32) -> i32 {
  return if ((*s) != 0) {
    strlen(s.offset(1_i32 as isize), n + 1_i32)
  } else { n };
}
let mut s: [i8; 4] = [('s' as i8), ('t' as i8),
  ('r' as i8), ('\0' as i8)];
let mut len: i32 = unsafe {
  strlen(s.as_mut_ptr(), 0_i32) };
```

Listing 19: Translation of C++ recursive `strlen` function to unsafe Rust

4.2.3 Unsafe Model

Table 4.3 illustrates the transformations performed by the Unsafe model. The primary goal of this model is to minimize manual effort compared to the previous model. It achieves this by downgrading the unsafe reference type in C++ to the unsafe pointer type in Rust. Again, it is important to note that, upon examining the reference type conversions in Table 4.3, one can observe that the reference is never qualified with the initial `mut`, unlike the pointer type. This distinction implies that the reference cannot be rebound in Rust, maintaining the same semantics as in C++. Finally, the use of `std::ptr::NonNull` could also have been considered to preserve the non-null reference semantics of C++ in Rust.

The examples from the Naive model are equivalent to those from the Unsafe model, with one exception explained next. Using the Unsafe model, the translation of the version of the C++ `max` function that returns a value by reference requires no manual effort, as lifetime annotations are no longer need to be included manually, as shown in Listing 20. This is due to the fact that the return type has become an

C++ Variable Declaration	Rust Variable Declaration
`Type v`	`let mut v: Type`
`const Type v`	`let v: Type`
`Type *p`	`let mut p: *mut Type`
`const Type *p`	`let mut p: *const Type`
`Type &r`	`let r: *mut Type`
`const Type &r`	`let r: *const Type`

Table 4.3: Translation of C++ to Rust using the Unsafe Model

unsafe mutable raw pointer, rather than a safe constant reference, as shown in line 1.

In this way, by downgrading the safe reference type to the unsafe pointer type, this model completely bypasses the borrow checker. Therefore, multiple mutable borrows or missing lifetime annotations errors are no longer problematic. However, this also means sacrificing safety guarantees throughout the code, semantically identically to C++.

```
1 pub unsafe fn max(mut a: *const i32, mut b: *const i32) -> *const i32 {
2     if *a > *b { return &*a; }
3     return &*b;
4 }
```

Listing 20: Translation of the C++ `max` function that returns value by reference into Unsafe Rust

However, even within memory-safe C++ programs, some may not pass Rust's borrow checker unless the Unsafe model is used. The Rust compiler, being inherently conservative, cannot provide guarantees for the safety of every reference. For instance, in Listing 21, we present a memory-safe C++ program and its translations to Rust using both the Naive and Unsafe models. In the Naive Rust model, the Rust compiler fails to compile this simple program, while it compiles successfully using the Unsafe Rust model.

The program includes types and methods that are in the C++ Standard Library, but we will later explain how we dealt with that part of the translation. It begins by allocating a vector with 5 elements, all set to 1. Subsequently, it obtains a mutable reference to the vector's first element. Then, it clears the vector, invalidating the previously acquired mutable reference (i.e., causing it to become a dangling reference). Consequently, any subsequent access to this reference would result in undefined behavior and potential safety vulnerabilities.

However, this access will never occur because a condition checking the vector's emptiness is verified before attempting the access. Therefore, even though the original C++ program is memory-safe, its translation to Rust using a safer model may not work, as the Rust compiler cannot always prove memory safety of certain safe programs.

The Rust program using the Naive model does not compile because it attempts to borrow two mutable references simultaneously: the first one acquired in line 3 and the second one in line 5. The solution is to use the Unsafe model, which successfully bypasses the Rust borrow checker. Despite being marked as unsafe, it is still a memory-safe program in Rust.

```
1 std::vector<int> v(1, 5);
2 int &r = v[0];
3 v.clear();
4 if (!v.empty()) {
5   r += 1;
6 }
```

```
1 let mut v: Vec<i32> =
2   vec![5_i32; 1 as usize];
3 let r: &mut i32 =
4   &mut v[0_i32 as usize];
5 v.clear();
6 if !v.is_empty() {
7   (*r) += 1_i32;
8 }
```

```
1 let mut v: Vec<i32> =
2   vec![5_i32; 1 as usize];
3 let r: *mut i32 =
4   &mut v[0_i32 as usize];
5 v.clear();
6 if !v.is_empty() {
7   unsafe { (*r) += 1_i32; }
8 }
```

Listing 21: A memory-safe C++ program translated into Naive and Unsafe Rust

Overall, the Unsafe model is a strong choice for translating the **WOFF2** codebase into Rust, much like the previous Naive model. This preference arises because, despite the model not generating idiomatic Rust, it eliminates the need for manual intervention by entirely bypassing the borrow checker. Additionally, it is highly efficient in terms of memory and execution time and provides excellent interoperability, given the close semantics between the types used in C++ and Rust. Most importantly, it effectively preserves the safety (or lack thereof) of the original C++. As a result, we have chosen to implement and utilize this model for the translation of **WOFF2**, a process explained in a later chapter, and it will be used for comparison with the reference counting models in the upcoming chapters.

4.3 Reference Counting Models

The Reference Counting models provide a novel approach to modeling C++ and automatically translating it into safe Rust, with no additional manual effort, using only Rust's standard library. However, this comes at an additional run-time cost. As mentioned earlier, these models are built around reference counting types (i.e., `Rc` and `Weak`), which check lifetimes dynamically, and shared mutable containers (i.e., `RefCell`), which dynamically check for multiple mutable borrows.

This chapter begins by describing how these types can be combined to implement temporal safety and how they are used to translate C++ into safe Rust. It then proceeds to upgrade the model to support safe pointer arithmetic by introducing a new fat pointer, that works as an additional layer of safety. The chapter also demonstrates how these techniques can be employed as a memory sanitizer, with several examples that panic in Rust if undefined behavior is detected at run time.

However, since all these models share the attribute of incurring additional run-time costs, an optimization based on the Lazy Initialization Pattern is also implemented and analyzed. Finally, the limitations of the reference counting models are detailed.

It is important to highlight that the model exhibits similarities to Swift programming language's Automatic Reference Counting (ARC) memory management mechanism [31].

4.3.1 Lifetimes and Mutability

C++ Variable Declaration	Rust Variable Declaration
`Type v`	`let v: Rc<RefCell<Type>>`
`const Type v`	`let v: Rc<RefCell<Type>>`
`Type *p`	`let p: Option<Weak<RefCell<Type>>>`
`const Type *p`	`let p: Option<Weak<RefCell<Type>>>`
`Type &r`	`let r: Weak<RefCell<Type>>`
`const Type &r`	`let r: Weak<RefCell<Type>>`

Table 4.4: Translation of C++ to Rust using the Reference Counting Model

Table 4.4 demonstrates the conversions generated by the initial version of the Reference Counting model.

Starting with the data type translation, we encapsulate the `Type` around two Rust standard library types: `Rc` and `RefCell`. Rust's `Rc`[3] type is a single-threaded reference-counting pointer, similar to the `std::shared_ptr`[4] smart pointer found in the C++ Standard Library, with the distinction of not having an atomic counter. Additionally, it sets itself apart by only supporting multiple immutable pointers. Rust's `RefCell` type is a mutable memory location that shifts borrow checking from compile time to run time. Let's explore each of these types separately to understand their capabilities.

The program on the left side of the Listing 22 demonstrates some basic operations with the `Rc` type. It starts by creating a new `Rc` object stored in the variable `rc_a`, which is allocated on the heap. This example already incurs an additional cost, as each reference counting memory location is stored on the heap. However, we will explore optimizations in this regard in a later chapter.

In lines 2 and 3, more basic operations are performed. First, we dereference the smart pointer to access the inner value to verify that it is equal to 1 using an assertion. Next, we determine how many strong reference counting variables point to this memory location, which currently amounts to just one.

The program proceeds and starts a new scope in line 4. It creates another reference counting variable that points to the same memory location in line 5. This means the number of strong reference counting variables is incremented to 2, as shown in line 7. When this last variable goes out of scope in line 8, the number of strong reference counting variables drops to 1 again, as shown in line 9. Finally,

[3] https://doc.rust-lang.org/std/rc/struct.Rc.html
[4] https://en.cppreference.com/w/cpp/memory/shared_ptr

when the first variable (i.e., `rc_a`) goes out of scope, the number of strong reference counting variables drops to zero, and its allocated memory is automatically freed.

To conclude this example, the program demonstrates an alternative method to have several immutable variables referencing the same memory location, all without using safe references and, consequently, bypassing the conservative borrow checker, in this case regarding lifetimes. Previously, we showed the Unsafe model that could circumvent the borrow checker. However, notably, this Reference Counting model achieves the same but without resorting to unsafe code.

It is worth noting that, in the end, references and pointers will be modeled using weak references (i.e., using the `Weak` type) instead of incrementing the strong reference counting variable as shown earlier (i.e., using the `Rc::clone` operation), but the semantic differences will be further detailed.

```
1 let rc_a: Rc<i32> = Rc::new(1_i32);
2 assert_eq!(*rc_a, 1);
3 assert_eq!(Rc::strong_count(&rc_a), 1);
4 {
5   let rc_b: Rc<i32> = Rc::clone(&rc_a);
6   assert_eq!(*rc_b, 1);
7   assert_eq!(Rc::strong_count(&rc_a), 2);
8 }
9 assert_eq!(Rc::strong_count(&rc_a), 1);
```

```
1 let cell: RefCell<i32> = RefCell::new(1_i32);
2 let ref_a: &RefCell<i32> = &cell;
3 let ref_b: &RefCell<i32> = &cell;
4 *ref_a.borrow_mut() += 1;
5 *ref_b.borrow_mut() += 1;
6 assert_eq!(*cell.borrow(), 3);
```

Listing 22: Demo of Rust's `Rc` type and `RefCell` type.

The program on the right side of Listing 22 shows a demo of how to use Rust's `RefCell` type. The program begins with the creation of a new `RefCell` that contains an integer in line 1 and stores it in a variable named `cell`. Despite the keyword `mut` never being used, this memory location is mutable, and its borrow rules are checked dynamically. Then, the program proceeds to borrow two immutable references in lines 2 and 3, storing them in variables named `ref_a` and `ref_b`, respectively. Here, the program satisfies the Rust's borrow checker since the simultaneous existence of multiple immutable borrows is allowed.

Notably, the following lines are the more interesting ones. Even though we had borrowed two immutable references, the program is capable of safely modifying their referred inner values by incrementing them, as shown in lines 4 and 5 (i.e., calling and dereferencing `borrow_mut()`). Finally, the program checks that those immutable references were capable of changing the original inner value stored in the cell's owner in line 6.

Actually, this program exemplifies a commonly used design pattern in Rust called the **Interior Mutability Pattern**,[5] which enables you to modify data even in the presence of multiple immutable references to that data. This is an action that would typically violate the rules of the borrow checker. The pattern achieves this by encapsulating mutability and shifting the verification of borrowing rules from compile-

[5] https://doc.rust-lang.org/book/ch15-05-interior-mutability.html

time to run-time. Finally, it is worth noting that, in contrast to the `Rc` type, the `RefCell` type stores the inner memory location on the stack, avoiding additional memory allocation costs at run time.[6] [7]

From the previous two examples, it becomes evident that the `Rc` type lacks the capability to modify inner data while making the use of safe references unnecessary, as illustrated in Listing 22. On the other hand, the `RefCell` type cannot eliminate the use of safe references, and as a result, cannot remove explicit lifetime annotations, but it allows the safe modification of inner immutable data, as demonstrated in Listing 22. Therefore, we combine these two types together: the `Rc` type allows modeling safe references at run-time and the `RefCell` type enables modeling multiple mutable references simultaneously, offering a mechanism to safely manage both references and mutability in Rust.

The program in Listing 23 achieves the same functionality as the program in Listing 22 without using safe references and mutability, relying solely on safe Rust code. It starts by creating a new `Rc` object in line 1, which allocates a `RefCell` on the heap with an integer as its inner value. The program then safely creates additional owners of the inner value by calling `Rc::clone` in lines 2 and 3. These operations semantically mean borrowing two mutable references that exist simultaneously, which is the key point: we are bypassing the borrow checker's rule of having no more than one mutable reference at a time using safe Rust.

These additional strong reference counting variables, which represent mutable references, are then used to modify the inner value. This is achieved by first borrowing the `RefCell` mutably (i.e., by calling `borrow_mut()`) and then dereferencing its mutably borrowed value (i.e., by dereferencing the returned value of `RefMut` type) in lines 4 and 5.[8] Finally, in line 6, the program verifies that the first owner of the inner value was successfully incremented twice.

```
1 let cell: Rc<RefCell<i32>> = Rc::new(RefCell::new(1_i32));
2 let ref_a: Rc<RefCell<i32>> = Rc::clone(&cell);
3 let ref_b: Rc<RefCell<i32>> = Rc::clone(&cell);
4 *ref_a.borrow_mut() += 1;
5 *ref_b.borrow_mut() += 1;
6 assert_eq!(*cell.borrow(), 3);
```

Listing 23: Managing mutable data with multiple owners combining `Rc` and `RefCell`.

While it is possible to represent C++ references as another strong reference counting pointer in Rust, as we have done in the previous examples, this modeling approach would introduce different semantics into the translated code. Specifically, destructors and the deallocation of corresponding memory would only be executed once the last reference counting pointer went out of scope.

As a consequence, all dangling pointers and references would be eliminated and made safe in one

[6] https://doc.rust-lang.org/src/core/cell.rs.html#678
[7] https://doc.rust-lang.org/stable/src/core/cell.rs.html#1987
[8] https://doc.rust-lang.org/std/cell/struct.RefCell.html#method.borrow_mut

step. While this approach might seem promising in terms of safety, it could introduce other issues, such as moving the destructor call to a different point in the program. Therefore, this could potentially lead to additional memory safety bugs, memory leaks (i.e., through strong reference cycles between class instances) or incorrect outputs, particularly when dealing with user-defined destructors that have side-effects. Nevertheless, it is a modeling technique to be considered.

In this Reference Counting model, we used Rust's safe type `Weak` to represent the reference type of C++, as illustrated in Listing 24. The first difference from the program in Listing 23 is that, instead of calling `Rc::clone` to create a strong reference, it calls `Rc::downgrade` to create a weak reference, as demonstrated in lines 2 and 3.

Secondly, to access the inner value from a weak reference, it must first be upgraded to a strong reference by calling `upgrade()`. This method returns an `Option` that may result in either `Some` value or `None`, depending on whether there is still a strong reference or not. In other words, if the call to `upgrade()` returns `None`, the Rust program will panic and print the error message passed to the method `expect()`. Essentially, a program that accesses a dangling reference in C++ panics when translated to Rust using the Reference Counting model. Finally, once we have safe access to the strong reference, we can dereference the inner borrowed value, much like the previous program (lines 4, 5 and 6).

The final state is characterized by a single strong reference counting variable that represents the owner, along with two weak references that represent the two mutable references, as shown in lines 7 and 8, respectively. It is important to emphasize that this model involves single ownership, which distinguishes it from the previous modeling strategy without weak references that results in multiple owners. In Listing 23, the program concludes with a strong reference counter set to 3, while in contrast, the program in Listing 24 concludes with just one.

```
1 let cell: Rc<RefCell<i32>> = Rc::new(RefCell::new(1_i32));
2 let ref_a: Weak<RefCell<i32>> = Rc::downgrade(&cell);
3 let ref_b: Weak<RefCell<i32>> = Rc::downgrade(&cell);
4 *ref_a.upgrade().expect("ub: dangling reference").borrow_mut() += 1;
5 *ref_b.upgrade().expect("ub: dangling reference").borrow_mut() += 1;
6 assert_eq!(*cell.borrow(), 3);
7 assert_eq!(Rc::strong_count(&cell), 1);
8 assert_eq!(Rc::weak_count(&cell), 2);
```

Listing 24: Managing mutable data with a single owner using the Reference Counting model.

The Interior Mutability Pattern is also visible in Table 4.4 where the `mut` qualifier is no longer used. Moreover, the primary difference between the translation of the pointer type and the reference type is the additional level of indirection represented by the `Option` type, which semantically accounts for the nullability difference: references cannot be null in C++, but pointers can.

The program in Listing 25 demonstrates the translation of the C++ `max` function from the previous

chapter using reference counting. In contrast to the compile-time models, it is apparent that this Reference Counting model requires considerably more lines, making it more verbose than the other models. For example, when comparing it to the program in Listing 20.

Moreover, in line 15, the distinction between creating a new pointer and creating a new reference can be observed, which corresponds to the difference between the address-of and borrowing operations. Creating a reference is as simple as calling `Rc::downgrade` and passing the owner as argument, while creating a pointer involves the additional step of wrapping the weak reference within the `Some` variant of Rust's `Option` enum, as shown in line 7.

To access the inner value of a pointer, the process is similar to accessing the inner value of a weak reference, as explained in the previous example. However, an additional check must be performed to ensure the pointer is not null. In line 3, for instance, the program first calls `as_ref()` to avoid moving the inner value and then calls `expect()`. If the pointer is null (i.e., the `Option` has a `None` value), `expect()` will panic with the specified error message. Otherwise, it returns an immutable reference to the unwrapped value, justifying the first dereference operation, as seen in line 3. This is identical to what occurred in the Safe model explained in the previous chapter.

Only after successfully passing this additional nullability safety check can one proceed to access the inner value of the strong reference counting object. This is done identically to how it happened with the reference type: first by borrowing immutably and then dereferencing its return value. This process occurs whenever the program dereferences and accesses the inner value, as seen in the operations starting in lines 3 and 5.

It is worth noting that in the original C++ code, although the dereferencing pointer operations in the binary comparison operator and the return statement are syntactically identical, they have different semantics. Specifically, these dereferencing operations are represented differently in Clang's Abstract Syntax Tree due to the presence of an additional implicit lvalue-to-rvalue cast. Consequently, in the program shown in Listing 25, the translated Rust code does not execute the final borrowing and dereferencing operations in the return statement, as it does for the left-hand and right-hand side variables of the binary comparison operator. In the return statement, the expression returns a reference, whereas the expressions of the left and right-hand side operands return rvalues.

However, the program still passes all safety checks, including nullability and temporal safety checks, before creating another reference (as demonstrated in line 10, for instance, when `Rc::downgrade` is called). This could lead to a premature panic in Rust because, according to the C++ standard, a reference may dangle, but then the code is not allowed to use it (i.e., the behavior is undefined). Nevertheless, with this translation model, the program panics if a reference is created from a dangling pointer before an access occurs.

As a final note on the program in Listing 25, it is clear that besides the extra heap allocations in

comparison to the original C++ code, there are also numerous additional run-time safety checks being executed by the method call `expect()`, present in lines 3, 4, 5, 6, 7, 8, 10, and 11. An interesting optimization can already be considered to reduce the number of local run-time checks by removing the redundant and unnecessary ones, such as those in lines 7, 8, 10, and 11.

```
1  pub fn max(a: Option<Weak<RefCell<i32>>>,
2             b: Option<Weak<RefCell<i32>>>) -> Weak<RefCell<i32>> {
3    if (*(*a.as_ref().expect("ub: null pointer")).upgrade()
4            .expect("ub: dangling pointer").borrow())
5      > (*(*b.as_ref().expect("ub: null pointer")).upgrade()
6            .expect("ub: dangling pointer").borrow()) {
7      return Rc::downgrade(&(*a.as_ref().expect("ub: null pointer"))
8        .upgrade().expect("ub: dangling pointer"));
9    }
10   return Rc::downgrade(&(*b.as_ref().expect("ub: null pointer"))
11     .upgrade().expect("ub: dangling pointer"));
12 }
13 let a: Rc<RefCell<i32>> = Rc::new(RefCell::new(1_i32));
14 let b: Rc<RefCell<i32>> = Rc::new(RefCell::new(2_i32));
15 let c: Weak<RefCell<i32>> = max(Some(Rc::downgrade(&a)), Some(Rc::downgrade(&b)));
```

Listing 25: Translation of the C++ `max` function that returns value by reference into Rust using Reference Counting

The program in Listing 26 demonstrates more basic C++ operations translated into Rust using the Reference Counting model. Specifically, it shows that it is possible to create new references and pointers without panicking in Rust before encountering undefined behavior. This is in contrast to the previous example, as illustrated in lines 3, 4, and 5, by calling `clone()` without the need for run-time safety checks.

```
1  int x = 1;
2  int y = x;
3  int &r1 = x;
4  int &r2 = r1;
5  int *p1 = &r1;
6  int *p2 = p1;
```

```
1  let x: Rc<RefCell<i32>> = Rc::new(RefCell::new(1_i32));
2  let y: Rc<RefCell<i32>> = Rc::new(RefCell::new(*x.borrow()));
3  let r1: Weak<RefCell<i32>> = Rc::downgrade(&x);
4  let r2: Weak<RefCell<i32>> = r1.clone();
5  let p1: Option<Weak<RefCell<i32>>> = Some(r1.clone());
6  let p2: Option<Weak<RefCell<i32>>> = p1.clone();
```

Listing 26: Translation of basic operations in C++ to Rust using Reference Counting

Some of the basic operations of the Reference Counting model have already been demonstrated. However, there are still problems with the translation of the pointer type, which become apparent in the program presented in Listing 27. The current translation model does not support pointer reassignment, as seen in line 3 of C++ code, or the creation of a pointer to a pointer type, as seen in line 4 of C++ code. To address both of these issues, it is necessary to add an additional layer to the current translation of the pointer type. This can be achieved by wrapping it with another layer of `Rc` and `RefCell`. First, the Interior Mutability Pattern allows us to change the inner value without requiring mutability, enabling pointer reassignment, as demonstrated in line 4 of Rust code. Second, a variable of pointer to a pointer

type can be created by downgrading the outer `Rc` and `RefCell`, as shown in line 6 of Rust code.

Therefore, with this latest update, there is an additional run time cost associated with unwrapping and allocating a pointer on the heap. Additionally, a static analysis to remove this extra layer could be implemented in a later optimization pass.

```
1  int x = 0;
2  int *p = &x;
3  p = nullptr;
4  int **pp = &p;
```

```
1  let x: Rc<RefCell<i32>> = Rc::new(RefCell::new(0_i32));
2  let p: Rc<RefCell<Option<Weak<RefCell<i32>>>>>
3    = Rc::new(RefCell::new(Some(Rc::downgrade(&x))));
4  *p.borrow_mut() = None;
5  let pp: Rc<RefCell<Option<Weak<RefCell<Option<Weak<RefCell<i32>>>>>>>>
6    = Rc::new(RefCell::new(Some(Rc::downgrade(&p))));
```

Listing 27: Translating a C++ program with basic pointer operations into Rust using the Reference Counting model

The types in the variable declarations in the program in Listing 27 have clearly become too complex and verbose. Fortunately, Rust offers a type alias operation, which permits the definition of a new name for an existing type, as demonstrated in Listing 28.[9] This way, the types become simpler, increasing the readability of the programs. For example, a pointer-to-pointer to an integer type becomes `Value<Pointer<Pointer<i32>>>`, as opposed to the type name in line 5 of the Rust code.

```
1  pub type Value<T>     = Rc<RefCell<T>>;
2  pub type Reference<T> = Weak<RefCell<T>>;
3  pub type Pointer<T>   = Option<Reference<T>>;
```

Listing 28: Type aliasing in the Reference Counting model

C++ Variable Declaration	Rust Variable Declaration
`Type v`	`let v: Value<Type>`
`const Type v`	`let v: Value<Type>`
`Type *p`	`let p: Value<Pointer<Type>>`
`const Type *p`	`let p: Value<Pointer<Type>>`
`Type &r`	`let r: Reference<Type>`
`const Type &r`	`let r: Reference<Type>`

Table 4.5: Translation of C++ to Rust using the Reference Counting Model with type aliasing

The Table 4.5 shows the updates of the Reference Counting model with type aliasing. From now on, types are going to be represented according to these new names.

Up to this point, we have observed the effectiveness of this model in ensuring temporal safety for stack-allocated values. To conclude this section, the programs showcased in Listing 29 demonstrate

[9] https://doc.rust-lang.org/reference/items/type-aliases.html

a memory-unsafe C++ program translated into a memory-safe Rust program using reference counting. The program starts by declaring a null pointer in line 1. Subsequently, the if-condition evaluates to true since the pointer points to null in line 2 (or to `None` in Rust code). Consequently, the program enters the if-then block, initiating a new scope.

Within this scope, an integer variable named `value` is declared in line 3, and the pointer is assigned to point to it in line 4. However, once the variable `value` goes out of scope in line 5, in Rust, the strong reference counter drops to zero, leading to the deallocation of its memory, and the pointer becomes dangling. Therefore, in the final variable declaration of the program, a dangling pointer is dereferenced. While in C++, such an operation leads to undefined behavior, the Reference Counting model in Rust takes a different approach. Instead of resulting in undefined behavior, it panics and halts execution in line 8.

In this way, the Reference Counting model relies on a single strong reference to represent the owner of a particular value, following the **Resource Acquisition Is Initialization** (RAII) programming idiom. Once the owner goes out of scope, all weak references referring to it, associated with either pointers or references, become dangling and unsafe to access. Consequently, the safe Rust program panics and prints an error message indicating the occurrence of undefined behavior. With this property, the Reference Counting model in Rust ensures that every memory location access through a reference or a pointer is safe.

```
1  int *ptr = nullptr;
2  if (!ptr) {
3    int value = 0;
4    ptr = &value;
5  }
6  int other = *ptr;
```

```
1  let ptr: Value<Pointer<i32>> = Rc::new(RefCell::new(None));
2  if !(*ptr.borrow()).is_some() {
3    let value: Value<i32> = Rc::new(RefCell::new(0_i32));
4    *ptr.borrow_mut() = Some(Rc::downgrade(&value));
5  }
6  let other: Value<i32> = Rc::new(RefCell::new(
7    *(*(*ptr.borrow()).as_ref().expect("ub: null pointer"))
8      .upgrade().expect("ub: dangling pointer").borrow()));
```

Listing 29: A memory unsafe program in C++ translated into safe Rust using the Reference Counting model

As of now, this model still faces limitations. It lacks support for C++ `new` and `delete` operators and does not handle safe pointer arithmetic.

C++ has two fundamental operations to manage memory on the heap: the `new` and `delete` operators. The program in Listing 30 demonstrates the translation of these two operations into unsafe Rust. Essentially, the unsafe model in Rust allocates an integer on the heap using `Box::new`, storing it inside a `Box`, which is similar to the `std::unique_ptr` type present in the C++ Standard Library. However, to maintain the same semantics as the original C++ code, this allocated memory cannot be deallocated at the end of the scope. Therefore, the program releases its ownership by calling `Box::leak`, as shown in line 1. That is, if no one frees this allocated memory in the future, the program leaks memory, identically

to C++. However, it is worth noting that Rust considers the leaking operation to be memory-safe.[10] This work classifies operations as safe if they align with Rust's own safety criteria.

The unsafe part of the program appears in line 2 because the function call to `Box::from_raw` is considered unsafe by Rust. This function allows the creation of a new `Box` from an unsafe raw pointer. The reason why an unsafe block is needed is because the Rust compiler cannot statically validate the preconditions of this function (e.g., that the code does not call `Box::from_raw()` twice on the same pointer). In this program, it is obviously safe, and therefore, a new `Box` is created. Finally, the call to `::std::mem::drop`[11] just invokes the drop destructor of the `Box` and the respective destructor of the pointee type, if there is one, freeing the allocated memory pointed to by the pointer `p`. Consequently, any access to this memory location through the pointer `p` results in undefined behavior.

It is worth noting that, even without explicitly calling `::std::mem::drop`, Rust would free this memory because the `Box` here is a temporary, whose scope is the statement - up until the semicolon. Therefore, the destructor would be run automatically at the end of the statement.

```
1  int *p = new int(5);
2  delete p;
```

```
1  let mut p: *mut i32 = Box::leak(Box::new(5_i32));
2  ::std::mem::drop(unsafe { Box::from_raw(p) });
```

Listing 30: Translating the `new` and `delete` operators into unsafe Rust

Contrary to the Unsafe model but following the same pattern of releasing the ownership and reacquiring it from and to an owner-pointer, it is possible to translate the `new` and `delete` operators to safe Rust using the Reference Counting model. However, instead of using the `Box` type as the owner it uses the `Rc` type.

Since the translation of the `new` operator into Rust using reference counting involves several steps, a helper function was implemented to improve the readability of the programs, as shown in the program on the left side of the Listing 31. As mentioned before, the idea is the same as in the Unsafe model. The function `alloc` starts by declaring a variable that represents an owner in line 1, similar to the function call `Box::new`. It then creates a pointer in line 2 by wrapping the weak reference around an optional value, following the same process as described earlier.

The program proceeds, and in line 3, `Rc::into_raw` is called, essentially releasing the ownership held by `Rc` by moving it into an unsafe raw pointer, similar to the function call `Box::leak`. The purpose is to avoid calling the destructor at the end of the scope, which would render the locally returned pointer dangling. As previously mentioned, if the program does not free the allocated memory, it results in a memory leak. However, it is important to emphasize that Rust treats memory leaks as memory-safe. Finally, the optional weak reference, representing the translated pointer, is returned in line 5.

As a side note, the variable `t` has a generic type `T`, which is a template type argument passed during

[10] https://doc.rust-lang.org/std/boxed/struct.Box.html#method.leak
[11] https://doc.rust-lang.org/std/mem/fn.drop.html

the function call, as indicated in line 1. It is important to be aware that template type parameters differ between C++ and Rust, and these differences will be explained in a future chapter.

```
1  pub fn alloc<T>(t: T) -> Pointer<T> {
2    let owner = Rc::new(RefCell::new(t));
3    let ptr = Some(Rc::downgrade(&owner));
4    let _ = Rc::into_raw(owner);
5    ptr
6  }
```

```
1  pub fn dealloc<T>(ptr: &Pointer<T>) {
2    let weak = ptr.as_ref()
3      .expect("ub: deleting a null pointer");
4    assert_eq!(Weak::strong_count(weak),
5      1, "ub: double free");
6    unsafe {
7      let strong = weak.upgrade()
8        .expect("ub: dangling pointer");
9      let _ = Rc::from_raw(Rc::as_ptr(&strong));
10   }
11   assert_eq!(Weak::strong_count(weak), 0);
12   assert_eq!(Weak::weak_count(weak), 0);
13 }
```

Listing 31: Implementing the `new` and `delete` operators in Rust using Reference Counting

The `new` operator involves only safe Rust code. However, implementing a safe `delete` operator in Rust using reference counting is more complex. To address this complexity, a helper function that deallocates a safe pointer has been implemented, as demonstrated in the program on the right side of the Listing 31. The `dealloc` function takes an immutable reference to a reference-counted pointer, as illustrated in line 1. It begins by unwrapping the inner weak reference within the pointer, checking if the pointer is null, in which case it panics with the appropriate error message, as shown in lines 2 and 3.

Despite this function using unsafe code, it performs an assertion check before execution. Under this premise, the execution is considered safe, although there is an issue that will be detailed later. In line 4, the program verifies that the value to which the pointer points to has a single owner by calling `Weak::strong_count`. Having more than one owner is a state the program should not reach, and having zero owners indicates a double-free error. In C++, this would result in undefined behavior, but in Rust using reference counting, it leads to a panic.

With this assumption verified, in line 6, the program opens an unsafe block which is safe to execute. Within this block, another strong reference counting variable is created by upgrading the weak reference, as shown in line 7. This brings the program to a state with two strong reference counts: the previously leaked one during the call to function `alloc` and this new one. Finally, in line 9, the program follows the same pattern as in the Unsafe model. The program first obtains a valid unsafe raw pointer to the memory location managed with reference counting by calling `Rc::as_ptr`, which neither consumes the `Rc` (passed by reference) nor affects the strong counts.[12] This unsafe raw pointer is valid because the assertion in line 4 ensures that there is one strong count in the `Rc`.

Next, this valid unsafe raw pointer is passed into the function `Rc::from_raw` to create a new strong reference counting variable, similar to the function call `Box::from_raw`. It is important to note that this

[12] https://doc.rust-lang.org/std/rc/struct.Rc.html#method.as_ptr

last call does not affect the reference counter. Consequently, the program maintains only two strong reference counting variables. In essence, this step recovers the reference to the previously leaked strong reference, which occurred during the `alloc` function call when `Rc::into_raw` was used. At the end of the unsafe block's scope, Rust automatically takes care of calling the destructor of the two reference counting variables, effectively reducing the strong reference count to zero and deallocating the memory (i.e., the program does not need to explicitly call `::std::mem::drop`).

After deallocation is complete, the number of strong counts must be equal to zero, as verified by the assertion in line 11. The Rust Standard Library documentation states that if no strong pointers remain, `Weak::weak_count()` will return zero,[13] as verified by the assertion in line 12. From this point on, any access through a weak reference to this memory location will panic, since the program will be accessing a dangling pointer or reference.

This function demonstrates the use of an unsafe block that is safe to execute, much like many functions in Rust's Standard Library. The function makes the unsafe block safe for execution by asserting conditions before use. By line 5, it is evident that the call to `upgrade()` will yield a `Some` value. As an optimization, the unsafe function `unwrap_unchecked()` could be used instead of `expect()` to unwrap the option without additional checks.[14]

Finally, the program in Listing 32 shows the translation of the C++ program present in Listing 30, which calls the `new` and `delete` operators, into safe Rust using Reference Counting. However, as mentioned before, there is still a semantics problem with this translation: pointers that are stack-allocated are not distinguished from pointers that are heap-allocated. In other words, they both have the same type, `Value<Pointer<T>>`.

Deleting a stack-allocated variable in C++ is undefined behavior, and currently the same applies in translated Rust using reference counting (i.e., it causes a double-free error when the stack-allocated value goes out of scope). As a result, this can currently be considered an unsafe operation. The issue arises because the assertion in line 3 of Listing 31 evaluates to true if the pointer points to a stack-allocated value. However, this problem is resolved by introducing a fat pointer as an additional layer of safety to address safe pointer arithmetic, distinguishing between the types of pointers that point to stack-allocated or heap-allocated values. This fat pointer carries additional information, as explained in the next subsection.

```
1 let ptr: Value<Pointer<i32>> = Rc::new(RefCell::new(alloc(5)));
2 dealloc(&(*ptr.borrow()));
```

Listing 32: Translating the `new` and `delete` operators into Rust using Reference Counting

[13] https://doc.rust-lang.org/std/rc/struct.Weak.html#method.weak_count
[14] https://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap_unchecked

4.3.2 Bounds Checking

Indexing operations in Rust are safe[15] because Rust performs bounds checking by default. Up to this point, the Reference Counting translation mechanism has leveraged Rust's run-time safety mechanisms to model C++ code into a safe version in Rust. In this way, a fat pointer has been implemented, which utilizes Rust's bounds checking, providing both temporal and spatial safety for stack-allocated memory, as demonstrated in Listing 33.

An `enum` in Rust is different from an `enum` in C++. In Rust, `enums` are **Algebraic Data Types**, a common structure in functional programming languages.[16] They can be used identically, but as will be demonstrated, in Rust, they are more powerful.

The program on the left side of the Listing 33 begins by defining an `enum`, named `PointerKind`, which specifies various variants, depending on the memory location to which the pointer points. In C++, a pointer can be null, and in Rust, when a pointer is null, it is assigned to the `Null` variant, as illustrated in line 4. Additionally, a pointer can point to a single memory location on the stack. In this case, the pointer is assigned to the `Single` variant, as shown in line 3. Essentially, this is the simplified version that supports temporal safety, as explained in the previous chapters.

It is worth noting that, as nullability is already represented by the `Null` variant, the `Reference` type no longer needs to be wrapped around an `Option`, as demonstrated in lines 3 and 4.

Moreover, a pointer can point to an element located within a stack-allocated array, in which case it will have the variant `Slice`, as shown in line 2. Here, it is important to emphasize several aspects. First, in order to take advantage of Rust's bounds checking a reference to the entire array must be taken, justifying the outer `Reference` present in line 2.

Finally, a pointer might point to arrays of different sizes during the program's execution. Consequently, the size of the array must be defined dynamically. This is why a stack-allocated array in C++ is translated into a heap-allocated array in Rust using the Reference Counting model with bounds checking, as illustrated by the `Box` type in line 2.

This way, the C++ pointer type is translated into Rust's pointer type as defined in line 1 of program on the right side of the Listing 33. This new fat pointer has two fields: `ptr` of type `PointerKind`, representing the memory location where the pointed value is stored, and `offset` of type `isize`, representing the current offset.

This new definition of the pointer type in the Reference Counting model obviously adds additional complexity to support safe pointer arithmetic. It expands the reference counting memory management mechanisms by adding an additional safety layer, and consequently, incurs run time costs.

There are two fundamental operations regarding this fat pointer: `as_pointer()` and `as_reference()`.

[15] https://doc.rust-lang.org/std/ops/trait.Index.html
[16] https://doc.rust-lang.org/std/keyword.enum.html

```
pub enum PointerKind<T> {
    Slice(Reference<Box<[Value<T>]>>),
    Single(Reference<T>),
    Null,
}
```

```
#[derive(Clone, Default)]
pub struct Pointer<T> {
    ptr: PointerKind<T>,
    offset: isize,
}
```

Listing 33: Implementing a fat pointer in Rust that performs bounds checking

The `as_pointer()` operation is responsible for creating a new pointer and classifying the memory location accordingly. On the other hand, the `as_reference()` operation is responsible for performing the bounds checking, returning back the respective weak reference in case of success, and panicking otherwise.

The program in Listing 34 demonstrates the operations to create temporal and spatial safe pointers. A `trait` was defined to facilitate the translation of these operations by the compiler. A `trait` is similar to a concept found in object-oriented programming languages called `interface`, albeit with some distinctions.[17] Essentially, a `trait` defines abstract functionality that certain types can share, enabling shared behavior among them.

In this case, the shared behavior is the address-of operation that creates a new pointer. The program defines a trait named `AsPointer` with a generic template type parameter `T`, representing the pointee or array element type, as shown in line 1 in the program on the right side of the Listing 34. A type that implements this `trait` must define the `as_pointer()` method, which given a variable of that type (i.e., `&self`) creates a pointer pointing to itself, as specified in line 2.

Additionally, the program on the left side of the Listing 34 presents a portion of the pointer structure's implementation, starting at line 1. This part of the code currently includes only one method named `new` which is responsible for creating a new `Pointer` (represented as `Self`) by taking a `PointerKind` and an `offset` as arguments, as showcased in lines 2 and 3.

The implementation of the `AsPointer trait` for the reference counted Rust type that corresponds to the stack-allocated array in C++ is defined in line 5 of the program on the right side. In this implementation, the `AsPointer trait` is implemented for `Value<Box<[Value<T>]>>`, representing the type for an owner of a stack-allocated array. As mentioned before, due to the pointer's capability to point to arrays of varying sizes, the array is allocated on the heap, as indicated by the presence of the `Box` type.

Regarding the method's implementation, the program returns a new pointer classified as the `Slice` variant with an initial `offset` of zero, as seen in line 7. This sets the pointer to point to the first element of the array. Notably, unlike the previous reference counting models that downgraded a single element of the array, the pointer's structure captures a weak reference to the entire array, as demonstrated by the `Rc::downgrade(self)` call in line 8. This approach is used to later take advantage of Rust's safe indexing and bounds checking capabilities.

[17] https://doc.rust-lang.org/book/ch10-02-traits.html

Furthermore, the implementation of this `trait` for a pointer to a single-element type is depicted in line 7 of the program on the left side. The implementation is similar to the previous one. However, in this case, the new pointer is classified with the `Single` variant, as shown in line 9, in contrast to the `Slice` variant. This distinction is crucial to explicitly convey that the pointer refers to a single element, as opposed to an element located within a stack-allocated array.

Finally, the definition of this auxiliary `trait` simplifies the translation process, in the same way an `interface` simplifies the utilization of a shared operation among various types through polymorphism. However, Rust claims to offer zero-cost abstractions, and therefore, these method calls should incur zero run time cost.

```
1  impl<T> Pointer<T> {
2    fn new(weak: PointerKind<T>,
3           offset: isize) -> Self {
4      Pointer { ptr: weak, offset }
5    }
6  }
7  impl<T> AsPointer<T> for Value<T> {
8    fn as_pointer(&self) -> Pointer<T> {
9      Pointer::new(PointerKind::Single(
10       Rc::downgrade(self)), 0)
11   }
12 }
```

```
1  pub trait AsPointer<T> {
2    fn as_pointer(&self) -> Pointer<T>;
3  }
4
5  impl<T> AsPointer<T> for Value<Box<[Value<T>]>> {
6    fn as_pointer(&self) -> Pointer<T> {
7      Pointer::new(PointerKind::Slice(
8        Rc::downgrade(self)), 0)
9    }
10 }
```

Listing 34: Implementing a trait in Rust for the creation of temporal and spatial safe pointers

On the other hand, there is the `as_reference()` operation, as illustrated in Listing 35. It is part of the `Pointer` structure's implementation. In contrast to the previous operation in Listing 34, which returns a `Pointer`, this one returns a `Reference` to the pointee value, as shown in line 1, after performing bounds checking, thus providing spatial safety.

The function starts by performing pattern matching on the pointer, using the `match` keyword, as shown in line 2. The program will enter the first matching arm that corresponds to the kind of memory location the pointer points to. If the pointer is null, it matches the third and final arm (i.e., `PointerKind::Null`) and the program panics because we are trying to access a null pointer, as illustrated in line 12.

Additionally, if the pointer points to a single element, it matches the second arm in the `match` statement in line 8 (i.e., `PointerKind::Single`). In this case, the necessary bounds checking involves verifying that the current `offset` is zero, since a pointer pointing to a single element cannot move around in memory, as showcased in line 9 (although there are exceptions to this, which will be discussed in a future section). After passing the bounds check, `clone()` is called to create another weak reference, as shown in line 10.

Finally, if the pointer points to an element within a stack-allocated array, it matches the first arm in the `match` statement in line 3 (i.e., `PointerKind::Slice`). Here, three safety checks are performed. First,

the `offset` is converted from an `isize` to a `usize`, causing a panic if it is negative, thereby ensuring that the index used to access the array element is non-negative, as illustrated in line 4. Next, an additional use-after-free safety check is conducted on the owner of the stack-allocated array in line 5. At last, with a valid index and a strong reference to the array, we can safely access the array element and return a reference to it, performing bounds checking, as demonstrated in line 6.

```
1  pub fn as_reference(&self) -> Reference<T> {
2    match self.ptr {
3      PointerKind::Slice(ref weak) => {
4        let offset: usize = self.offset.try_into().expect("ub: index must be positive");
5        let strong = weak.upgrade().expect("ub: dangling pointer");
6        Rc::downgrade(&(*strong.borrow())[offset])
7      }
8      PointerKind::Single(ref weak) => {
9        assert_eq!(self.offset, 0, "ub: index out of bounds");
10       weak.clone()
11     }
12     PointerKind::Null => panic!("ub: null pointer"),
13   }
14 }
```

Listing 35: Implementing a method responsible for bounds checking

The program in Listing 36 demonstrates the use of the `as_pointer()` operation, showcasing the pointer's ability to safely point to an element within an array (line 6), a single element (line 7), and a null pointer (line 8). All of this is achieved through the use of safe Rust code.

```
1  int arr[3] = {
2      1,
3      2,
4      3,
5  };
6  int *ptr = arr;
7  int elem = 0;
8  ptr = &elem;
9  ptr = nullptr;
```

```
1  let arr: Value<Box<[Value<i32>]>> = Rc::new(RefCell::new(Box::new([
2      Rc::new(RefCell::new(1_i32)),
3      Rc::new(RefCell::new(2_i32)),
4      Rc::new(RefCell::new(3_i32)),
5  ])));
6  let ptr: Value<Pointer<i32>> = Rc::new(RefCell::new(arr.as_pointer()));
7  let elem: Value<i32> = Rc::new(RefCell::new(0_i32));
8  (*ptr.borrow_mut()) = elem.as_pointer();
9  (*ptr.borrow_mut()) = Pointer::null();
```

Listing 36: Translating a C++ program with a pointer to various memory locations into reference counted Rust

Finally, the program in Listing 37 displays the translation of a recursive `strlen` function, implemented in Listing 19, into safe Rust using reference counting, providing both temporal and spatial safety.

The program starts by creating a heap-allocated array that represents a stack-allocated array of characters in C++, as shown in line 9. It then calls the recursive `strlen` function passing by value a pointer, that results from an implicit array to pointer decay cast, and the initial size (line 13).

Inside the `strlen` function, the program safely dereferences the pointer `s` and compares its value to the null character, as shown in the if-condition starting in line 4. It ensures safety by first performing bounds checking (using `as_reference()` in line 3) and temporal safety checking (using `upgrade().expect()`

in lines 3 and 4).

If the pointer `s` points to a non-null character, the `strlen` function is called recursively. A pointer to the next character is passed by value into the function, as demonstrated in line 5, computed by calling a method named `offset` which essentially returns a new `Pointer` with its `offset` field incremented by one. Otherwise, it returns the current computed size present in the variable `n`, as shown in line 7.

```
1  pub fn strlen(s: Value<Pointer<i8>>, n: Value<i32>) -> Value<i32> {
2    return Rc::new(RefCell::new(
3      if ((*(*s.borrow()).as_reference().upgrade()
4               .expect("ub: dangling pointer").borrow()) != 0) {
5        (*strlen(Rc::new(RefCell::new((*s.borrow()).offset(1_i32 as isize))),
6                 Rc::new(RefCell::new((*n.borrow()) + 1_i32))).borrow())
7      } else { (*n.borrow()) }));
8  }
9  let s: Value<Box<[Value<i8>]>> = Rc::new(RefCell::new(Box::new([
10     Rc::new(RefCell::new(('s' as i8))), Rc::new(RefCell::new(('t' as i8))),
11     Rc::new(RefCell::new(('r' as i8))), Rc::new(RefCell::new(('\0' as i8)))])));
12 let len: Value<i32> = Rc::new(RefCell::new(
13     (*strlen(Rc::new(RefCell::new(s.as_pointer())), Rc::new(RefCell::new(0_i32))).borrow())));
```

Listing 37: Translation of C++ recursive `strlen` function to safe Rust

The `new[]` and `delete[]` operators in C++ are responsible for allocating and deallocating arrays on the heap, respectively. Besides the allocation and deallocation, these operators also call the constructor and destructor of each array element object, respectively.

To support these operations, we need to update the `enum PointerKind` definition to distinguish between arrays that are allocated on the heap and on the stack. Additionally, we apply the same idea to differentiate between pointers that point to single elements located on the stack and those on the heap. This resolves the issue previously mentioned in Listing 31.

The Listing 38 demonstrates how to update the `enum` and the `as_reference()` method to support these new features. The variants `AllocSlice` and `AllocSingle` were added to the `enum PointerKind` to represent pointers that point to elements within heap-allocated arrays (i.e., `new[]`) and single elements stored on the heap (i.e., `new`), respectively, as shown in lines 4 and 5 (left-side), effectively distinguishing them from values stored on the stack. Additionally, the match arms present in the `as_reference` method, as defined in Listing 35, were updated to accommodate these two new variants, as illustrated in lines 2 and 5 (right-side).

Finally, the instruction in Listing 39 (left-side) updates the first instruction of the `dealloc` function, as defined in Listing 31. This way, the program safely ensures that the call to `delete` is made over a pointer that points to memory allocated on the heap with a single element (left-side). The same concept can be applied to implement a translation for a safe `delete[]` operation (right-side). Whenever a program calls `delete` or `delete[]` on a pointer that does not point to a single element stored on the heap or the first element of a heap-allocated array (asserted with the pointer's `offset` field), respectively, the program

```
pub enum PointerKind<T> {
  Slice(Reference<Box<[Value<T>]>>),
  Single(Reference<T>),
  AllocSlice(Reference<Box<[Value<T>]>>),
  AllocSingle(Reference<T>),
  Null,
}
```

```
PointerKind::Slice(ref weak) |
  PointerKind::AllocSlice(ref weak) =>
// ...
PointerKind::Single(ref weak) |
  PointerKind::AllocSingle(ref weak) =>
// ...
```

Listing 38: Update `enum PointerKind` and `as_reference()` method

will safely halt execution.

```
let weak = match self.ptr {
  PointerKind::AllocSingle(ref w) => w,
  _ => panic!("ub: invalid delete"),
};
```

```
let weak = match self.ptr {
  PointerKind::AllocSlice(ref w) => w,
  _ => panic!("ub: invalid delete"),
};
```

Listing 39: Update safety checks to `delete` and `delete[]` translations into Rust using reference counting

Overall, the Reference Counting model, which ensures both temporal and spatial safety, stands as a robust choice for automatically translating the WOFF2 codebase into Rust. This model offers the same features as the Unsafe model but relies exclusively on safe Rust code. Although it may generate code that is less idiomatic and less interoperable with C++ compared to the Unsafe model, it also eliminates the need for manual work, but by relying on run-time borrow and lifetime checking. Furthermore, it serves as an intriguing case study for discussing and measuring the trade-offs between safety, memory consumption, and execution time.

4.3.3 Undefined Behavior

The model provides temporal and spatial safety by ensuring that whenever a program accesses an invalid memory location (e.g., out of bounds or deallocated memory location) it halts execution with an appropriate error message. In other words, translated Rust programs using the Reference Counting model panic at runtime whenever undefined behavior happens during the execution of the input C++ program. In this section, the Reference Counting model's ability to catch memory safety bugs at run time is demonstrated.

The C++ function `dangling` in Listing 40 creates a dangling reference by returning a reference to the local variable `x`, as illustrated in line 4. The action that triggers undefined behavior happens in the return statement of the function `main`, in line 8. Therefore, the translated Rust program using Reference Counting panics in line 11 printing the appropriate error message.

Moreover, the C++ function `release` in Listing 41 deletes a pointer in line 2, and thus every access to this memory location after line 6 is undefined behavior. In the return statement of the function `main`, in line 7, a dangling pointer is dereferenced, resulting in undefined behavior. In the same way, the

```
int &dangling() {
  int x = 1;
  int *p = &x;
  return *p;
}
int main() {
  int &x = dangling();
  return x;
}
```

```
pub fn dangling() -> Reference<i32> {
  let x: Value<i32> = Rc::new(RefCell::new(1_i32));
  let p: Value<Pointer<i32>> = Rc::new(RefCell::new(
    x.as_pointer()));
  return Rc::downgrade(&(*p.borrow()).as_reference()
    .upgrade().expect("ub: dangling pointer"));
}
pub fn main_0() -> Value<i32> {
  let x: Reference<i32> = dangling().clone();
  return Rc::new(RefCell::new(
    (*x.upgrade().expect("ub: dangling reference").borrow())));
}
```

Listing 40: A dangling reference caught at run-time using reference counting

translated Rust program using Reference Counting panics in line 9 with the appropriate error message.

```
void release(int *p) {
  delete p;
}
int main() {
  int *p1 = new int(1);
  release(p1);
  return *p1;
}
```

```
pub fn release(p: Value<Pointer<i32>>) {
  (*p.borrow()).delete();
}
pub fn main_0() -> Value<i32> {
  let p1: Value<Pointer<i32>> = Rc::new(RefCell::new(
    Pointer::alloc(1_i32)));
  release(Rc::new(RefCell::new((*p1.borrow()).clone())));
  return Rc::new(RefCell::new((*(*p1.borrow()).as_reference()
    .upgrade().expect("ub: dangling pointer").borrow())));
}
```

Listing 41: A dangling pointer caught at run-time using reference counting

The previous two examples demonstrate the Reference Counting model's capability to ensure temporal safety. In Listing 42, the C++ function `strlen` returns the number of characters in a C string, assuming it is null-terminated (line 3). If the caller passes a non-null-terminated string, this C++ function call triggers undefined behavior because it involves an out-of-bounds access when dereferencing the raw pointer `s` in line 3. Consequently, the Rust program that uses Reference Counting on the right panics at runtime (line 4) when calling `as_reference()`, leading to a runtime out-of-bounds exception, ensuring spatial safety. It is worth noting that the C++ postfix increment operator in line 3 is translated into the `postfix_inc()` method call in line 4, which is a helper trait implemented to ease the translation process (the same happens with the C++ prefix increment operator and Rust's `prefix_inc` method call).

4.3.4 Optimizing Reference Counting

The additional runtime overhead incurred by the Reference Counting models is significantly higher compared to the Compile Time models.

First, all variables are allocated on the heap, regardless of whether their types own a heap allocation (e.g., `std::vector`, `std::string`) or not (e.g., `int`, `double`, `char`). In concrete terms, the declaration

```cpp
size_t strlen(const char *s) {
  size_t count = 0;
  while (*s++) ++count;
  return count;
}
```

```rust
pub fn strlen(s: Value<Pointer<i8>>) -> Value<u64> {
  let count: Value<u64> = Rc::new(RefCell::new(
    (0_i32 as u64)));
  while ((*(*s.borrow_mut()).postfix_inc().as_reference()
        .upgrade().expect("ub: dangling pointer")
        .borrow()) != 0) {
    (*count.borrow_mut()).prefix_inc();
  }
  return Rc::new(RefCell::new((*count.borrow())));
}
```

Listing 42: An out of bounds access caught at run-time using reference counting

of N variables in C++ leads to N additional heap allocations in Rust when employing the Reference Counting model. But when reasoning about complexity, the memory allocation cost transitions from $O(1)$ to $O(N)$ for types that do not own a heap allocation. This contrasts to the memory allocation cost for types that own a heap allocation, which shifts from $O(N)$ to $O(N + N) = O(N)$, resulting in no additional cost in terms of complexity.

Additionally, pointers perform runtime bounds checking using the `as_reference()` method call and temporal safety checking using the `upgrade().expect()` method call. In contrast, references only need to pass temporal safety checks since they cannot move around memory as pointers can in C++. Lastly, to finally access the inner value, the borrow rules are dynamically checked, utilizing the `borrow()` or `borrow_mut()` method calls. These are the principal operations responsible for the runtime overhead required to ensure safety in the translated Rust code.

Furthermore, there are other aspects that contribute to runtime overhead, including the following: the pointer type is allocated on the heap to be managed by reference counting, enabling pointer-to-pointer type conversions; a new allocation occurs every time a variable that is implicitly cast from lvalue to prvalue is returned, potentially requiring optimizations such as return value optimization (as illustrated in line 13 in Listing 42); stack and heap-allocated arrays allocate every array element on the heap, even though their lifetimes are bounded by the array's owner; and there is only a single strong reference for each value. These are some of the issues within the Reference Counting model that could be optimized, leading to an improvement in the translated Rust code.

4.3.4.A Lazy Initialization

While all of these issues can be addressed through optimization in a subsequent static analysis following the initial translation step, this work aimed to evaluate the impact of implementing a custom Reference Counting Smart Pointer designed to optimize the excessive heap allocation calls utilized by the Reference Counting model.

The optimization is based on the **Lazy Initialization Pattern**, which postpones the construction of

an object until it is actually required. In this case, we want to delay the heap allocation of a given value, required by `Rc`, until a moment of aliasing either by an address-of operation (i.e., pointer type) or through a borrow operation (i.e., reference type). In essence, a value is initially instantiated on the stack. It is only when aliasing operations on the value occur that it is relocated to the heap, where it can be safely managed through reference counting.

The key observation behind this optimization is the fact that the majority of variables are never borrowed, making the reference counting management entirely unnecessary for them. These variables include those of trivially copyable types, local variables, values within stack- or heap-allocated arrays, among others.

The programs in Listing 43 demonstrate some basic operations present in the implementation of this custom `Rc` structure. The top-left program shows the structure's definition, starting in line 1. It consists of two attributes: `init`, representing the initial value stored on the stack, as shown in line 2; and `value`, whose inner value is constructed and allocated on the heap, safely managed by Reference Counting, as shown in line 3. It is worth noting that this implementation uses the `Rc` and `RefCell` types under the hood, following the same patterns as the Reference Counting model to achieve the same goals, safely bypassing lifetime and borrow checking. Additionally, it utilizes the `OnceCell` type, which is a cell capable of safely mutating the inner value only once.[18]

The top-right program in Listing 43 demonstrates the function `new`, responsible for creating a new `LazyRc` object. In this function, the argument `t` is moved into the `init` field, which allocates the inner value on the stack, as shown in line 3. This approach delays or avoids the expensive heap allocation. On the other hand, the `value` field starts as a new empty cell, as illustrated in line 4.

The bottom-left program in Listing 43 demonstrates the lazy initialization pattern in action. This method is semantically equivalent to the aforementioned `Rc::downgrade`, as shown in line 9. However, it begins by checking whether the `value` field has already been instantiated or not, which means verifying whether the reference counting management has already initialized or not, as shown in line 2. In the latter case, it proceeds to move the inner value from the `init` field, as illustrated in lines 3 and 4. Finally, it moves the inner value into a reference counting variable stored in the `value` field, as showcased in lines 5 and 6.

Lastly, the bottom-right program in Listing 43 demonstrates how to access the inner value based on whether the current inner value is under reference counting or not, as shown in lines 2 and 6.

It is worth noting that this implementation does not directly affect the Reference Counting model. Instead, it functions as an optimization for the underlying reference counting mechanism. As demonstrated in the evaluation chapter, it notably improves the performance of most programs.

[18] https://doc.rust-lang.org/nightly/std/cell/struct.OnceCell.html

```rust
pub struct LazyRc<T> {
  init: Option<RefCell<T>>,
  value: OnceCell<Rc<RefCell<T>>>,
}
```

```rust
pub fn new(t: T) -> Self {
  LazyRc {
    init: Some(RefCell::new(t)),
    value: OnceCell::new(),
  }
}
```

```rust
pub fn downgrade(&self) -> LazyWeak<T> {
  if self.value.get().is_none() {
    let cell = self.init.as_ref().unwrap();
    let moved_value = cell.take();
    let rc = Rc::new(RefCell::new(moved_value));
    let _ = self.value.set(rc);
  }
  LazyWeak {
    weak: Rc::downgrade(self.value.get().unwrap())
  }
}
```

```rust
fn inner(&self) -> &RefCell<T> {
  if let Some(value) =
        self.value.get() {
    return value;
  }
  self.init.as_ref().unwrap()
}
```

Listing 43: Implementing a custom Reference Counting Smart Pointer based on Lazy Initialization Pattern

4.3.5 Problems and Limitations

There are more problems arising from using reference counting to model the translation of C++ to safe Rust. For instance, while the `RefCell` type enables having multiple mutable references by dynamically checking the borrow rules, an exceptional problem arose and needed to be solved. The problem lies on the binary assignment operator, when the left-hand side expression is borrowed mutably (i.e., `borrow_mut()`) and the right-hand side borrows immutably the same expression (i.e., `borrow()`). For instance, the statement `x = x + 1` in C++ cannot be directly translated into `*x.borrow_mut() = *x.borrow() + 1` in Rust using the Reference Counting model, since it would panic with an already mutable borrowed error (i.e., `'already borrowed: BorrowMutError'`).

This occurs because the lifetime of the mutable borrow taken on the left-hand side lasts until the end of the statement,[19] overlapping with the lifetime of the immutable borrow taken on the right-hand side. Since this violates a fundamental borrow checking rule, Rust panics at run time.

To prevent this runtime error, the assignment operator can be split into two assignment statements, as follows: `let rhs = *x.borrow() + 1; *x.borrow_mut() = rhs`. However, since this can make the code cumbersome to read, a simple algorithm was implemented to detect common cases where this split is unnecessary (e.g., there is no need for splitting the assignment `x = 0`). The algorithm collects the left-hand side and right-hand side variable expressions and checks if their names overlap or include pointers or references of the same type. If they do not, the split is not performed. This algorithm is executed during the translation process.

Additionally, there is a minor implementation detail: all user-defined structures need an explicitly

[19] https://doc.rust-lang.org/std/cell/struct.RefCell.html#method.borrow_mut

defined `clone` method (implementing the `trait Clone`) that performs a deep clone of the structure's fields. This is necessary because a simple call to `clone()` on a structure would only increment the reference counter of its fields, and a cloned structure would consequently end up with fields pointing to the same memory locations as the fields of the original structure.

The Reference Counting model has limitations. These include its inability to effectively handle global variables, which are translated in the same manner as in Compile Time models. This is because the function call `Rc::new` is not marked as constant, and calls in statics are limited to constant functions. However, this problem could be addressed by lazily constructing the heap allocation (e.g., using the custom `LazyRc` type, with the `LazyRc::new` marked as a constant function).

Another challenge arises from the different memory layout employed by this model, which can complicate low-level implementations, such as pointer type casts. Furthermore, the model's approach necessitates the allocation of more objects on the heap than actually needed, resulting in increased memory usage. These are some of the constraints and considerations associated with the Reference Counting model.

Moreover, the model automatically eliminates certain memory safety bugs that arise from specific operations resulting in undefined behavior in C++. These bugs include reading uninitialized memory, since the model currently initializes memory with default zero values. Another scenario involves a pointer that originally points to a value within a `std::vector`, and the vector is reallocated. In C++, the pointer becomes dangling, but in the translation model that uses bounds checking, it is automatically updated to point to the new memory location. Lastly, the model is not **field sensitive**, meaning it treats the fields of a structure as separate areas in memory. Therefore, Rust will panic if there is an attempt to access these fields using pointer arithmetic between them. Nevertheless, these are not theoretical limitations as they all have solutions.

4.4 Architecture, Design and Implementation

This work proposes a translation pipeline with two main steps. First, translate the input C++ program to Rust using one of the previously described translation models. Lastly, improve the modeled Rust code using static or dynamic analysis.

This work focused on the first step of the translation pipeline. It involved implementing a compiler that automatically translates C++ programs into Rust, offering users a choice between two models: the **Unsafe** model and the **Reference Counting** model with bounds checking.

These two models were selected from the various described translation models for their ability to automatically translate a wider range of C++ programs into Rust, eliminating the need for manual effort. Additionally, these selected models provide contrasting features. Specifically, the **Unsafe** model offers

better interoperability and is closer to C++ in terms of memory usage and execution time than the **Reference Counting** model. However, it retains the memory-unsafe characteristics of the original C++ program, unlike the **Reference Counting** model. Lastly, neither of them fully supports the generation of idiomatic Rust code.

Figure 4.1 illustrates the translation pipeline, which includes the initial **Transpiler** step. As shown, the **Transpiler** generates either unsafe or safe Rust code, depending on whether the **Unsafe** or **Reference Counting** model was selected. Furthermore, the second **Refactoring** step can be viewed as either a safety enhancement or an optimization process, depending on the selected model. Consequently, the translation models serve as an intermediate representation of the input C++ program within the translation pipeline. Ultimately, the primary goal is to implement a translation pipeline capable of generating safe, efficient, and idiomatic Rust code, without forgetting the theoretical limitations of static and dynamic analysis during the **Refactoring** step.

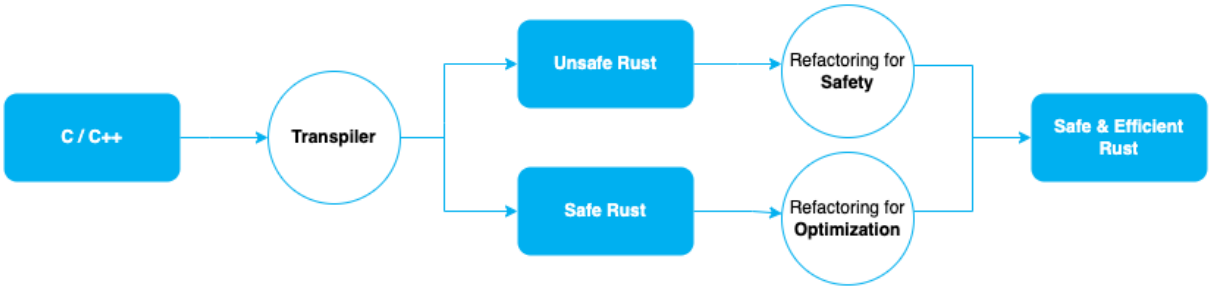

Figure 4.1: Translation Pipeline for Automatic Translation of C++ to Rust

The scheme in Figure 4.2 demonstrates the method calls needed to safely dereference a pointer in Rust using **Reference Counting** to access the inner value. The **Bounds Checking** step ensures spatial safety and null safety, while the **Lifetime** and **Mutability** steps ensure temporal safety. It demonstrates the run-time safety checks that a pointer, the most complex type in the translation model, has to pass to access the inner value. In contrast, a reference only needs to pass the **Lifetime** and **Mutability** run-time safety checks.

Furthermore, Figure 4.2 offers a systematic overview of the static or dynamic analyses required during the **Refactoring** step. To optimize the pointer type (e.g., to achieve static safety for a pointer), it is necessary to eliminate each additional level of indirection, which necessitates distinct analyses for addressing each of these layers.

It is worth mentioning that a subset of C++ was defined, essentially to support the entire automatic conversion of **WOFF2**, which is a real C++ codebase. Additionally, a few more features were included, such as the `std::move`, `new` and `delete` expressions, although these are not extensively used in **WOFF2**. In the final chapter, we discuss the automatic translation of **WOFF2** to Rust using the **Unsafe** and **Reference Counting** models.

Figure 4.2: Steps for safely dereferencing a pointer in Rust using reference counting

The implementation involved using multiple programming languages and libraries. First, the compiler was written in C++, and the compiler code consisted of over 5000 lines, making use of Clang LibTooling.[20] Essentially, the compiler implements a class named `clang::RecursiveASTVisitor`,[21] which is a component of the Clang library designed to traverse the entire Clang Abstract Syntax Tree (AST), providing visitors for each node. During this traversal, the compiler concatenates Rust code according to each visitor.

As previously mentioned, the compiler supports two distinct translation models. The output generated by the compiler depends on the specific model chosen as input. It can produce either unsafe Rust code or Rust code with reference counting. Furthermore, the implementation of these two translation models was efficiently reused through inheritance. This approach was chosen because their implementations were largely equivalent in handling visitors for declaration and statement nodes. The primary distinctions between the models were observed in their treatment of expression and type nodes.

Additionally, a cross-checking tool was implemented, written in Python, and it utilizes the LLVM Integrated Tester (LIT) library.[22] Essentially, the tool runs a test suite concurrently with more than 100 C++ programs, translates them into Rust using one of the available translation models, runs the original C++ and the translated Rust programs, which may include assertions, and compares the final output. Each C++ program in the test suite tests a specific C++ feature.

Moreover, a Rust library comprising 1000 lines of code was developed to support the translation process. The library consists of two modules: one handles the implementation of prefix and postfix unary operators, which are not available in Rust but are commonly used in C++; the other module is responsible for implementing reference counting functionality. Therefore, the compiler generates code assuming this helper library exists.

Finally, to translate the **WOFF2** codebase, several Bash scripts were developed. These scripts are responsible for various tasks, including building **WOFF2** with CMake (which generates the compile commands JSON file that serves as input to the compiler), building and calling the compiler using Bazel, and building the translated Rust **WOFF2** library using Cargo. In the end, the translated Rust **WOFF2** libraries, modeled with both unsafe and safe reference counting code, were cross-checked with the original C++ **WOFF2** library, involving the compression and decompression of more than 100 fonts.

[20] https://clang.llvm.org/docs/LibTooling.html
[21] https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html
[22] https://llvm.org/docs/CommandGuide/lit.html

4.5 Evaluation: Micro-benchmarking

This section evaluates the translation models previously described in this chapter, primarily categorized as compile-time and reference counting models. The criteria for evaluating each translation model were listed at the beginning of this chapter but can be further categorized as quantitative and qualitative. The quantitative evaluation criteria include memory usage and execution time efficiency, while the qualitative criteria encompass the amount of required manual intervention, ability to generate idiomatic Rust code, safety, and interoperability.

It is worth noting that while in the quantitative evaluation, the several reference counting models are assessed separately, during the qualitative evaluation, the reference counting model with bounds checking is examined as a representative case for reference counting models. The primary goal is to evaluate the memory and execution time cost of adding bounds checking to the first reference counting model and to assess the optimization of the reference counting model with lazy initialization on some translated programs.

In the end, a table summarizing the different characteristics of each translation model, along with their advantages and drawbacks regarding the aforementioned criteria, is provided.

To quantitatively evaluate these models, we conducted performance measurements to assess memory consumption and execution time. The experiments were performed on a MacBook Pro computer (13-inch, 2018, Four Thunderbolt 3 Ports) with a 2.3 GHz Quad-Core Intel Core i5 processor and 8 GB 2133 MHz LPDDR3 memory. Furthermore, C++ and Rust programs were compiled with the highest optimization level, using `-O3` and `-C opt-level=3` compiler flags, respectively. Finally, to ensure the same backend is used when compiling both C++ and Rust programs, LLVM version 16 was employed for both (i.e., `clang++ 16` for C++ and `rustc 1.72.0` for Rust).

The benchmarked compile-time translation models include the **Safe**, **Naive**, and **Unsafe** models. In contrast, the reference counting models encompass the **Reference Counting** model without bounds checking, the model with bounds checking, and the model without bounds checking optimized using lazy initialization. This experiment aims to measure the execution time efficiency and memory usage efficiency of the translated Rust programs, using the input C++ program as the optimal reference.

The input C++ programs chosen for benchmarking are featured in Listing 44. This section encompasses a total of 8 benchmarked C++ programs, with only 6 of them showcased in the same listing for brevity. Additionally, their corresponding conversions in Rust, using each translation model, are omitted for brevity as well.

These C++ programs encompass variations designed to measure specific behaviors. In the programs that execute the function `fib`, the Fibonacci number is computed recursively (i.e., the C++ programs denoted as (a), (c), and (d) in Listing 44). These programs differ from each other as they pass the argument by value, by reference, and by pointer, leading to the execution of distinct operations in the

function body. For instance, dereferencing a pointer differs from accessing a variable's value.

Furthermore, the (b) program in Listing 44 performs an extensive iteration using a `for` loop, wherein it adds two integers that vary as the loop progresses. The primary goal is to measure the cost of accessing local variables of trivially copyable types by value, as these variables are most likely stored in registers after the optimization of this C++ program.

Finally, the (e) and (f) programs in Listing 44 involve repetitive extensive array allocations on the heap. These programs include iterations that access an array by value, by reference and pointer, respectively, with the same reasoning mentioned previously. The last two omitted programs execute a function to determine if a given number is a prime number, passing arguments by both value and by reference.

```cpp
unsigned long long fib(
  unsigned long long n) {
  return n == 0 || n == 1
    ? n : fib(n - 1) + fib(n - 2);
}
```

(a) Recursive Fibonacci (pass by value)

```cpp
long N = 25000000000;
long sum = 0;
for (long i = 0, j = N; i < j; ++i, --j) {
  sum += i + j;
}
```

(b) Summing two integers in an extensive `for` loop

```cpp
void fib(unsigned long long &n) {
  if (n == 0 || n == 1)
    return;
  unsigned long long n_1 = n - 1;
  unsigned long long n_2 = n - 2;
  fib(n_1);
  fib(n_2);
  n = n_1 + n_2;
}
```

(c) Recursive Fibonacci (pass by reference)

```cpp
void fib(unsigned long long *n) {
  if (*n == 0 || *n == 1)
    return;
  unsigned long long n_1 = *n - 1;
  unsigned long long n_2 = *n - 2;
  fib(&n_1);
  fib(&n_2);
  *n = n_1 + n_2;
}
```

(d) Recursive Fibonacci (pass by pointer)

```cpp
int N = 100000000;
long sum = 0;
for (int k = 0; k < 35; ++k) {
  std::unique_ptr<int[]> array =
    std::make_unique<int[]>(N);
  for (int i = 0; i < N; ++i) {
    array[i] = i;
  }
  for (int i = 0; i < N; ++i) {
    sum += array[i];
  }
}
```

(e) Allocate and iterate long array (by value)

```cpp
int N = 100000000;
long out = 0;
for (int k = 0; k < 35; ++k) {
  std::unique_ptr<int[]> array =
    std::make_unique<int[]>(N);
  initialize(array, N);
  out += sum(&array, N);
}
```

(f) Allocate and iterate long array (by reference)

Listing 44: Input C++ programs for benchmarking (6 out of 8)

Measuring the memory consumption of a process is not a trivial task, as there are several methods and variables to consider, such as the presence of other concurrently running processes. In this context,

the selected method involved measuring the maximum Resident Set Size (RSS), which quantifies the largest amount of physical memory (RAM) used by a process at any one instant, including heap and stack memory.

	C++	Unsafe Rust	Naive Rust	Safe Rust	Ref Count Rust	Ref Count Rust (lazy)	Ref Count Rust (bounds)
Fibonacci	721	819	795	807	889	799	889
Fibonacci (ref)	721	807	807	807	889	889	889
Fibonacci (ptr)	721	807	807	811	897	889	901
Sum	721	807	807	807	811	795	811
Alloc	400749	400785	400785	400785	4650881	4376457	4057772
Alloc (ref)	400749	400785	400785	400785	4589822	4333859	4058620
Prime	721	795	827	807	885	807	885
Prime (ref)	721	807	807	807	860	811	860
Avg Memory (KB)	100728	100802	100803	100802	1155742	1089413	1015203
Avg Overhead (x)		1.09	1.09	1.09	3.79	3.58	3.44

Table 4.6: Memory usage (KB) benchmark results

Table 4.6 shows the memory usage benchmark results in kilobytes (KB). Clearly, C++ is optimal, always having the lowest memory consumption, as shown by the first column in Table 4.6.

The memory consumed by C++, compile-time Rust, and reference counting Rust programs that compute the `Fibonacci` number recursively is approximately 700 KB, 800 KB, and 900 KB, respectively, as shown by the first three rows in Table 4.6.

However, the reference counting Rust program that computes the `Fibonacci` number and passes the argument by value exhibits lower memory consumption when executed with lazy initialization. Its maximum RSS is around 800 KB, as indicated in the first cell regarding reference counting Rust with lazy initialization in Table 4.6. This value is much closer to the maximum RSS of compile-time models than to the other reference counting models. This is due to the optimization that delays the reference counting management, which is never actually initialized in this particular program since no references or pointers are declared.

Furthermore, the programs `Sum` and `Prime` exhibit the same memory usage behavior as the programs computing the `Fibonacci` number by value, as demonstrated in rows 1, 4, and 7 in Table 4.6. As before, the table reveals that memory usage is approximately 700 KB for C++, 800 KB for compile-time Rust, and 900 KB for reference counting Rust programs.

Identically, the reference counting Rust program with lazy initialization differs from other reference counting models, showing memory consumption values of 795 KB and 807 KB in `Sum` and `Prime`, respectively. As before, these values are much closer to the memory usage of compile-time models.

Additionally, the programs that compute `Prime (ref)` exhibit an identical memory usage pattern, with slightly more memory consumed by lazy initialization when compared to the program that computes `Prime`. Specifically, 811 KB and 807 KB, respectively. This difference can be attributed to the use

of references, which trigger the reference counting mechanism and allocate additional memory on the heap.

Finally, the C++ and Rust programs that compute `Alloc` and `Alloc (ref)` exhibit significantly higher memory usage compared to the previous examples. This increase is due to the extensive array allocation on the heap, with the C++ program consuming 400749 KB of memory. The memory usage pattern is once again similar, with the compile-time Rust programs having a memory usage of 400785 KB, very close to the optimal C++ performance. However, in this case, the reference counting Rust programs consume approximately 4000000 KB, which is ten times more memory than the optimal C++ and compile-time Rust programs. This highlights the need to design an optimization for removing reference counting management within array elements.

In summary, Table 4.6 presents the average memory usage and overhead among C++ and the various translated Rust programs. It highlights the optimal performance of C++ and the lower memory usage of compile-time translation models (no overhead in general and approximately 100000 KB on average) compared to the reference counting models (3 times overhead in general and approximately 1000000 KB on average) in Rust. Notably, the primary reason for this difference lies in the `Alloc` and `Alloc (ref)` programs. Finally, bounds checking in reference counting incurs no additional memory overhead compared to the default reference counting model. However, utilizing lazy initialization may lead to some memory savings, especially when references and pointers are not extensively used.

To measure the execution time of each of these programs, the approach involved measuring the amount of CPU time spent in user mode during each process.

	C++	Unsafe Rust	Naive Rust	Safe Rust	Ref Count Rust	Ref Count Rust (lazy)	Ref Count Rust (bounds)
Fibonacci	5.95	5.13	5.14	5.12	1058.68	39.38	1059.53
Fibonacci (ref)	6.31	5.98	5.97	5.95	384.68	393.89	384.18
Fibonacci (ptr)	6.32	5.96	5.98	6.23	704.18	424.9	739.91
Sum	4.98	5.04	5.03	5.03	3.33	4.98	3.33
Alloc	4.73	4.74	4.73	4.74	592.61	136.61	562.91
Alloc (ref)	4.76	4.76	4.81	4.8	608.6	139.48	579.13
Prime	5.66	6.48	6.46	6.46	6.55	20.96	6.54
Prime (ref)	5.67	6.47	6.47	6.5	6.5	26.61	6.51
Avg Time (s)	5.55	5.57	5.57	5.6	420.64	148.35	417.76
Avg Slowdown (x)		1.01	1.01	1.01	75.8	25.48	74.96

Table 4.7: Execution time (s) benchmark results

Table 4.7 displays the benchmark results for execution time in seconds (s). Once again, C++ proves to be optimal in terms of execution time, as shown by the first column, just as it was for memory usage. However, what is truly surprising is that in some instances within the benchmark, both compile-time Rust and reference counting Rust programs outperform the original C++ in terms of performance.

All compile-time Rust programs demonstrate better execution times compared to the original C++

when running programs that calculate the `Fibonacci` number, as evidenced by the results in the first three rows and the first four columns of Table 4.7. For instance, when computing the `Fibonacci` number by value, C++ takes 5.95 seconds, while Safe Rust only takes 5.12 seconds. Furthermore, it is worth noting that all compile-time Rust programs exhibit nearly identical execution times on these `Fibonacci` computations, and thus, compile-time safety in Rust does not incur performance costs.

On the other hand, reference counting Rust programs demonstrate significantly worse execution times compared to the original C++ when running the same `Fibonacci` programs, as indicated by the results in the first three rows and the last three columns of Table 4.7. For example, when computing the `Fibonacci` number by value, C++ takes 5.95 seconds, while reference counting Rust, with or without bounds checking, requires approximately 1059 seconds.

Furthermore, it is worth noting that reference counting Rust program with bounds checking incurs execution time overhead when computing `Fibonacci` by pointer compared to the reference counting Rust program without bounds checking, with execution time values of 739.91 seconds and 704.18 seconds, respectively. This can be attributed to the additional bounds checking layer. Regarding the remaining execution times for these two models, they are nearly identical, with times of 1058 seconds and 384 seconds.

Finally, the reference counting Rust program with lazy initialization demonstrates significant optimizations in terms of execution time when computing `Fibonacci` by both value and pointer. The times for these executions are 39.38 seconds and 424.9 seconds, respectively, compared to the approximately 1059 seconds and 700 seconds required by the other two reference counting models. This can be justified by the decreased number of heap allocations made on these programs achieved with lazy initialization.

The C++ and Rust programs calculating `Sum` display nearly identical execution times, each taking approximately 5 seconds, as shown in row number 4 of Table 4.7. However, this is a unique benchmark case, where the reference counting models with temporal safety, both with and without bounds checking, outperformed both the original C++ and compile-time Rust programs, each with execution times of 3.33 seconds. An LLVM backend optimization may explain this difference.

Additionally, the C++ and compile-time Rust programs that allocate extremely long arrays on the heap present nearly identical execution times, each taking approximately 4.75 seconds, as shown by row number 5 and 6 of Table 4.7. It is worth noting that the compile-time Rust programs do not incur any performance costs even though they execute additional array indexing operations with dynamic bounds checking.

On the other hand, when compared to other models, the reference counting Rust programs exhibit a performance overhead, with the lazy initialization variant performing better. They have execution times of approximately 600 seconds and 135 seconds, respectively. This is attributed to the large number of

additional heap allocation performed for each array element and additional run time safety checks.

Lastly, the C++ and reference counting Rust programs showcase similar execution times, both taking approximately 6 seconds for prime number checking programs. However, it is noteworthy that the performance of lazy initialization is notably poorer compared to the others, taking around 20 seconds. This difference is possibly attributed to a loop optimization issue that the program encounters due to an `inline` implementation in the reference counting smart pointer.

The last two rows of Table 4.7 provide a summary of the execution time benchmark results. On average, C++ performs optimally with an execution time of 5.55 seconds. Nevertheless, the compile-time Rust programs are not far behind, averaging 5.57 seconds. On the other hand, the reference counting Rust programs take significantly longer than the other programs, with an overhead of approximately 75 times. However, the reference counting with lazy initialization shows some optimization cases, being only 25 times worse on average than the optimal programs.

However, it is noteworthy that reference counting Rust programs offer memory-safe code that outperforms other memory-safe programming languages. For example, when running the `Prime` program in Python, it took approximately 175.64 seconds, while the reference counting Rust programs completed the same task in just 6.55 seconds. This demonstrates the efficiency and performance advantages of Rust in memory-safe programming.

To conclude the evaluation section, we qualitatively compare the described models based on the previously mentioned criteria.

First, among the translation models, only the **Unsafe** and **Reference Counting** (with and without bounds checking) models utilize Rust types that possess all the functionalities of their C++ counterparts, such as pointer arithmetic, and can bypass Rust's borrow checker, thereby completely avoiding the need for manual intervention. The **Unsafe** model utilizes unsafe Rust code (e.g., unsafe raw pointers), while the **Reference Counting** model employs safe Rust code (e.g., reference counting smart pointers) to achieve this.

In contrast, the **Safe** and **Naive** models may require manual effort as they optimistically convert unsafe C++ references to safe Rust references. This conversion exposes them to evaluation by Rust's borrow checker, potentially resulting in issues like missing lifetime annotations and multiple mutable borrow errors. Additionally, the **Safe** model has limitations in supporting the pointer type conversion, as it translates pointers into optional safe references, restricting operations such as pointer arithmetic.

Nevertheless, the **Unsafe** and **Reference Counting** models are less suitable for generating idiomatic Rust code. Clearly, generating entirely unsafe or reference-counting Rust code is not idiomatic. As a result, the refactoring and optimization steps will need to perform a higher number of passes to improve the generated code, making it more readable and maintainable. On the other hand, the **Safe** and **Naive** models generate more idiomatic Rust code by making more assumptions on input C++ programs and

using safe and idiomatic Rust types.

Most importantly, the **Safe** and **Reference Counting** models are the only models capable of generating safe Rust code. However, only the **Reference Counting** model allows the compiler to automatically translate C++ into valid Rust programs. Additionally, in contrast to the **Safe** model, it expands its scope by supporting pointer arithmetic.

Furthermore, generating valid and safe Rust programs is extremely helpful because it helps in identifying and fixing bugs injected during the automatic translation process. For example, casting `&T` to `*const T` and subsequently casting to `*mut T` constitutes undefined behavior if the program later mutates `T&` through `*mut T`.[23] There were specific cases where the compiler, using the **Unsafe** or **Naive** model (both unsafe models), generated this code, resulting in undefined behavior in Rust. However, this problem would not have occurred if the compiler had used the **Reference Counting** model, as the pointer type contains the safe `RefCell` type. Furthermore, if the compiler had employed the **Safe** model, Rust's borrow checker would have rejected the program due to an attempt to create a mutable reference from an immutable reference.

Finally, the **Unsafe** and **Naive** models offer improved C++/Rust interoperability compared to the other models because they ensure the layout preservation of types during the translation process. Certainly, the **Unsafe** model is the most suitable option in this context, given that C++ is a memory-unsafe programming language, and any combination of safe and unsafe Rust code can potentially break the translated Rust program, as mentioned previously.

Table 4.8 provides a summary of the advantages and disadvantages of each translation model. A cross in a cell indicates that the corresponding translation model satisfies the respective criteria.

Translation Models	Memory Efficiency	Time Efficiency	(NO) Manual Effort	Idiomatic	Safety	Interoperability
Safe	X	X		X	X	
Naive	X	X		X		X
Unsafe	X	X	X			X
Reference Counting			X		X	

Table 4.8: Translation models comparison

[23] https://github.com/rust-lang/rust/issues/66136

5

Case Study: Automatic Translation of WOFF2 to Safe Rust

Contents

In this chapter, the process of automatically converting the **WOFF2** library from C++ to both unsafe and safe Rust is described in detail. This automatic translation employed both the **Unsafe** and **Reference Counting** models. As a result, this chapter delves into the automatic translation of advanced C++ features, highlighting the differences between translating them into unsafe and safe Rust. This chapter also shares lessons learned from this process. Detailed descriptions of the **WOFF2** C++ library and the **Brotli** C++ library, which is a dependency library of **WOFF2**, are also provided.

Additionally, this section provides insights into the implementation details of the translation process, such as the translation of multiple files and compilation modes. The section concludes with a benchmark

that compares **WOFF2** implementations in C++, unsafe Rust, and safe Rust, focusing on runtime performance and memory consumption overhead. It also offers final considerations regarding the feasibility of automatically porting C++ code to Rust, taking into account possible need for manual intervention and the idiomatic, safe, and C++-interoperable nature of the generated Rust code.

5.1 WOFF2

WOFF2[1] is a C++ library which converts fonts between TTF and WOFF2 formats. This C++ library primarily comprises classes that resemble C structs, with very few methods and no special constructors or destructors. It predominantly uses the `int` type for most integer values, relying heavily on implicit integer type coercions and truncations. Moreover, it utilizes the low-level C aspects of C++, such as raw pointers to buffers and pointer arithmetic. However, **WOFF2** also leverages advanced C++ features, including polymorphism, function overloading, templates, and advanced data structures available in the C++ Standard Library. These include `std::unique_ptr`, `std::map`, `std::vector`, `std::string`, and their associated methods and functions.

Furthermore, **WOFF2** depends on **Brotli** which is a C library to compress and decompress data. This C++ library consists of numerous header and source files, totaling approximately 6700 lines of code.

The choice of this C++ library was influenced by a prior experience in manually porting the **WOFF2** library to Rust.[2] The goal of this porting experience was to identify challenges in manually converting a C++ library to Rust. Several issues encountered in this process were similar to those encountered when translating Rust code into the **Safe** and **Naive** models. These challenges included annotating lifetimes and enforcing the exclusivity rule.

Another reported issue was determining the trade-off between using unsafe code, such as raw pointers, and safer alternatives like `RefCell`. This choice parallels the decision between the **Unsafe** and **Reference Counting** models. Additionally, ensuring that the ported Rust code remained idiomatic was also an issue, which is a common concern across all translation models. In the worst case, this requires rethinking the software design in Rust.

Identical to the automatic approach proposed in this work, the manual migration process consisted of two passes. The first pass involved a literal and mechanical translation with minimal semantic alterations, mirroring our initial translation modeling of Rust code. The second pass encompassed a final manual refactoring step aimed at making the Rust code more idiomatic.

Nevertheless, the **WOFF2** C++ library to Rust manual conversion process took 20 days, during which multiple bugs were introduced due to human errors in the manual translation process. In contrast, our

[1] https://github.com/google/woff2
[2] https://www.youtube.com/watch?v=kcMAiTg5j1w

automatic approach can complete the same task in less than 5 minutes without introducing any bugs. Additionally, the compiler generates both unsafe and safe reference counting Rust code.[3]

5.2 Translation of Advanced C++ features

A typical C++ codebase comprises multiple source files, including headers, which are distributed across various translation units. Each translation unit is compiled independently by the compiler. Currently, this C++ to Rust transpiler converts all Rust code and outputs it as a single file to simplify the translation process while avoiding duplicate declarations.

To build **WOFF2** in Rust, a Cargo library with a single Rust output file was utilized. Additionally, the three `main` **WOFF2** functions (found in the source C++ files `woff2_compress.cc`, `woff2_decompress.cc`, and `woff2_info.cc`) were manually placed in separate Rust source files within a `/bin` folder (represented by the source Rust files `woff2_compress.rs`, `woff2_decompress.rs`, and `woff2_info.rs`). This file organization enables the execution of the **WOFF2** Rust library for compressing and decompressing TTF and WOFF2 fonts, respectively.

It is worth noting that the C parts of C++ were converted similarly to the translations made by c2rust, such as loop control statements and wrapped unsigned binary operators. However, unary prefix and postfix operators were an exception. These specific unary operators in C caused complex translations when used within conditional expressions. Therefore, these unary operators were implemented in a helper library in Rust, using `trait` definitions, for all the different signed and unsigned integer types, to facilitate the C++ to Rust translation process and improve the quality of the generated code. This translation difference between c2rust and this C++ to Rust transpiler can be seen in Listing 45.

The c2rust conversion, as shown in program `(b)` in Listing 45, creates an outer `loop` and introduces a new variable named `fresh0`, along with an inner `if` statement, as depicted in lines 2, 3, and 5, respectively. This complexity is simplified in programs `(c)` and `(d)` through the use of the `postfix_inc()` and `prefix_inc()` method calls.

Clearly, there is a trade-off between designing less idiomatic solutions that convert C++ into valid Rust code 100% of the time and more idiomatic solutions that work 99% of the time. In this implementation, the choice was to generate idiomatic Rust code if it works for 99% of the conversions. Therefore, some translation solutions in this chapter may have certain limitations, but they are still generic enough to work for the majority of cases. This includes their capability to handle the automatic translation of real C++ codebases, specifically, **WOFF2**. Nevertheless, the limitations and their respective less idiomatic solutions are pointed out whenever necessary.

[3] The automatic conversion of **WOFF2** C++ library into Rust using reference counting had some syntax errors due to bugs in code generation, but they were fixed manually in less than 1 hour.

```
1  int x = 0;
2  while (x++ < 100 && x != 50) {
3    ++x;
4  }
```

(a) C program with post and prefix unary operators

```
1  let mut x: libc::c_int = 0 as libc::c_int;
2  loop {
3    let fresh0 = x;
4    x = x + 1;
5    if !(fresh0 < 100 as libc::c_int &&
6         x != 50 as libc::c_int) {
7      break;
8    }
9    x += 1;
10 }
```

(b) Translation to Rust using c2rust

```
1  let mut x: i32 = 0_i32;
2  while x.postfix_inc() < 100_i32 &&
3        x != 50_i32 {
4    x.prefix_inc();
5  }
```

(c) Translation to Rust using **Unsafe** model

```
1  let x: Value<i32> = Rc::new(RefCell::new(0_i32));
2  while (*x.borrow_mut()).postfix_inc() < 100_i32 &&
3        (*x.borrow()) != 50_i32 {
4    (*x.borrow_mut()).prefix_inc();
5  }
```

(d) Translation to Rust using **Reference Counting** model

Listing 45: Differences when translating C++ unary prefix and postfix operators into Rust

5.2.1 Classes

C++ classes are translated into Rust structs. Furthermore, C++ class methods are translated into methods within the Rust struct's implementation. Specifically, the C++ constructor is translated into a static method defined within the Rust struct's implementation, since Rust does not have built-in constructors as C++ does. An illustration of the automatic translation of a C++ class into a Rust struct can be seen in Listing 46. This listing displays only a portion of the **WOFF2** Buffer class[4] and presents only the unsafe Rust version, for brevity.

There are a couple of things worth noting. Firstly, the C++ `public` and `private` access specifiers are converted to Rust as `pub` and an empty access specifier, respectively, because everything in Rust is private by default. This is demonstrated in Rust on lines 3, 4, and 5 for the translation of the `private` access specifier and on lines 8, 20, and 23 for the translation of the `public` access specifier. This is identical in safe Rust code.

Moreover, in Rust, every static and class method is qualified as unsafe, as illustrated in lines 8, 20, and 23. In contrast, the translation of these methods using safe Rust is never qualified as unsafe.

Additionally, the qualification of the `self` argument with `mut` depends on whether the original C++ method is marked as `const` or not, as exemplified by the `length` and `set_offset` methods in Listing 46. In contrast, when translating these qualifiers into safe Rust, the use of `mut` is never required, thanks to the use of the **Interior Mutability Pattern**.

The validation of `&mut self` in Rust depends on the borrow checker, which may result in compile-

[4] https://github.com/google/woff2/blob/master/src/buffer.h

```cpp
class Buffer {
public:
  Buffer(
    const uint8_t *data,
    size_t len
  ) : buffer_(data),
      length_(len),
      offset_(0) { }
  // ...
  size_t length() const {
    return length_;
  }
  void set_offset(size_t newoffset) {
    offset_ = newoffset;
  }
private:
  const uint8_t * const buffer_;
  const size_t length_;
  size_t offset_;
};
```

```rust
#[repr(C)]
pub struct Buffer {
  buffer_: *const u8,
  length_: u64,
  offset_: u64,
}
impl Buffer {
  pub unsafe fn Buffer(
    mut data: *const u8, mut len: u64
  ) -> Buffer {
    let mut this = Buffer {
      buffer_: data,
      length_: len,
      offset_: (0_i32 as u64),
    };
    this
  }
  // ...
  pub unsafe fn length(&self) -> u64 {
    return self.length_;
  }
  pub unsafe fn set_offset(
    &mut self, mut newoffset: u64
  ) {
    self.offset_ = newoffset;
  }
}
```

Listing 46: Translating C++ **WOFF2** `Buffer` class into unsafe Rust

time errors. For example, if a field inside a non-`const` method is mutably borrowed for another function call. While this is not a problem when using safe Rust, there is another limitation when using reference counting Rust code: it is not possible to execute `return this;` since `this` is an immutable safe reference (`&self`) and not a weak reference.

In both of these cases, a less idiomatic solution would be to use functions instead of methods and pass a pointer to the structure (i.e., `*mut Buffer` or `Pointer<Buffer>` in unsafe and safe Rust, respectively), rather than using `&mut self`, similar to how structs are implemented in C.

Finally, a new variable named `this` is created within the constructor in Rust to represent the `this` pointer in C++, as shown in line 12. The limitation of this approach is that the `this` variable in Rust is local and does not have the same stack address as the variable being created, unlike the `this` pointer in C++. However, there is a solution for implementing C++ constructors in Rust generically, but it is more intricate and less idiomatic.[5]

[5] https://mcyoung.xyz/2021/04/26/move-ctors/

5.2.2 Polymorphism

C++ abstract classes are translated into Rust trait objects. In C++, abstract classes cannot be used directly to instantiate objects, permitting only a pointer referencing the base abstract class. In Rust, the same happens with trait objects but the keyword `dyn` must be included in the pointer type of the abstract class (i.e., `*mut dyn T` or `Pointer<dyn T>` in unsafe Rust code and safe Rust code with reference counting, respectively, being `T` a trait object).

Listing 47 demonstrates a portion of the translation of the C++ **WOFF2** `WOFF2Out`[6] abstract base class and its subclasses into safe Rust using reference counting. The unsafe Rust code and the remaining class implementations are omitted for brevity.

```
1  class WOFF2Out {
2  public:
3    // ...
4    virtual size_t Size() = 0;
5  };
6  class WOFF2StringOut : public WOFF2Out {
7  public:
8    // ...
9    size_t Size() override {
10     return offset_;
11   }
12 private:
13   std::string *buf_;
14   size_t max_size_;
15   size_t offset_;
16 };
17 class WOFF2MemoryOut : public WOFF2Out {
18 public:
19   // ...
20   size_t Size() override {
21     return offset_;
22   }
23 private:
24   uint8_t* buf_;
25   size_t buf_size_;
26   size_t offset_;
27 };
```

```
1  pub trait WOFF2Out {
2    // ...
3    fn Size(&self) -> Value<u64>;
4  }
5  #[repr(C)]
6  pub struct WOFF2StringOut {
7    buf_: Value<Pointer<Vec<Value<i8>>>>,
8    max_size_: Value<u64>,
9    offset_: Value<u64>,
10 }
11 impl WOFF2Out for WOFF2StringOut {
12   // ...
13   fn Size(&self) -> Value<u64> {
14     return Rc::new(RefCell::new(
15       (*self.offset_.borrow())));
16   }
17 }
18 #[repr(C)]
19 pub struct WOFF2MemoryOut {
20   buf_: Value<Pointer<u8>>,
21   buf_size_: Value<u64>,
22   offset_: Value<u64>,
23 }
24 impl WOFF2Out for WOFF2MemoryOut {
25   // ...
26   fn Size(&self) -> Value<u64> {
27     return Rc::new(RefCell::new(
28       (*self.offset_.borrow())));
29   }
30 }
```

Listing 47: Translating abstract class and subclasses in **WOFF2** library into safe Rust

The Listing 47 illustrates the Rust's `trait` definition in line 1 and `trait` implementations in lines 11 and 24. It is worth noting that neither the `trait` nor the methods are classified as unsafe. In contrast, in the translation using unsafe Rust code, the `trait` and methods are classified as unsafe (e.g., `pub`

[6] https://github.com/google/woff2/blob/master/include/woff2/output.h

`unsafe trait WOFF2Out` and `unsafe fn Size`). Most importantly, the syntax for dereferencing a pointer to a `trait` object remains unchanged, in both unsafe and safe Rust.

Another important point to note is that, although the C++ pure virtual method `Size` is not marked with `const` qualifier, in safe Rust, the translated method is immutable, using an immutable self-reference (i.e., `Size(&self)`), as shown in lines 13 and 26. In contrast, unsafe Rust generates a mutable method (i.e., `Size(&mut self)`).

To wrap this section up, in unsafe Rust, this approach has a limitation: a pointer to a `trait` object cannot be null. Fortunately, this issue does not arise in safe Rust.

5.2.3 Function Overloading

Rust does not support function overloading directly as C++ does. There are several approaches to solve this problem, such as using traits or generating a new and unique function name for each overloaded function. This work selected the latter, due to its simplicity and reusability when generating template function specializations. To generate the new and unique function name, the compiler uses the following function declaration elements: the number of arguments, the types of arguments, reference qualifiers, and the `const` qualifier.

Listing 48 demonstrates the translation of function overloading present in the **WOFF2** library into safe Rust, specifically the methods `FindTable` and functions `WriteFont` in the `Font` structure's header file.[7]

```
1  struct Font {
2    // ...
3    Table* FindTable(
4      uint32_t tag);
5    const Table* FindTable(
6      uint32_t tag) const;
7  };
8  // ...
9  bool WriteFont(
10   const Font& font,
11   uint8_t* dst,
12   size_t dst_size);
13 bool WriteFont(
14   const Font& font,
15   size_t* offset,
16   uint8_t* dst,
17   size_t dst_size);
```

```
1  impl Font {
2    // ...
3    pub fn FindTable_u32(
4      &self, tag: Value<u32>
5    ) -> Value<Pointer<Font_Table>> { /*...*/ }
6    pub fn FindTable_u32_const(
7      &self, tag: Value<u32>
8    ) -> Value<Pointer<Font_Table>> { /*...*/ }
9  }
10 // ...
11 pub fn WriteFont_pconstFont_pmutu8_u64(
12   font: Reference<Font>, dst: Value<Pointer<u8>>,
13   dst_size: Value<u64>,
14 ) -> Value<bool> { /*...*/ }
15 pub fn WriteFont_pconstFont_pmutu64_pmutu8_u64(
16   font: Reference<Font>, offset: Value<Pointer<u64>>,
17   dst: Value<Pointer<u8>>, dst_size: Value<u64>,
18 ) -> Value<bool> { /*...*/ }
```

Listing 48: Translating function overloading in **WOFF2** library into safe Rust

[7] https://github.com/google/woff2/blob/master/src/font.h

5.2.4 Templates

Rust does not support type templates as C++ does. Rust has generics, which are a similar concept to templates in C++. However, the key distinction is that generics in Rust are type-safe by design. In Rust, generic type correctness is enforced at compile-time through the use of `trait` bounds, which dictate the behaviors and constraints that generic types must adhere to. While this approach enhances type safety, determining the appropriate `trait` bounds for each generic type is in general a very hard problem.

Therefore, the selected approach involved generating a distinct function for each template function specialization, each with a different name, generated identically to function overloading. However, this method results in Rust having multiple function definitions compared to the single original C++ template function definition.

Listing 49 demonstrates this approach, as evidenced by the generation of functions `Round4_i32` and `Round4_u64` in Rust from the function template specializations of the `Round4` function in C++. It is worth noting the differences between the function bodies of the function template specializations in the translated Rust programs (i.e., `Round4_i32` and `Round4_u64`).

```
template<typename T>
T Round4(T value) {
  if (std::numeric_limits<T>::max()
      - value < 3) {
    return value;
  }
  return (value + 3) & ~3;
}
```

```
pub fn Round4_i32(
    value: Value<i32>) -> Value<i32> {
  if i32::MAX - (*value.borrow()) < 3_i32 {
    return Rc::new(RefCell::new((*value.borrow())));
  }
  return Rc::new(RefCell::new(
    ((*value.borrow()) + 3_i32) & !3_i32));
}
```

```
pub fn Round4_u64(value: Value<u64>) -> Value<u64> {
  if (u64::MAX as u64).wrapping_sub((*value.borrow())) < (3_i32 as u64) {
    return Rc::new(RefCell::new((*value.borrow())));
  }
  return Rc::new(RefCell::new(
    ((*value.borrow()).wrapping_add((3_i32 as u64))) & (!3_i32 as u64)));
}
```

Listing 49: Translating templates in **WOFF2** library into safe Rust

5.2.5 C++ Standard Library

To translate the C++ Standard Library, the compiler would have to support all features used internally. Since it does not, a different approach had to be designed. Moreover, the compiler could have used a **Foreign Function Interface** (FFI) mechanism, but this is not supported for C++/Rust yet. Therefore, the compiler translates C++ Standard Library types and functions using a map together with pattern matching, mapping each type and function to the respective implementation present in the Rust Standard Library, as demonstrated in Listing 50.

This approach successfully translated all C++ Standard Library types and functions used in **WOFF2**, totaling approximately 100 map entries. Additionally, it made the generated Rust code more idiomatic by replacing C++ Standard Library types and functions with native Rust equivalents and eliminating the need for FFI function calls, which are inherently unsafe.

However, this approach is not very scalable, as in order to support the translation of each new C++ Standard Library type or function, a new entry in the map must be added (even though the most commonly used ones are not added very often). Furthermore, using native safe Rust might lead to compile-time or run-time errors, depending on whether unsafe Rust code or safe reference counting Rust code is utilized, respectively, such as errors related to mutable borrows. Finally, limitations are evident when employing advanced features of the C++ Standard Library.

```
{
  // ...
  {"class std::unique_ptr", "Box<%>"},
  {"class std::vector", "Vec<%>"},
  {"class std::map", "Vec<(%, %)>"},
  {"struct std::pair", "(%, %)"},
  {"std::string", "Vec<%>"},
}
```

```
{
  // ...
  { "std::vector::vector()", "Vec::new()" },
  { "std::vector::push_back", "%o.push(%a0)" },
  { "std::vector::pop_back", "%o.pop()" },
  { "std::vector::size", "%o.len() as u64" },
  { "std::vector::empty", "%o.is_empty()" },
}
```

Listing 50: Translating C++ Standard Library types and functions into Rust using pattern matching

5.3 Brotli

The **WOFF2** C++ library uses only two functions from the **Brotli** C library: `BrotliEncoderCompress`[8] and `BrotliDecoderDecompress`[9]. The translation of these functions into Rust was handled separately to avoid the need for supporting the automatic conversion of another complete C++ codebase into Rust. Therefore, the **rust-brotli**[10] library was used, which is an implementation of the `Brotli` C library in Rust, to complete the automatic translation of **WOFF2** into Rust.

The translation of **WOFF2** into Rust using the **Unsafe** model involves calling an unsafe Rust function from the **rust-brotli** library, specifically named `::brotli::ffi::compressor::BrotliEncoderCompress` (C FFI), which is equivalent to the `BrotliEncoderCompress` function in the **Brotli** C library. On the other hand, the translation of **WOFF2** into Rust using the **Reference Counting** model permitted calling a safe and idiomatic Rust function from the **rust-brotli** library, specifically named `::brotli::BrotliCompress`, which is again equivalent to the `BrotliEncoderCompress` function in the **Brotli** C library. Additionally, to call this function, it was required to implement the `std::io::Read` and `std::io::Write` traits for the `Pointer<i8>` type. The same approach was followed for the decompression function.

[8] https://github.com/google/woff2/blob/master/src/woff2_enc.cc
[9] https://github.com/google/woff2/blob/master/src/woff2_dec.cc
[10] https://github.com/dropbox/rust-brotli

5.4 Problems and Limitations

In addition to the previously mentioned limitations regarding the automatic translation of **WOFF2** library in C++ into Rust, there are a few more to consider: C++ `namespaces` were ignored because Rust's `modules` are not an equivalent concept; the `void*` type was translated into the `Pointer<dyn std::any::Any>` type in safe Rust; safe versions of `memset` and `memcpy` were specifically implemented for the `Pointer<T>` type, but they are not semantically equivalent; and the expression `*reinterpret_cast<uint16_t*>(dst + offset)` had to be manually translated into safe Rust due to differences in memory layout.

5.5 Evaluation: Macro-benchmarking

In this section, we benchmark the **WOFF2** C++ library and its automatic translation into unsafe Rust and safe reference counting Rust for both compression and decompression of TTF and WOFF2 font formats. These benchmarks measure the following metrics: execution time, peak memory usage (i.e., maximum Resident Set Size), and lines of code.

The benchmark settings for this experiment mirror those of the previous chapter's micro-benchmark, ensuring that both the **WOFF2** libraries in C++ and Rust are built in release mode to maximize optimization. In addition, more than 100 fonts of various sizes were compressed and decompressed in order to collect these results, the majority sourced from Google Fonts.[11]

	Avg (s)	Max (s)	Min (s)	Avg (KB)	Max (KB)	Min (KB)
C++	0.62	2.30	0.02	17800	49230	3273
Unsafe Rust	0.46	1.87	0.02	20062	63586	3396
Ref Count Rust (bounds)	0.88	6.76	0.05	70494	408383	7414

Table 5.1: WOFF2 compression benchmark results

Table 5.1 demonstrates the **WOFF2** compression benchmark results. Unsafe Rust and C++ exhibit nearly identical average execution times, with unsafe Rust being slightly faster than C++, averaging 0.46 and 0.62 seconds, respectively. This observation is supported by comparing the maximum execution times of C++ and unsafe Rust, which are 2.30 seconds and 1.87 seconds, respectively. On the other hand, unsafe Rust has higher memory usage than C++, with an average of 20062 KB compared to 17800 KB. This observation is supported when comparing the maximum peak memory consumption of C++ and unsafe Rust, which is 49230 KB and 63586 KB, respectively. Nevertheless, the **WOFF2** compression performance of C++ and unsafe Rust is similar.

As expected, reference counting Rust has the slowest compression execution time, averaging 0.88 seconds. This is roughly 1.5 times longer than the execution times of C++ and twice as long as those

[11] https://fonts.google.com/

of unsafe Rust. Furthermore, it has the highest memory usage, averaging 70494 KB, which is approximately 4 times and 3.5 times higher than the memory usage of C++ and unsafe Rust, respectively. These disparities in performance become evident when analyzing the maximum execution time and maximum peak memory usage of safe Rust, which are 6.72 seconds (approximately 3.5 times longer on average than C++ and unsafe Rust) and 408383 KB (approximately 7.5 times higher on average than C++ and unsafe Rust), respectively.

However, these compression benchmark results are particularly noteworthy, given the substantial use of reference counting in the safe Rust implementation. For instance, the minimum execution time and minimum peak memory usage of safe Rust are 0.05 seconds (similar to that of the original C++ and unsafe Rust, with times of 0.02 seconds) and 7414 KB (2 times higher than the original C++ and unsafe Rust), respectively.

	Avg (s)	Max (s)	Min (s)	Avg (KB)	Max (KB)	Min (KB)
C++	0.01	0.05	0.01	2191	7840	1298
Unsafe Rust	0.06	0.43	0.01	2293	7401	1155
Ref Count Rust (bounds)	0.29	3.18	0.02	27929	172630	3293

Table 5.2: WOFF2 decompression benchmark results

Table 5.2 demonstrates the **WOFF2** decompression benchmark results, which are similar to the compression performance. However, C++ has better decompression performance on average than unsafe Rust regarding execution time (0.01 seconds and 0.06 seconds, respectively) and memory consumption (2191 KB and 2293 KB, respectively). Additionally, the maximum decompression execution time for C++ is 0.05 seconds, while for unsafe Rust, it is 0.43 seconds.

Similar to the compression performance, safe Rust also exhibits the longest execution time and the highest memory consumption on average. This time, the disparity increased further, with reference counting Rust averaging 0.29 seconds and 172630 KB, which is approximately 20 times worse than the original C++. Nevertheless, it is still capable of decompressing some WOFF2 font files in as little as 0.02 seconds, identical to the times of C++ and unsafe Rust (both 0.01 seconds).

It is worth noting that the compression and decompression execution times and memory consumptions scale in proportion to the size of the input font in TTF and WOFF2 format, meaning that larger files result in increased values. Moreover, larger files also lead to a greater disparity in the compression and decompression benchmark results of **WOFF2** in C++, unsafe Rust, and safe Rust.

Moreover, the runtime of the **WOFF2** library in C++ is dominated by **Brotli**, not by the **WOFF2** code itself. Consequently, it could be useful to conduct a more in-depth profiling of this library to unveil specific performance bottlenecks in the **WOFF2** library in Rust, especially when using reference counting.

In conclusion, the original **WOFF2** C++ library comprises approximately 6000 lines of code. While the translation of **WOFF2** into unsafe Rust also consists of around 6000 lines of code, the safe Rust version

extends to approximately 14000 lines, which is more than double the lines of code in comparison to C++ and unsafe Rust. This significantly affects the quality of the generated code.

6

Conclusion

Contents

6.1 Future Work

This work paves the way for future research and development in the field of automatic translation between programming languages. Given the extensive nature of the C++ programming language, a complete automatic source-to-source compiler that translates C++ into Rust would necessitate handling the translation of a broader range of C++ features. These features include user-defined move and copy constructors, virtual destructors, lambda expressions, converting constructors, strongly-typed enums, forwarding references, variadic templates, and many others.

Additionally, the Rust code generated initially has substantial room for improvement through automatic refactoring and optimization, with the aim of making it more idiomatic and efficient. For example, when converting C++ into safe Rust using reference counting, notable inefficiencies become apparent. These issues include performing redundant safety checks at runtime, making an excessive number of

heap allocations, and applying reference counting to variables that do not actually require an expensive automatic memory management mechanism. These are just a few examples of areas where improvements can be made through optimization.

6.2 Conclusion

In this thesis, we began by highlighting a crucial aspect: C and C++, while highly efficient programming languages, lack comprehensive support for memory-safety features. As an alternative, Rust emerges as a promising candidate due to its efficiency comparable to C and C++ coupled with its intrinsic memory-safety features. However, manually rewriting C++ software systems in Rust is a daunting and expensive task. It is within this context that the necessity for developing a compiler to perform the automatic translation of C++ into Rust became evident. Furthermore, the existing related work primarily focused on the automatic translation of C into unsafe Rust and utilized static analysis to enhance the generated Rust code in terms of safety. In this thesis, we aimed to build upon this by addressing the direct automatic translation of C++ code into safe Rust, offering a novel contrast to existing approaches.

The fundamental differences between C++ and Rust were explicitly described, such as the move operation, the support for multiple mutable references simultaneously, and the need for manual lifetime annotations. Consequently, a straightforward translation of C++ into Rust would yield numerous compile-time errors, requiring manual intervention, which was not our primary goal. In this thesis, we introduced multiple translation models capable of converting C++ into Rust, each with its unique advantages and drawbacks, including the amount of manual work required, the quality of the generated Rust code (e.g., safety), and performance. This work focused on two translation models that perform C++ to Rust translation entirely automatically. First, the **Unsafe** model translates C++ code into unsafe Rust, preserving the performance of the original C++ code. On the other hand, the **Reference Counting** model translates C++ code directly into safe Rust, losing the performance of the original C++ code. Nevertheless, the safety-first approach allows for a subsequent optimization step that enhances execution time and memory consumption of the safe Rust generated code without compromising safety. Furthermore, generating unsafe Rust code from C++ can be error-prone, particularly when combined with safe Rust's built-in mechanisms, potentially leading to undefined behavior.

Finally, the **WOFF2** library in C++ was automatically translated into both unsafe and safe Rust. This involved implementing advanced C++ features in the transpiler, including classes, polymorphism, function overloading, templates, and functions and types from the C++ Standard Library. Performance measurements revealed that the unsafe Rust version closely matched the original C++ code in terms of execution time, memory usage, and lines of code. In contrast, the safe Rust version exhibited slightly higher execution times, increased memory consumption, and more than double the lines of code.

Bibliography

[1] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021.

[2] Microsoft. (2019) Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Visited on 2023-01-12. [Online]. Available: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[3] Google. (2021) An update on memory safety in chrome. Visited on 2023-01-12. [Online]. Available: https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

[4] T. N. S. A. (NSA). (2022) Software memory safety. Visited on 2023-01-11. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[5] (2023) Us government mandate on memory safety. Visited on 2023-10-30. [Online]. Available: https://www.armed-services.senate.gov/imo/media/doc/fy24_ndaa_bill_text.pdf

[6] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018. [Online]. Available: https://doc.rust-lang.org/book/

[7] N. Ivanov, "Is rust c++-fast? benchmarking system languages on everyday routines," 2022.

[8] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," 2022.

[9] E. Saykol and T. Uzlu, "On utilizing rust programming language for internet of things," in *CICN*, 2017.

[10] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin, "Experience report: Developing the servo web browser engine using rust," 2015.

[11] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership is theft: Experiences building an embedded os in rust," in *PLOS*, 2015.

[12] Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, “Rust as a language for high performance gc implementation,” in *ISMM*, 2016.

[13] D. Bryant. (2016) A quantum leap for the web. [Online]. Available: https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12

[14] (2022) Rust - the linux kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/next/rust/index.html

[15] J. Vander Stoep and S. Hines. (2021) Rust in the android platform. [Online]. Available: https://security.googleblog.com/2021/04/rust-in-android-platform.html

[16] J. V. Stoep. (2022) Memory safe languages in android 13. Visited on 2023-01-12. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[17] W. Crichton, “The usability of ownership,” 2020.

[18] M. Tofte and L. Birkedal, “A region inference algorithm,” *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, p. 724–767, jul 1998.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, “Region-based memory management in cyclone,” in *PLDI*, 2002.

[20] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill, “An overview of the singularity project,” Tech. Rep. MSR-TR-2005-135, 2005. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/10/tr-2005-135.pdf

[21] D. J. Pearce, “A lightweight formalism for reference lifetimes and borrowing in rust,” *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 1, apr 2021.

[22] I. inc. (2023) The c2rust repository. [Online]. Available: https://github.com/immunant/c2rust

[23] C. Developers. (2023) Citrus: Convert c to rust. [Online]. Available: https://gitlab.com/citrus-rs/citrus

[24] J. Sharp. (2023) Corrode: Automatic semantics-preserving translation from c to rust. [Online]. Available: https://github.com/jameysharp/corrode

[25] I. inc. (2020) c2rust manual. Visited on 2023-01-04. [Online]. Available: https://c2rust.com/manual/

[26] *ISO/IEC 9899:1999 Programming languages — C*, 2nd ed., 1999. [Online]. Available: https://www.iso.org/standard/29237.html

[27] B. Steensgaard, “Points-to analysis in almost linear time,” in *POPL*, 1996.

[28] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, “Rudra: Finding memory safety bugs in rust at the ecosystem scale,” in *SOSP*, 2021.

[29] P. Ferrara, F. Logozzo, and M. Fahndrich, “Safer unsafe code for .net,” in *Proceedings of the 23rd ACM Conference on Object-Oriented Programming (OOPSLA’08)*. Association for Computing Machinery, Inc., October 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/safer-unsafe-code-for-net/

[30] S. Olson. (2016) Miri: An interpreter for rust’s mid-level intermediate representation. Visited on 2023-01-12. [Online]. Available: https://solson.me/miri-report.pdf

[31] (2023) Automatic reference counting (arc). Visited on 2023-10-30. [Online]. Available: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/