TÉCNICO
UNIVERSIDADE
DE LISBOA

Optimizing Safe Rust Programs With Dynamic Ownership and Mutability Checks

Francisco Catanho Barreto Tomé Gouveia

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Nuno Claudino Pereira Lopes

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Nuno Claudino Pereira Lopes
Member of the Committee: Prof. Bernardo Parente Coutinho Fernandes Toninho

October 2026

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.

Aos meus pais, e aos meus avós,

que me moldaram e me permitiram estar aqui hoje.

"O que impede de saber não são nem o tempo nem a inteligência,
mas somente a falta de curiosidade."

— Agostinho da Silva

"Eles não sabem, nem sonham,
que o sonho comanda a vida."

— António Gedeão, *Pedra Filosofal*

Acknowledgments

I would like to thank my supervisor, Prof. Nuno Lopes, for his expertise, mentorship, and patience throughout this work. I am particularly grateful for the wonderful opportunities, especially RustWeek, for the many discussions, and for the freedom he gave me to explore my own ideas.

I would also like to thank my colleagues João, Henrique, José, Lucian, Manuel, and Xufan, whose discussions and feedback helped shape this work and made it considerably more enjoyable.

My sincere thanks also go to Prof. Pedro Monteiro for the invaluable guidance during my bachelor's and master's, Tiago Ribeiro for making a health diagnosis a strength instead of a weakness, and Prof. Ana Valente for teaching me the work ethic necessary to take on any challenge in life. I am also grateful to my cousins, especially Gonçalo, for always listening to my ideas since I was a kid and for motivating me to go to Técnico.

Most importantly, I would like to thank my parents, Luís Miguel and Ana Rita. Thank you for always supporting my endeavors, whether it was tennis, piano, or, ultimately, computer science. I am who I am today because I had the opportunity and the support to pursue so many enriching activities. Thank you for your interest in my work and for your patience as I talked about it endlessly. You are the people I aspire to be like, and much of who I am today is owed to your example.

I would also like to thank my grandparents Celda, Manuel, Catarina, and Orlando. I may no longer be able to spend all my summers with you, but I offer my sincere gratitude for all the moments we shared, the teachings, and the childhood memories I will forever carry with me. They are the best ones.

This thesis would not have been possible without the support of all of you. To each and every one of you, thank you, not just for being part of my academic journey, but of my life.

Abstract

Memory safety bugs remain the primary source of security vulnerabilities in systems software. While memory-safe languages such as Rust offer a compelling alternative to C/C++, the amount of existing legacy code makes manual migrations impractical. As such, automatic translation is an attractive approach to improving safety in practice.

Existing translation tools, such as C2Rust, are fast but produce Rust code outside of the safe subset. Cpp2Rust, a novel tool, instead produces safe Rust from the outset, but its use of reference-counted mechanisms to circumvent the borrow checker introduces run-time checks and heap allocations, degrading performance.

This thesis presents DAEDALUS, a rule-based source-to-source rewriter that removes superfluous constructs introduced during translation when they can be shown to be unnecessary. A rule is written in ordinary Rust as a pair of source and target patterns, optionally accompanied by pre- and post-conditions. Rules are loaded at run time, making DAEDALUS extensible without modifying or recompiling the tool.

We evaluated DAEDALUS on a set of microbenchmarks and observed that our tool can recover the performance of some of the original C++ programs. On larger benchmarks, such as the Cpp2Rust translations of WOFF2 and Brunsli, DAEDALUS removes half of the reference-counting and borrow operations. To evaluate DAEDALUS as an independent tool, we ran it over the 1,000 most downloaded Rust crates, discovering 5,794 rewrites over 133 crates. This study resulted in 55 pull requests, with 39 merged, including contributions to repositories from the Rust Project itself, Mozilla, and Google.

Keywords

Rust; Source-to-Source Transformations; Rule-Based Rewriting; Automatic Code Translation.

Resumo

As vulnerabilidades de memória continuam a constituir a principal fonte de vulnerabilidades de segurança em software. Embora linguagens seguras como Rust constituam uma alternativa promissora a C/C++, a enorme quantidade de código existente exige mecanismos de tradução automática para que o processo de migração seja prático.

As ferramentas existentes, como o C2Rust, são rápidas, mas produzem código Rust fora do subconjunto seguro da linguagem. O Cpp2Rust, uma nova ferramenta, produz Rust seguro desde o início, mas a utilização excessiva de mecanismos de contagem de referências para contornar as regras do compilador resulta em verificações e alocações adicionais no heap, degradando o desempenho.

Para colmatar esta lacuna, esta dissertação apresenta o DAEDALUS, uma ferramenta de reescrita baseada em regras, que remove abstrações supérfluas introduzidas durante a tradução. As regras são escritas em Rust como um par de padrões de origem e destino, opcionalmente acompanhadas por pré- e pós-condições.

Avaliámos o DAEDALUS num conjunto de microbenchmarks e observámos que a ferramenta consegue recuperar, ou até superar, o desempenho de alguns programas C++ originais. Em benchmarks de maior dimensão, como as traduções do WOFF2 e do Brunsli pelo Cpp2Rust, o DAEDALUS removeu metade dos mecanismos de contagem de referências e operações de borrow. Para avaliar o DAEDALUS enquanto ferramenta independente, executámo-lo sobre as 1000 crates mais descarregadas, descobrindo 5794 modificações em 133 crates. Este estudo resultou na criação de 55 pull requests, dos quais 39 já foram aceites, incluindo contribuições para repositórios do The Rust Project, Mozilla e Google.

Palavras Chave

Rust; Optimização de Programas; Reescrita Baseada em Regras; Tradução Automática de Código.

Contents

1 Introduction **1**

1.1 Contributions 4

1.2 Outline 4

2 Rust **5**

2.1 Ownership and Borrowing 6

2.1.1 Borrowing 7

2.1.2 Lifetimes 9

2.2 Shared Ownership and Interior Mutability 10

2.2.1 Reference Counting 10

2.2.2 Interior Mutability 12

2.3 Optional Values 13

2.4 Traits 13

2.5 Compilation 14

3 Cpp2Rust **15**

3.1 Boxing Variables 16

3.2 Pointers 17

3.2.1 Reinterpretation 19

3.3 Optimization Opportunities 19

4 Daedalus **21**

4.1 Rewriting Loop 22

4.1.1 Passes 23

4.1.2 Applying Rewrites 24

4.1.3 Profitability of Rewrites 25

4.2 Rule Language 26

4.2.1 Rules as Functions 27

4.2.2 Conditions 27

4.2.3 Propagation 29

4.3 Correctness . . . 30
4.4 Consulting the Compiler . . . 30
4.5 Summary . . . 32

5 Implementation **33**
5.1 Choice of Front-End . . . 34
5.1.1 Requirements . . . 34
5.1.2 Candidates . . . 35
5.1.3 Comparison (and Decision) . . . 38
5.2 Rewrite Rules and the Intermediate Representation . . . 38
5.3 From Rules to Edits . . . 40
5.4 Deciding Conditions . . . 42
5.5 Limitations . . . 44

6 Evaluation **47**
6.1 Experiment Setup . . . 48
6.2 Microbenchmarks . . . 48
6.3 Case Studies: WOFF2 and Brunsli . . . 51
6.4 Optimizing the Top 1k Crates . . . 55
6.5 Summary . . . 57

7 Related Work **59**
7.1 Source-to-Source Translation . . . 60
7.2 Refining a Translation . . . 64
7.3 Rule-Based Rewriting . . . 68
7.4 Summary . . . 70

8 Conclusion **71**
8.1 Future Work . . . 73

Bibliography **73**

A DAEDALUS's Rules **81**
A.1 The `rc` Pass . . . 81
A.2 The `refcell` Pass . . . 83
A.3 The `cast` Pass . . . 83
A.4 The `clone` Pass . . . 84
A.5 The `option` Pass . . . 85
A.6 The `ptr` Pass . . . 87
A.7 The `peephole` Pass . . . 87

List of Figures

4.1 The rewriting loop 22
4.2 The default pass schedule 23

6.1 Microbenchmark ablation plot 49
6.2 Rewrites per crate 56

List of Tables

5.1 Candidate front-ends evaluated against DAEDALUS's requirements 38
5.2 DAEDALUS's conditions 43

6.1 The microbenchmark suite 49
6.2 Microbenchmark wall-clock times 50
6.3 Microbenchmarks' wrappers removed by DAEDALUS 50
6.4 Microbenchmark code sizes 51
6.5 Evaluation of WOFF2 52
6.6 Evaluation of Brunsli 53
6.7 Constructs removed from WOFF2 and Brunsli 53
6.8 Statistics of the 1,000 most downloaded crates 55

7.1 Source-to-source translators into Rust 64
7.2 Tools for refining a Rust translation 68
7.3 Summary of the available rewriting tools against DAEDALUS 70

List of Listings

1.1 A translated loop, before and after optimization . . . 3
2.1 Ownership differences: a copy and a move . . . 6
2.2 Borrowing instead of transferring ownership . . . 7
2.3 Violations of the aliasing rule . . . 8
2.4 A dangling reference . . . 8
2.5 A reference within and beyond its value's lifetime . . . 9
2.6 A function that needs a lifetime annotation . . . 9
2.7 A linked list implemented using `Rc`s . . . 11
2.8 Aliasing XOR mutability at run time . . . 12
3.1 A simple Cpp2Rust translation . . . 16
3.2 Preamble of a Cpp2Rust-translated function . . . 16
3.3 The variants of Cpp2Rust's `Ptr` . . . 17
3.4 Cpp2Rust's translation of pointers . . . 18
3.5 A C++ reference under the reference-counting model . . . 19
4.1 An example of a rewriting rule . . . 26
4.2 A metavariable binding to a code fragment . . . 27
4.3 A pre-condition deciding between two matches . . . 28
4.4 A propagating rule and a propagatee . . . 29
5.1 A rewrite rule parsed into the intermediate representation . . . 39
5.2 A rule whose metavariable must agree with itself . . . 42
7.1 A function in C and in CheckedC . . . 61
7.2 A C2Rust translation . . . 63
7.3 Aliasing in Crown's ownership inference . . . 67

Acronyms

AST Abstract Syntax Tree

CI Continuous Integration

FFI Foreign Function Interface

HIR High-Level Intermediate Representation

IR Intermediate Representation

LLM Large Language Model

LLVM Low-Level Virtual Machine

MIR Mid-Level Intermediate Representation

UB Undefined Behavior

1

Introduction

Contents

1.1 Contributions 4
1.2 Outline 4

As memory-safety bugs account for approximately 70% of reported security vulnerabilities in modern software systems [1,2], there is a growing effort to steer away from lower-level languages such as C/C++, which are particularly susceptible to these issues. To overcome this systemic flaw, many codebases are being rewritten in memory-safe languages [3], eliminating and preventing most memory-related vulnerabilities by design. This trend is even endorsed by the NSA, which issued a press release on the topic [4].

Hardening techniques may seem sufficient to address these memory vulnerabilities; however, security is often an *all-or-nothing* property, rendering efforts to address individual vulnerabilities ultimately insufficient. Although transitioning parts of a program to memory-safe languages has an immediate impact on security [5], developers often implement performance-critical software in non-memory-safe languages, fearing the performance drawbacks of memory-safe languages, such as run-time checks and garbage collection. The problem with this idiosyncrasy is that performance-critical software is often security-sensitive as well.

Rust, a relatively recent memory-safe programming language, stands out for its performance, as it achieves memory safety without the runtime overhead of garbage collection (unlike Java, Go, or Python). It instead relies on code adhering to an ownership model and passing through a strict borrow checker. As developers recognize that Rust can provide memory-safety guarantees without any performance degradation, while also improving developer productivity [6], manual rewrites of legacy applications in Rust are becoming increasingly popular [7, 8, 9]. A notable example of Rust's adoption in performance-critical environments is the Linux kernel, which now supports drivers written in Rust, marking a significant step toward integrating the language into critical systems and encouraging community adoption [10, 11].

As one might expect, rewriting software in memory-safe languages is neither quick nor straightforward and carries a significant risk of introducing errors [12, 13]. Often, developers attempting to improve the readability and maintainability of existing codebases may inadvertently alter functionality they do not fully understand, leading to subtle and hard-to-detect bugs [14, 15]. Furthermore, transitions between programming languages are often complicated because developers with deep expertise in the source language may lack equivalent proficiency in the target language, and vice versa, hindering collaboration and making accurate translation more difficult.

In response to these challenges, as well as the NSA's report, the automatic translation of existing software to memory-safe languages has emerged as a promising area of research. C2Rust [16] is the leading tool for translating C code into Rust, but it has a significant limitation: the generated Rust code does not belong to Rust's safe subset, thereby canceling the expected memory-safety benefits. Within `unsafe` blocks, the compiler still enforces borrowing rules where possible, but raw pointer dereferences are exempt from those checks, and C2Rust's output uses them pervasively—they are a major source of security vulnerabilities. Despite efforts to replace raw pointers with safe references [17, 18], C2Rust cannot currently serve as a viable solution for generating safe Rust code.

More recently, a different approach has emerged: Cpp2Rust [19] adopts the reverse strategy and produces safe Rust code from the outset. It achieves this by circumventing the problem's inherent undecidability and shifting Rust's ownership checks to runtime via reference-counting mechanisms. However, this approach comes at a considerable cost in terms of performance and readability, undermining Cpp2Rust's applicability in performance-critical contexts and hindering its long-term maintainability.

Listing 1.1 presents an example of a representative translation in which a simple C function that computes a sum is automatically translated into safe Rust by Cpp2Rust and subsequently optimized by hand. This example illustrates the impact on both readability and performance of automatic translation, as well as the potential benefits of targeted optimization of the generated code.

```c
__attribute__((noinline))
int f(int x) {
    return x;
}
int benchmark() {
    int sum = 0;
    for (int i = 0; i < 500000; ++i)
    ↪  {
        sum += f(i);
    }
    return sum;
}
```

(a) Original C version

```rust
#[inline(never)]
pub fn f(x: i32) -> i32 {
    let x: Value<i32> =
    ↪  Rc::new(RefCell::new(x));
    return (*x.borrow());
}
pub fn benchmark() -> i32 {
    let sum: Value<i32> =
    ↪  Rc::new(RefCell::new(0));
    let i: Value<i32> =
    ↪  Rc::new(RefCell::new(0));
    while (*i.borrow() < 500000) {
        (*sum.borrow_mut()) += ({
            let _x: i32 = *i.borrow();
            f(_x)
        });
        (*i.borrow_mut()).prefix_inc();
    }
    return *sum.borrow();
}
```

(b) Cpp2Rust Translation

```rust
#[inline(never)]
pub fn f(x: i32) -> i32 {
    x
}
pub fn benchmark() -> i32 {
    let mut sum: i32 = 0;
    for i in 0..500000 {
        sum += f(i);
    }
    sum
}
```

(c) Cpp2Rust Manually Optimized

Listing 1.1: Example translation of a program containing simple loops. The code on the left represents the original C program; in the middle, the Cpp2Rust translation; and on the right, a manually optimized version of Cpp2Rust's output. The C version was compiled with `-O3` and the Rust versions with `-release`, ensuring they have access to the same set of optimizations. A separate, non-inlined function produces the summed value so LLVM cannot recognize the loop as a reducible recurrence and collapse it into a Gaussian sum.

Compared to the original C implementation, Cpp2Rust's output is $13.7\times$ slower. However, after manually optimizing the generated Rust code, the performance gap was substantially reduced: the optimized version was only $1.14\times$ slower than the original C implementation.

This overhead is inherent to the translation approach; it is the price of a decision taken without sufficient information. As Cpp2Rust translates one construct at a time and cannot determine how a variable will ultimately be used, it adopts a conservative approach. Every variable is boxed inside an `Rc<RefCell<T>>` to provide *shared interior mutability*, since its address may be taken. For pointers, an additional `Option` is introduced, since every pointer may be null. For most variables in typical programs, these properties are not required, but this additional machinery is nevertheless retained. Eliminating this overhead is a rewriting problem over the translated source, where the program-wide information that Cpp2Rust lacked—how each variable is used—is now available.

This thesis presents DAEDALUS, a source-to-source, rule-based rewriter that eliminates unnecessary constructs introduced during translation. It consists of 54 rules, organized into passes, where each rule specifies the construct it matches, the code into which it rewrites that construct, and the conditions under which the rewrite is valid. Rules are written in ordinary Rust: two functions defining the source and target patterns, and two optional functions defining the pre- and post-conditions. Adding a new rewrite therefore requires only a new pair of functions, without needing to modify the tool itself.

1.1 Contributions

DAEDALUS is, to the best of our knowledge, the first rule-based source-to-source rewriter for Rust whose rules are expressed in *pure* Rust, with no additional dialect. Adding a rule requires neither tinkering with the tool's infrastructure nor recompiling it.

The code DAEDALUS leaves behind always type-checks, and a rule may state `well_typed()` as a post-condition, thereby delegating the validation of the resulting code to the Rust compiler rather than the rule's author. When a rewrite is rejected, DAEDALUS reports the compiler diagnostics responsible for its rejection, and identifies the responsible rule whenever the diagnostics can be attributed to a single rewrite.

DAEDALUS is not limited to Cpp2Rust-translated code. In a study of the 1,000 most downloaded crates on `crates.io`, all hand-written Rust, it found optimization opportunities in 133 crates, resulting in 39 already-merged pull requests in repositories maintained by organizations such as the Rust Project, Mozilla, and Google. These results demonstrate that DAEDALUS can be integrated into ordinary development workflows, including CI pipelines.

Across translated microbenchmarks and two realistic-sized translated programs, DAEDALUS narrows the performance gap with the original C++ implementations. Combined with the results of the crate study, these results demonstrate the potential of Cpp2Rust and DAEDALUS as a pathway for migrating legacy C/C++ software to safe Rust.

1.2 Outline

The document is structured as follows. Chapter 2 introduces the Rust programming language, providing the background necessary for this work: ownership and borrowing, as well as the constructs that enforce these properties at runtime. Chapter 3 describes how Cpp2Rust translates C/C++ programs, its strengths and limitations. Chapter 4 presents DAEDALUS, explaining what constitutes a rule, how rules are grouped into passes, and how rewrites are decided, applied, and rolled back. Chapter 5 describes the implementation: the front-end on which it is built, the intermediate representation into which rules are lifted, and how pre-conditions are evaluated. Chapter 6 measures DAEDALUS on translated microbenchmarks, two realistic case studies, and the 1,000 most downloaded crates on `crates.io`. Chapter 7 places this work among tools for program translation, translation refinement, and rule-based program rewriting. Finally, Chapter 8 summarizes the contributions of this dissertation and discusses possible directions for future work.

2

Rust

Contents

2.1 Ownership and Borrowing 6
2.2 Shared Ownership and Interior Mutability 10
2.3 Optional Values 13
2.4 Traits 13
2.5 Compilation 14

Rust is a programming language that combines performance, concurrency, and memory safety without run-time garbage collection. In some languages, such as Java or Go, a garbage collection system handles memory management; in others, such as C and C++, memory is the programmer's responsibility. Rust takes a third route, *ownership*: a set of rules the compiler enforces that rules out dangling pointers, double frees, and data races at compile time, while out-of-bounds accesses are prevented by run-time checks. The absence of run-time overhead for temporal memory safety makes Rust well suited for performance- and safety-critical systems.

Before exploring ways to translate C++ to Rust, or to optimize Rust code itself, it is essential to understand how Rust enforces memory safety and what its main features are.

2.1 Ownership and Borrowing

Static ownership is the idea that responsibility for an object is assigned, at compile time, to exactly one variable. So, the question "is this object still needed?" can be answered without any run-time garbage collection mechanism. Thus, the compiler can determine exactly who owns a given piece of memory, and whether that owner is still in scope. With that information, Rust can surgically insert memory allocation and deallocation instructions without the programmer's intervention. This lets Rust retain the performance of a manually memory-managed language while providing the safety of a garbage-collected one.

That guarantee rests on three rules, each of which the compiler checks: (a) each object has an *owner*; (b) there is only one owner at a time; and (c) when the owner goes out of *scope*, the object is deleted. Two aspects matter: (1) when an object is assigned to another variable, ownership is transferred, unless its type implements `Copy` (Section 2.4), in which case the value is duplicated instead; and (2) when a value goes out of scope, Rust invokes its destructor, the `Drop` trait, although the programmer can also invoke it earlier.

Listing 2.1 illustrates the concept of ownership.

```
1 fn ok() {
2     let x = 2;
3     let y = x;
4     println!("{x}");
5 }
```

```
1 fn not_ok() {
2     let x = String::from("a");
3     let y = x;
4     println!("{x}");
5 }
```

Listing 2.1: The assignment on the left copies the value, the one on the right moves it (i.e., its ownership). As a result, the left-side function type checks, whilst the right-side one does not.

In the example above, a variable `y` is declared and assigned the contents of variable `x`. Normally this would represent an ownership transfer, from `x` to variable `y`. However, this is not the case. Because `x` is `i32`—which Rust automatically infers for numeric literals—and `i32` implements the `Copy` trait, the contents are copied from one variable to another. At line 4, the variable `x` is still valid because its ownership was never transferred.

The example on the right, however, does not compile. That is because `x` is now of type `String`, which, unlike `i32`, does not implement the `Copy` trait. As such, in line 3, ownership is transferred from `x` to `y`. Now that ownership has been transferred out of `x`, using it later in line 4 causes a compilation error; `x` no longer owns the object and is invalid. This is how ownership works: when a non-`Copy` variable is passed, either to another let-binding or as an argument to a function, its ownership is transferred right away. For a function argument, this means the caller cannot use the variable after the call unless the function returns it.

2.1.1 Borrowing

Returning ownership from every function is impractical. A function would need to return every argument whose ownership it receives, quickly making this approach impractical. The concept of *borrowing* solves this problem: a value can be accessed temporarily without losing ownership. A function can now receive references to objects and access them freely, without worrying about giving back ownership. All of this is possible because the variable's ownership never left its place; the variable was simply *borrowed*. Rust achieves this through *references*, which come in two kinds: immutable (`&T`), which gives shared read-only access, and mutable (`&mut T`), which gives exclusive read-write access.

Listing 2.2 illustrates this borrowing concept.

```
1  fn not_owner(s: &String) {
2      println!("{}", s.len());
3  }
4  fn main() {
5      let x = String::from("a");
6      not_owner(&x);
7      println!("{x}");
8  }
```

Listing 2.2: Passing a reference instead of the value; `x` still owns the `String` after the call.

In the code snippet above, instead of passing a `String` as an argument—which would result in `x` losing ownership and the program failing to compile—a reference to that `String` (i.e., `&String`) is passed instead. When the function `not_owner` returns, `x` is still the rightful owner of the `String`, so the instruction on line 7 does not fail. However, passing it by reference has a caveat: the function `not_owner` can no longer change its value; it can only read it. If, instead, the function wanted to modify the value of `s`, the parameter would have to be typed as `&mut String`.

That is all it takes to write through a borrow, but a mutable reference comes with two conditions. First, the binding must be mutable, and variables are immutable by default. A `let` binding cannot be reassigned or modified unless declared with the `mut` modifier. Second, a mutable reference is *exclusive*: for as long as it is in use, no other reference to the same value—mutable or not—may be used. That second condition is the rule stated as *aliasing XOR mutability*: at any point in the program a value may be shared, through any number of *immutable* borrows, **or** mutated, through exactly one mutable borrow, but never both at once. Much of what Rust rules out follows from it. Take, for example, a data race: two accesses to the same value with at least one active writer. By not allowing more references while a mutable reference is in use, Rust automatically protects against these errors. Together with ownership, it gives safe Rust a property C and C++ lack: a program the compiler accepts has no undefined behavior. The cases C leaves undefined—reading freed memory, writing through an alias, racing on a value—are exactly the ones these rules reject, and what remains undefined is confined to code marked `unsafe`.

```
1 fn not_ok1() {
2     let mut s = String::from("a");
3     let ref1 = &s;
4     let ref2 = &mut s;
5     println!("{ref1}");
6 }
```

```
1 fn not_ok2() {
2     let mut x = String::from("a");
3     let ref1 = &mut x;
4     let ref2 = &mut x;
5     println!("{ref1}");
6 }
```

Listing 2.3: Two violations of the aliasing rule: a mutable borrow while an immutable one is still in scope (left), and two mutable borrows in scope at once (right).

Listing 2.3 above shows two situations where the borrowing rules prevent a program from compiling. On the left, a mutable reference `ref2` is created right after an immutable one. When that immutable reference is used later, the compiler issues an error. This is because `ref1` is still live at line 4—a reference's scope ends at its last use, and `ref1` is used on line 5—so the mutable borrow cannot be taken there, even though `ref2` will never itself be read. Had line 5 printed `ref2` instead, `ref1` would have been dead by line 4, and the function would compile.

On the right, the rule is violated by two mutable references to `x`, and the error is again at the second borrow: `ref1` is read after it was taken, so it is still live there. Had `ref1` not been used after line 4, the function would compile. These examples illustrate the *aliasing XOR mutability* concept in Rust: if we have multiple references, all must be immutable; if there is a mutable reference, it must be the only one.

Notably, all of these borrowing rules are checked at compile time. The mechanism that enforces them, called the *borrow checker*, is a compile-time analysis, so it introduces no performance overhead whatsoever. In summary, its task is to enforce the following properties: (1) a reference must not outlive the value it points to; (2) a value cannot be dropped while any reference to it is still active; (3) at any point in the program, all active references must satisfy the aliasing rules above.

Listing 2.4 shows the borrow checker at work.

```
1 fn dangling(x: String) -> &String {
2     &x
3 }
```

Listing 2.4: Because the borrow checker cannot guarantee the value will live as long as the reference, it rejects the function.

The above function does not compile as `x` owns the `String`. Since `x` is deallocated when the function returns—because it does not return ownership—the returned reference would outlive its value, breaking rule (1) above. A reference that the borrow checker accepts is a plain pointer at run time. Everything about it was decided before the program ran, so borrowing costs nothing; at most, it constrains the programming style. The constructs of Section 2.2 relax these constraints by adding run-time checks, albeit at the cost of performance overhead and the possibility of a run-time crash.

2.1.2 Lifetimes

For each reference, the borrow checker reasons about its *lifetime*: the scope during which the reference can be safely used. For a reference to be valid, the value it points to must not have been deallocated, which rules out *use-after-free* errors. In Listing 2.5, the example on the left is valid because the reference is used while the value it points to is still alive (in scope). On the right, the value is bound inside an inner block and dropped at its end, while the reference to it is used afterward, so the reference outlives what it points to.

```rust
fn ok() {
    let s = String::from("a");
    let ref1 = &s;
    println!("{ref1}");
}
```

```rust
fn not_ok() {
    let ref1;
    {
        let s = String::from("a");
        ref1 = &s;
    }
    println!("{ref1}");
}
```

Listing 2.5: A reference used while its value lives (left), and one used after its value went out of scope (right).

In Rust, most lifetimes are inferred automatically by the borrow checker. This *elision* is possible under three rules: (a) each input reference gets a lifetime of its own; (b) if there is exactly one input lifetime, the output references get it; and (c) in a method, the output references get the lifetime of `self`. However, the compiler cannot always unambiguously determine the relationship between reference lifetimes. In such cases, developers must write explicit lifetime annotations to clarify the intended borrowing semantics. This arises because the borrow checker is intra-procedural: it analyses one function at a time and never looks at its callers, so the checker cannot know what a signature does not say about how its references relate. An annotation does not change how long a reference is valid; it states how the lifetimes of several references relate, for the checker to verify. For instance, the type `&'a T` denotes a reference to a value of type `T` that is guaranteed to remain valid for at least the lifetime `'a`.

```rust
fn longest<'a>(x: &'a String, y: &'a String) -> &'a String {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 2.6: A function returning one of two references; without `'a` it does not compile.

Listing 2.6 shows a function that needs such an annotation. Without the annotations, the function looks straightforward: it takes two references to `String` variables and returns the longer one. The compiler, however, cannot determine which input's lifetime the returned reference depends on—it could be tied to `x`, `y`, or some other lifetime independent of both.

If only one of the references could be returned, say `x`, with `y` used only for intermediate calculations, the code would compile, since the borrow checker would know which reference is being returned. As either may be returned, the annotation has to encode the following constraint:

$$\text{lifetime}(\text{return}) \subseteq min(\text{lifetime}(x), \text{lifetime}(y))$$

The lifetime parameter `'a` does so: it expresses that both `x` and `y` live at least as long as `'a`, which is also the lifetime of the returned reference. The annotated version in the listing disambiguates the lifetimes, so that the compiler can enforce that the returned value does not outlive its inputs. With two different lifetimes, `<'a, 'b>`, the function would not compile either, since `y` might be returned under `'a` while living only for `'b`. To allow it, we would need a bound, such as `<'a, 'b: 'a>`, which reads "`'b` outlives `'a`".

Lifetimes appear in type definitions as well as in signatures. If a struct has a reference as a field, that reference must have a lifetime annotation. This ensures the reference remains valid as long as the struct is used. The checker enforces this as it does for a function, since every use of the struct uses the reference inside it. This is what normally makes a reference the wrong tool for a data structure whose parts must outlive one another; such a structure has to own its parts instead.

2.2 Shared Ownership and Interior Mutability

A structure that owns its parts runs into the single-owner rule as soon as one of them is needed in two places: a node reached from two others, a value held by two collections. Ownership admits exactly one owner, and the borrow checker ties every reference to that owner's lifetime, so neither a move nor a borrow suffices for these situations. For such cases, Rust provides a run-time mechanism, *reference counting*, which lets several owners share one heap-allocated value. The single-owner rule is not eliminated; it is shifted: the checks move from compile time to run time, incurring a performance overhead.

2.2.1 Reference Counting

The simplest heap allocation possible is `Box<T>`. It keeps to the rules above: it has one *single* owner, and the value is freed when that owner goes out of scope. To allow multiple owners, *shared ownership* is implemented through the `Rc<T>` and `Arc<T>` types instead; the former is reserved for single-threaded contexts, while the latter is an atomic variant designed for multi-threaded programs. These constructs automatically manage a value's lifetime via a counter of how many `Rc` handles to it exist. Every `.clone()` of an `Rc` increments the counter, and every `Rc` going out of scope decrements it; when it reaches zero, the value is deallocated.

Listing 2.7 represents what could be a struct definition for a *linked list*.

```
1  use std::rc::Rc;
2  struct Node {
3      value: i32,
4      next: Option<Rc<Node>>,
5  }
6  fn main() {
7      let first = Rc::new(Node { value: 1, next: None });
8      let second = Rc::new(Node { value: 2, next: Some(first.clone()) });
9  }
```

Listing 2.7: A singly linked list whose nodes share ownership of their successors through `Rc`.

In the example above, the first node is shared between its own binding and the `next` field of the second node—an `Option`, since a node may have no successor (Section 2.3). Cloning the `Rc<Node>` in line 8 increments the reference count. As such, the first `Node` is deallocated only when the count reaches zero, which means the second `Node` must be destroyed first; otherwise, `first`'s count would never reach zero, and the node would never be deallocated. A node that lives as long as either of its owners is what a reference cannot give: as Section 2.1.2 showed, each node must own its successor, and only shared ownership lets two nodes own the same one.

Every `Rc` reference counted so far keeps the value alive (i.e., the `Rc` is only dropped when there are no more active references). However, that is sometimes more than a link should do; sometimes we need only a reference that can observe the value without preventing it from being freed. The `Rc` references counted so far are *strong* references, and their count is the *strong count*. Besides tracking the number of strong references, the `Rc` construct also makes available a `Weak<T>`, a non-owning reference that does not count towards it. Instead of incrementing the strong count, it increments another field of an `Rc`, the *weak count*. These weak references are particularly useful for observing or linking to shared data without preventing its deallocation, effectively allowing programmers to break potential reference cycles. Moreover, accessing the underlying value through a weak reference requires an explicit upgrade to a strong reference, which may fail if the value has already been dropped (strong count reached zero).

An example is a *doubly linked list*, where each node points to both its next and its previous node. With strong `Rc` references in both directions, the two would form a cycle that keeps both counts above zero, and neither node would ever be deallocated.The solution, then, is to keep `next` as a strong `Rc` reference and `prev` as a `Weak` reference, upgrading it whenever the backward link is followed.

Although reference counting is useful for circumventing the borrow checker, it has a price. `Rc<T>` relieves the programmer of the ownership rules where a structure needs sharing, but it does so at run time. By storing its inner type `T` on the heap, rather than the stack, it incurs extra run-time costs for allocation and deallocation, but also for resolving the indirection associated with it, which would be absent if `T` were stack-allocated. Moreover, it maintains a reference count, and lightweight as each increment and decrement is, they add up: each is a memory read and a write.

2.2.2 Interior Mutability

While reference counting can provide shared ownership, it does not, by itself, allow mutation of the underlying value: an `Rc` hands out only shared references, and aliasing XOR mutability forbids writing through one. However, shared ownership and mutation must sometimes coexist. For these situations, Rust introduces the concept of *interior mutability*: the ability to mutate data even when there are immutable references to it. This is accomplished through the `RefCell<T>` type. Unlike compile-time borrow checking, `RefCell<T>` enforces aliasing XOR mutability at run time—naturally implying a run-time performance overhead. This run-time checking allows patterns the compiler would otherwise reject, such as mutating a value inside an `Rc<T>`. A `RefCell` tracks active borrows and panics if the rule is violated: what the compiler would have rejected at compile time, the program now discovers at runtime and stops.

Listing 2.8 shows both the pattern and its failure: on the left, a value shared through `Rc` is mutated through one reference and read through the other; on the right, a second borrow is taken while a first is still live, which Listing 2.3 rejected at compile time and which here panics at line 4.

```rust
fn ok() {
    let a = Rc::new(RefCell::new(1));
    let b = Rc::clone(&a);
    *b.borrow_mut() += 1;
    assert_eq!(*a.borrow(), 2);
}
```

```rust
fn panics() {
    let a = RefCell::new(1);
    let r = a.borrow();
    *a.borrow_mut() += 1;
    println!("{r}");
}
```

Listing 2.8: Aliasing XOR mutability checked at run time: a shared value mutated through `RefCell` (left), and a second borrow taken while the first is live (right), which panics at run time instead of failing to compile.

The type on the left, `Rc<RefCell<T>>`, is Rust's idiomatic way of achieving shared mutable data: the outer layer counts owners, the inner one counts borrows, and between them a value may be both shared and mutated. For the linked list of Listing 2.7, this is what a mutable node looks like: the successor field becomes an `Option<Rc<RefCell<Node>>>`, so that any node holding a successor may also change it. Every access then goes through `borrow()` or `borrow_mut()`, each of which checks the borrow flag and returns a guard that holds the borrow until it is dropped; a guard held too long is exactly what the right-hand function pays for. What was a field read in the plain struct is now an indirection and a borrow check on a heap-allocated node, and that is the shape shared, mutable data takes in Rust whenever the compile-time rules cannot express it.

2.3 Optional Values

The outermost layer of that node type, `Option`, is the last construct the list needs, and it stands in for what C and C++ do with a null pointer. `Option<T>` is an enum with two variants, `Some(v)` holding a value and `None` holding nothing, so the absence of a value is itself a value of the type rather than a special address; Listing 2.7 used it because a node may have no successor. Reading an `Option` means matching on it, which, although somewhat cumbersome, ensures that the program has decided whether a value exists before it tries to use one. The match can be skipped by calling `unwrap()`, which returns the value if it exists and panics if it does not; it trades the compile-time check for a run-time one and is generally discouraged for that reason.

2.4 Traits

Every construct mentioned so far—`Rc`, `RefCell`, `Option`—is a type, and a type may define methods of its own in an `impl` block. A *trait* names a set of methods that several types may share, and a type implements it by providing them in an `impl` block for that trait, so that code can be written against the capability rather than against any one type. Such code is generic, with the acting trait as a *bound* on a type parameter: `fn smallest<T: PartialOrd>(items: &[T]) -> &T` accepts a slice of any type whose values can be compared (i.e., which implements `PartialOrd`) and may call the comparison on them. The compiler checks the bound at each call, so a generic function is instantiated only with types that support what it does. If, for example, we were to call `smallest` with a slice of a type `T` which did not implement `PartialOrd`, the code would fail to compile.

Three important traits shape how values move, and Section 2.1 has already named two of them. `Copy` marks a type whose values may be duplicated by copying their bits: assigning or passing a `Copy` object leaves the original usable, which is why the `i32` of Listing 2.1 was not moved. `Clone` marks a type whose values can be duplicated explicitly by calling `.clone()`. Every `Copy` type is also `Clone`, but a `String` is only `Clone`, since duplicating it means allocating; cloning an `Rc`, as Section 2.2.1 showed, only increments a count. `Drop` is the destructor: a type that implements it runs code when a value goes out of scope, and it is what makes the end of a scope a deterministic point of cleanup.

2.5 Compilation

The compiler works on a *crate*: a tree of modules compiled together into a library or an executable, where names are resolved, types and borrows are checked, and errors are reported. A crate is not compiled from its text directly. `rustc` lowers it to representations of its own—the High-Level Intermediate Representation (HIR), on which type checking runs, and then the Mid-Level Intermediate Representation (MIR), a control-flow graph carrying the borrow checker and most of the remaining analyses—before handing the result to LLVM—the back-end it shares with `clang`. A *package* gathers one or more crates together with a manifest, the `Cargo.toml` file, which names the package and lists its dependencies, its lints, and its compilation profiles. Cargo, the build tool of the Rust Project, is what reads that manifest, fetches the dependencies it names, and invokes the compiler; in practice, every Rust project is a Cargo project.[1] It offers two ways to put a crate to the compiler: `cargo build`, which compiles it into an artifact, and `cargo check`, which runs only the front-end—parsing, name resolution, type and borrow checking—and stops before code generation. A check reports all errors that would be reported during a full build, but in a fraction of the time. Its diagnostics include the file, source span, and message associated with each error in a machine-readable form, making it an inexpensive way to determine whether a crate still compiles.

A crate does not correspond to a single program, but rather to a family of possible builds. Code guarded by `#[cfg(...)]`—by a named feature, operating system, or target architecture—is included only in the builds satisfying the corresponding condition. Similarly, a crate's tests, examples, and benchmarks are compiled only when explicitly selected, so the scope of a check depends on the build targets and configurations it is given.

What that check sees is the crate as the compiler has it, which differs from the original source in two important aspects. The first concerns macros. Macros—`println!`, `vec!`, and those defined using `macro_rules!` or procedural macros—are expanded before type checking and operate on tokens rather than typed syntax. As such, the check operates on the expansion, which does not appear explicitly in the source file; a source-based tool sees the macro invocation, whereas the compiler sees the code generated by its expansion.

The second concerns `unsafe` code, where the compiler imposes fewer restrictions. Within an `unsafe` block or function, the programmer may dereference raw pointers, call foreign functions, and perform other operations whose safety the compiler cannot verify. The borrow checker continues to enforce borrowing rules for references, but raw pointer dereferences are not subject to the same guarantees. Consequently, the safety guarantees described in Section 2.1 depend on the programmer rather than being enforced by the compiler.

[1] Some low-level projects cannot depend on Cargo and are built from standalone files instead; the best-known example is the Rust code in the Linux kernel.

3

Cpp2Rust

Contents

3.1 Boxing Variables 16

3.2 Pointers 17

3.3 Optimization Opportunities 19

Cpp2Rust [19] is the first tool to automatically translate C++ into safe Rust. Earlier translation tools, such as C2Rust [16], emit unsafe Rust. Cpp2Rust achieves this by sidestepping a problem that is undecidable in the general case—statically establishing ownership in the presence of unrestricted aliasing—by replacing compile-time ownership and borrowing checks with runtime mechanisms. `Rc` tracks shared ownership through reference counts, while `RefCell` enforces the borrowing rules dynamically. The translated program is therefore safe from the outset, and the memory bugs of the original (e.g., use-after-frees, double-frees, and out-of-bounds accesses) are detected at runtime and result in panics rather than undefined behavior.

This safety comes at a performance cost: rather than satisfying the borrow checker, Cpp2Rust instead relies on runtime ownership and borrowing mechanisms, which introduce heap allocations, reference-count updates, indirection, and runtime borrowing checks.

3.1 Boxing Variables

Every C/C++ variable is translated by Cpp2Rust into a `Value<T>`, which is an alias for `Rc<RefCell<T>>`. As Section 2.2 showed, this gives both shared ownership and interior mutability, with the rules checked only at run time. The reason for this is *aliasing*. Any C/C++ variable may have its address taken, and any number of pointers may then read or write it. This cannot be expressed with regular Rust references; the borrow checker would promptly reject them. Boxing every variable in this `Value<T>` abstraction in advance means that any of them can be pointed to later, without Cpp2Rust having to decide which. Listing 3.1 shows an example of an automatic translation of a simple program made by Cpp2Rust.

Original C/C++

```
int f() {
    int a = 2;
    int b;
    b = 3;
    return a + b;
}
```

Translated Rust

```
pub fn f() {
    let a: Value<i32> = Rc::new(RefCell::new(2));
    let b: Value<i32> = Rc::new(RefCell::new(0));
    *b.borrow_mut() = 3;
    return *a.borrow() + *b.borrow();
}
```

Listing 3.1: A C/C++ program (on the left) adding two integers and returning their sum. Every variable in the translation (on the right) is boxed inside a `Value`, and every access is a run-time borrow of it.

Every variable is boxed in an `Rc<RefCell<T>>`, so stack-allocated variables now live on the heap. This is problematic in three ways. First, the memory usage increases. Second, performance takes a hit: every access to that variable (i.e., a read or write) needs to go through another level of indirection and then through the `RefCell`'s run-time borrow check. Third, it hurts code readability.

Function parameters are boxed too, but not in the signature. A C++ argument passed by value stays a plain value in the Rust signature. However, the function's body must be able to treat the parameter like any other variable. To that end, Cpp2Rust adds a preamble to every function that shadows each parameter with a boxed version of itself. Listing 3.2 illustrates this preamble.

Original C++

```
int sum(int a, int b) {
    return a + b;
}
```

Translated Rust

```
pub fn sum(a: i32, b: i32) -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(a));
    let b: Value<i32> = Rc::new(RefCell::new(b));
    return *a.borrow() + *b.borrow();
}
```

Listing 3.2: Parameters reach the function unboxed and are shadowed right away in a preamble.

Boxing in the body rather than in the signature has one advantage: whether a parameter needs boxing is decided by the function that receives it, not by its callers. Since most function arguments never have their address taken, most of these boxes are unnecessary, and thus can be eliminated locally.

3.2 Pointers

Cpp2Rust needs a way to represent C++ pointers in safe Rust. Rust references are subject to the ownership rules described in Section 2.1—one writer, or many readers, and none outliving its value—which makes them unsuitable for translating programs that freely alias memory, as C/C++ programs often do. Rust raw pointers, on the other hand, do not impose these restrictions and would be a natural counterpart to C++ pointers; however, dereferencing them requires unsafe code. Cpp2Rust therefore introduces a custom type, `Ptr<T>`, whose operations—arithmetic, array-to-pointer decay, comparison, and dereferencing—are implemented using safe Rust and supplemented with run-time checks.

This custom `Ptr<T>` consists of a pair: an offset, for pointer arithmetic, and a `Weak` reference into the allocation it points to. Note that this reference is weak because of destructors. C++ programs often rely on objects being destroyed at a precise point (e.g., a mutex released at the end of a scope, a file closed when its handle dies), and a strong reference held by a stray pointer would keep the object alive past that point, thereby altering the timing of destructor invocation. By using a weak reference, the object is destroyed at the same point as in the original C++ program, and a pointer that outlives the object becomes dangling without extending its lifetime. This also allows Cpp2Rust to detect memory bugs such as *use-after-free*s: when an attempt is made to access an already freed object through a `Ptr<T>`, the upgrade to a strong reference fails. Cpp2Rust relies on this property: the memory safety bugs of the original program result in panics in the translated program.[1]

`Ptr<T>` supports multiple kinds of pointed-to objects: null pointer, a single value, a fixed-size array, a growable buffer (equivalent to, e.g., `std::vector`), or a pointer cast to a different pointee type. Listing 3.3 shows the definition of `Ptr<T>`.

```
enum PtrKind<T> {
    Null,
    StackSingle(Weak<RefCell<T>>),
    StackArray(Weak<RefCell<Box<[T]>>>),
    StackVec(Weak<RefCell<Vec<T>>>),
    HeapSingle(Weak<RefCell<T>>),
    HeapArray(Weak<RefCell<Box<[T]>>>),
    HeapVec(Weak<RefCell<Vec<[T]>>>),
    Reinterpreted(Rc<ReinterpretedView>),
}
```

Listing 3.3: Definition of Cpp2Rust's `Ptr<T>`.

Two things in the listing deserve a note. The stack and heap variants are kept apart because only a heap allocation has a leaked owning reference for `delete` to recover, so a `delete` on a stack value—which is UB in C++—is caught as such here. Regarding the array variants, they carry one reference counter for the whole allocation rather than one per element: the pointer holds a weak reference to the array and the offset of the element it points to, keeping the cost of an array close to that of a single value.

[1] In some cases, such as integer overflow, Cpp2Rust generates safe wrapping alternatives.

Three operations cover what a program can do with a pointer. Taking the address of a variable is `as_pointer()`, which downgrades the variable's `Rc` to a weak reference and wraps it in a `Ptr`. Pointer arithmetic is `offset()`, which adds to the offset field and nothing more. Dereference is the one operation the type cannot do the obvious way: a `Ptr` cannot return a `&T` or a `&mut T` to its pointee, since the reference would hold the `RefCell` borrowed with nothing to bound its lifetime. Instead, `read()` and `write()` clone a value out of or into the allocation, and, where a reference is genuinely needed, `with()` and `with_mut()` take a closure and hold the borrow for exactly as long as it runs. Listing 3.4 shows all three on a small function.

Original C++

```
1  int f() {
2      int a = 2;
3      int b[3];
4      int *p = &a;
5      int *q = b + 1;
6      g(q);
7      return *p + *q;
8  }
```

Translated Rust

```
1  pub fn f() -> i32 {
2      let a: Value<i32> = Rc::new(RefCell::new(2));
3      let b: Value<Box<[i32]>> = Rc::new(RefCell::new(Box::new([0; 3])));
4      let p: Value<Ptr<i32>> = Rc::new(RefCell::new(a.as_pointer()));
5      let q: Value<Ptr<i32>> = Rc::new(RefCell::new(b.as_pointer().offset(1)));
6      g(q.borrow().clone());
7      return p.borrow().read() + q.borrow().read();
8  }
```

Listing 3.4: A C++ function taking the address of a variable and of an array element (above), and its translation (below): each address becomes a `Ptr`, and each dereference a checked read.

In the listing above, lines 4 and 5 of the translation take an address and step through an array; the pointer variables `p` and `q` are themselves boxed—being variables like any other—and the argument to `g` is a clone of the pointer, so one more weak reference to `b`. The two dereferences in the return are `read`s, and each is an upgrade of the weak reference, a borrow-flag check, and an offset—so every pointer access in the translated program pays three checks whereas the original paid none. Notably, as C++ references are non-null constant pointers, Cpp2Rust skips the emission of the `Value<T>` wrapper and just emits them as plain `Ptr<T>`. Listing 3.5 illustrates this optimization.

Heap allocation goes through this same `Ptr<T>` type. C++'s `new` becomes `Ptr::alloc`, and `malloc`, `calloc`, `realloc`, and `strdup` allocate through the same call, which creates the allocation's `Rc` and then deliberately leaks it, so that the allocation outlives the statement that made it. C++'s `delete` recovers the leaked reference and drops it, and panics if the object has been freed already. A `std::unique_ptr<T>` owns its allocation and may be null, so it becomes an `Option<Value<T>>`.

Original C++

```
1  int f() {
2      int a = 1;
3      int &p = a;
4      p = 2;
5      return a;
6  }
```

Translated Rust

```
1  pub fn f() -> i32 {
2      let a: Value<i32> = Rc::new(RefCell::new(1));
3      let p: Ptr<i32> = a.as_pointer(); // no Value<T> boxing
4      p.write(2);
5      return *a.borrow();
6  }
```

Listing 3.5: A C++ reference and its translation. The reference is a `Ptr` with no `Value` around it, and the assignment through it on line 4 is a single `write`.

3.2.1 Reinterpretation

C++ supports reading a memory location with a pointer of a different type than the stored data in some situations. Consider a `long` inspected byte by byte through a `char*`, a buffer from `malloc` used as an array of structs, or a `sockaddr_in` passed where a `sockaddr` is expected. Each of these is a `reinterpret_cast`. The translation, however, has no bytes for such a cast to point at: it holds typed Rust values inside `Value` cells, not a flat block of memory that two pointers of different types could both address. Cpp2Rust therefore supplies those bytes *on demand*. Every translated type carries its C byte representation, generated beside the struct: a size, a serialization of each field at its C offset, and the inverse. A `reinterpret_cast` then yields a `Ptr` of the reinterpreted kind—a handle to the original allocation and a byte offset—through which a read serializes the overlapping elements of the original and parses the target out of the bytes, and a write is a read-modify-write back into it, so every view sees every write. This is the *most* expensive access in the translation, a whole serialization per access.

3.3 Optimization Opportunities

Cpp2Rust can also emit an unsafe translation in the style of C2Rust. Every C++ pointer or reference turns into a Rust raw pointer, nothing is boxed and no run-time checks are introduced. That version carries none of the machinery of this chapter and runs at the speed of its C++ counterpart. It is, however, *unsafe*: the memory bugs of the original program remain in the translated program instead of becoming panics. The safe translation is therefore the input of DAEDALUS, and the question is how much of that speed can be recovered without leaving the safe subset.

Everything that the translation model adds is added on the assumption that it might be needed. Cpp2Rust does not know, at translation time, how a variable will be used, and prioritizes a program that compiles over one that is fast. A variable is boxed because its address might be taken, an `Option` wraps an owner because it may be null, *etc.* For most variables in most programs none of this happens, and the translated code carries this pessimistic machinery anyway.[2] That is where the opportunity lies:

[2] This is by design: Cpp2Rust decides per construct, and never looks at how a construct is used.

removing these unnecessary run-time checks, returning the heap-allocated objects to the stack, and recovering the lost performance.

The simplest wrapper to remove is the `Option` type. A variable that only ever holds `Some` does not need it, and taking it away shortens every access to that variable. Although not an optimization *per se*, it simplifies a variable's type, by removing the `Option` wrapper.

Removing a `Value` is the next step, and the one where most of the performance lies. A variable whose address is never taken—no `as_pointer()` on it and no clone of its `Rc`—holds the only reference to its value, and its `Rc` wrapper type can safely disappear. Once it has, the `RefCell` inside can go too, wherever the borrows never overlap, which for translated code they never do, since each `borrow()` lasts one expression; the variable becomes a plain `let`, or `let mut` if it was ever borrowed mutably. Removing both the `Rc` and the `RefCell` thus removes the indirection and the run-time checks, respectively.

Every one of these decisions is local to a function body, and that locality is what makes them tractable. It is also exactly what the remaining construct lacks.

What makes `Ptr<T>` hard to do without is not any one of its operations but the relation it creates. A variable can lose its `Value` box only if no `Ptr` points into it, since the box is precisely what a `Ptr` holds a handle to. Any single pointer can be turned into a reference on its own, and a `Ptr` and a reference to the same variable can coexist; but the box can only be removed when no `Ptr` is left. In practice, this means that the last `Ptr`s and the box have to go in one single change, with a pointer passed to another function, making that function's signature also part of the same change. The unit of the rewrite is therefore not a pointer but an alias set, and one that crosses function boundaries. That is what sets pointers apart from the boxing of Section 3.1: a `Value` on a variable nobody points to can be removed on its own, and a `Ptr` never can. Whether such a rewrite is safe is, in general, undecidable, and an analysis can only approximate it; but a wrong approximation need not be fatal, because a rewrite that was not safe produces a program the compiler rejects. That observation—that the analysis may guess as long as the compiler decides—is what the rest of this thesis builds on.

4

Daedalus

Contents

4.1 Rewriting Loop 22

4.2 Rule Language 26

4.3 Correctness 30

4.4 Consulting the Compiler 30

4.5 Summary 32

DAEDALUS is a source-to-source rewriter for Rust programs. It reads a crate and writes it back with a set of rewrites applied; these may be readability- or performance-oriented. It emits source code and is lossless, meaning comments, formatting, and everything the rewrites do not touch are fully preserved.

Every rewrite rule is specified in a DSL as a pair of Rust functions: one that gives the code to be matched, and the other that gives the code that replaces it. A rule may also have pre- and post-conditions. A set of static analyses decides where the conditions hold. The rewrites are all based on rules; none is hard-coded, so adding a new rewrite means simply writing another rule: no visitor to implement, no multi-file changes.

4.1 Rewriting Loop

DAEDALUS rewrites a crate in rounds. A round is a set of logically organized collections of rules, which we call passes, each targeting its own type of rewrites. Simply put, a round goes through every file of a crate and applies the rewrites it finds. These rounds repeat until one of them changes nothing, that is, until a fixed point is reached: no rule can find any other rewrite.

To fully rewrite a crate, a single round (i.e., a single fixed-point iteration) would not suffice. This follows from the rewrites themselves, not from how they are scheduled. A rewrite changes two things: what the next rule will see, and what that rule is allowed to do.

The first is a matter of exposure: what the rule sees when trying to match. Take, for example, an `Rc<RefCell<Rc<T>>>`. According to the implementation of DAEDALUS and the rules for removals of `Rc`—which do not support the removal of nested constructs—the inner `Rc` can only be removed once it is visible (i.e., when the `RefCell` wrapper type is removed). So, if we want to remove both `Rc`s, a single round will never be enough, regardless of the passes' ordering.

The second is a matter of pre-conditions. A pre-condition is judged where the rule matches, so it depends on the surrounding code. Rewriting that code can turn a failed pre-condition into one that holds, making a rewrite that was refused earlier possible. A single round would, therefore, stop well short of what the rules could actually reach and rewrite; thus DAEDALUS runs until a fixed point.

Figure 4.1 visually illustrates the architecture of DAEDALUS.

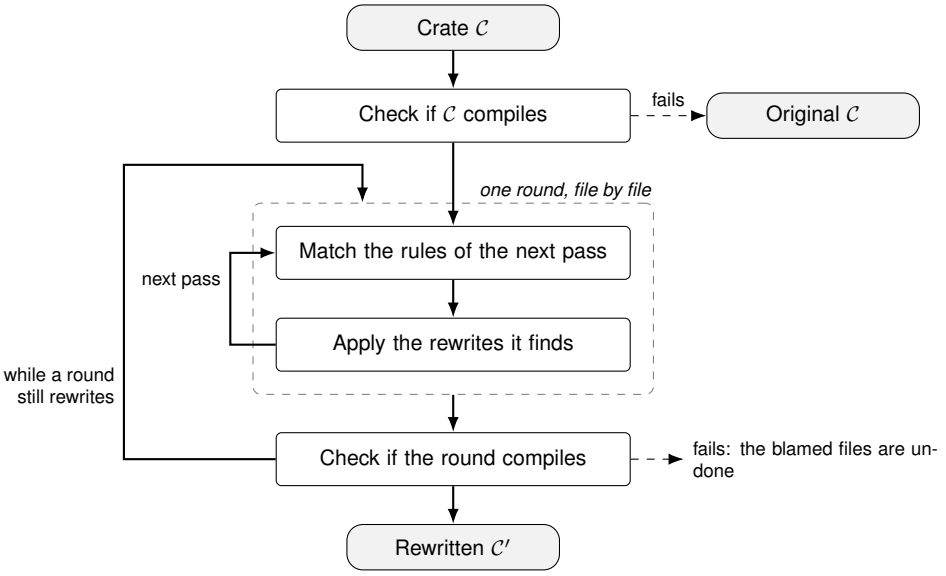


Figure 4.1: The rewriting loop. A crate that does not compile is never rewritten. A round takes each file through the passes in turn, compiling the round afterward: if it fails, the files whose errors are named are put back—or the whole round if that is not enough—with the rules that caused the unsound rewrites reported. This serves as a safety net to catch any unsound rules the user writes. These rounds run until reaching a fixed point, or, if preferable, until a maximum number of iterations.

4.1.1 Passes

Within a round, each file is taken through a schedule of passes. A pass is a set of rules, and normally it aims to eliminate the same construct. For example, an `Rc` pass aims to eliminate unnecessary `Rc`s, or a `clone` pass aims to remove redundant clones. A user can thus organize their rewrites in passes, logically fitting rules together. Grouping rules into passes leaves two separate orders to consider, both of which matter.

The first ordering is within a pass: the first rule that matches a node is the one applied. This implies that the rule for the most specific case should be placed above the general rule; otherwise, the specific case would always be shadowed and never matched. The second ordering is between passes and defines what constructs should be removed first.

The latter ordering is a design choice rather than a requirement. It does not influence the outcome of DAEDALUS, but it does affect the number of rounds needed to reach the fixed point. Take, for example, the common pattern of combining shared ownership and interior mutability: `Rc<RefCell<T>>`. As we know, this is a very common pattern—and definitely more common than `RefCell<Rc<T>>`—so it would be preferable to have the pass in charge of removing `Rc`s come before the one in charge of removing `RefCell`s. Although it does not affect DAEDALUS's final output by any measure, it would improve performance, as it would require one fewer round to reach the fixed point.

Ordering is not the schedule's only concern: what a pass leaves behind matters too. Sometimes, rewrites may create new, temporary bindings to maintain semantics. Commonly, those temporary constructs are actually redundant (e.g., a binding which can be constant-folded) and can be safely deleted. To this end, the pass schedule detects these cases and implements a solution: if a pass holds a rule that rewrites a binding, and with it the uses of that binding, then a special pass called `peephole` is appended to it, and the two run as a unit until the `peephole` reaches a fixed point.

Figure 4.2 illustrates an example pass schedule and the appended `peephole` passes.

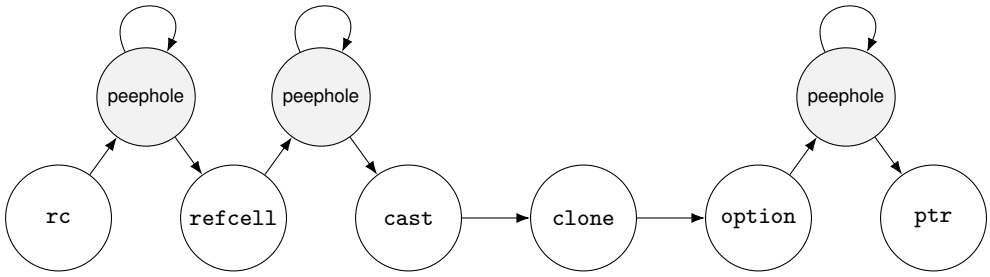


Figure 4.2: The default pass schedule. A pass whose rules rewrite a let-binding, and with it the uses of that binding, appends the special peephole pass, which in turn runs until it reaches its own fixed point. The remaining passes rewrite expressions in place and leave nothing for the peephole to collect, so they can run straight on to the next pass in the schedule.

This `peephole` pass is never scheduled independently. DAEDALUS detects which passes carry *propagating* rewrites—those that rewrite a let-binding and, with it, every use of that binding—and appends the peephole to them. It removes the garbage a rewrite leaves behind, such as bindings, checks, or unnecessary scopes.

The simple case is a let-binding that is never used, in which case a dedicated rule in the `peephole` removes it. The harder case is detecting a binding that is still used, but could be deleted. To this end, the `peephole` pass can propagate constants and fold them until nothing further can be folded. If it proves that the binding is indeed redundant, it can safely delete it. That is why the `peephole` pass runs to a fixed point: to apply constant propagation and folding.

4.1.2 Applying Rewrites

Inside a pass, the smallest unit of work is a function, and a rewrite may affect only its body. It can be a free function, a method of an `impl`, or a function declared in an inline module. Each function's body is scanned with the rules of the current pass, and the matches that scan yields constitute that body's worklist. The pass drives every body at once, taking matches from those worklists until none has anything left; a body that runs out early simply stops contributing new edits.

Not every candidate in a worklist can be committed on the spot. A rule may require that the code it produces typechecks—as a post-condition—and a rewrite from such a rule is considered final only after it has been validated by the compiler (see 4.2). Checking each one individually would be rather costly, but a rewrite never reaches outside the body that produced it, so bodies are independent units of work. This allows the pass to take one tentative rewrite from each body and pass them to the compiler (see 4.4); two bodies can be checked at the same time without either explaining the other's failure. The worklists are ordered so that the rewrites needing no check come first: those are committed together, and the rest are put to the compiler.

Whatever the compiler eventually rejects costs nothing to take back, and that is by construction: a rewrite is never applied where or when it is found. The pass collects edits, each one a syntax node paired with the text that replaces it, and only materializes the rewrites at the end. These edits may not overlap, because the whole scheme rests on being able to drop any one of them without touching the rest. The scanner therefore never proposes a rewrite inside a node that another rewrite is already replacing; that nested opportunity is found by a later scan, once the outer rewrite has landed. Reverting is therefore an omission rather than an undo: the edit is simply not applied to the file's tree, so nothing has to be *put back*. That materialized text is then handed back to the analysis, so the next pass matches against what this one produced rather than against the file as it arrived.

4.1.3 Profitability of Rewrites

Not every post-condition is about correctness. A rewrite that changes a binding's type leaves temporary variables and logic in its place. It could be a flag (e.g., recording whether an `Option` is holding a value), a counter, or additional checks. And that rewrite may have a post-condition to assert that this temporary logic is gone by the end of the rewrite. The `peephole` is what makes that true: it propagates the constants the rewrite introduced and folds them away, and when nothing names these introduced temporary constructs, the rewrite is said to be worth it, to be profitable. This is another way to revert a rewrite, not by type checking, but by satisfying a profitability function; in this case, folding the introduced temporary constructs. The system therefore checks profitability conditions only after the pass and its peephole have reached a fixed point. If the extra constructs and checks survive, the proposed edit is abandoned.

At the round level, validation proceeds differently. When every file has been through its schedule, the crate is checked once, as a whole. This check is a best-effort attempt to find faulty rules that propose rewrites that do not compile and are therefore incorrect. Restoring every file would be suboptimal: thousands of rewrites may be correct, and a single incorrect one would cause the compiler to discard them all. The compiler names the file each error is in, so it restores those files to the text they had when the round began, then checks the crate a second time. That second check makes the selection safe, since the files the errors name are not necessarily the files at fault: a rewrite in one file may break another by changing a signature it exports. If the crate compiles once the named files are back, the rest of the round stands; if it does not, every file the round rewrote is restored. The price is one further compilation, paid only by a round that failed.

Restoring the files an error names is as far as DAEDALUS goes. It undoes what a rule broke, never repairs it, and never tries to use certain rewrites. The premise is that a rejected rewrite is a defect in a rule rather than an accident to work around. A rule whose rewrites are often rejected is one whose pre-conditions are too weak, and the appropriate response is to report this so that the rule can be corrected or its pre-conditions refined. DAEDALUS does not work around an unsound rule; instead, it reports exactly which one is responsible, along with the edit it proposed and the error it caused, grouped so that a rule rejected many times stands out. If the user is not confident about the correctness of the rule they wrote, they should use a specific post-condition to assert that the rewrite typechecks (see 4.2.2). For these reasons, each round either lands what the compiler accepts or puts the rest back, which keeps the crate on disk in a compilable state between rounds and at the end.

4.2 Rule Language

Rules are the core of DAEDALUS. A rule is written as a small group of ordinary Rust functions, sharing a name and told apart by a suffix. Rules have a source pattern, which they try to match in source code, and a target pattern that represents the desired substitution. Because these rules are the core of DAEDALUS and support its extensibility claim, we want them to be practical to write while carrying as much expressiveness as possible. To that end, rules can also carry pre- and post-conditions.

Listing 4.1 represents an example of a real DAEDALUS rule, exercising many of its features.

```
fn clone1_pre<T>(a0: Rc<T>) -> bool {
    !rules::is_ref(&a0) && rules::last_use(&a0)
}

fn clone1_src<T>(a0: Rc<T>) {
    a0.clone();
}

fn clone1_tgt<T>(a0: Rc<T>) {
    a0;
}

fn clone1_pos() -> bool {
    rules::well_typed()
}
```

Listing 4.1: Rewrite rule that drops a clone of a reference-counted `Rc` value. It matches `a0.clone()`, where `a0` needs to be of type `Rc<T>`, and replaces it with just `a0`. Additionally, the rule also expresses the pre-condition that it will not fire unless `a0` is not a reference and this is its last use. After application, as a safety measure, what it produces may be typechecked.

In the above rule, there is no bespoke syntax. Notably, to the best of our knowledge, this is the first tool that is capable of describing rewrites in plain, unmodified Rust. A rule says what it matches, when it may fire, what it promises, and what it produces. And it says all of it in ordinary Rust, with the type system doing the work a pattern language would otherwise have to invent. The file that contains the rules compiles with the rest of DAEDALUS, so the compiler checks every rule before the tool runs: a rule that does not typecheck is a build failure, not a runtime surprise. Nevertheless, to facilitate rapid development, you do not need to recompile DAEDALUS just to add a new rule. DAEDALUS parses the rules and alerts the user if a rule is malformed.

4.2.1 Rules as Functions

Rules are all about functions. Every rule must always have two functions: `_src` and `_tgt`. For more expressiveness, it also admits `_pre` and `_pos` functions. The function names matter: for functions to be part of the same rule, they must share the same prefix; for example, the functions in Listing 4.1 all share the `clone1` prefix. Then we can add the appropriate suffix to indicate whether the function refers to the source or target pattern, a pre-condition, or a post-condition.

The bodies of the rules are read as patterns, rather than as programs—DAEDALUS never calls them—and their parameters are holes in those patterns. These parameters have a special name: the rule's *metavariables*.

When trying to match a rule's source pattern with some source code, each metavariable is bound to a fragment of the program under rewrite, and the target is rebuilt with those bindings put back in.

Listing 4.2 demonstrates this substitution.

The rule

```
fn clone1_src<T>(a0: Rc<T>) {
    a0.clone();
}
fn clone1_tgt<T>(a0: Rc<T>) {
    a0;
}
```

Before

```
consume(self.cache.clone());
```

After

```
consume(self.cache);
```

Listing 4.2: The rule presents an `a0` metavariable that will be matched, and then substituted in the target pattern. The matching is not restricted to just a name; it can be whatever expression stands in that position, in this case `a0` bound to `self.cache`. The target is then rebuilt with that fragment put back in.

The above match was more than a shape search. When devising a rule, the parameters, which we now call metavariables, are *typed*. The parameter in Listing 4.2 is `a0: Rc<T>`, meaning that the match will only happen if the receiver of the `clone` call is actually of type `Rc<T>`. Hence, not every receiver of a `clone` is matched; only those with the right type. Besides ensuring a type, we can also extend matching expressiveness with traits. For example, we could specify `T: Copy`, which would narrow the match further: besides needing to be an `Rc<T>`, the `T` would also need to implement the `Copy` trait. In contrast to a custom pattern language, DAEDALUS can leverage Rust's type system to ensure the signature is well-typed.

4.2.2 Conditions

A rule may carry two further functions, and both return a boolean over the same metavariables. The first, `_pre`, is a pre-condition; it is evaluated at the match site, and a match that does not satisfy the condition is never proposed. It is built from a vocabulary of predicates, such as `last_use` or `never_calls`, combined with the ordinary Rust operators for conjunction, disjunction, and negation.

Currently, DAEDALUS supports 15 such predicates; Table 5.2 describes them. Notably, although it is not as straightforward as adding a new rule, users can also craft their own conditions. Conditions may assert, for example, that a given type does not have a manual implementation of a given trait, that a given variable is never used again, that a given method is never called on the variable to be matched, or even that a type does not implement a certain trait at all (similar to Rust's under-development feature of *negative-impls*).[1]

Listing 4.3 shows how a condition decides between two similar matches.

The rule

```
fn clone1_pre<T>(a0: Rc<T>) -> bool {
    !rules::is_ref(&a0)
        && rules::last_use(&a0)
}
fn clone1_src<T>(a0: Rc<T>) {
    a0.clone();
}
fn clone1_tgt<T>(a0: Rc<T>) {
    a0;
}
```

Before

```
let x = data.clone();
consume(data.clone());
```

After

```
let x = data.clone();
consume(data);
```

Listing 4.3: The rule of Listing 4.2 again, now with the pre-condition it actually carries, on a `data` of type `Rc<T>`. Both lines match the source pattern and are indistinguishable by shape; only the second is rewritten because only there does `last_use` hold. A syntactic-only matcher could not tell the two apart.

The other function, `_pos`, is a post-condition, and it has two kinds. The first, `well_typed()`, asserts that the result still compiles. The second, `folded()`, asserts that a named binding is gone by the time the pass and its peephole have settled.

Both kinds of conditions are written the same way. In essence, each is a propositional logic formula where each predicate asserts something about the code. Each predicate is an ordinary function, declared to return `true` and carrying a comment that says what it means. The engine supplies the meaning by reading a condition rather than running it. So a condition, too, is written as Rust and nothing else.

A pre-condition never asks about shape; it is essentially semantic information about the code surrounding the rewrite. For example, `last_use` is a question about liveness; `side_effect_free` about what an expression may do; `no_manual_impl_drop` about which impls the crate provides. A set of over-approximating static analyses evaluates these pre-conditions.

[1] https://doc.rust-lang.org/beta/unstable-book/language-features/negative-impls.html

4.2.3 Propagation

Rules have a taxonomy. They have roles: `Simple`, `Propagating`, or `Propagatee`. A rule's role is not declared; it follows from the shape of its source. For example, a rule that matches an expression is a `Simple` rule; it rewrites that expression and nothing else. If, however, the rule's source pattern is a let-statement, the situation is different. If rewriting the statement changes the type of what the binding holds, it is called a `Propagating` rule. This is because the rewrite is not self-contained; it propagates/affects all uses of that let-binding. To this end, we have a third role: the `Propagatee` rules, which rewrite the uses of a type-modified let-binding. Notably, this classification has an exception: if the source pattern is a let-binding but the target pattern is empty, it removes the statement immediately and leaves nothing to propagate. Every rule of this kind lives in the `peephole` pass, whose job is to remove the bindings that earlier rewrites leave dead.

We call the set of rewrites constituted by a `Propagating` rewrite and the `Propagatee`s it drags along a *bundle*. The rewrites in a bundle are atomic, meaning that either all of them land, or none does. This is necessary because changing a binding's type, while leaving its uses unchanged, results in an ill-typed program.

A `Propagating` target therefore has to say two things its source cannot, and a naming convention covers each. A metavariable ending in `_type`, such as `a1_type`, stands for the type of the metavariable it names, not a fragment of code. Names appearing in a target but not in its source, such as `a0_s_count`, are the bindings the rewrite introduces to stand in for what the removed type used to track. Listing 4.4 shows a `Propagating` rule with one of its `Propagatee` rules, and uses both.

```
fn rc1_pre<T>(a0: Rc<T>) -> bool {
    rules::never_calls(&a0, "as_pointer")
}
fn rc1_src<T>(a0: Rc<T>, a1: T) {
    let a0 = Rc::new(a1);
}
fn rc1_tgt<T, a1_type>(a0: a1_type, a1: a1_type) {
    let a0: a1_type = a1;
    let mut a0_s_count = 1;
    let mut a0_w_count = 0;
}

fn rc3_src<T>(a0: Rc<T>) { Rc::strong_count(&a0); }
fn rc3_tgt<T>(a0: T, a0_s_count: usize) { a0_s_count; }
```

Listing 4.4: A propagating rule, `rc1`, and one of its propagatees, `rc3`. Because the `rc1` rule matches a let-statement, it is propagating: it strips the `Rc` from the binding and, in its place, introduces the counters that stand in for what the discarded wrapper type used to track. On one hand, the suffixed parameters carry that information, in this case, the strong and weak counts of `Rc`. On the other hand, `a1_type` stands for the type `a1` was inferred to have, rather than for a value. Every use of the binding must then be rewritten to read the counters instead, which is what `rc3` does for `Rc::strong_count`. Neither rule is applied alone. Their application is atomic: either all rewrites are applied, or none is applied.

Notice what a rule never had to do. It did not declare its role, name the pass it belongs to, say which other rules it depends on, or state that its match is type-sensitive. All of that is inferred from code the author would have written anyway, so adding a rewrite costs two functions, four when it carries conditions, and nothing else.

4.3 Correctness

DAEDALUS preserves compilation. Given a crate $\mathcal{P}$, if it compiles beforehand, then the rewritten crate $\mathcal{P}'$ is guaranteed to compile as well. This is not a claim about the rules; it is a property of DAEDALUS and its rewrite pipeline. The compiler is run on the rewritten crate, and whatever it rejects is reverted. Even a rule that produces ill-typed code cannot pass DAEDALUS.

In contrast, behavioral equivalence is not guaranteed. If a user is trying to delete the `Rc` wrapper type and, in one of its `Propagatee` rules—say the one substituting the uses of `Rc::clone()`—they forget to increment the strong count (i.e., `a0_s_count`) in the target pattern, then the behavior may differ. A straightforward example is a control-flow branch that depends on the strong count of the `Rc`. Although the compiler accepts the program, its behavior is altered.

This is deliberate: altering a program's behavior may be intentional, so DAEDALUS allows it. As such, DAEDALUS's correctness guarantee is only this: whatever the rules propose, a crate that compiled before rewriting *always* compiles after.

The rules DAEDALUS ships with aim past that: each carries pre-conditions meant to leave behavior intact, refusing to remove an owner that is aliased or whose type implements `Drop` by hand, and refusing to remove a clone whose original is used again or whose `Clone` does more than copy. Those pre-conditions are approximate, so one can misjudge a site, and when it does, the crate still compiles—it is the behavior the rules meant to preserve that gives way.

4.4 Consulting the Compiler

The compiler decides which rewrites survive, and nothing is rewritten before the crate is known to build. In this first *sanity check*, DAEDALUS compiles the crate as it was given, with all the features and targets enabled (i.e., with the `-all-features` and `-all-targets` flags). This often fails—some code is architecture-specific, and some depends on libraries that are not present—so DAEDALUS tries narrower combinations in turn, ending with no flags at all: a plain `cargo check`. If no combination compiles, the run ends there, with no file touched.

What the compiler rejects is informative only against a clean baseline: on a crate that already fails, an error after a rewrite would say nothing about the rewrite. The same dependence explains the order in

which the combinations are tried: the broader the configuration that compiles, the more of the crate falls under every check that follows. Whatever that configuration does not build—typically the tests, examples, and benchmarks—is left as it stood rather than rewritten unchecked. So the guarantee covers everything DAEDALUS rewrites: code outside the chosen configuration goes unchecked, but it also goes untouched.

Putting a rewrite to the compiler is expensive. A check is nothing less than a `cargo check` over the whole package, not over the line that changed. Cargo's caching keeps that from being a full build—dependencies are not re-examined, only the crate that changed—but a check still implies a process launch and a fresh pass over that crate, and one per rewrite would dominate the run. The number of invocations is kept down in two ways. First, not every rewrite needs checking; only a rule whose post-condition asks for `well_typed` delegates its decision to the compiler; the rest are committed as proposed. Second, rewrites that do require checking are tested together. DAEDALUS keeps a worklist per function body, and a check round takes the first pending rewrite that requires checking (i.e., whose post-condition includes `well_typed`) from every body at once, applies them on top of everything already committed, and runs a single check over the result. As each function has a distinct body, one invocation can test as many rewrites as the file has functions.

A failing batch has to be blamed on something. The compiler reports each error with a span, and DAEDALUS takes the bodies whose text overlaps one of those spans as the culprits: it drops their tentative rewrites and keeps the rest of the batch. This rests on the assumption that rewrites are *local* and do not produce errors outside their functions' bodies. This is *almost* what happens every time, but sometimes, due to trait resolution, type inference, or a signature change, an error may be reported at the use rather than the cause, landing in a body that was never touched.

The *blaming* mechanism therefore cannot be trusted as the final word. When a round finishes, DAEDALUS checks the crate once more, and if that check fails—or cannot be carried out at all—the files the errors name are restored to what they held when the round began, and the crate is checked again; if it still fails, every file the round rewrote goes back. A misattributed error costs rewrites and not correctness: what survived a bad blame is discarded rather than kept on trust. The case is rare, and this last check is what keeps it from mattering: whatever the blame gets wrong, the crate that remains on disk compiles.

That assurance costs a bounded amount, and the bound is not the number of rewrites. Within a file, one invocation serves every function body at once, so the count follows the deepest per-body worklist. One further check closes each pass. That is what makes checking by compilation affordable: the compiler is consulted often enough to have the last word on every rewrite that asks for it, and seldom enough that it does not dominate the run.

4.5 Summary

DAEDALUS is a rewriter tool that, paired with the Rust compiler, lets users confidently apply rewrites in their codebases. All its rewrites are based on user-defined rules; nothing is hard-coded, making it accessible for anyone to extend DAEDALUS with their own rewrites for their own purposes. Because Rust is one of the few languages that combine performance with a strong type system, we leverage its attributes and trust the compiler to ensure every rewrite is compilable.

Although DAEDALUS applies only compilable rewrites, these may change the program's behavior. This is by design: the user should have the power, and, consequently, the responsibility, for changing their program. Because the compiler has the last word, everything upstream of it may be cheap: pre-conditions are answered by an approximate analysis, blame for a failed batch is assigned by where the errors land, and neither is trusted further than a compile can check.

Rules in DAEDALUS are written as ordinary Rust functions. This is a bonus: users have no learning curve to write their first rewrites. A rule is a source and a target function, with a pre- and post-condition when it needs them. And that is it; nothing else is needed. The rewrite's role, its pass, the rules it depends on, and how it matches the source code are all the responsibility of DAEDALUS. An approximate analysis with a compiler that has the last word is what makes DAEDALUS extensible: a rule can be cheap to write and still wrong, and the cost of being wrong is a reverted rewrite.

5

Implementation

Contents

5.1 Choice of Front-End 34
5.2 Rewrite Rules and the Intermediate Representation 38
5.3 From Rules to Edits 40
5.4 Deciding Conditions 42
5.5 Limitations 44

DAEDALUS is implemented in roughly 6,000 lines of Rust, and is distributed as two executables: `daedalus`, which rewrites a single file, and `daedalus-cargo`, which rewrites a Cargo workspace. Both share the same pipeline; single-file mode wraps its input in a synthetic crate and calls the workspace path. Currently, DAEDALUS has 54 transformations, grouped by the construct they eliminate. The tool derives its rewrites mechanically from this set of (valid) Rust functions, so adding a new transformation does not require writing a new visitor to traverse the AST. The implementation of the tool consists of the machinery that turns these functions into applicable edits: a lifter and parser that reduce them to an intermediate representation, a matcher that finds if the source patterns appear in a crate, and a pass manager that applies the resulting edits whilst ensuring that the result is always a valid program.

5.1 Choice of Front-End

As a source-to-source rewriter, DAEDALUS is expected to consume the source code of a Rust crate and emit source code for that same crate, with a set of transformations applied. On top of that, DAEDALUS must also accurately infer type information to guide its rewrites. Therefore, it is built on an existing Rust front-end, where the choice of the concrete front-end dictates what the tool can and cannot do. For this purpose, we considered four candidates: Clippy, Syn, the Rust compiler's internal representation, and rust-analyzer. They differ less in what they can parse than in what they retain of the original text, what they know about it, and whether they offer any way to change it.

5.1.1 Requirements

Four requirements follow from DAEDALUS's intended behavior. The first three are properties of the front-end itself. The fourth is a practical constraint for adoption.

R1 – Lossless source representation. The output of DAEDALUS is expected to be readable, reviewable, and maintainable by a programmer. A front-end that produces source code only by pretty-printing an internal representation discards everything that representation does not model, such as comments or formatting. Therefore, the front-end must preserve the original text *verbatim*, without any loss.

R2 – Types, traits, and resolved names. Most rewrite rules in DAEDALUS are guarded by conditions that cannot be decided syntactically. Removing a clone, for example, may require the receiver to implement `Copy`, and deciding that requires a prior semantic analysis. Other conditions are formulated over a variable's uses rather than its type, such as a predicate asserting that a use is the last one. Enumerating those uses syntactically is not sufficient: a traversal of the source text does not see uses inside unexpanded macro invocations, and a rewriter that would ignore those uses would be unsound. Hence, the front-end must be capable of resolving the types of arbitrary expressions, answering trait queries, and indexing uses by the resolved name rather than by source text.

R3 – Tentative rewriting. DAEDALUS may apply a rewrite before knowing whether it is admissible. It writes a batch of rewrites, invokes the compiler, and retracts those the compiler rejected. Applying speculatively implies that the front-end must ergonomically support taking a rewrite back. An edit to the program must be a first-class construct that can be held and discarded, rather than a mutation already made to a shared tree that could only be undone by implementing non-trivial undo logs or re-parsing techniques. The speculative application has a second consequence: while a tentative batch is in place, the crate may not typecheck, and the front-end must keep answering queries about it, even if the crate is ill-typed.

R4 – Availability and stability. The front-end must be usable as a library, have a well-documented

interface, and be stable enough that the tool does not break between releases.

5.1.2 Candidates

As a new, emerging programming language, Rust does not yet have a *de facto* library or API for analyzing program source code. Rust does have a considerable set of tools built on MIR, the control-flow representation on which the compiler runs its borrow check, such as Miri [20] and the deductive verifiers Creusot [21] and Verus [22]. None of them works at the source level, and the community has not yet elected or developed a standard way to analyze Rust source code. Below are presented the main candidates for reading, analyzing, and modifying Rust code at the source level.

Clippy: As the official Rust linting tool, Clippy [23] is the most established tool for generating automated Rust code suggestions. The majority of these code suggestions, or as Clippy calls them, lints, target code readability: not passing too many arguments to a function, using iterators correctly, or avoiding cloning an object. And almost all Rust projects have Clippy running on their CIs to help them catch these anti-patterns before code gets to production.

This was the first alternative considered, since it already provided a good rule abstraction, a whole-crate traversal, and full access to the compiler's type information through its use of `rustc_private`. Adding a new rule to DAEDALUS would amount to adding a new lint to our custom version of Clippy. However, we quickly rejected Clippy for several reasons.

1. Every new rule would imply a new Clippy lint. The code for the new lint could be generated: the declaration, the pass skeleton, and the registration are mechanical. However, the matcher, type checks, and the conditions are not, and a generator that emitted them would contain the whole matching engine anyway, so Clippy would contribute only its traversal. Every new rule would require rebuilding the linter, whereas DAEDALUS reads its rules from source at run-time.

2. To rewrite a crate, there is often the need to apply and revert edits many times during the rewriting process, a feature Clippy does not support. For Clippy, a lint is merely a diagnostic; even a machine-applicable suggestion depends on other machinery to be applied, in this case `rustfix` [24].

3. Lints are expected to be only intra-procedural, which would confine DAEDALUS to just intra-procedural rewrites. This is not a limitation of Clippy's implementation that could be worked around; it is a deliberate property enforced by Clippy's developers.

4. Clippy does not make use of the compiler as an oracle, nor does it have a notion of borrow-checking. In contrast, our rewriter runs in a fixed-point loop using the compiler as an oracle to verify the soundness of the produced rewrites.

Although Clippy meets **R2** fully, it fails to be a lossless representation (**R1**), and cannot apply edits and reverts on its own, thus failing requirement **R3**.

Syn: This is the most famous crate the Rust community uses for programmatic manipulation of Rust syntax; in fact, `syn` [25] is recommended by the compiler team for creating new macros. It provides a complete parser for the language, a conventional algebraic representation of the AST, and, together with `prettyplease` [26], a pretty-printer to get back the source code. `syn`'s design reflects its intended use: macros. It takes a token stream produced by the compiler, produces a mutable AST, and outputs a token stream back to the compiler, so neither comments nor formatting are meaningful, and neither is preserved.

This philosophy makes it somewhat incompatible with DAEDALUS's objectives. First, it applies to a single file or snippet of Rust code, so it does not support whole-crate traversal. Second, it is not lossless; it loses comments and custom formatting. And, more decisively, `syn` performs no name resolution or type inference; it is merely syntactic. Therefore, neither **R1** nor **R2** is satisfied, rendering `syn` insufficient for the needs of DAEDALUS.

Rustc's internal APIs: Using the internal, private compiler crates gives access to all possible type information. And this semantic information is *sound*: the types are the ones the compiler itself computed, so a pre-condition evaluated against them is as accurate as the compiler's own type checking. This is the strongest possible position with respect to **R2**. In fact, DAEDALUS was originally implemented this way and remained so for 225 commits; unfortunately, it had to be replaced, since three problems emerged:

1. **The AST is mutable, but untyped.** The rewrites must be expressed over the AST, which retains the source program's structure and can be mutated, but has no notion of types. Type inference and name resolution only happen after the AST has been lowered to the (immutable) HIR level.[1] Moreover, the compiler provides no mapping between the two, and there is no supported way to obtain one.[2] The first implementation worked around this by keying type information on source spans, which is flaky: it works only against the pristine program and breaks for code inside macro expansions.

2. **There is no edit API.** The compiler's AST is designed to be *lowered*, not edited. It does offer a `MutVisitor`, but it is used only to implement the compiler's own desugaring passes (e.g., macro expansion or name resolution). There is no notion of an edit as a value, so a proposed change cannot be constructed, held without being applied, combined with other changes, or discarded.

[1] One could argue that lowering to MIR would allow us to recover the necessary mutability information. However, this approach is problematic because there is no way to faithfully reconstruct the original source code from MIR.

[2] https://internals.rust-lang.org/t/mutation-creation-of-hir/24214

Every rewrite is an in-place mutation, which makes **R3** unattainable: retracting a tentative rewrite would imply either a hand-written undo log or a full re-parse.

3. **The representation is not lossless.** The only way to emit a modified AST is to pretty-print it with `rustc_ast_pretty`. The problem with that is that it discards all non-doc comments, since the parser does not retain them in the first place. This was discussed with the parser team: their position was that `rustc_ast_pretty` carries no correctness guarantees and is simply a debugging mechanism. Therefore, the requirement **R1** cannot be satisfied.

It is also necessary to account for the fact that the `rustc_private` crates are only available on nightly toolchains, implying no stability guarantees. They are not documented anywhere, and the only way to learn how to use them is to read the compiler's source code. This is a cost rather than an obstacle, but it means that requirement **R4** is not fulfilled. Also, the compiler team, which was regularly consulted throughout this work, advised against building on these APIs.[3]

Rust-analyzer: This is the official Language Server Protocol (LSP) implementation for Rust. Rust-analyzer [27] is, at its core, a second, independent front-end for the language, written for a setting whose demands are close to those of DAEDALUS. Like an IDE, our rewriter must analyze code (that may not compile), produce modifications to source files (including macros), and have a notion, even if incomplete, of type-checking. Rust-analyzer can be easily used as a library, even with a stable toolchain, fulfilling requirement **R4**. Three further properties address the remaining requirements:

1. Its syntax trees are built on `rowan` [28], a concrete syntax tree (CST) rather than the typical abstract syntax tree (AST) compilers normally use. This means it retains every token of the input, including whitespace and comments, so it reproduces an unmodified region of code exactly; it is lossless.

2. It performs its own name resolution and type inference, and exposes the results through an easy-to-use semantic API. The API can answer trait queries and reach into unexpanded macro invocations, mapping what it finds there back to offsets in the original file. These analyses reimplement the compiler rather than reuse it; however, the two do not always agree, so it cannot be treated as a source of truth the way the `rustc_private` APIs can. To this end, the rust-analyzer team is working towards using the compiler's own implementation.

3. Rust-analyzer represents a change in the source code as a value (e.g., a text edit over a range) rather than as a tree mutation, so a set of edits can be constructed, applied, and reverted with ease. Its analyses, memoized through `salsa` [29], can answer queries about ill-typed or incomplete code, so the state left behind by an unverified rewrite remains analyzable.

[3] https://rust-lang.zulipchat.com/#narrow/channel/131828-t-compiler/topic/Using.20rustc.20APIs.20for.20source-to-source.20transformations/near/574853757

Rust-analyzer therefore satisfies **R1** and **R3**, but meets **R2** and **R4** only partially: the types it exposes are its own, not the compiler's, and the library interfaces are not well documented.

5.1.3 Comparison (and Decision)

None of the four candidate front-ends satisfies every requirement. Both `syn` and the compiler's internal APIs fail three of the four requirements. Clippy meets **R1** only partially because a machine-applicable suggestion is a text replacement and therefore does not reformat the rest of the code; however, it needs to be applied by an external tool which does not satisfy **R3**. Moreover, it only meets **R4** partially as it would need a non-trivial amount of extension for the implementation to support tentative and inter-procedural rewrites. Rust-analyzer, by contrast, satisfies all the requirements, even if only partially. It can only meet **R4** partially as its crates are not well-documented; however, the community is very active and helpful.[4] Regarding **R2**, the rust-analyzer team is making progress on switching from its own analyses to rustc's and already reuses multiple compiler analyses.

Table 5.1 summarizes the four candidates against the requirements of Section 5.1.1.

	clippy	syn	rustc APIs	rust-analyzer
R1 – Lossless source	◑	○	○	●
R2 – Types, traits, names	●	○	●	◑
R3 – Tentative rewriting	○	○	○	●
R4 – Available and stable	◑	●	○	◑

Table 5.1: Candidate front-ends evaluated against the requirements of Section 5.1.1. The selected alternative is highlighted in yellow. ● denotes a requirement that is met, ○ one that is not, and ◑ one that is met only partially.

The implementation initially used rustc's private APIs; however, DAEDALUS was soon migrated to rust-analyzer. The migration replaced the front-end while leaving the rule language, intermediate representation (parser and lifter), pass manager, and pipeline in place. This was needed as **R1** and **R3** are non-negotiable, and only rust-analyzer could fulfill them.

5.2 Rewrite Rules and the Intermediate Representation

A rule is written in Rust and compiled with the rest of DAEDALUS, but it is never run. The rules of a pass are read from their source file (under `src/rules/`) when the passes are built, parsed with the same front-end DAEDALUS uses for matching rewrites, and lifted into a list of `ir::Rule` values. This IR is nothing more than a pattern tree for the rule source, a second for the target, and a condition tree for each pre- and post-condition, if any. Nothing in Chapter 4 depends on the shape of that structure, and

[4] https://rust-lang.zulipchat.com/#narrow/channel/185405-t-compiler.2Frust-analyzer/topic/Applying.20source-to-source.20rewrites.20with.20type.20information/with/598372999

it could be replaced without changing any of that chapter's claims; it is, however, where the conventions of Section 4.2 acquire meaning, and the only place they are enforced.

Because the rule files are read at run time rather than at compile time, a new rule takes effect without rebuilding the tool. Parsing them on every invocation may look like overhead, but it accounts for less than 2% of the time spent rewriting a single file and less than 0.1% of rewriting an entire crate.

Describing a rule takes three things: a shape to look for, a shape to put in its place, and the circumstances under which the exchange is allowed. The first two are patterns—Rust syntax with holes in it—and they are built from one set of node types, which mirror the fragment of Rust a rule may mention, and no more. An `ir::PatBlock` is a list of `ir::PatStmt`, and a statement can be a let-binding or an expression. An `ir::PatExpr` covers the forms the rules have needed so far: method calls, calls, references, casts, unary and binary operators, assignments, control flow, blocks, literals, and paths. This list is deliberately short. The IR is not a full intermediate representation of Rust, unlike the HIR, so the rule parser does not accept all valid Rust. Listing 5.1 shows the `clone1` rule on both sides of the lift.

Specification

```
fn clone1_pre<T>(a0: Rc<T>) -> bool {
    !rules::is_ref(&a0)
        && rules::last_use(&a0)
}
fn clone1_src<T>(a0: Rc<T>) {
    a0.clone();
}
fn clone1_tgt<T>(a0: Rc<T>) {
    a0;
}
fn clone1_pos() -> bool {
    rules::well_typed()
}
```

Intermediate Representation

```
Rule {
  name: "clone1",
  rule_ty: Simple,
  src_vars: { a0: Rc<T> },
  tgt_vars: { a0: Rc<T> },
  src: Semi(MethodCall(MetaVar("a0").clone())),
  tgt: [Semi(MetaVar("a0"))],
  pre_cond: And(Not(IsRef(a0)), LastUse(a0)),
  post_cond: WellTyped,
}
```

Listing 5.1: The `clone1` rule of Listing 4.1 with the `ir::Rule` that the parser lifts it to. The four functions become one value: each side declares its metavariables once, with what it demands of them, and the patterns name them; the source statement becomes a pattern holding a `MetaVar` where `a0` stood; the target becomes a block of a single statement; and the two boolean expressions become condition trees. Paths are shortened, the `ir::PatBlock` wrapper is written as a list, and empty trait lists are elided.

The one node in the listing that is not parsed as regular Rust is the `MetaVar`: a hole, or *metavariable*, in the place of `a0` in the rule. It is where the parameters of Section 4.2.1 end up: each parameter of a `_src` declares a metavariable, and the pattern refers to it by name. What the parameter's type and bounds say is kept beside the pattern, in `src_vars`, as it is a condition on what the hole may bind and not a property of any occurrence of it: a metavariable used ten times is declared only once. The two sides declare separately because a rewrite may change the type of a metavariable—`rc1` holds `a0: Rc<T>` in its `_src` and `a0: a1_type` in its `_tgt`—and it is that difference which says what the rule is for. A bound is attached only when the parameter's type is the bounded type parameter itself: `a0: T` under `T: Copy` yields a metavariable that matches only a `Copy` expression, whereas in `a0: Rc<T>` the `T` is part of the

type being matched and may carry no bound of its own. A type parameter may carry more than one bound, and the metavariable then matches only an expression whose type satisfies all of them.

The circumstances that dictate when and where a rewrite may happen are what `pre_cond` and `post_cond` hold in the listing: a condition, a propositional tree over those same metavariables. This `ir::Condition` offers `Pred`, `Not`, `And`, and `Or`, and the boolean expression that a `_pre` or a `_pos` returns is parsed into one, with each call becoming a `Predicate` and each `&a0` argument the metavariable it names. The argument is a reference by necessity rather than by convention: a condition, as in ordinary Rust, must compile, and `a0` passed by value would be moved into the first predicate, making it unavailable to the next. Hence, the borrow lets one metavariable appear in several predicates; the parser requires the `&`, but discards it when lifting to the IR.

Two checks then run on the resulting parsed rule. The first rejects a rule that states a post-condition in its `_pre`, or a pre-condition in its `_pos`, since the two are decided at different times and by different means. The second confines `well_typed()` to a top-level conjunct of a `_pos`: a rewrite may promise to typecheck alongside other promises, but a negated or disjoined `well_typed()` is not something the pipeline could act on. That restriction also makes one question cheap. Whether a rule needs the compiler at all is read off its post-condition once, when the rule is lifted, by walking that conjunction; a rule whose post-condition never mentions `well_typed` is committed as it is proposed. In Listing 5.1, `clone1`'s `post_cond: WellTyped` is what sends it to the compiler.

That leaves one field of the listing, and the rule never mentions it: `rule_ty: Simple`, the role of the rule. Section 4.2.3 says the role of a rule follows from the shape of the source and the rule's place among the rules around it, and the parser reads both. A source pattern is a single statement—more than one is an error—and when that statement is a let-binding, the rule is `Propagating`, or `LetRewrite` if the target is empty. Anything else is `Simple`, unless a `Propagating` rule has already been seen in the file, in which case it is a `Propagatee`, one of the rules that rewrite the uses of the let-binding the propagating rule changed. That last case is why the parser keeps an ordered list of rule names beside the map it groups the functions into: iterating the map would lose the order of the file, and with it the only signal separating a propagatee from a simple rule.

5.3 From Rules to Edits

Each pass lives in its own file, and the set of passes DAEDALUS offers is the set of those files. `DEFAULT_PASSES` names six of them: `rc`, `refcell`, `cast`, `clone`, `option`, and `ptr`. This is just a default. A user who wants only some of the passes, or another order, names them with the `-passes=a,b,c` flag; a name no file provides is rejected, and the run does not start. Notably, the `peephole` pass is not on the list and cannot be named there. Instead, it is appended to every pass with a `Propagating` rule, which Section 4.1.1

describes from the other side: a pass whose rewrites leave temporary bindings and operations behind needs rules to fold them away.

Those passes run over a crate, one file at a time, and within a file, one function body at a time; each body has its own worklist. When the input is not a Cargo project but a single file, the executable wraps it in a temporary crate first, since rust-analyzer analyzes crates, not files, and copies the rewritten file back out. The worklist is constructed by scanning the body against the pass's rules, ordered so that the rewrites needing no typecheck come first, and drained from the front. When it empties, the pass finishes, and the next pass over the file scans the body again, now rewritten.

Scanning a body is a preorder walk of its syntax tree, although not every body is walked. DAEDALUS supports unsafe blocks and treats them the same way, except for rewrites which have the `well_typed` post-condition. In that case, the compiler cannot be trusted to verify unsafe code, so those rules do not apply. Conversely, code gated on a feature or target the chosen configuration does not enable is not considered for rewriting. This is similar to the above, because the compiler could never truthfully check whether the code compiles.

Where a body is walked, the rules are tried at each node in the order they appear in their file, and the first that matches wins: a let-binding is offered to the propagating rules, every expression to the simple ones, and a rule matches only if its pattern fits and its pre-condition holds there. A match consumes its subtree. In practice, this means the walk does not descend into a node it has matched, since the rewrite replaces it; whatever was nested inside is found on the next fixed-point iteration, once the outer rewrite has landed. Propagatees are scanned in a second phase, and only against the bindings a propagating match has just produced, so that each joins the bundle of the rewrite that made it necessary and none fires on its own.

Macros need one more step. A rule never fires on an unexpanded macro call, because there is no syntax there to match; instead, the scanner asks rust-analyzer to expand the call and matches inside the expansion. Only the parts of an expansion that came *ipsis verbis* from the call site can be rewritten, since only those map back to a range in the source. Anything the macro synthesized is left alone, and such an edit replaces a token range rather than a node.

Whether the scan offers a node of the body or an expansion, fitting a pattern means walking the pattern and the expression together, with parentheses looked through. A metavariable binds on first sight and must agree afterward: the first occurrence of a metavariable captures the expression under it, and every later occurrence of the same name matches only an expression with identical text; not exactly the *same*, since parentheses are looked through. Patterns are therefore non-linear, and equality is textual. A typed metavariable asks rust-analyzer for the expression's type and compares it with the type written in the rule structurally, generic argument by generic argument, a bare type parameter standing for anything; a trait bound is checked through rust-analyzer's trait queries, which at present cover `Clone` and

`Copy`. The comparison is conservative in one deliberate way: an expression whose type rust-analyzer cannot resolve never matches a constrained metavariable, since a rewrite admitted on a guess is one the compiler would probably have to refuse later.[5] Listing 5.2 shows the smallest rule that relies on a metavariable agreeing with itself.

The rule

```
fn self_let_binding_pre<T>(a0: T) -> bool {
    rules::no_manual_impl_drop(&a0)
}
fn self_let_binding_src<T>(a0: T) {
    let a0 = a0;
}
fn self_let_binding_tgt() {}
```

Before

```
let x = x;
let y = x;
```

After

```
let y = x;
```

Listing 5.2: A peephole rule that removes a binding of a name to itself, and what it does to two bindings. The source uses `a0` on both sides of the let-binding, so the statement matches only when the name and the initializer are the same identifier: the first line matches and, the target being empty, is deleted; the second does not, because `y` and `x` differ.

Once matching has bound the metavariables, building the replacement is a walk of the target pattern, substituting each metavariable with what it is bound to; the result is a detached syntax node ready to be spliced in. Two things happen here that the source pattern could not express. A metavariable with the `_type` suffix becomes a written type annotation, taken from the type recorded when the metavariable it is named after was bound, which is how a target states a type the source never mentioned. A name that appears only in the target becomes a fresh binding.

Together, the match and its replacement make an `Edit`: the node—or in the case of a macro, a token range—it replaces, the nodes that replace it, the rule that produced it, and the bundle it belongs to. `Edit`s are collected across every body in the file; should one bundle's edit enclose another's, as when a propagatee lands inside a later let-binding, the outer bundle waits for the next round. Those that remain are applied in a single `SyntaxEditor` pass, which is what makes reverting a tentative rewrite trivial: an edit that is not committed is simply not applied. The result of a pass is a tree rather than text, so the next scan reads it without re-parsing.

5.4 Deciding Conditions

A pre-condition is a propositional logic formula built on predicates. These predicates are declared as functions that return either `true` or `false`, depending on what the analysis infers. In fact, a pre-condition is decided only at the match site, on the metavariable it names, meaning all predicates are evaluated lazily. Only after a match has happened (including types and bounds) do the predicates' analyses take place.

[5] *Probably*, because rust-analyzer's type information, though very close to the compiler's, is not the same, and the two do diverge on occasion.

Predicate	Holds when	Decided by
Syntactic		
`side_effect_free`	The expression is a literal, a path, or a reference to or parenthesization of one	The shape of the expression
Local semantic		
`is_ref`	The expression's type is a reference	The type of the expression
`no_manual_impl_clone`	The type has no hand-written `Clone` impl in the crate	The impls of the type, excluding derives
`no_manual_impl_drop`	Neither the type nor, for a field access, any type it is a field of has a hand-written `Drop`	The impls of the type, walking up field accesses
`src_mutably_borrowed`	The call on the expression is itself under `&mut`, or the receiver of a `&mut self` method	The enclosing expression and the callee's signature
`no_outer_shadow`	No let-binding in the block shadows a local of the enclosing scope	Scope resolution at the block
`bindings_are_copy`	Every let-binding in the block binds a `Copy` type	The type of each binding
Flow-sensitive		
`last_use`	No later reference to the local, and the use is not in a loop, not captured by a closure or a non-`move` async block, and the local is not borrowed past it	The uses of the resolved local, in source order
`last_field_use`	As `last_use` for a field `x.f`, where `x` must own `f` and later uses of other fields do not count	The uses of the resolved local, in source order
`unused`	Nothing references the local after its statement, outside a loop and not borrowed past it	The uses of the resolved local
`known_true`, `known_false`	The boolean constant-folds to that value along the writes that reach it	Reaching writes, then constant evaluation
`known_none`	The last write reaching the `Option` is `None`	Reaching writes
`always_some`	Every write to the `Option` is `Some`, and it is never mutably borrowed	All writes to the local
`never_calls`	No use of the local is the receiver of the named method	The uses of the resolved local, descending into macros
Post-conditions		
`well_typed`	The rewritten crate compiles	The compiler, after the batch
`folded`	No identifier with the binding's name remains in the body	A token scan, after the peephole reaches a fixed point

Table 5.2: The available predicates for constructing conditions, grouped by what deciding each one requires. A *local* is the binding rust-analyzer resolves the metavariable's expression to; *reaching writes* are the assignments to that local that dominate the use on a straight-line path, with the analysis giving up wherever a loop, a conditional write, or a mutable borrow exists.

There are 15 predicates for pre-conditions, and they fall into three tiers. These can be read off the signatures of the functions that decide them: one takes only the expression, six take the expression and the semantic database, and eight take the whole function body as well. The first tier is syntactic, the second queries rust-analyzer for semantic information, and the third walks the body.

Table 5.2 lists them, with the two post-conditions below; every predicate takes only the metavariable it judges as an argument. The only exception to this is the `never_calls` predicate, which also takes a method name.

The evaluation of most of these predicates is not exact given that rust-analyzer's information is approximate. Two flow-sensitive predicates are not implemented as a dataflow analysis: `last_use` asks whether a later reference exists in source order and refuses inside loops, closures, and async blocks rather than reasoning about them, and the *reaching-write* predicates give up rather than guess. DAEDALUS strives for soundness, not completeness. The syntactic predicate is incomplete in the same direction: `side_effect_free` accepts a literal, a path, or a reference to one, and refuses every call—

not because a call is known to have an effect, but because nothing short of an analysis of the callee could say that it does not. It would be highly complex to prove a given expression is side-effect-free. A function could call another function, loops, recursion, or even FFI could be involved. If we were instead dealing with a simpler representation, such as the HIR, or MIR, it would be an easier task, as these representations are significantly simpler. For source code, this analysis is a considerable challenge in itself. Either way, in every such case the predicate is false, so the rewrite does not fire. Once again, we seek soundness, not completeness.

Assurance comes from the post-conditions instead, and it is paid for after the rewrite: `well_typed` is decided by the compiler, `folded` by DAEDALUS itself. This check runs `cargo check -message-format json`, whose errors carry spans in the candidate file on disk. Errors are positioned in the candidate file, whereas bodies are known by their position in the pristine one, and every earlier edit has shifted what follows by the difference between what it inserted and what it removed. Summing those differences over the edits before a body gives where the body now sits, and the body whose moved range overlaps an error is the one blamed. A check that cannot be carried out at all—`cargo` fails to run, or exits with an error but no diagnostics—is a third improbable, but possible, outcome, `Unverifiable`, which DAEDALUS treats as a failure at whichever level it occurs: a batch loses every tentative rewrite in it, and a round has every file it touched restored. A flag, `-keep-failures`, leaves a failing round on disk instead, for inspection during development.

The `folded` predicate is decided once the pass and its peephole have saturated. It holds for a binding when no identifier with that name occurs in the body anymore, which is the only sense of *folded* the engine needs: what a propagating rewrite introduced was consumed by the peephole, or it was not. A body whose post-condition fails is restored *in pleno* to a snapshot taken before the group began, and its rewrites no longer count toward the round's total. The unit of that revert is the body, not the rewrite: a rewrite that did not pay off is taken back, as Section 4.1.3 says, and everything the group did to that body goes with it. Of the two verdicts, one costs a compiler run and the other a name scan; the cheap one decides whether a rewrite was worth making, the dear one only whether it was well-formed.

5.5 Limitations

The design has limits of its own, apart from what the implementation has not yet reached, and they are worth stating together. First, everything is intra-procedural. The function body is the unit of scanning, of bundling, of blame, and of revert, so a rewrite that must change a signature and every call of it at once can neither be stated as a rule nor checked as one. Making DAEDALUS inter-procedural is doable and belongs to future plans, but it requires a non-trivial amount of effort. Second, a rule is a syntactic pattern. It matches shape, with metavariables agreeing by text, and cannot ask for an expression *equivalent* to

another or for one that merely *contains* a construct; what the rule set can say is bounded by what a pattern can spell.

The guarantees are also deliberately narrow. DAEDALUS preserves only compilation; behavior is left to whoever writes the rules, and the shipped rules pursue it with pre-conditions that are approximate by design. Moreover, in many cases, this approximation is rather conservative, which is sub-optimal; correctness, however, is never at stake. The guarantee is stated for one build configuration, the broadest that builds, and what that configuration does not build is left as it was. The front-end's types are not the compiler's, a price paid in Section 5.1 for a lossless, editable representation; the compiler has the last word where the two disagree. That last word has a cost: a compiler run per batch and one more per round, and when blame lands on the wrong body, the round is undone rather than kept on trust.

6

Evaluation

Contents

6.1 Experiment Setup 48
6.2 Microbenchmarks 48
6.3 Case Studies: WOFF2 and Brunsli 51
6.4 Optimizing the Top 1k Crates 55
6.5 Summary 57

One of the initial purposes of designing DAEDALUS was to remove a substantial share of the `Rc`s, `RefCell`s, and `Option`s that Cpp2Rust leaves behind. Removing these wrappers reduces runtime overhead and improves the readability of the output.

Three sets of benchmarks measure how much difference it makes: (1) a suite of microbenchmarks, translated from small C++ programs that exercise simple algorithms, covering array and matrix manipulations, recursion, and a binary search tree; (2) two real case studies that Cpp2Rust translated, WOFF2 and Brunsli, showcasing intricate C++ patterns such as abstract classes, inheritance, and lambdas; and (3) the 1,000 most downloaded crates on `crates.io`, all of them hand-written Rust. Together they show how much DAEDALUS can optimize, without ever breaking a build or a test. This is not a proof of DAEDALUS's correctness, but it is strong evidence of our tool's effectiveness.

6.1 Experiment Setup

We evaluated DAEDALUS on one machine: an AMD EPYC 9554P with 64 cores and 754 GB of RAM, running CentOS Stream 9. All programs were compiled with `rustc 1.98.1` and `clang 22`, both built on LLVM 22, so that both the C++ and Rust programs share the same set of compiler optimizations. We built the original, translated, and rewritten programs in release mode (`-O3` for C/C++), and we formatted all generated Rust with `rustfmt` so the reported code sizes are comparable. Cpp2Rust, pinned at commit `79a9acd1` on 21/09/2026, produced every translation so that the only variable between a measurement and its baseline is the rewriting.

For performance measurements, including the microbenchmarks and case studies, we ran each experiment 10 times and used the median to mitigate system noise. We obtained both the wall-clock time and peak memory usage for each run from the `/usr/bin/time` command.

For the crate study, we downloaded the 1,000 most downloaded crates on `crates.io` on 21/09/2026, at their latest published version that day. Each crate was given five minutes: DAEDALUS first selected the broadest build configuration under which the crate compiled, as Section 4.4 describes; it recorded any crate that did not compile or timed out and did not rewrite it. After rewriting, we rebuilt every crate under the same configuration, as Section 4.3 promises—that every program DAEDALUS rewrites compiles.

6.2 Microbenchmarks

The microbenchmark suite consists of ten small C++ programs, none longer than 70 lines. Each program focuses on a single way of using memory (i.e., passing by value, by reference, or by pointer), and part of the suite is built in pairs for that reason. Four programs have a `ref_` variant, which splits the same computation across functions that take arrays and accumulators by reference or by raw pointer. The `fibonacci` benchmark is a special case, appearing a third time as `ptr_fibonacci` so we can measure the two passing styles independently.

Cpp2Rust translates both reference and pointer parameters as `Ptr` (see Section 3.2). These variants let us see when and how DAEDALUS can recover performance lost to either boxed locals or pointers. The only benchmark without such a variant is the `bst`, since it already passes pointers by nature. Table 6.1 lists all the microbenchmarks and what they aim to exercise.

Each program is measured in seven configurations: the C++ original, compiled with `clang -O3`, as the reference at 1.0; the raw Cpp2Rust translation as the starting point DAEDALUS is asked to improve on; and five stages, each adding a pass to the ones before it—`rc`, `refcell`, `option`, `clone`, and `cast`, which is every pass DAEDALUS has. That order is the order in which the wrappers can be reached—since variables, in the worst case, arrive boxed as `Value<Option<Value<T>>>`—and each pass uncovers the type the next one matches on. This is also the order of decreasing effect, as moving a heap-allocated

Benchmark	What it exercises
`array_sum`	A heap array, boxed, written and read in a flat loop
`ref_array_sum`	The same array (and accumulator), passed to functions by reference
`matmul`	Three heap arrays, indexed in nested loops
`ref_matmul`	The same arrays, passed to functions by reference
`fibonacci`	Deep recursion over boxed scalars, passed by value
`ref_fibonacci`	The same recursion, with its accumulator passed by reference
`ptr_fibonacci`	The same recursion, with its accumulator passed by pointer
`prime`	Boxed scalars in a tight nested loop
`ref_prime`	The same loop, with its bound passed by constant reference
`bst`	Nodes owned by `unique_ptr`, linked by raw pointers

Table 6.1: The microbenchmark suite and what each benchmark aims to exercise.

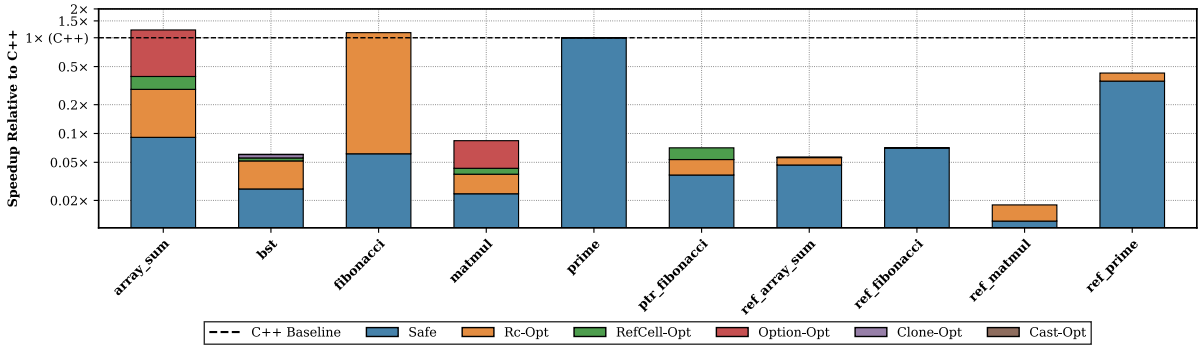


Figure 6.1: Speedup of each stage relative to the C++ original, on a logarithmic axis. The bottom segment of each bar is the untouched Cpp2Rust translation, labeled `safe`, and every segment above it is the ground the next stage gains; a bar therefore shows the best reached up to each stage, and a stage that leaves a program slower than the one before it adds no segment of its own. A bar reaching $1\times$ matches the speed of the C++ the program was translated from, and anything above it is faster.

object to the stack buys more than dropping a clone or a cast, which is why those two passes come last.

Figure 6.1 shows what each stage contributes, Table 6.2 reports the corresponding times, and Table 6.3 shows what each stage removes.

The initial translation runs between $1\times$ and $82\times$ slower than the original C++. `prime` matches its original exactly: the compiler can remove the cells outright, so there is nothing for DAEDALUS to recover but readability. `ref_matmul` sits at the other end, at $0.012\times$: every element access in its 512×512 multiplication goes through a `Ptr`, implying a cumbersome amount of additional run-time checks.

Notably, three programs return to the original C++ speed or surpass it. `fibonacci` does so at the first stage, moving from $0.06\times$ to $1.13\times$ as soon as `ptr` and `rc` unbox its scalars; `array_sum` climbs through every stage in turn, $0.09\times$ to $0.29\times$, then $0.39\times$, and finally $1.21\times$ once `option` removes the wrapper that stood between its two cells; and `prime` was never slowed down to begin with. `matmul` shows the same mechanism without reaching the same end: it goes 58.4 s, 36.5 s, 31.6 s and then 16.3 s, the last of those at the `option` stage, since until the `Option` is gone the `Rc<RefCell<Box<[f64]>>>` inside it is not a type any pass can match on.

`matmul` ends with no wrapper of any kind—every binding is a local, and the three matrices are a plain

Benchmark	C++	Cpp2Rust	rc	+refcell	+option	+clone	+cast
array_sum	1.99	21.83	6.89	5.04	1.65	1.64	1.64
ref_array_sum	1.98	42.42	35.53	35.45	35.27	35.27	35.48
matmul	1.37	58.45	36.56	31.64	16.30	16.28	16.29
ref_matmul	1.37	112.64	76.31	79.78	78.78	78.96	79.95
fibonacci	0.26	4.20	0.23	0.23	0.23	0.23	0.23
ref_fibonacci	0.47	6.67	6.64	6.65	6.64	6.62	6.65
ptr_fibonacci	0.47	12.61	9.01	6.64	6.62	6.64	6.64
prime	4.87	4.88	4.87	4.87	4.87	4.87	4.87
ref_prime	4.87	13.82	11.38	11.38	11.38	11.38	11.38
bst	0.71	27.60	13.89	12.88	13.00	11.90	11.90

Table 6.2: Median wall-clock time, in seconds, behind Figure 6.1. The `rc` column includes the `ptr` pass, and each subsequent column adds one pass to the ones on its left. Every configuration of every program was confirmed to produce the same output as the C++ original.

Benchmark	Value	Option
array_sum	14 of 14	1 of 1
ref_array_sum	19 of 25	1 of 4
matmul	32 of 32	3 of 3
ref_matmul	37 of 56	3 of 13
fibonacci	2 of 2	0 of 0
ref_fibonacci	0 of 6	0 of 0
ptr_fibonacci	2 of 8	0 of 0
prime	14 of 14	0 of 0
ref_prime	4 of 8	0 of 0
bst	26 of 50	2 of 2

Table 6.3: What DAEDALUS's full schedule of passes removes, against what the Cpp2Rust translation introduced: each cell reads *removed* of *present*. A `Value` is counted wherever the alias, or the `Rc<RefCell<T>>` it abbreviates, is written; `Rc` and `RefCell` share a column because every benchmark loses the same number of each. Clones and casts have their own passes but are omitted, since only the `bst` benchmark includes them.

`Box<[f64]>`—and still runs at $0.084\times$, twelve times slower than the C++ it was translated from. What remains is not the translation machinery but an innate property of Rust itself. The program contains three nested loops of array accesses, and safe Rust bounds-checks every index at runtime, whereas the C++ compiles to unchecked pointer arithmetic.

Whenever a program makes pervasive use of references (i.e., the `ref_` variants), performance is hardly recovered. Removing some `Rc`s and `RefCell`s yields some improvements, but they are very limited. This stems from the fact that DAEDALUS works intra-procedurally, so we cannot yet track a variable once it is passed to another function. This is also why we cannot optimize any of Cpp2Rust's `Ptr` constructs: they are created to be passed as function arguments, so we cannot simplify them in the current state of DAEDALUS.

Where no pointer is involved, the wrappers—`Rc`, `RefCell`, and `Option`—are entirely removed: `array_sum`, `fibonacci`, and `prime` have all of their wrappers removed, as does `matmul`. Where a pointer is involved, almost nothing moves—only boxed loop counters are unboxed. `ref_matmul` is left with 19 `Value` and 10 `Option`, `ref_prime` with half of its 8, and `ref_fibonacci` with all 6, being the one program nothing reaches. The reason is the same in each: a `Ptr` is a `Weak` into the very cell it points at, so the cell cannot

Benchmark	C++	Cpp2Rust	+ DAEDALUS	Removed
`array_sum`	15	41	38	3 (7%)
`ref_array_sum`	23	54	51	3 (6%)
`matmul`	29	88	76	12 (14%)
`ref_matmul`	42	137	129	8 (6%)
`fibonacci`	4	25	24	1 (4%)
`ref_fibonacci`	14	29	29	0 (0%)
`ptr_fibonacci`	14	30	29	1 (3%)
`prime`	22	41	39	2 (5%)
`ref_prime`	21	48	48	0 (0%)
`bst`	53	172	160	12 (7%)
Total	**237**	**665**	**623**	**42 (6%)**

Table 6.4: The number of lines of code for each microbenchmark: the C++ original, its Cpp2Rust translation, and the translation after DAEDALUS rewrote it. All generated Rust was formatted with `rustfmt`, so the counts are comparable.

be removed while a pointer to it survives, which is what the `never_calls` pre-condition checks before unwrapping a binding. The `bst` benchmark falls between the two, losing 26 of its 50: loop variables are unboxed, but nodes that are reachable through pointers cannot be.

A pointer is also where the rewrite stops being local. A binding whose address is passed to another function cannot be unboxed without changing that function's parameter type, and with it every other call of it. The unit of the rewrite is therefore the alias set rather than the binding, which is what Section 3.3 predicted of `Ptr`, and in `bst`, where the pointers are the data structure rather than a way of passing an argument, all 17 of them survive.

This is barely reflected in the code size. The rewritten programs are between 0 and 14% shorter (see Table 6.4) than the baseline Cpp2Rust translation, and still two to six times the length of the C++ originals—an innate property of Cpp2Rust. In translations of larger programs, like those in Section 6.3, DAEDALUS can remove more constructs, including unused functions, showing it is not only fit for improving performance.

6.3 Case Studies: WOFF2 and Brunsli

To evaluate DAEDALUS on real-world codebases of considerable size, we chose the two flagship translations Cpp2Rust presented in their paper [19]: WOFF2 and Brunsli. WOFF2 [30] is a font compression library, mainly used by browsers like Google Chrome. It compresses fonts in the `.ttf` format into its own compressed format, `.woff2`. WOFF2 is built in 4.7k lines of C++ code around abstract classes, inheritance, and STL containers, which is why Cpp2Rust decided to target it—a reasonably sized codebase, no multi-threading, and simple use of the STL.

Brunsli [31] is also a compression library, but it targets JPEG images instead. It is a lossless compression tool built in 9.9k lines of code, exercising what the first does not: lambdas, function pointers passed as arguments, multidimensional arrays whose address is taken, and pointer arithmetic through-

Configuration	Lines	Time (s)		Peak RSS (MiB)		Binary Size (KiB)	
		comp.	decomp.	comp.	decomp.	comp.	decomp.
C++ original	4,673	53.60	0.77	47.3	7.0	828	828
Cpp2Rust	13,621	58.35	2.99	45.1	8.0	2,038	1,322
Cpp2Rust + DAEDALUS	12,823	58.36	2.17	46.3	7.0	1,933	1,227

Table 6.5: WOFF2 compresses and then decompresses 117 TTF fonts, totaling 26.97 MiB, reducing them to 9.10 MiB and restoring them exactly. All three configurations emitted the same bytes—they are functionally equivalent. Times and memory are medians over ten repetitions. The line count covers the three generated binaries together: `woff2_compress`, `woff2_decompress`, and `woff2_info`. The binary sizes are those of the compression and decompression binaries.

out. The case studies are thus two non-trivial programs that Cpp2Rust has translated, and between them they cover the C++ that the microbenchmarks of Section 6.2 leave out, representing a good opportunity to test DAEDALUS in a real-world scenario to evaluate the difference it can make and decide on its next steps.

With Google's recent stance on Rust as a safe alternative to C++ [2], an engineer at Google also ported WOFF2 to Rust by hand and reported their experience [32]. That port is the comparison this work most wants to make. Instead of having engineers manually translate and rewrite a codebase, Cpp2Rust, together with DAEDALUS, can do it in a few minutes. Moreover, Google's engineer reported that this migration took 20 days and introduced 20 bugs. In contrast, a pipeline that automatically executes translation and performance-oriented rewriting takes much less time and introduces fewer bugs.

To evaluate DAEDALUS in these two case studies, we measure each program in three configurations: the original C++ program, the Cpp2Rust translation, and the translation after DAEDALUS's rewrites. We do not repeat the ablation in Section 6.2 here, since the question is what DAEDALUS achieves in a real-world program, not which pass contributes most. We measure WOFF2 by compressing and then decompressing a directory of TTF fonts, and we measure Brunsli by compressing and decompressing a directory of JPEG images. Table 6.5 summarizes the results.

The compression times remain largely unchanged. This is because much of the compression time is spent in `Brotli` [33], a library that both `WOFF2` and `Brotli` leverage. For decompression, DAEDALUS made the translation only $2.8\times$ slower, compared to the original $3.9\times$ that Cpp2Rust achieved. Compression is therefore dominated by code the translation did not produce and, consequently, DAEDALUS cannot reach, while decompression overhead can be measured and mitigated. We also observed little to no difference in peak memory usage during execution. In terms of binary size, both binaries shrink by around 5% while remaining more than twice the size of the C++ ones. This is expected because Rust programs require additional run-time checks and panic mechanisms. Finally, regarding code size, DAEDALUS eliminated 798 lines across all three WOFF2 binaries.

Table 6.6 showcases the results of the `Brunsli` case study. On Brunsli, both compression and decompression improve, and by a larger margin. Compression falls from 634 seconds to 419, a $1.51\times$ speedup. Decompression went from 479 to 289, a $1.66\times$ speedup. Both also start much further away

Configuration	Lines	Time (s)		Peak RSS (MiB)		Binary Size (KiB)	
		comp.	decomp.	comp.	decomp.	comp.	decomp.
C++ original	9,851	11.81	13.46	351	149	881	399
Cpp2Rust	34,926	634.28	479.07	4,557	161	3,233	2,815
Cpp2Rust + DAEDALUS	32,985	419.06	289.19	4,556	161	2,799	2,359

Table 6.6: Brunsli compresses and then decompresses 60 JPEG images, 75.17 MiB in total, reducing them to 64.59 MiB and restoring them byte for byte. Times and memory are medians over ten repetitions. The line count covers the two generated binaries, `cbrunsli` and `dbrunsli`. The binary sizes are those of the compression and decompression binaries.

Construct	WOFF2			Brunsli		
	before	after	removed	before	after	removed
`Rc`	1,452	721	731 (50%)	3,853	2,069	1,784 (46%)
`RefCell`	1,452	789	663 (46%)	3,853	2,196	1,657 (43%)
`Option<T>`	5	5	0 (0%)	24	24	0 (0%)
`Ptr<T>`	1,316	1,040	276 (21%)	3,008	2,439	569 (19%)
`borrow/borrow_mut`	4,272	2,213	2,059 (48%)	11,658	6,136	5,522 (47%)
`clone`	693	549	144 (21%)	1,317	1,021	296 (22%)

Table 6.7: Wrappers and accesses in the generated source of each program, before and after DAEDALUS's rewrites.

from the original. The Cpp2Rust translation is about $53\times$ slower than the original C++ program for compression and $35\times$ for decompression; the rewrites bring these numbers down to $35\times$ and $21\times$, respectively.

Brunsli suffers a more acute performance degradation than WOFF2. Not because of how much wrapping the translation carries, but because Brunsli's work loops over the coefficients of every block of every image, so each wrapper that survives is paid once per coefficient, while WOFF2's per-table and per-glyph access is much more efficient. Peak memory also degrades in the translation—about $13\times$—and DAEDALUS cannot recover it.

Table 6.7 summarises what DAEDALUS could remove in both of the case studies. In both cases, DAEDALUS removes close to half of the occurrences: 50% of `Rc` and 46% of `RefCell` wrappers in WOFF2, and 46% and 43%, respectively, in Brunsli. As a result, roughly half of `.borrow()` and `.borrow_mut()` are removed, meaning fewer additional run-time checks. Although introduced together by Cpp2Rust, `Rc` and `RefCell` are not necessarily removed together. Half of WOFF2's `Rc` go against 46% of its `RefCell`; this gap is smaller on Brunsli. This is because each pass judges its own wrapper, so a binding can lose its reference-count mechanism while keeping interior mutability. This is still a win for DAEDALUS: although the shared ownership is kept, and with it the additional run-time checks, a heap allocation was transferred back to the stack.

Although the Cpp2Rust `Ptr<T>` construct is used inter-procedurally, and thus is outside the scope of DAEDALUS, we could still remove some of them—21% in WOFF2 and 19% in Brunsli. This is because the `Ptr` is used in a very specific manner: a `Value<T>` calls `.as_pointer()`, creating a `Ptr`, calls `.read()` on it immediately, and drops it right after. DAEDALUS detects these cases and is able to transform the `a.as_pointer().read()` into an `a.borrow()`, which the `rc` and `refcell` passes can later detect and

remove. Although DAEDALUS can remove these situations, we expect Cpp2Rust to detect them and not emit them. During the development of DAEDALUS some of these irregularities were detected, and subsequently addressed through commits to Cpp2Rust.[1,2]

The `Option` wrapper is never removed in either program. Dropping an `Option` means rewriting every assignment to the binding as well, and the owners here that begin as `None` are later assigned from a call that returns an `Option` without constructing one in place—`Ptr::alloc_array(..).to_owned_opt()` in WOFF2's case. As DAEDALUS can rewrite `Option` only when it statically knows its value at all points in its lifetime, it refrains from deleting any of them.

As for DAEDALUS's execution time, it took two and a half minutes on WOFF2, reaching a fixed point in four rounds, and thirteen minutes on Brunsli, which needed ten. Notably, neither program was tuned for: the schedule, the rules, and their pre-conditions are the same as in every other measurement in this chapter. One of DAEDALUS's objectives is to be easily usable in any Rust project, without any adjustments.

Compared to the microbenchmarks, where DAEDALUS took a few seconds to optimize them, it is natural to question the tool's performance, since that also plays a big part in its applicability and overall ease of use. This is because programs translated by Cpp2Rust are condensed into a single file—each of Brunsli's binaries is over ten thousand lines of code—which makes `rust-analyzer`'s job harder, as it cannot parallelize its analyses. Moreover, when DAEDALUS leverages Cargo for sanity checks or its `well_typed` post-condition, it pays this toll twice.

Because Cpp2Rust merges everything into a single binary, sometimes functions are translated when they do not need to be. Normally, Clippy or even the Rust compiler would catch this, but in this case they do not. That is because all functions translated by Cpp2Rust are marked as `pub`, so no tool can be sure they are unused.

To bridge this gap, DAEDALUS has a small flag `-delete-unused-functions`. With the flag enabled, DAEDALUS removed 54 of WOFF2's 418 functions, accounting for 2,055 lines. This is a 15% reduction in code size. In Brunsli, we removed 53 of the total 844 functions, representing a 5% reduction in code size.

The hand-written WOFF2 migration [32] is 6.2k lines of Rust, compared with the original's 4.7k. Cpp2Rust and DAEDALUS result in 12.8k lines of code, close to three times the original and twice the size of the port. The difference is the cost of producing them. The port was written over weeks, and its author reports 20 bugs introduced in the process; Cpp2Rust and DAEDALUS together took under five minutes. This pipeline does not replace manual intervention completely, but it immediately yields a starting point that developers can refine. In addition, and contrary to modern LLM practices, this is a deterministic approach, leaving the door open to formalizing translation and refinement later.

[1] https://github.com/Cpp2Rust/cpp2rust/commit/8d33dbff3a9273ba45d001c0dcf52af116968084
[2] https://github.com/Cpp2Rust/cpp2rust/commit/c2ce47478d4f802c480847a54384021dfd66d6a0

Stage	Crates
Fetched	1,000
Compiled under some configuration	978
Rewritten at least once	133
Compiled after rewriting	133

Table 6.8: The 1,000 most downloaded crates on `crates.io`, at each stage of the study. We did not rewrite any crate that did not compile under any configuration. Every surviving rewrite has been accepted by the compiler, so the last row is living proof that DAEDALUS never kept an uncompilable rewrite, confirming the promise of our design.

6.4 Optimizing the Top 1k Crates

To avoid limiting DAEDALUS's evaluation to microbenchmarks and Cpp2Rust-translated programs, we ran our tool on the 1,000 most downloaded Rust crates. Contrary to our case studies, these are hand-written Rust. They are therefore the case where DAEDALUS should find next to nothing: the wrappers the `rc`, `refcell`, and `option` passes look for are artifacts of the Cpp2Rust ownership model, and a human writing Rust has no reason to introduce them. Of DAEDALUS's available passes, only `clone` and `cast` remain applicable, since redundant clones and casts are not uncommon. This part of the evaluation therefore does not quantify how much performance our tool can recover. Its objectives are twofold: on one hand, we want to attest whether we can find any redundancies in real-world code; on the other hand, we want to be as sure as possible that our tool never damages a crate (i.e., leaves it in a non-compilable state).[3]

Table 6.8 summarizes the main findings of this study. Of the thousand crates, 22 of them did not compile under any configuration tried by DAEDALUS. As such, we did not rewrite them. These were due to differences in architecture and dependencies on external libraries; problems no rewriter tool could circumvent. The drop-off from 978 *rewritable* crates to 133 *rewritten* ones is expected. Given that these are long-standing open-source projects and among the most-used Rust code in the world, we did not expect many redundancies; still, we found rewrites in 14% of the analyzed crates.

Figure 6.2 shows how many rewrites we found across our corpus of crates. The 5,794 rewrites are distributed unevenly: nearly two-thirds of the crates that were touched at all received one or two rewrites, and four crates account for 89% of the total rewrites. Those four are worth naming, because none of them is really hand-written. `fiat-crypto` alone takes 3,945 of the 4,639 casts, and it is machine-generated field arithmetic, emitted by a compiler from a formal specification and full of casts between fixed-width integers that are already the right width. `lalrpop` supplies 927 of the 1,155 clones, and it is a parser generator whose own tables are generated. `jni` and `p384` contribute 144 and 138 casts respectively, with the latter being generated elliptic-curve arithmetic of the same kind as `fiat-crypto`.

This study corresponds to DAEDALUS analyzing 9.5 million lines of hand-written Rust code and find-

[3] We argue that, although not comparable to a formalized version of DAEDALUS, guaranteeing that the tool never breaks any code in the most downloaded crates is a good approximation.

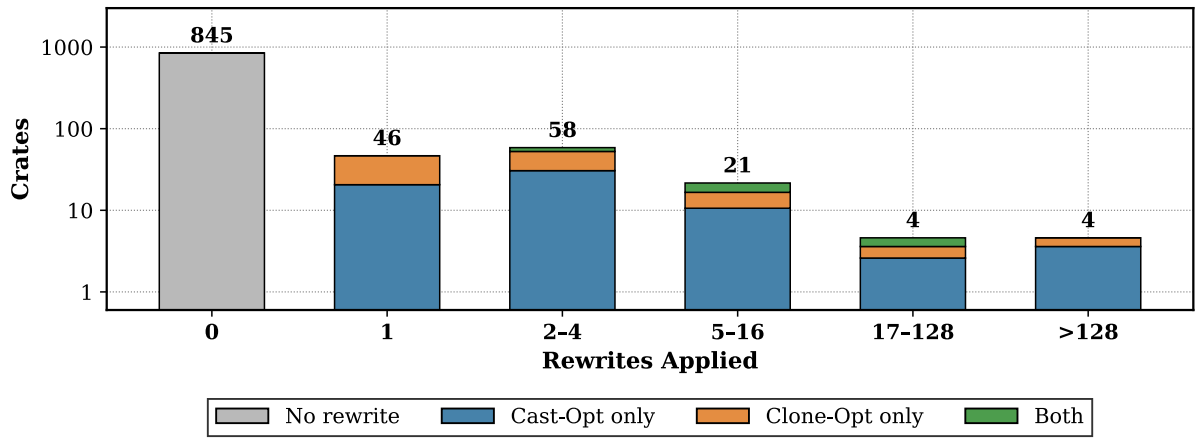


Figure 6.2: Rewrites applied per crate, over the 978 that compiled. The left axis is logarithmic. The leftmost bucket, crates with no rewrites, holds most of them.

ing hundreds of redundant clones and casts. Notably, no crate failed to compile after being rewritten. That is not unexpected: as Section 4.4 explains, a rewrite is kept only once the compiler accepts the file it produced, so a surviving rewrite that breaks the build is a contradiction in terms rather than an unlikely event.

What DAEDALUS found here was not left in the evaluation. 55 pull requests were opened against 52 projects, carrying 265 rewrites in total: 99 from the `clone` pass and 166 from the `cast` pass. Thirty-nine have been merged, accounting for 113 of those rewrites, among them changes to `rust-lang/libc`, `servo/rust-url`, `mozilla/cbindgen`, `google/zerocopy`, `PyO3/pyo3`, `rustls/rustls`, and `seanmonstar/reqwest`.

A merged pull request is a strong result. We show not only that DAEDALUS finds rewrites, but also, through their maintainers, that those rewrites make sense and are welcome. A maintainer's acceptance of the change indicates that someone with in-depth knowledge of the codebase considers it an improvement and is willing to maintain it. The fact that 39 were merged across projects maintained by the Rust Project, Mozilla, Google, and a number of independent authors shows that DAEDALUS is usable on any available Rust project. In four of the submitted pull requests, the discussion turned from the change in front of the maintainer to the tool that had produced it—many were interested in DAEDALUS itself, showing that it indeed has an impact on the community: `biodivine-algo-smt-inference`,[4] `zlib-rs`,[5] `rust_radix_trie`,[6] and `x11rb`.[7]

This study took two hours and eight minutes for 9.5 million lines of code, with a median of six seconds per crate, 967 of the 978 finishing within thirty seconds, and only two taking longer than a minute. That time depends on the crate size, not the number of rewrites applied: `fiat-crypto` took 24 seconds for

[4] https://github.com/sybila/biodivine-algo-smt-inference/pull/9
[5] https://github.com/trifectatechfoundation/zlib-rs/pull/549
[6] https://github.com/michaelsproul/rust_radix_trie/pull/81
[7] https://github.com/psychon/x11rb/pull/980

its 3,945 rewrites, while `parquet` took 80 seconds for none at all. This is what Section 4.4 describes. Loading a crate into `rust-analyzer` and resolving its types costs whether or not a rule ever matches; what varies on top of that is the number of check rounds, since each round costs one compiler invocation, however many rewrites it carries.

6.5 Summary

DAEDALUS removes about half of what Cpp2Rust's ownership model costs: 50% of WOFF2's `Rc`s and 48% of its borrows, 46 and 47 of Brunsli's, respectively. The two programs share their source: both originate from a translation made with Cpp2Rust. This lets us evaluate how much DAEDALUS can recover, in both performance and readability, from what Cpp2Rust introduced. Time-wise, this is worth a third of WOFF2's decompression overhead, and a third of Brunsli's in both directions.

The microbenchmarks show why, and the other half of the picture: when no pointer is involved, every wrapper comes off, and `array_sum`, `fibonacci`, and `prime` end at or past the speed of the C++ they were translated from. Their `ref_` variants differ only in passing those same values to other functions, and keep almost everything. What survives is what a `Ptr` reaches, and a `Ptr` cannot be taken off one function at a time. We thus show that, whenever the wrappers can be deleted, we can match the original C++ performance, or even surpass it, and that removing the custom `Ptr` types will finally bridge the performance gap Cpp2Rust introduced.

Regarding real-world Rust code, our tool never silently introduces an unsound rewrite and is never harmful. Of the 978 crates that compiled from the outset, 133 had redundant operations that DAEDALUS caught. The findings were submitted as 55 pull requests, 39 of which were merged, including in repositories of important organizations such as the Rust Project, Mozilla, and Google.

7

Related Work

Contents

7.1 Source-to-Source Translation 60

7.2 Refining a Translation 64

7.3 Rule-Based Rewriting 68

7.4 Summary 70

Cpp2Rust and DAEDALUS are not the only tools working on the translation of C/C++ into safe Rust. Research on migrating from unsafe to safe languages, and on refining the resulting translations for safety or speed, has grown accordingly.

Three lines of work are important for DAEDALUS. The translators decide what the code will look like in the new language. The refinement tools are the nearest neighbors, and the comparison that matters is not what each removes but how they decide whether a rewrite is safe. The rule-based systems share with DAEDALUS the decision to state the rewrites as rules, so it is interesting to see how they differ and why. One thing, however, is absent from all three lines of work. No prior tool combines rewrites written in the host language with a compiler that has the last word on every rewrite; each line has one of the two, never both.

7.1 Source-to-Source Translation

As programming languages evolve, software systems often require migration to new technologies. Although common, the process is complex, time-consuming and error-prone: developers must understand both the source and the target language, and semantic differences between them can make a faithful translation difficult, introducing subtle bugs. Functional equivalence—the same output for the same input—is the essential requirement, and yet a translation may preserve it while still creating semantic mismatches, such as stack versus heap allocation, that affect security and performance. The tools below differ in how much of that they promise, and in what they ask of the programmer to deliver it.

COBOL to Java. Modernizing legacy systems, particularly migrating from COBOL to Java, has been a major industry challenge since the early 2000s [12]. Many large-scale efforts spanned years and some failed entirely [34]; the successful ones revealed that the primary difficulty lies less in performing the translation than in demonstrating its correctness, that is, in establishing functional equivalence. Some migrations remain extremely costly: the Commonwealth Bank of Australia spent over 750 million dollars in 2012 to replace its COBOL-based platform [35]. Several reports find that an initial abstraction phase, followed by incremental modernisation, improves outcomes [36]; even so, tightly coupled, hardware-specific code often prevents full conversion, and the promised gains in cost and maintainability are frequently only partly achieved. To reduce costs, some organisations have used automated COBOL-to-Java translators [37], but the generated code is typically too generic and prone to performance problems [13]. LLM-based translation has shown recent progress, without formal guarantees [38], and across both kinds of tool there is a growing focus on automated validation [39] to build confidence in the result.

Hardening C/C++ programs. C and C++ are unsafe languages: memory vulnerabilities can be introduced inadvertently, and the languages define hundreds of cases of undefined behaviour [40]. Several attempts have been made to add partial memory safety to C/C++ programs while balancing run-time overhead against the impact on developers' productivity.

Cyclone [41] is a safe dialect of C designed to prevent buffer overflows, format-string attacks and memory-management errors. Its compiler enforces non-null and initialised pointers, restricts unsafe casts and pointer arithmetic, and rejects programs it cannot verify statically, inserting checks where necessary. To support controlled pointer arithmetic it introduces “fat" pointers, which carry bounds information and enable run-time bounds checks, and never-null pointers that eliminate redundant null checks; a region-based memory-management system, supplemented by an optional garbage collector, prevents dangling pointers. Cyclone's adoption cost remains high, since programs must be rewritten by hand to satisfy its stricter type system, which makes migration from legacy C cumbersome.

CCured [42] retrofits memory safety onto existing C programs by classifying pointers into three categories: SAFE, which forbid pointer arithmetic and need only null checks; SEQ, which support arithmetic and carry bounds information; and WILD, which arise from unverifiable casts and need full run-time checking. Pointer types are inferred by generating constraints from pointer operations, followed by a linear-time algorithm that minimises the use of WILD pointers, then SEQ, assigning SAFE wherever possible. This converts legacy C automatically into a memory-safe variant, at a 5–90% performance overhead from the inserted checks.

CheckedC [43], an extension of C developed at Microsoft, improves spatial memory safety while remaining backwards compatible with existing C. It adds two checked pointer types, `_Ptr<T>`, for a pointer that is only dereferenced and never subject to arithmetic, and `_Array_ptr<T>`, which supports arithmetic if a bounds declaration is provided in the type; array types are checked by prefixing their dimensions with `_Checked`. A CheckedC program is expected to mix *checked* and *unchecked* regions, but a block or function designated as checked carries stronger restrictions on unchecked pointers, since those could corrupt already-checked ones through aliases. Listing 7.1 shows an unsafe function and a partial rewrite of it in CheckedC.

C

```c
void read_n(int *b, int idx, int *out) {
    int tmp = *(b+idx);
    *out = tmp;
}
```

CheckedC

```c
void read_n(int *b, int idx, _Ptr<int> out) {
    int tmp = *(b+idx);
    *out = tmp;
}
```

Listing 7.1: A C function (left) and a partial rewrite in CheckedC (right): the output pointer becomes a checked `_Ptr`, while `b`, which is subject to arithmetic, stays unchecked until a bounds declaration can be supplied.

CheckedC imposes several restrictions: checked pointers, and aggregates containing them, must be initialized at declaration; heap objects involving checked pointers require zero-initialization through `calloc`; and taking the address of a variable or member involved in a bounds expression is forbidden. Within those limits, the dialect has a small run-time overhead (under 20%) and allows code to be ported from legacy C incrementally. To make that porting smoother, 3C [44], a semi-automated process for converting legacy C to CheckedC, achieves a 70% conversion rate for pointer annotations. It runs two static analyses: `typ3c`, which determines which legacy pointers can be made checked, and `boun3c`, which infers the bounds of the array varieties. `typ3c` assigns a qualifier variable to each indirection level of a pointer and generates constraints $x \sqsubseteq y$ whose solution must respect the order $chk \sqsubseteq wild$. Its key feature is localized wildness: rather than letting an unsafe use in one function propagate wildness to its callers, it models each parameter with two nodes in the flow graph, an internal one for uses within the body and an external one for interactions at call sites, with call edges directed from the external parameter node to the argument node—the reverse of the data flow—so that callers and callees are made independent.

A parameter whose nodes are both *chk* receives a checked type; one whose external node is *chk* but internal node *wild* receives an interop type, which allows checked arguments at call sites despite internal unsafety and defers the refactoring. A later stage settles the exact checked type, and `boun3c` analyses indexing patterns, pointer arithmetic, and control flow to insert bounds declarations, emitting diagnostics to guide manual work where a bound cannot be inferred soundly. That localization—confining a hard case to the function it occurs in so that the rest of the program can still be converted—is the same decision Cpp2Rust makes with its parameter preamble (Section 3.1) and DAEDALUS inherits from it.

Fil-C [45] takes the opposite approach: it changes nothing in the source code and pays for everything at runtime. It is a fork of Clang that compiles unmodified C and C++ into a memory-safe program, and it has built a complete Linux user space, with modifications only to the most complex programs. Every pointer carries a capability, stored in an auxiliary allocation so that the pointer itself retains its native size and C programs that manipulate address bits continue to work. Every potentially unsafe operation checks this capability, so spatial violations are detected at the point of access. A concurrent, precise garbage collector allows `free` to atomically invalidate every capability associated with the freed memory, causing use-after-free, double-free, and invalid-free errors to panic rather than corrupt memory. The cost is that most programs run about four times slower, and that Fil-C objects have an ABI of their own and cannot link against ordinary compiled code. Of all the tools here, it is the closest in spirit to Cpp2Rust's model, since both pursue the idea of memory safety through new run-time checks that turn the original's bugs into panics; the difference is where the program ends up. In Fil-C, the checks end up in C, and the "safer" program now depends on Fil-C's compiler, whereas in Cpp2Rust, safety comes from translation to a different programming language whose compiler ensures it. A Fil-C program is therefore as slow on its last day as on its first. Whilst a Cpp2Rust program can shift these added run-time checks to compile time, Fil-C cannot, leaving an irreparable performance overhead.

C to Rust. Although recent developments have retrofitted dynamically typed languages with a region-based ownership model [46], Rust remains the only language to implement a memory-safe model and see consistent industrial use. Automatic translation is therefore an emerging area, enabling a simpler transition. C2Rust [16] is the most mature tool for translating full C programs into Rust while preserving their semantics, converting complex C features such as pointers, structs, and memory-management APIs into their Rust counterparts; its evaluation translated real systems such as SQLite, Lua, and curl with all of their original test suites passing. Since many C constructs lack a direct counterpart in Rust's ownership model, C2Rust's output relies heavily on `unsafe` blocks to maintain functional equivalence. It does not reach a safe counterpart, but it is a first step: developers can refine the unsafe regions iteratively, and the refinement happens entirely on the generated Rust, with no need to revisit the C code or be an expert in both languages. Listing 7.2 shows an example translation.

Original C

```c
int sum(int *arr, int len) {
    int s = 0;
    for (int i = 0; i < len; i++) {
        s += arr[i];
    }
    return s;
}
```

C2Rust

```rust
pub unsafe extern "C" fn sum(
    mut arr: *mut libc::c_int,
    mut len: libc::c_int,
) -> libc::c_int {
    let mut s: libc::c_int = 0 as libc::c_int;
    let mut i: libc::c_int = 0 as libc::c_int;
    while i < len {
        s += *arr.offset(i as isize);
        i += 1;
        i;
    }
    return s;
}
```

Listing 7.2: A C function and C2Rust's translation of it: valid Rust that compiles, but unsafe, dependent on `libc`, and far from idiomatic.

The generated code compiles, but it is unsafe, dependent on `libc`, and far from idiomatic, which defeats much of the purpose of translating from an unsafe language to a safe one; what it offers is a working Rust base to start from. Other tools attempt the same transpilation. Corrode [47] targets Rust that is ABI-compatible with the original and usable as a drop-in replacement; it produces cleaner code than C2Rust but does not preserve semantics, leaving it a proof of concept. Citrus [48] translates C syntax into readable Rust with no promise that the result compiles or preserves the original's behavior; it can serve as a starting point for a manual refactoring. Cpp2Rust [19], described in Chapter 3, is the one translator whose output is safe from the outset, and it pays for that with the run-time constructs this thesis removes.

LLM-based translation. Large language models (LLMs) have emerged as an alternative to rule-based transpilers. Where a rule-based approach provides human-readable transformation rules that can be proved correct, a model generates code from statistical patterns learned over large datasets, and so often produces more idiomatic code than a deterministic mapping does—without any formal guarantee that the result is semantically equivalent to the original. Models do output incorrect or inconsistent code [49], and idiomatic translations that look right but alter the program's behavior [15]; their use in critical or large-scale applications remains limited by the absence of guarantees. The systems built on them differ in how they recover one.

VERT [50] compiles the source program to WebAssembly, which makes it independent of the source language, then lifts the binary to Rust with rWasm [51], yielding a correct-by-construction but unreadable translation. The lifted code is passed to a model, which proposes an idiomatic version that may introduce syntactic or semantic errors, and VERT enters a repair-and-verify loop—hand-written rules and re-prompting, then compilation and an equivalence check against the Wasm reference—until it produces a verified Rust program. RustAssistant [52] drives a model with compiler diagnostics, and op-

tionally Clippy, to fix compilation errors in existing Rust; Fluorine [53] extends the same prompting loop to translation from C or Go, and once a candidate compiles, runs I/O-based behavioral tests against the original, re-prompting on any divergence. As its authors note, the stochastic nature of next-token prediction limits translation of large programs in one pass: if each token carries an error probability e, the probability of n correct tokens is $(1-e)^n$, which vanishes as n grows. C2SaferRust [54] combines C2Rust with a model, lifting the C with C2Rust and then decomposing the unsafe Rust into slices rewritten independently to shorten the model's input and reduce its error rate; it relies on end-to-end functional tests and cannot rule out semantic divergence introduced by the model. Syzygy [55] translates incrementally in dependency order, with dynamic-analysis-guided test generation validating each unit before attempting its dependants.

Overall, existing source-to-source translation approaches reveal a trade-off between semantic preservation, safety, and code quality. Rule-based tools provide strong correctness guarantees but often yield unidiomatic or inefficient code, while LLM-based methods improve readability and are much easier to develop, but at the cost of formal assurance. Table 7.1 summarizes these techniques. In the context of C/C++ to Rust, this motivates a pipeline like Cpp2Rust's that guarantees safety and correctness by construction and permits refinement afterward without reintroducing unsoundness—which is exactly the refinement DAEDALUS performs.

Tool	Objective	Technique
Citrus	C syntax to readable Rust	Syntax-only translation; no guarantee of compilation or correctness
C2Rust	Semantics-preserving C to Rust	AST lifting, raw pointers, unsafe blocks, libc mappings
Cpp2Rust	C/C++ to safe Rust	Defers ownership and mutability checks to run time
VERT	LLM-assisted Rust translation	LLM refinement loop over a Wasm lifting
Fluorine	C/Go to Rust with functional guarantees	Compiler-driven LLM repair loop with I/O behavioural tests
C2SaferRust	Safer Rust from C2Rust's output	Program decomposition for LLM rewriting
Syzygy	Correctness-preserving incremental translation	Dependency-ordered translation and dynamic test generation

Table 7.1: The leading tools for source-to-source translation into Rust. The highlighted tool, Cpp2Rust, is the main target of the rewrites that DAEDALUS produces.

7.2 Refining a Translation

Prior work on C-to-Rust translation has focused mainly on improving the safety of C2Rust's output, reflecting the core motivation for migrating from C/C++. Because C2Rust provides a mature baseline, many approaches build on it, applying automated transformations to improve memory safety and idiomatic quality through structured post-translation analyses, large language models, or both. DAEDALUS starts from an already safe translation and refines it for performance instead, but these efforts are the closest work to it, and the comparison that matters is how each decides a rewrite is safe.

Laertes. An empirical investigation into the challenges of automatically translating C programs into safer Rust [17] analyzed the sources of memory unsafety that emerge after an initial translation step and classified the most common patterns. Its central finding is that a large fraction of these issues stems from the pervasive use of raw pointers: C2Rust maps every C pointer to a Rust raw pointer, so every dereference is unsafe, whereas in idiomatic Rust, ownership-checked references are the norm. Beyond characterizing these patterns, the authors propose a tool, *Laertes*, that automatically replaces selected raw pointers with safe references annotated with explicit lifetimes, thereby improving the safety of C2Rust's output.

The transformation process relies on iteratively querying the Rust compiler, using the emitted errors to infer ownership and lifetime information for raw pointers whose unsafety arises solely from missing aliasing metadata—rather than from inherently unsafe constructs such as pointer arithmetic or unchecked casts. While, in theory, this information could be recovered using a static analysis on either the original C program or the translated Rust code, the authors argue that such analyses would be complex and redundant, given that the borrow checker already encodes the required reasoning, whilst an in-house mechanism would need to be perfectly tuned to be no less precise than the Rust compiler. That is the argument of Section 4.3, and *Laertes* most closely resembles DAEDALUS in how it discharges this responsibility to the compiler; the difference lies in what it asks of the compiler. For *Laertes*, it is the analysis, consulted once per candidate pointer; for DAEDALUS, it confirms only an approximate analysis that ran first, batched across every function of a file. Although *Laertes* does introduce additional memory-management mechanisms (e.g., reference counting), the authors note that doing so could enlarge the set of pointers that can be migrated safely, aligning with the direction explored in Cpp2Rust.

CRustS. CRustS [56] is a fully automated source-to-source transformation that aims to reduce the use of the `unsafe` keyword in translated programs, on the premise that reducing explicit unsafety increases the share of code that passes Rust's checks and thus reduces the likelihood of memory vulnerabilities. Rather than repairing the unsafe constructs, it eliminates unnecessary `unsafe` annotations; one may argue that this merely shifts unsound operations across function boundaries, but it meaningfully reduces the code that needs manual auditing. It applies 220 TXL-based [57] rules over the Rust syntax tree, targeting unsafe function qualifiers and blocks while leaving traits and their implementations unchanged, and removes at least 95% of the reported unsafe constructs on its six benchmarks. The analysis does not establish whether the transformed code is genuinely safer or whether it merely obscures unsafety; it also reminds us that starting from an unsafe translation can hinder later safety transformations. This is, however, the only prior tool whose rewrites are rules over source text rather than a hand-written analysis or interaction with an LLM; this rule-based approach is the same one DAEDALUS follows.

The limits of aliasing. A follow-up study by *Laertes*' authors [18] examined the aliasing constraints that limit how far C can be translated automatically into safe Rust—concretely, how much safer C2Rust's output can be made. In the earlier work, many raw pointers were unsafe for reasons unrelated to ownership or lifetimes, such as pointer arithmetic or unchecked casts, and those confounding factors kept *Laertes* from analyzing the aliasing structure of most pointers: only about 12% could be rewritten as references. To determine whether a given pointer could be modeled as a safe reference, the study introduced the notion of *pseudo-safety*, in which unsafe pointer operations are replaced by constructs that preserve the static properties relevant to Rust's type and borrow checks while disregarding run-time behavior. An expression e is rewritten as e' with the same lifetime, using the same objects as e and pointing to the same abstract region; for example, `arr.offset(i)` becomes `{i; arr}`. The goal is not a correct transformation but removing extraneous causes of unsafety so ownership and lifetime inference can be studied alone; under it, 75% more raw pointers could have been translated safely.

The authors also investigated why their earlier system, *Laertes*, could not transform such a substantial portion of raw pointers. The core limitation stems from its equality-based interpretation of Rust's type checker: raw pointers are grouped into equivalence classes, and if any pointer in a class is marked unsafe, all members of that class must also be marked unsafe. To test this hypothesis, analyses were built on the SVF framework [58] over LLVM IR, compiled with optimizations and safety checks disabled to preserve the original aliasing structure, constructing four increasingly precise data-flow graphs, from field-based and context-insensitive to one whose nodes encode call contexts. The degree distribution of unsafe pointers across them quantifies how unsafety propagates and which pointers taint the most others, and it supports the hypothesis: unsafety spreading, rather than ownership limits alone, accounts for much of *Laertes*' restricted effectiveness. That finding is why Section 3.2 treats `Ptr<T>` as a single construct with a known cost rather than as a raw pointer: Cpp2Rust has already paid for arithmetic and casts at run time, so what remains for a rewriter is exactly the aliasing question Laertes could not separate out.

Crown. With the same objective of converting raw pointers into safe reference types, Crown [59] takes a different path from *Laertes*: instead of depending on the compiler as an oracle, it performs its own static analyses on Rust's MIR to infer ownership for code translated by C2Rust, and retypes pointers accordingly. A pointer that owns its object at any point becomes a `Box`, wrapped in an `Option` where it may be null; an output parameter `*mut T` becomes `Option<&mut T>`; and after retyping, every pointer use is rewritten to dereference through safe references, eliminating the `unsafe` blocks the translation introduced. The inference is a flow- and field-sensitive analysis that builds ownership constraints over access paths—memory locations reached by traversing fields from a variable, such as `(*list).head`—each associated with a boolean ownership variable, and the system is solved with Z3 [60].

The central difficulty is aliasing, and Listing 7.3 illustrates it: if the comments describe the intended state, line 4 requires `(*p).next` to take ownership of the object `r` points to, an ownership transfer; but if `(*p).next` and `(*q).next` may alias before that line, both access paths would have to be owning, which contradicts Rust's requirement of a unique owner. Computing sufficiently precise aliasing for user-defined data structures proved infeasible, and Crown breaks the impasse with a strengthening assumption, *ownership monotonicity*: ownership may only decrease along an access path, so that if a is a prefix of b then $O_a \geq O_b$. Applied to the example, if `(*p).next` becomes owning then `p` must already be, and every alias of `p` must therefore be non-owning, which propagates to the aliases of `(*p).next` and `(*q).next` and removes the ambiguity. The assumption is stricter than Rust's own rules and rejects programs Rust accepts; the authors identify the two most common valid exceptions, local borrows and reference-typed parameters, and admit them explicitly.

```
1  let p, r, q : *mut Node;
2  // p and (*p).next non-owning;
3  // q, (*q).next and r owning;
4  (*p).next = r;
5  // (*p).next must have ownership
```

Listing 7.3: The aliasing problem in Crown's ownership inference: the assignment on line 4 transfers ownership to `(*p).next`, which is only consistent if nothing that may alias it is owning.

The constraints come from a suite of intra- and inter-procedural rules that encode how ownership flows through assignments, calls, and memory-management operations, and a central question is when an assignment transfers ownership along every access path reachable from the assigned variables. For `p = q` with both of type `Box<Box<i32>>`, ownership of the object `q` references passes to `p`, and since ownership must be unique, subsequent uses of `q` and of any path rooted at it are disallowed, whereas `*p = *q` transfers nothing at the level of `p` and `q` themselves. The assignment rule turns on one equality, $O_{a'} + O_{b'} = O_b$, over the access paths a and b reachable from the two sides: if a transfer occurs, every path rooted at `p` takes the ownership its counterpart under `q` held, and if none does, both sides are unchanged; paths with shorter prefixes than the assigned variables keep their ownership either way. The rule carries the pre-condition $O_a = 0$, requiring the target's paths to be non-owning beforehand; without it, the translation could silently repair a memory leak that C permits, but Rust does not, breaking the correspondence between source and translation. Analogous rules cover `malloc`, `calloc`, `free`, and `memset`. Crown retyped 37% of the raw pointers in C2Rust's output, and its approach is directly relevant to retyping `Ptr<T>` in Cpp2Rust's. Crown and DAEDALUS draw the same line in different places: Crown makes its analysis sound by strengthening its assumptions, and accepts that the assumptions exclude valid programs; DAEDALUS lets its analysis be unsound, and accepts that the compiler will refuse some of what it proposes. The first never proposes a wrong rewrite; the second never keeps one.

Clippy. Clippy [23], Rust's official linter, adds over 750 lint checks for correctness, style, performance, and complexity to the language's static-analysis ecosystem. It is not a research artifact, but it is the most comprehensive tool operating on Rust's HIR (and also on the AST for lints that do not require type information), and in the context of translation it plays a complementary role: its purpose is not memory safety, but it is effective at identifying unidiomatic or suboptimal constructs in generated Rust, which is exactly what a translator that guarantees safety at the expense of style, such as Cpp2Rust, produces. Section 5.1 considers it a front-end; its lints are hand-written visitors rather than rules, each one a program, and it cannot apply and retract a rewrite, which a compiler-checked rewriter needs most.

The tools above improve the safety of a translation whose performance already matches the C/C++ original. DAEDALUS faces the converse problem: a translation that is safe from the outset and must be made fast. Table 7.2 summarizes the available tools.

Tool	Objective	Technique
Laertes	Replace raw pointers with safe references	Iterative compiler-feedback inference of lifetimes and aliasing
CRustS	Reduce unsafe usage in translated Rust	220+ TXL transformation rules over the syntax tree
Crown	Reduce raw pointers in translated code	SMT-based reasoning on ownership and aliasing constraints
Clippy	Improve idiomatic quality and performance	HIR-based linting with 750+ checks

Table 7.2: The main tools for refining a Rust translation, also applicable to general Rust code.

7.3 Rule-Based Rewriting

Creating rewrites through dedicated static analyses and syntactic substitutions is tedious, error-prone, and gets worse as the number of rules and optimizations increases. Refactoring tools and dedicated languages have been developed to make this rewriting process more scalable. Every system here made the same decision DAEDALUS makes in Section 4.2, to state a rewrite as a rule, and each paid for it with a language of its own. DAEDALUS does not create a new language; its rewrite rules are pure Rust.

TXL. TXL [57] is a language for source-to-source transformation with a three-part structure: a context-free base grammar, a set of grammar overrides, and a rooted set of transformation rules. Parsing follows a direct, top-down functional interpretation of the grammar with full backtracking; ambiguities are resolved deterministically through ordered alternatives, and a transformation may produce structures the grammar itself cannot parse. Its semantics combine first-order functional programming with term rewriting: transformations are typed by the grammar and stated as "native patterns" written in the syntax of the target language; rules have fixed-point semantics, applied until nothing matches, while functions apply once; `deconstruct` clauses impose structural constraints, `construct` clauses build intermediate

subtrees, and `where` clauses provide explicit semantic guards. Grammar overrides let non-terminals be modified or extended without touching the base grammar, which permits catch-all constructs for incomplete or malformed code and island grammars, in which relevant structures such as SQL are parsed inside otherwise uninterpreted text such as COBOL. A TXL rule has the shape of a DAEDALUS rule—a typed source pattern, a guard, and a target—except that TXL's types are grammatical and its guards are syntactic, whereas DAEDALUS's are in the type system and its guards are answered by semantic analysis.

Cobalt and Rhodium. Cobalt [61] is a domain-specific language for compiler optimizations as guarded rewrite rules, and it separates the pattern that states when a rewrite is legal from the heuristic that decides whether it is profitable, so that the legal conditions can be reasoned about independently. A pattern, ψ_1 *followed by* ψ_2 *until* $s \Rightarrow s'$ *with witness* P, names an enabling statement, the innocuous statements that preserve its effect along every path to s, and a predicate P over the execution state; the Simplify theorem prover [62] then discharges three obligations—that ψ_1 establishes P, that ψ_2 preserves it, and that s and s' leave identical states from any state satisfying P—and the optimization is thus proved sound. Rhodium [63], built on Cobalt to spare the programmer the witness, restates optimizations as local rules over an abstract state that a theorem prover checks against the concrete semantics without a user-provided inductive hypothesis. DAEDALUS keeps Cobalt's separation between the pre-condition that decides where a rule may fire and the post-condition that decides whether it paid off (Section 4.2.2), but discharges none of the three obligations; it only asks the compiler whether s' typechecks.

Coccinelle. Coccinelle [64] is the nearest in spirit to DAEDALUS's rule language: a *semantic patch* states a transformation as a diff over a C source, with metavariables declared in a header standing for whatever fragment matches at each site, and it is matched against the control-flow graph through a CTL model checker, which is what lets a pattern say that a call is followed on every path by a release. It has been used for two decades to apply collateral evolutions across the Linux kernel, and Coccinelle for Rust [65], by the same group, brings the approach to the kernel's Rust code, parsing with rust-analyzer's syntax crate and reusing the CTL engine. A patch, read against Listing 4.1, is the same object—a source, a target, and metavariables for what varies—as a diff rather than a pair of functions, with one construct a rule has no counterpart for: the ellipsis, `...`, which stands for any path between two lines, where DAEDALUS matches a subtree and nothing around it. The main difference is that `expression` is the only type a metavariable can have: Coccinelle for Rust takes rust-analyzer's parser and not its type inference, so a patch cannot ask whether a value is `Copy`, whether a type has a hand-written `Drop`, or whether a use is the last. This is the main contrast with DAEDALUS: it does not leverage type information to guide the rewrites, nor does it have the expressiveness to declare post-conditions for the rewrites.

7.4 Summary

Each of the three sections above shares something with DAEDALUS and lacks something it has. The translators set the problem. C2Rust shows that a faithful mechanical translation from C is achievable, at the cost of code no Rust programmer would write; the model-based tools show that idiomatic output is achievable too, at the cost of any guarantee about semantic equivalence. Cpp2Rust alone produces output that is safe from the outset, and its cost—the constructs of Chapter 3—is what DAEDALUS works on. None of them refines its own output.

The refinement tools are the nearest neighbors, and they divide on what decides that a rewrite is safe. Laertes asks the compiler alone, one candidate at a time, and rewrites 12% of its pointers; Crown asks an analysis of its own, made sound by an assumption stricter than Rust's rules, and rewrites 37%; CRustS asks neither, and removes annotations rather than operations. DAEDALUS asks both, in an order none of them uses: an approximate analysis proposes, and the compiler has the final word. It keeps Laertes' oracle but consults it once per batch rather than once per candidate (Section 4.4), and it needs none of Crown's assumptions, since whatever an assumption would have ruled out, the compiler rules out instead.

The rule-based systems share the decision to state a rewrite as a rule, and each pays for it with a language of its own: TXL's grammar, Cobalt's guarded patterns and their proof obligations, or Coccinelle's semantic patches over a control-flow graph. Each offers something DAEDALUS does not have—Coccinelle's ellipsis, Cobalt's proofs—and each lacks what DAEDALUS's rules get for nothing by being Rust: typed metavariables, and the compiler's own judgment of whether a target is well formed. Coccinelle for Rust is the closest of all, and the gap is exactly one thing: type information. None of the above tools has the combination: rewrites stated as rules in the host language—with access to type information—and the compiler as the final judge of each one. Table 7.3 summarizes the available tools against these two properties.

Tool	Rewrites	Rules in host language	Compiler decides
Laertes	raw pointers → references	○	●
Crown	raw pointers → `Box`/references	○	○
CRustS	`unsafe` annotations	○	○
Clippy	lints, applicable suggestions	○	○
TXL	any, over a grammar	○	○
Cobalt	compiler optimisations	○	○
Coccinelle	any, over C or Rust	◑	○
DAEDALUS	`Rc`, `RefCell`, `Option`, `Ptr`, clones	●	●

Table 7.3: The rewriting tools of this chapter against the two properties DAEDALUS combines. Coccinelle's patches are fragments of the rewritten language, but they are not programs in it, nor do they provide any type information to guide the rewrites.

8

Conclusion

Contents

8.1 Future Work 73

To address the emerging problems of modernizing legacy C/C++ software systems, Cpp2Rust offers an automatic, fast solution that generates only safe Rust. However, the code it outputs, albeit safe, is slow, and many systems are not only security-sensitive but also performance-critical. In Cpp2Rust translations, every object is owned through an `Rc<RefCell<T>>` and, if nullable, wrapped in an `Option` type. This is what is necessary to translate a language that allows aliasing and mutation at once into one that is less permissive, without having a human in the loop. In essence, this is how Cpp2Rust circumvents Rust's borrow checker and always emits safe Rust. But because of the additional run-time checks and heap allocations, it incurs considerable performance degradation.

DAEDALUS, the tool presented in this thesis, removes the unnecessary part of that cost, one rewrite at a time, and keeps only rewrites accepted by the Rust compiler. It functions in a rule-based manner. A rule is a pair of function bodies naming the shape to find and the shape to put in its place (i.e., a source and target pattern), with pre- and post-conditions written as ordinary calls. This makes adding a new rewrite as simple as writing a new rule. It is pure Rust code, so there is no learning curve, and it requires no changes to the engine or recompilation. Since a rule cannot argue its own soundness, it may have a

special `well_typed()` post-condition that uses the Rust compiler as an oracle to confirm the rule does not produce any unsound rewrites.

For Cpp2Rust translations, what DAEDALUS can achieve depends directly on how the translated program uses pointers. On the microbenchmarks of Section 6.2, programs that do not take the address of their values lose every wrapper they were given, and can recover their original performance, or even surpass it. If, however, the program passes those same values by pointer, the wrappers stay, and with them, their associated heap allocations and additional run-time checks. WOFF2 and Brunsli represented two real-world examples of considerably sized codebases translated by Cpp2Rust. Running DAEDALUS on them removed half of the heap allocations and run-time checks, reducing WOFF2's decompression from 3.88 to 2.82 times its C++ original and Brunsli's from 35.6 to 21.5. What DAEDALUS left behind in the translated code were `Ptr`s. A `Ptr` exists to be passed to another function, and DAEDALUS is an intra-procedural rewriter, so every wrapper that survives in these programs does so for the same reason: unwrapping it would imply a non-localized rewrite, including signature modifications and everything that flows through it.

DAEDALUS is not specific to Cpp2Rust-translated code. Run over the 1,000 most downloaded crates of `crates.io`—hand-written Rust representing the most used libraries in the world—it applied 5,794 rewrites to 133 of them and returned 845 untouched, taking two hours for nine and a half million lines. Of what was found this way, 265 rewrites were offered upstream in 55 pull requests, and 39 of those have already been merged. Notably, some of these contributions were made to repositories from the Rust Project, Mozilla, and Google.

DAEDALUS bridges the gap that Cpp2Rust created. Cpp2Rust produces safe Rust code from the outset, but it pays for that safety at runtime. The tool presented in this thesis recovers some of that lost performance and offers a path to recover all of it, even surpassing the performance of the original C++ code. DAEDALUS promises that every program it rewrites—given that it is previously compilable—will always compile after being rewritten. Moreover, it is extensible and easy to use: adding a new rewrite means adding a new set of rules in a dedicated file; DAEDALUS takes care of the rest. Thus, we consider DAEDALUS a first step towards removing unnecessary constructs in Rust code, and complementing Cpp2Rust in the mission of automatically migrating C/C++ codebases to safe Rust.

8.1 Future Work

Every rewrite DAEDALUS applies is intra-procedural: the function body is the unit of scanning, blame, and revert. This bounds what the engine can remove and rules out any construct whose removal requires inter-procedural analysis. The clearest case is `Ptr`, which is what stands between the rewritten WOFF2 and Brunsli and their C++ originals. Supporting rewrites that span more than one function—and that may change a signature along with every call—is therefore the most valuable extension DAEDALUS could receive. Although it does not directly affect the generality of Rust code, it would be the final step to make Cpp2Rust a usable tool in the industry for large-scale migrations.

Verification is a second direction. DAEDALUS can presently establish that a rewrite still compiles, not that it preserves the program's behavior. This matters more as the rule set grows, and would be essential if rules were generated automatically, which the current generation of LLMs makes an obvious thing to attempt: a rule may be plausible but wrong, altering behavior in ways no compiler reports. A formal semantics for MIR would allow DAEDALUS to establish that a rule's source and target patterns are semantically equivalent, or that the target refines the source. That is a promising line of work, but a substantial one, and it is unlikely to be within reach until Rust's own operational semantics are settled.

A third direction is to close the gap between `rust-analyzer`'s analyses and the compiler's. The type information `rust-analyzer` exposes approximates what `rustc` would infer rather than the thing itself, and where the two differ, a pre-condition may answer incorrectly and admit a rewrite that should have been refused. Building `rust-analyzer` on `rustc`'s own APIs would be a first step towards closing it, although that depends on how stable those APIs become, along with the name and trait resolution they expose.

Bibliography

[1] M. Miller, “Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape,” in *BlueHat*, 2019. [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

[2] A. Taylor, A. Whalley, D. Jansens, and N. Oskov, “An update on memory safety in Chrome,” 2021. [Online]. Available: https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

[3] J. Protzenko, S. Lee, S. Khadeer, S. Ho, O. Oleksenko, M. Naehrig, and C. Fournet, “Rewriting SymCrypt in Rust to modernize Microsoft’s cryptographic library,” 2025. [Online]. Available: https://www.microsoft.com/en-us/research/blog/rewriting-symcrypt-in-rust-to-modernize-microsofts-cryptographic-library/

[4] NSA, “The case for memory safe roadmaps,” 2023. [Online]. Available: https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF

[5] J. V. Stoep, “Memory safe languages in Android 13,” 2022. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[6] L. Bergstorm, “Beyond safety and speed: How Rust fuels team productivity,” 2022. [Online]. Available: https://www.youtube.com/live/6mZRWFQRvmw?t=26575s

[7] T. Fontana, S. Vigna, and S. Zacchiroli, “WebGraph: The next generation (is in Rust),” in *WWW*, 2024.

[8] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, “An empirical study of Rust-for-Linux: the success, dissatisfaction, and compromise,” in *USENIX ATC*, 2024. [Online]. Available: https://www.usenix.org/system/files/atc24-li-hongyu.pdf

[9] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, “Engineering the Servo web browser engine using Rust,” in *ICSE-C*, 2016.

[10] S.-F. Chen and Y.-S. Wu, “Linux kernel module development with Rust,” in *DSC*, 2022.

[11] S. Panter and N. Eisty, “Rusty Linux: Advances in Rust for Linux kernel development,” in *ESEM*, 2024.

[12] A. A. Terekhov and C. Verhoef, “The realities of language conversions,” *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2002.

[13] T. Suganuma, T. Yasue, T. Onodera, and T. Nakatani, “Performance pitfalls in large-scale java applications translated from COBOL,” in *OOPSLA*, 2008. [Online]. Available: https://doi.org/10.1145/1449814.1449822

[14] I. Ferreira, L. Arkoh, A. Uchôa, A. C. Bibiano, A. Garcia, and W. K. G. Assunção, “Assessing the bug-proneness of refactored code: A longitudinal multi-project study,” 2025. [Online]. Available: https://arxiv.org/abs/2505.08005

[15] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in translation: A study of bugs introduced by large language models while translating code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[16] I. Immunant and I. Galois, “C2Rust: Migrate C code to Rust,” 2024. [Online]. Available: https://github.com/immunant/c2rust

[17] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, “Translating C to safer Rust,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021.

[18] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, “Aliasing limits on translating C to safe Rust,” *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023.

[19] L. Popescu, F. Gouveia, H. Preto, J. Silveira, D. Hrybenko, J. F. Santos, and N. P. Lopes, “Cpp2Rust: Automatic translation of c++ to safe rust,” *Proc. of the ACM on Programming Languages*, vol. 10, no. PLDI, Jun. 2026.

[20] R. Jung, B. Kimock, C. Poveda, E. S. Muñoz, O. Scherer, and Q. Wang, “Miri: Practical undefined behavior detection for rust,” vol. 10, no. POPL, 2026. [Online]. Available: https://doi.org/10.1145/3776690

[21] X. Denis, J.-H. Jourdan, and C. Marché, “Creusot: A foundry for the deductive verification of rust programs,” in *International Conference on Formal Engineering Methods*. Springer, 2022, pp. 90–105.

[22] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, “Verus: Verifying rust programs using linear ghost types,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[23] T. R. P. Developers, “Clippy: A collection of lints to catch common mistakes and improve your Rust code.” [Online]. Available: https://github.com/rust-lang/rust-clippy

[24] ——, “rustfix: Automatically apply the suggestions made by `rustc`.” [Online]. Available: https://github.com/rust-lang/cargo/tree/master/crates/rustfix

[25] D. Tolnay, “`syn`: Parser for Rust source code,” https://crates.io/crates/syn, 2026, accessed: 2026-08-25.

[26] ——, “prettyplease: A minimal `syn` syntax tree pretty-printer.” [Online]. Available: https://crates.io/crates/prettyplease

[27] rust-analyzer contributors, “rust-analyzer: A Rust compiler front-end for IDEs,” https://rust-analyzer.github.io/, 2026, accessed: 2026-08-25.

[28] A. Kladov and rust-analyzer contributors, “`rowan`: A library for lossless syntax trees,” https://github.com/rust-analyzer/rowan, 2026, accessed: 2026-08-25.

[29] N. Matsakis and salsa contributors, “`salsa`: A generic framework for on-demand, incrementalized computation,” https://github.com/salsa-rs/salsa, 2026, accessed: 2026-08-25.

[30] Google, “Woff2.” [Online]. Available: https://github.com/google/woff2

[31] ——, “Brunsli.” [Online]. Available: https://github.com/google/brunsli

[32] F. de Albuquerque Mello Pereira, “Lessons learned from porting the woff2 library from c++ to rust,” 2022. [Online]. Available: https://www.youtube.com/watch?v=kcMAiTg5j1w

[33] Google, “Brotli.” [Online]. Available: https://github.com/google/brotli

[34] A. De Marco, V. Iancu, and I. Asinofsky, “Cobol to java and newspapers still get delivered,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 583–586.

[35] A. Irrera, “Banks scramble to fix old systems as it "cowboys" ride into sunset,” 2017. [Online]. Available: https://www.reuters.com/article/us-usa-banks-cobol/banks-scramble-to-fix-old-systems-as-it-cowboys-ride-into-sunset-idUSKBN17C0D8/

[36] R. Khadka, P. Shrestha, B. Klein, A. Saeidi, J. Hage, S. Jansen, E. van Dis, and M. Bruntink, “Does software modernization deliver what it aimed for? a post modernization analysis of five software modernization case studies,” in *ICSME*, 2015.

[37] M. Mossienko, “Automated COBOL to Java recycling,” in *CSMR*, 2003.

[38] S. Hans, A. Kumar, T. Yasue, K. Ono, S. Krishnan, D. Sondhi, F. Satoh, G. Mitchell, S. Kumar, and D. Saha, “Automated testing of COBOL to Java transformation,” in *FSE*, 2025. [Online]. Available: https://doi.org/10.1145/3696630.3728548

[39] A. Kumar, D. Saha, T. Yasue, K. Ono, S. Krishnan, S. Hans, F. Satoh, G. Mitchell, and S. Kumar, “Automated validation of cobol to java transformation,” in *ASE*, 2024. [Online]. Available: https://doi.org/10.1145/3691620.3695365

[40] L. Popescu and N. P. Lopes, “Exploiting undefined behavior in c/c++ programs for optimization: A study on the performance impact,” *Proc. of the ACM on Programming Languages*, vol. 9, no. PLDI, Jun. 2025.

[41] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of c,” in *USENIX ATC*, 2002. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix02/full_papers/jim/jim.pdf

[42] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: type-safe retrofitting of legacy software,” *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, p. 477–526, May 2005.

[43] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, “Checked c: Making c safe by extension,” in *SecDev*, 2018.

[44] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, “C to checked C by 3c,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, Apr. 2022.

[45] F. Pizlo, “Fil-c: A memory-safe implementation of c and c++,” https://fil-c.org, 2025.

[46] F. P. Stoldt, B. Bucher, S. Clebsch, M. A. Johnson, M. J. Parkinson, G. van Rossum, E. Snow, and T. Wrigstad, “Dynamic region ownership for concurrency safety,” vol. 9, no. PLDI, 2025. [Online]. Available: https://doi.org/10.1145/3729313

[47] J. Sharp, “Corrode: A c to rust translator,” 2016. [Online]. Available: https://github.com/jameysharp/corrode

[48] Kornel, “Citrus: C to rust syntax converter,” 2017. [Online]. Available: https://lib.rs/crates/citrus

[49] Z. Xu, S. Jain, and M. Kankanhalli, “Hallucination is inevitable: An innate limitation of large language models,” 2025. [Online]. Available: https://arxiv.org/abs/2401.11817

[50] A. Z. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, “Vert: Verified equivalent rust transpilation with large language models as few-shot learners,” *arXiv preprint arXiv:2404.18852*, 2024.

[51] J. Bosamiya, W. S. Lim, and B. Parno, “Provably-Safe multilingual software sandboxing using WebAssembly,” in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022, pp. 1975–1992. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[52] P. Deligiannis, A. Lal, N. Mehrotra, R. Poddar, and A. Rastogi, “Rustassistant: Using llms to fix compilation errors in rust code,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 3097–3109.

[53] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, “Towards translating real-world code with llms: A study of translating to rust,” *arXiv preprint arXiv:2405.11514*, 2024.

[54] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray, “C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques,” *arXiv preprint arXiv:2501.14257*, 2025.

[55] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, “Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis,” *arXiv preprint arXiv:2412.14234*, 2024.

[56] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, “In rust we trust: a transpiler from unsafe c to safer rust,” in *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, 2022, pp. 354–355.

[57] J. R. Cordy, “The txl source transformation language,” *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[58] Y. Sui and J. Xue, “SVF: interprocedural static value-flow analysis in LLVM,” in *CC*, 2016.

[59] H. Zhang, C. David, Y. Yu, and M. Wang, “Ownership guided c to rust translation,” in *CAV*, 2023.

[60] L. M. de Moura and N. S. Bjørner, “Z3: An efficient SMT solver,” in *TACAS*, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[61] S. Lerner, T. Millstein, and C. Chambers, “Cobalt: A language for writing provably-sound compiler optimizations,” *Electronic Notes in Theoretical Computer Science*, vol. 132, no. 1, pp. 5–17, 2005.

[62] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: a theorem prover for program checking,” *Journal of the ACM (JACM)*, vol. 52, no. 3, pp. 365–473, 2005.

[63] S. Lerner, T. Millstein, E. Rice, and C. Chambers, “Automated soundness proofs for dataflow analyses and transformations via local rules,” *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 364–377, 2005.

[64] J. Lawall, “Coccinelle: reducing the barriers to modularization in a large c code base,” in *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, ser. MODULARITY ’14. New York, NY, USA: Association for Computing Machinery, 2014.

[65] T. Roy and J. Lawall, “Coccinelle for rust.” [Online]. Available: https://gitlab.inria.fr/coccinelle/coccinelleforrust

A

DAEDALUS's Rules

A.1 The `rc` Pass

```
fn rc1_pre<T>(a0: Rc<T>) -> bool {
    rules::never_calls(&a0, "as_pointer")
}
fn rc1_src<T>(a0: Rc<T>, a1: T) {
    let a0 = Rc::new(a1);
}
fn rc1_tgt<T, a1_type>(a0: a1_type, a1: a1_type) {
    let a0: a1_type = a1;
    let mut a0_s_count = 1;
    let mut a0_w_count = 0;
}


fn rc1_mut_pre<T>(a0: Rc<T>) -> bool {
    rules::never_calls(&a0, "as_pointer")
}
fn rc1_mut_src<T>(a0: Rc<T>, a1: T) {
    let mut a0 = Rc::new(a1);
}
fn rc1_mut_tgt<T, a1_type>(a0: a1_type, a1: a1_type) {
    let mut a0: a1_type = a1;
    let mut a0_s_count = 1;
    let mut a0_w_count = 0;
}
```

```rust
fn rc2_src<T>(a0: Rc<T>) {
    Rc::try_unwrap(a0);
}
fn rc2_tgt<T>(a0: T, a0_s_count: usize, a0_w_count: usize) {
    if a0_s_count == 1 {
        Ok(a0)
    } else {
        Err(a0)
    };
}

fn rc3_src<T>(a0: Rc<T>) {
    Rc::strong_count(&a0);
}
fn rc3_tgt<T>(a0: T, a0_s_count: usize) {
    a0_s_count;
}

fn rc4_src<T>(a0: Rc<T>) {
    Rc::weak_count(&a0);
}
fn rc4_tgt<T>(a0: T, a0_w_count: usize) {
    a0_w_count;
}

fn rc10_src<T>(a0: Rc<T>) {
    Rc::into_inner(a0);
}
fn rc10_tgt<T>(a0: T, a0_s_count: usize) {
    if a0_s_count == 1 {
        Some(a0)
    } else {
        None
    };
}

fn rc11_src<T: Clone>(a0: Rc<T>) {
    Rc::unwrap_or_clone(a0);
}
fn rc11_tgt<T: Clone>(a0: T, a0_s_count: usize) {
    if a0_s_count == 1 {
        a0
    } else {
        a0.clone()
    };
}

fn rc12_src<T>(a0: Rc<T>) {
    Rc::clone(&a0);
}
fn rc12_tgt<T>(a0: T) {
    &a0;
}

fn rc6_src<T>(a0: Rc<T>) {
    a0.clone();
}
fn rc6_tgt<T>(a0: T) {
    &a0;
}
```

```
fn rc7_src<T>(a0: Rc<T>) {
    Rc::as_ptr(&a0);
}
fn rc7_tgt<T>(a0: T) {
    std::ptr::from_ref(&a0);
}
```

A.2 The `refcell` Pass

```
fn refcell1_src<T>(a0: RefCell<T>, a1: T) {
    let a0 = RefCell::new(a1);
}
fn refcell1_tgt<T, a1_type>(a0: a1_type, a1: a1_type) {
    let mut a0: a1_type = a1;
}
fn refcell1_mut_src<T>(a0: RefCell<T>, a1: T) {
    let mut a0 = RefCell::new(a1);
}
fn refcell1_mut_tgt<T, a1_type>(a0: a1_type, a1: a1_type) {
    let mut a0: a1_type = a1;
}
fn refcell2_src<T: Copy>(a0: RefCell<T>) {
    *a0.borrow();
}
fn refcell2_tgt<T>(a0: T) {
    a0;
}
fn refcell3_src<T: Copy>(a0: RefCell<T>) {
    *a0.borrow_mut();
}
fn refcell3_tgt<T>(a0: T) {
    a0;
}
fn refcell4_src<T>(a0: RefCell<T>) {
    a0.borrow();
}
fn refcell4_tgt<T>(a0: T) {
    &a0;
}
fn refcell5_src<T>(a0: RefCell<T>) {
    a0.borrow_mut();
}
fn refcell5_tgt<T>(mut a0: T) {
    &mut a0;
}
```

A.3 The `cast` Pass

```
fn redundant_cast_pre<T>(a0: T) -> bool {
    rules::concrete_return(&a0)
}
fn redundant_cast_src<T>(a0: T) {
    a0 as T;
}
fn redundant_cast_tgt<T>(a0: T) {
    a0;
}
```

A.4 The `clone` Pass

```
fn clone1_pre<T>(a0: Rc<T>) -> bool {
    !rules::is_ref(&a0) && rules::last_use(&a0)
}
fn clone1_src<T>(a0: Rc<T>) {
    a0.clone();
}
fn clone1_tgt<T>(a0: Rc<T>) {
    a0;
}
fn clone1_pos() -> bool {
    rules::well_typed()
}

fn clone2_src<T>(a0: Weak<T>) {
    a0.clone();
}
fn clone2_tgt<T>(a0: Weak<T>) {
    a0;
}
fn clone2_pos() -> bool {
    rules::well_typed()
}

fn clone6_src<T>(a0: Ptr<T>) {
    a0.clone();
}
fn clone6_tgt<T>(a0: Ptr<T>) {
    a0;
}
fn clone6_pos() -> bool {
    rules::well_typed()
}

fn clone5_src<T>(a0: Arc<T>) {
    a0.clone();
}
fn clone5_tgt<T>(a0: Arc<T>) {
    a0;
}
fn clone5_pos() -> bool {
    rules::well_typed()
}

fn clone3_pre<T: Copy>(a0: T) -> bool {
    rules::no_manual_impl_clone(&a0)
        && !rules::is_ref(&a0)
        && !rules::src_mutably_borrowed(&a0)
}
fn clone3_src<T: Copy>(a0: T) {
    a0.clone();
}
fn clone3_tgt<T: Copy>(a0: T) {
    a0;
}
```

```
fn clone4_pre<T>(a0: T) -> bool {
    rules::no_manual_impl_clone(&a0)
        && rules::no_manual_impl_drop(&a0)
        && !rules::is_ref(&a0)
        && !rules::src_mutably_borrowed(&a0)
        && (rules::last_use(&a0) || rules::last_field_use(&a0))
}
fn clone4_src<T: Clone>(a0: T) {
    a0.clone();
}
fn clone4_tgt<T>(a0: T) {
    a0;
}
```

A.5 The `option` Pass

```
fn option1_pre<T>(a0: Option<T>) -> bool {
    rules::known_writes(&a0)
}
fn option1_src<T>(a0: Option<T>, a1: T) {
    let a0 = Some(a1);
}
fn option1_tgt<T>(a0: T, a1: T) {
    let a0 = a1;
    let a0_is_some = true;
}
fn option1_pos<T>(a0: T, a0_is_some: bool) -> bool {
    rules::folded(&a0_is_some) || rules::folded(&a0)
}

fn option2_pre<T>(a0: Option<T>) -> bool {
    rules::known_writes(&a0)
}
fn option2_src<T>(a0: Option<T>, a1: T) {
    let mut a0 = Some(a1);
}
fn option2_tgt<T>(a0: T, a1: T) {
    let mut a0 = a1;
    let mut a0_is_some = true;
}
fn option2_pos<T>(a0: T, a0_is_some: bool) -> bool {
    rules::folded(&a0_is_some) || rules::folded(&a0)
}

fn option6_pre<T>(a0: Option<T>) -> bool {
    rules::known_writes(&a0)
}
fn option6_src<T>(a0: Option<T>) {
    let a0: Option<T> = None;
}
fn option6_tgt<T>(a0: T) {
    let a0: T;
    let a0_is_some = false;
}
fn option6_pos<T>(a0: T, a0_is_some: bool) -> bool {
    rules::folded(&a0_is_some) || rules::folded(&a0)
}
```

```
fn option7_pre<T>(a0: Option<T>) -> bool {
    rules::known_writes(&a0)
}
fn option7_src<T>(a0: Option<T>) {
    let mut a0: Option<T> = None;
}
fn option7_tgt<T>(a0: T) {
    let mut a0: T;
    let mut a0_is_some = false;
}
fn option7_pos<T>(a0: T, a0_is_some: bool) -> bool {
    rules::folded(&a0_is_some) || rules::folded(&a0)
}


fn option3_src<T>(mut a0: Option<T>, a1: T) {
    a0 = Some(a1);
}
fn option3_tgt<T>(mut a0: T, a1: T, mut a0_is_some: bool) {
    a0 = a1;
    a0_is_some = true;
}


fn option8_src<T>(mut a0: Option<T>) {
    a0 = None;
}
fn option8_tgt<T>(mut a0: T, mut a0_is_some: bool) {
    drop(a0);
    a0_is_some = false;
}


fn option4_src<T>(a0: Option<T>) {
    a0.is_some();
}
fn option4_tgt<T>(a0_is_some: bool) {
    a0_is_some;
}


fn option5_src<T>(a0: Option<T>) {
    a0.is_none();
}
fn option5_tgt<T>(a0_is_some: bool) {
    !a0_is_some;
}


fn option9_src<T: PartialEq>(a0: Option<T>) {
    a0 == None;
}
fn option9_tgt<T>(a0_is_some: bool) {
    !a0_is_some;
}


fn option10_src<T: PartialEq>(a0: Option<T>) {
    a0 != None;
}
fn option10_tgt<T>(a0_is_some: bool) {
    a0_is_some;
}
```

```
fn option12_pre<T>(a0: Option<T>) -> bool {
    rules::known_none(&a0)
}
fn option12_src<T, U>(a0: Option<T>, a1: T, body: U) {
    if let Some(a1) = a0 {
        body;
    }
}
fn option12_tgt() {}

fn option13_pre<T>(a0: Option<T>) -> bool {
    rules::always_some(&a0)
}
fn option13_src<T, U>(a0: Option<T>, a1: T, body: U) {
    if let Some(a1) = a0 {
        body;
    }
}
fn option13_tgt<T, U>(a0: T, a1: T, a0_is_some: bool, body: U) {
    if a0_is_some && let a1 = a0 {
        body;
    }
}

fn option14_src<T>(a0: Option<T>) {
    a0.as_ref().unwrap();
}
fn option14_tgt<T>(a0: T) {
    a0;
}
```

A.6 The `ptr` Pass

```
fn ptr1_src<T: Clone>(a0: Rc<RefCell<T>>) {
    a0.as_pointer().read();
}
fn ptr1_tgt<T: Clone>(a0: Rc<RefCell<T>>) {
    (*a0.borrow()).clone();
}
fn ptr1_pos() -> bool {
    rules::well_typed()
}
```

A.7 The `peephole` Pass

```
fn constant_folding1_pre(a0: bool) -> bool {
    rules::known_true(&a0)
}
fn constant_folding1_src<T>(a0: bool, body: T) {
    if a0 {
        body;
    }
}
fn constant_folding1_tgt<T>(a0: bool, body: T) {
    body;
}
```

```
fn constant_folding2_pre(a0: bool) -> bool {
    rules::known_true(&a0)
}
fn constant_folding2_src<T, U>(a0: bool, then_body: T, else_body: U) {
    if a0 {
        then_body;
    } else {
        else_body;
    }
}
fn constant_folding2_tgt<T, U>(a0: bool, then_body: T, else_body: U) {
    then_body;
}


fn constant_folding3_pre(a0: bool) -> bool {
    rules::known_false(&a0)
}
fn constant_folding3_src<T, U>(a0: bool, then_body: T, else_body: U) {
    if a0 {
        then_body;
    } else {
        else_body;
    }
}
fn constant_folding3_tgt<T, U>(a0: bool, then_body: T, else_body: U) {
    else_body;
}


fn constant_folding4_pre(a0: bool) -> bool {
    rules::known_false(&a0)
}
fn constant_folding4_src<T>(a0: bool, body: T) {
    if a0 {
        body;
    }
}
fn constant_folding4_tgt<T>(a0: bool, body: T) {}


fn constant_folding5_pre(a0: bool) -> bool {
    rules::known_true(&a0)
}
fn constant_folding5_src(a0: bool, a1: bool) {
    a0 && a1;
}
fn constant_folding5_tgt(a1: bool) {
    a1;
}


fn constant_folding6_pre(a1: bool) -> bool {
    rules::known_true(&a1)
}
fn constant_folding6_src(a0: bool, a1: bool) {
    a0 && a1;
}
fn constant_folding6_tgt(a0: bool) {
    a0;
}
```

```
fn collapse_irrefutable_let_src<T, U>(a0: T, a1: T, body: U) {
    if let a1 = a0 {
        body;
    }
}
fn collapse_irrefutable_let_tgt<T, U>(a0: T, a1: T, body: U) {
    {
        let a1 = a0;
        body;
    }
}

fn self_let_binding_pre<T>(a0: T) -> bool {
    rules::no_manual_impl_drop(&a0)
}
fn self_let_binding_src<T>(a0: T) {
    let a0 = a0;
}
fn self_let_binding_tgt() {}

fn self_mut_let_binding_pre<T>(a0: T) -> bool {
    rules::no_manual_impl_drop(&a0)
}
fn self_mut_let_binding_src<T>(a0: T) {
    let mut a0 = a0;
}
fn self_mut_let_binding_tgt() {}
fn self_mut_let_binding_pos() -> bool {
    rules::well_typed()
}

fn dead_let_binding_pre<T: Copy>(a0: T, a1: T) -> bool {
    rules::unused(&a0) && rules::side_effect_free(&a1)
}
fn dead_let_binding_src<T: Copy>(a0: T, a1: T) {
    let a0 = a1;
}
fn dead_let_binding_tgt() {}

fn dead_mut_let_binding_pre<T: Copy>(a0: T, a1: T) -> bool {
    rules::unused(&a0) && rules::side_effect_free(&a1)
}
fn dead_mut_let_binding_src<T: Copy>(a0: T, a1: T) {
    let mut a0 = a1;
}
fn dead_mut_let_binding_tgt() {}

fn dead_uninit_let_pre<T>(a0: T) -> bool {
    rules::unused(&a0)
}
fn dead_uninit_let_src<T>(a0: T) {
    let a0: T;
}
fn dead_uninit_let_tgt() {}

fn dead_mut_uninit_let_pre<T>(a0: T) -> bool {
    rules::unused(&a0)
}
fn dead_mut_uninit_let_src<T>(a0: T) {
    let mut a0: T;
}
fn dead_mut_uninit_let_tgt() {}
```

```
fn dead_assign_pre<T: Copy>(a0: T, a1: T) -> bool {
    rules::unused(&a0) && rules::side_effect_free(&a1)
}
fn dead_assign_src<T: Copy>(mut a0: T, a1: T) {
    a0 = a1;
}
fn dead_assign_tgt() {}
fn flatten_block_pre<T>(body: T) -> bool {
    rules::no_outer_shadow(&body) && rules::bindings_are_copy(&body)
}
fn flatten_block_src<T>(body: T) {
    {
        body;
    }
}
fn flatten_block_tgt<T>(body: T) {
    body;
}
fn constant_folding7_pre(a1: bool) -> bool {
    rules::known_true(&a1)
}
fn constant_folding7_src<T, U>(a0: T, a1: bool, then_body: T, else_body: U) {
    let a0 = if a1 {
        then_body;
    } else {
        else_body;
    };
}
fn constant_folding7_tgt<a0_type>(a0: a0_type, then_body: a0_type) {
    let a0: a0_type = then_body;
}
fn constant_folding8_pre(a1: bool) -> bool {
    rules::known_false(&a1)
}
fn constant_folding8_src<T, U>(a0: T, a1: bool, then_body: T, else_body: U) {
    let a0 = if a1 {
        then_body;
    } else {
        else_body;
    };
}
fn constant_folding8_tgt<a0_type>(a0: a0_type, else_body: a0_type) {
    let a0: a0_type = else_body;
}
```