Towards Macro-Aware C-to-Rust Transpilation (WIP)

Robbe De Greef
Vrije Universiteit Brussel
Brussels, Belgium
robbe.de.greef@vub.be

Attilio Discepoli
Vrije Universiteit Brussel
Brussels, Belgium
attilio.discepoli@vub.be

Esteban Aguililla Klein
Université Libre de Bruxelles
Brussels, Belgium
esteban.aguililla.klein@ulb.be

Théo Engels
Royal Military Academy
Brussels, Belgium
theo.engels@mil.be

Ken Hasselmann
Royal Military Academy
Brussels, Belgium
ken.hasselmann@mil.be

Antonio Paolillo
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be

Abstract

The automatic translation of legacy C code to Rust presents significant challenges, particularly in handling preprocessor macros. C macros introduce metaprogramming constructs that operate at the text level, outside of C's syntax tree, making their direct translation to Rust non-trivial. Existing transpilers — source-to-source compilers — expand macros before translation, sacrificing their abstraction and reducing code maintainability. In this work, we introduce `Oxidize`, a macro-aware C-to-Rust transpilation framework that preserves macro semantics by translating C macros into Rust-compatible constructs while selectively expanding only those that interfere with Rust's stricter semantics. We evaluate our techniques on a small-scale study of real-world macros and find that the majority can be safely and idiomatically transpiled without full expansion.

CCS Concepts: • **Software and its engineering → Incremental compilers**; **Source code generation**; **Preprocessors**.

Keywords: Transpilation, C, Rust, Preprocessor, Macros, Abstract Syntax Tree, Metaprogramming, Embedded

ACM Reference Format:
Robbe De Greef, Attilio Discepoli, Esteban Aguililla Klein, Théo Engels, Ken Hasselmann, and Antonio Paolillo. 2025. Towards Macro-Aware C-to-Rust Transpilation (WIP). In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3735452.3735535

LCTES '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1921-9/25/06
https://doi.org/10.1145/3735452.3735535

1 Introduction

Embedded systems power critical applications ranging from robotics and autonomous vehicles to industrial control systems and defense infrastructure. These systems usually have to meet strict safety and reliability requirements. However, much of the software running on these systems is written in *memory-unsafe languages* such as C and C++, leading to security vulnerabilities that pose severe risks, including remote exploits, system crashes, and even physical damage.

Recognizing this, governments and industry leaders are advocating for a transition to memory-safe programming languages: the White House ONCD explicitly recommended moving away from C/C++ [10], Google's security analysis found that 70% of critical vulnerabilities in Android and Chrome stem from memory safety issues [6], and the DARPA has issued a call, the TRACTOR program [2], for proposals aimed at automating the translation of C to Rust.

Rust is seeing increasing adoption across various domains, including in embedded software [7]. Its strong memory safety guarantees, built-in concurrency model, and strict type system make it a compelling alternative to C and C++, particularly for mitigating security vulnerabilities and undefined behavior in low-level programming.

However, embedded and safety-critical software development relies heavily on large C/C++ stacks that include real-time operating systems, kernel subsystems, system-level middleware (*e.g.* CycloneDDS), ROS nodes, and sensor drivers. A full manual rewrite is often impractical due to cost, time, and maintainability constraints. To facilitate Rust adoption without abandoning these critical software stacks, automated transpilation offers a practical solution. By translating C code to Rust in a source-to-source manner, developers can incrementally migrate code while preserving compatibility with legacy systems. However, existing transpilation tools such as C2Rust and Corrode bypass preprocessor constructs, particularly macros, which are widely used in embedded software for configuration, code reuse, and low-level optimizations. Fully expanding macros during transpilation eliminates their abstraction benefits, increases code complexity, and reduces maintainability—making automated migration significantly less practical.

Table 1. `Oxidize` features compared with the related work. ✗ = not supported, ✛ = planned, ✓ = supported.

Project	Features				
	C	C++	Comments	Safe	Macros
`CRust [9]`	✓	✓	✓	✗	✗
`C2Rust [4]`	✓	✗	✗	✗	✗
`Corrode [8]`	✓	✗	✗	✗	✗
`CRustS [5]`	✓	✗	✗	✓	✗
`Oxidize`	✓	✛	✛	✛	✓

Contributions. This paper presents `Oxidize`, a macro-aware C-to-Rust transpiler that preserves macro semantics while ensuring correct and maintainable translation. Our key contributions include: (1) A novel macro handling strategy that classifies macros as either expandable or contractable, allowing selective translation into Rust macros rather than forcing expansion. (2) A technique for resolving identifier hygiene issues, ensuring macros do not introduce naming conflicts. (3) A method for reconstructing types of C macro arguments and implicit C type promotions in Rust to ensure type safety while preserving C semantics. (4) A study on macro usage in an existing codebase, analyzing transpilability in real-world scenarios.

2 Related Work

Transpilers. Existing C/C++ to Rust transpilers include CRust [9], C2Rust [4], and Corrode [8]. The CRustS [5] project, built on top of C2Rust, adds more passes to reduce the amount of `unsafe` blocks. None of these tools preserve preprocessor constructs such as macros – which are widely used in embedded software for configuration, code reuse, and low-level hardware control – hence limiting the practical value of automated migration. None of these tools aim to generate idiomatic Rust code. Table 1 summarizes the key differences between our tool, `Oxidize`, and related work. In contrast to prior tools, `Oxidize` focuses on: (i) preserving macros where possible, (ii) restructuring them into equivalent Rust idioms, and (iii) producing readable, idiomatic Rust.

Others. The Zig programming language includes a C compiler frontend capable of integrating C code directly into Zig projects, but it does not preserve C macros during the process. Zig's goal is interoperability, not transpilation [12]. Macroni [11] preserves macro information during C preprocessing by extracting structured representations of macros for later use, but is not integrated into a transpilation pipeline.

3 Designing a C Transpiler

Building a C-to-Rust transpiler involves multiple stages, beginning with preprocessing and macro handling, followed by parsing the C code into an Intermediate Representation (IR)

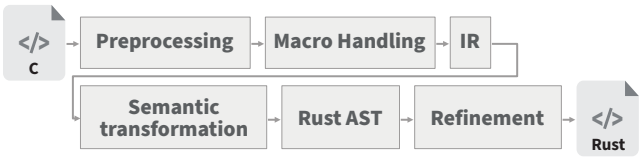

Figure 1. Overview of the various passes of the C-to-Rust `Oxidize` transpiler, starting with a preprocessing stage as frontend followed by a macro handling stage.

that faithfully captures program semantics. This IR serves as the foundation for subsequent semantic transformations, where C-specific constructs – such as implicit type conversions, pointer arithmetic, and struct layout assumptions – are adapted to fit Rust's stricter type system and ownership model. Once these adjustments are made, the IR is converted into a Rust Abstract Syntax Tree (AST), which is then refined further to ensure the resulting code is not merely syntactically correct but also idiomatic and maintainable, as if it had been originally written in Rust.

As illustrated in Figure 1, the design of our transpiler `Oxidize` follows these stages, starting with the frontend that preprocesses the file followed by the handling of macros. At the frontend, we use `libclang` to parse the C source into an AST, extracting syntactic elements while gathering information about macros. We focus in this paper on the macro handling stage. The descriptions of the next stages of `Oxidize` are out of scope and thus left as future work.

4 Transpiling Macros

A particularly challenging stage of the transpiler pipeline is also one of the first: *handling C macros*, which introduce metaprogramming constructs that significantly complicate transpilation. Unlike standard C constructs, macros exist entirely at the preprocessor level, meaning they operate outside the compiler's type system and can define anything from simple constants to complex functions and code fragments.

A common approach to handling macros is to expand them fully before transpilation, reducing all macro logic to standard C syntax [3]. While this preserves functional correctness, it removes the higher-level abstraction intended by the original developer, making the translated Rust code significantly harder to read and maintain. Instead, our approach seeks to preserve the *metaprogramming intent* behind macros by restructuring them into Rust-compatible constructs. Rather than expanding each macro, we analyze its role in the program and determine whether it should be retained as a Rust macro, converted into an equivalent Rust construct, or expanded only where necessary.

4.1 Macros as Text, Not AST

C macros are purely text-based substitutions, meaning they operate independently of C syntax. Unlike Rust macros,

```c
// Original C code
#define RETVAL 2
#define LPAR (
#define RPAR )
#define ASSIGN =
#define SCOLON ;

int main LPAR RPAR {
  int ret ASSIGN RETVAL SCOLON
  return ret SCOLON
}
```

```rust
// Code transpiled into Rust
macro_rules! RETVAL {
  ( ) => { 2 }
}
unsafe fn c_main() -> i32 {
  let mut ret: i32 = RETVAL!();
  return ret;
}
```

Figure 2. Transpiling macros that are not C expressions.

```c
#define STRCPY(x,y) strcpy((x),(y))

int main() {
  int (*strcpy)(char*, char*) = otherfunc;
  char a[] = "Hello";
  char b[] = "abc";
  STRCPY(a, b); // Expands to strcpy((a), (b));
}
```

Figure 3. Example of unhygienic C macro code.

which must form a valid AST, C macros act as token-based replacements and can generate syntactically incorrect constructs when expanded. Consequently, some macros do not have a direct analogue in Rust.

In Figure 2, the macros are used to reconstruct syntax. These macros do not represent valid C ASTs (except for `RETVAL`); rather, they encode *syntactic elements* such as parentheses, assignments, and semicolons. After expansion, the result could be invalid C, since the macro system is unaware of the broader syntactic context. Rust macros require a conforming AST representation, making this type of token-based substitution impossible to transpile directly.

Solution: restricting to valid C expressions. Most macros, as we observe in Section 5, are used in expression contexts and form valid C expressions. We use this property to decide whether a macro can be transpiled or must be expanded. Chained macro invocations are resolved recursively, with each submacro either expanded or retained based on this same criterion.

4.2 Hygiene Issues in C Macros

C macros lack *hygiene*, meaning they do not enforce scoping rules and can introduce identifier shadowing. Since macros are not aware of variable scopes, they can introduce name conflicts. In the example of Figure 3, `strcpy` has been redefined as a function pointer, thus the expanded macro refers to the *local identifier* instead of the standard `strcpy` function.

Solution: restricting outside identifiers. All identifiers used inside a macro must either be defined within the macro's parameters or be defined at the macro's definition without being shadowed at the macro call sites. If a macro depends on an external identifier that is shadowed, it is expanded with a transpilation warning. Rust maintains proper scoping rules, so all macros must follow Rust's hygiene constraints to prevent accidental shadowing.

```c
// Original C code
#define ADD(x, y) \
    ((x) + (y))

int main() {
    int x = 2;
    char y = 3;

    int foo = ADD(x, y);
}
```

```rust
// Code transpiled into Rust
macro_rules! ADD {
    ($x:expr, $y:expr, $ty_0:ty) => {
        (($x as $ty_0) + ($y as $ty_0))
    };
}

unsafe fn c_main() -> i32 {
    let mut x: i32 = 2;
    let mut y: i8 = (3 as i8);

    let mut foo: i32 = ADD!(x, y, i32);
}
```

Figure 4. Argument types are explicitly passed to transpiled macros at their call sites.

4.3 Typing Information and Implicit Conversions

The C preprocessor lacks any typing information. Additionnally, and unlike Rust, C implements features such as implicit type conversions, type promotions and pointer arithmetic, which must be then explicitly implemented in the transpiled code. However, Rust enforces strict typing and does not allow such implicit conversions.

When transpiling typical C code, these implicit conversions can be addressed by analyzing the corresponding types, identifying instances of conversions in the C code, and making them explicit through correct cast operations. However, within macros, there is no typing information available for macro input arguments, as those can change with each invocation. Consequently, it is impossible to explicitly perform casts or handle pointer arithmetic within macros.

Solution: type inference at macro call sites. In Rust, macros allow various types of arguments to be passed in, including types. At points where implicit promotion would typically occur, for example, in function arguments, assignments or arithmetic, we introduce a cast to a generic type name which is passed into the macro. During macro invocation, the transpiler determines the appropriate type based on the macro's input parameters and passes it as the generic type name. Although this solution adds type parameters to macros, which increases complexity, it guarantees type correctness in Rust. The increased complexity can be simplified by adding further code cleaning passes.

Figure 4 illustrates this technique. The macro `ADD` performs arithmetic on variables of different types. Due to C's implicit type promotion rules, the expression `ADD(x, y)` results in an implicit conversion of `char y` to an `int`.

4.4 Handling Chained Macros

Many C macros are not standalone entities but instead invoke other macros in a nested fashion. Handling such cases requires to integrate our prior techniques while maintaining a structured representation of macro dependencies. We construct a graph where each node represents a macro and directed edges indicate which macros invoke other macros. After the graph is constructed, we perform a depth-first traversal and evaluate whether the lowest-level macro in the

```c
#define THREE 3
#define PLUS +
#define BAZ THREE
#define FOO(a) a PLUS BAZ

int main() {
    return FOO(2);
}
```

```rust
macro_rules! THREE {() => { 3 }}
macro_rules! BAZ {() => { THREE!() }}
macro_rules! FOO {
    ($a: expr, $ty_0: ty) => {
      (($a as $ty_0) + (BAZ!() as $ty_0))
    }
}
unsafe fn c_main() -> i32 {
    return FOO!(2, i32);
}
```

Figure 5. Example of macros calling macros in C, transpiled to Rust with `Oxidize`.

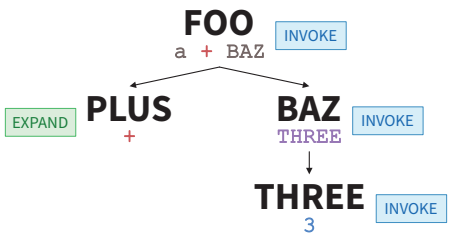

Figure 6. The AST corresponding to the code at Figure 5 with the expand or invoke selection of `Oxidize`.

tree is suitable for transpilation, using our previously described techniques. If it is valid, the macro is invoked rather than expanded wherever it is referenced, including in other macro definitions. By processing the graph in this bottom-up approach, we ensure that any macro that is called in another macro's body has been processed previously, defining whether it should be expanded or invoked.

Figure 5 presents an example of chained macros in C and how they are transpiled: FOO calls PLUS and BAZ, with BAZ further referencing THREE. THREE is processed first, followed by BAZ, and then PLUS. Since PLUS does not form a valid C expression, it needs to be expanded in the body of FOO. Consequently, once PLUS is expanded, FOO becomes a valid C expression and can be invoked directly in the source code rather than expanded. Figure 6 demonstrates the graph created by `Oxidize` for transpiling FOO.

5 Small-Scale Study

To assess the effectiveness of our macro transpilation approach, we conducted an initial evaluation on the open-source project **figlet**, a text-based banner generation tool written in C [1]. This preliminary study aimed to analyze the types of macros present in a real-world codebase and determine how many could be successfully translated into idiomatic Rust using the `Oxidize` macro-aware transpiler.

The **figlet** codebase comprises 8 C source files and contains a total of 82 macros. These include 68 object macros and 14 function macros. Each file was processed through our transpiler to identify which macros could be translated. The resulting macro definitions were then compared with the original macro definitions in the C source.

Table 2. C macros are either transpiled or expanded by `Oxidize`. The table presents the number of transpiled macros over the overall number of macros, per file in **figlet**.

Filename	#T/#M	Filename	#T/#M
`figlet.c`	31 / 31	`zipio.c`	8 / 20
`crc.c`	0 / 1	`utf8.c`	9 / 11
`inflate.c`	10 / 19	`chkfont.c`	4 / 4
`getopt.c`	3 / 4		

The relevant data is presented in Table 2. Notably, the type of macro usage in these files vary considerably. For instance, the **zipio.c** file makes extensive use of macros for defining and expanding statements. Since our current solution is limited to handling expressions, many of these macros are ineligible for direct transpilation and will require expansion.

However, a substantial majority of macros are expression-like macros and are able to be transpiled. This is promising, as it implies a significant portion of macros can be mapped to Rust constructs without requiring complex rewrites. Additionally, we are confident that our approach can be adapted to also support C statements in the future.

6 Conclusions and Future Work

We presented a method for transpiling C macros and functions into Rust, addressing an issue that has been persistently overlooked by other existing transpilation tools, including C2Rust, CRust, and Corrode.

We currently test `Oxidize` using a custom suite that compares the output of the original C code with its transpiled Rust version to check for equivalence. The next phase of the project involves fuzzing the transpiler and extending testing with additional C projects, including DOOM. Furthermore, we aim to focus on low-level libraries, embedded software and potentially certain components of the Linux kernel, such as drivers. As `Oxidize` currently only works with object and function macros, we need to develop it further so that it also handles other types of preprocessor statements (`#include`, `#undef`, `#ifdef`, variadic macros, etc). These could all be expanded, but that would heavily complexify the resulting code. We will prioritize macro transpilation mechanisms according to the use cases we will encounter.

With C++ adding the concept of template-based metaprogramming, more challenges will need to be addressed to transpile more legacy and unsafe code into Rust. The development of `Oxidize` as a transpilation tool will continue, so that we can transpile entire C and C++ codebases into Rust. We will also dedicate future work to the development of Rust-specific passes that reduce the usage of unsafe constructs. We envision the transpiler operating as an incremental, function-by-function tool, exploiting Rust's FFI mechanism to gradually integrate Rust functions into the C codebase to test the correctness of the transpiled functions.

Acknowledgments

This work is supported by the Belgian Ministry of Defence under the Defence-related Research Action (DEFRA), contract number 24DEFRA001, within the FORCES project.

References

[1] Glenn Chappell, Ian Chai, and Contributors. 1991. FIGlet: A program for making large letters out of ordinary text. Online. http://www.figlet.org/ Accessed: 2025-05-12. Maintained on GitHub at https://github.com/cmatsuoka/figlet.

[2] DEPT OF DEFENSE. 2024. Translating All C TO Rust (TRACTOR). Online. https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view Accessed: 2025-05-12.

[3] Immunant. 2018. Handling C Macros in C2Rust. https://github.com/immunant/c2rust/issues/16 Accessed: 2025-05-12.

[4] Immunant. 2025. immunant/c2rust. GitHub repository. https://github.com/immunant/c2rust Originally published 2018-04-20. Accessed: 2025-05-12.

[5] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 354–355. https://doi.org/10.1145/3510454.3528640

[6] Alex Rebert and Christoph Kern. 2024. *Secure by Design: Google's Perspective on Memory Safety.* Technical Report. Google Security Engineering.

[7] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2296–2310. https://doi.org/10.1145/3658644.3690275

[8] Jamey Sharp. 2025. Corrode: Automatic semantics-preserving translation from C to Rust. Online. https://github.com/jameysharp/corrode Accessed: 2025-05-12.

[9] Nishanth Shetty, Nikhil Saldanha, and M. N. Thippeswamy. 2019. CRUST: A C/C++ to Rust Transpiler Using a "Nano-parser Methodology" to Avoid C/C++ Safety Issues in Legacy Code. In *Emerging Research in Computing, Information, Communication and Applications*, N. R. Shetty, L. M. Patnaik, H. C. Nagaraj, Prasad Naik Hamsavath, and N. Nalini (Eds.). Springer, Singapore, 241–250. https://doi.org/10.1007/978-981-13-5953-8_21

[10] Stack Overflow. 2024. In Rust We Trust: White House Office Urges Memory Safety. Online. https://stackoverflow.blog/2024/12/30/in-rust-we-trust-white-house-office-urges-memory-safety/ Accessed: 2025-05-12.

[11] Trail of Bits. 2023. Holy Macroni! A Recipe for Progressive Language Enhancement. https://blog.trailofbits.com/2023/09/11/holy-macroni-a-recipe-for-progressive-language-enhancement/ Accessed: 2025-05-12.

[12] Zig Software Foundation. 2024. Working with C - Zig Guide: translate-c. https://zig.guide/working-with-c/translate-c/. Accessed: 2025-05-12.