RUSTY: Effective C to Rust Conversion via Unstructured Control Specialization

Xiangjun Han Baojian Hua* Yang Wang*
School of Software Engineering
University of Science and Technology of China
pnext@mail.ustc.edu.cn bjhua@ustc.edu.cn* angyan@ustc.edu.cn*

***Abstract*—Rust is an emerging programming language designed for both performance and security, thereby sparking recent study interests on migrating legacy C/C++ code bases to Rust to exploit Rust's safety benefits. Unfortunately, prior studies, as well as practical engineering efforts, on C to Rust conversion still suffer from three limitations: 1) complex structure; 2) code explosion; and 3) poor performance. These limitations greatly hinder the deployment of such conversions in practice.**

In this paper, we present RUSTY, the *first* system for effective C to Rust code conversion, via a novel technique of unstructured control specialization. First, we systematically analyzed the limitations of prior studies and investigated the root causes, which motivated our insights and novel solutions. Next, we propose RUSTY, an infrastructure for C to Rust conversion, in which the key technical novelty is to implement C syntax on top of Rust as syntactic sugars, thus eliminating the syntactic discrepancies between the two languages. We have implemented a prototype of RUSTY and conducted extensive experiments to evaluate its effectiveness, performance and usefulness. We first applied RUSTY to microbenchmarks and experimental results demonstrated that RUSTY is effective in eliminating unstructured controls. We then applied RUSTY to three real-world large C projects: 1) Vim; 2) cURL; and 3) the silver searcher. Experimental results showed that RUSTY successfully reduced the code size by 21.1% on average, and is performant in processing 817 lines of C code per second.

***Index Terms*—Rust security, code conversion, control specialization**

I. INTRODUCTION

Rust [1] is an emerging programming language with design goals of memory safety, type safety, and thread safety. First, Rust guarantees memory safety by a unique ownership model [2] as well as a borrow checker [3], without explicit garbage collections. Second, Rust ensures type safety with both compile-time static type checking and runtime checking [4]. Third, Rust offers thread safety via global immutable states and fearless concurrency [5]. Due to these safety advantages, since the first stable release in 2015 [6], Rust has been successfully used to build a diverse range of important infrastructures, such as operating system kernels [7], file systems [8], network protocol stacks [9], Web browsers [10], language runtimes [11], databases [12], cloud services [13], and blockchains [14]. Rust is arguably becoming the next generation of safe systems programming language.

* Corresponding authors.

In light of Rust's safety advantages, recently, there have been considerable academic studies [15] [16] [17], as well as industry efforts [18] [19], on migrating legacy C/C++ code bases to Rust. Such migrations bring three notable advantages: 1) safety; 2) efficiency; and 3) economy. First, although C/C++ has been the de facto system language due to their performance advantages and low-level programming flexibility, C/C++ has been well known to bring more flexibility in introducing subtle bugs [20]. To this end, migrating legacy C/C++ code bases to Rust can significantly improve safety [21]. Second, migrating legacy C/C++ code to Rust does not sacrifice efficiency due to Rust's zero-cost abstraction design philosophy and its competitive performance with C [22]. Third, it is often more economic to migrate legacy code to Rust than to rewrite every line of code from scratch, as the mature data structures and algorithm design in legacy projects can be largely reused, thus reducing the reimplementation efforts considerably. Due to these advantages, recent studies [15] [16] [17] and efforts [23] [24] on C to Rust migrations are quite successful and promising.

Technically, two approaches can be utilized for migrating legacy C projects to Rust: manual or automated migrations. First, in manual migration, (Rust) developers rewrite the functionalities of legacy C projects, using Rust. Unfortunately, although manual code migration might seem attractive and flexible as Rust-native data structures and algorithms can be selected, and runtime performance can be fine tuned in this approach, its engineering efforts are often considerable or even unacceptable, especially for large C projects with huge code bases. For example, the Linux kernel consists of more than 27.8 million lines of C code [25], thus migrating such projects manually is difficult, if not impossible.

Second, in automated migrations, legacy C code can be converted to Rust by leveraging *transpilers* in an automated manner. Unlike compilers, which compiles programs in high-level languages (e.g., C or Java) into programs in low-level languages (e.g., x86 or ARM), a transpiler translates programs between languages at (roughly) the same abstraction level (e.g., C to Java [26]), without changing the functionalities of the programs being translated. Due to its advantages of full automation and semantics preservation, transpiler-based automated code migration has been a hot research topic with a significant amount of academic studies [15] [16] [17] as well

as engineering efforts [27] [28] [29] [30], to investigate the theory underpinnings and practical development, respectively. With these research progress, current transpilers are quite effective and successful. For example, C2Rust [30], one of the state-of-the-art transpilers from C to Rust, might convert a C project of more than 145,000 lines of code, in less than one minute without any manual interventions

Limitations. Unfortunately, while prior studies on automated code migration have made considerable progress, they still suffer from three limitations: 1) complex structure; 2) code explosion; and 3) poor performance. First, the Rust code generated by the existing automated conversion algorithms and tools exhibits more complex structures than the original C code. For example, the widely used Relooper algorithm [31] will generate complex Rust programs with deeply nested loops, for even simple C programs with `goto` statements [32]. The complex structures of the generated Rust code not only bring challenges to code understanding, but also make subsequent software engineering tasks, such as code evolution and maintenance, difficult. Second, existing technologies and tools often generate Rust code of undesired sizes. For example, in converting cURL, a popular network library in C, C2Rust generates $7,700$ lines of Rust code for a 700 line C function [33], leading to a significant code size increase up to 10X. Third, exiting algorithms and tools often generate Rust code with poor performance, largely due to the unreasonable code size increase, which not only increases the compilation time but also may have undesired effects of increasing cache misses, which are unacceptable for performance-sensitive scenarios such as real-time systems. The effectiveness and practical usefulness of automated migration techniques are greatly reduced without addressing these limitations.

Our work. In this paper, we present RUSTY, the *first* system for effective C to Rust code migration via unstructured control specialization. First, we systematically analyzed the limitations of prior studies and investigated the root causes leading to these limitations. We found that the syntactic discrepancy of *control* structures between C and Rust is the key root cause for the aforementioned limitations. We argue that this root cause is not unique to transpilers (e.g., C to Rust migration) but is general even to compilers (e.g., C to WebAssembly compilation) [34]. With these findings and insights, we then proposed novel methodologies to address these limitations by presenting the design of RUSTY, an effective infrastructure for C to Rust conversion. The key technical insight of RUSTY is to implement C-oriented syntactic sugars on top of Rust, thus eliminating the discrepancies between the two languages. Finally, we present a prototype implementation of RUSTY, which consists of three main components: 1) a tree rewriter introducing specialized unstructured code patterns; 2) a library implementing the syntax extensions for Rust; and 3) a library syntactic expansion based on Rust's procedural macros [35][36].

We have conducted extensive experiments to evaluate RUSTY in terms of effectiveness, complexity, correctness, performance, and usefulness. First, to evaluate the effectiveness of RUSTY, we applied it to microbenchmarks, and experimental results demonstrated that RUSTY is effective in reducing the code sizes of the generated Rust target code by 21.1% on average. Second, to evaluate the complexity of the Rust code RUSTY generated, we applied RUSTY to three real-world large C projects: 1) Vim; 2) cURL; and 3) the silver searcher, and experimental results demonstrated that RUSTY is effective in reducing the cognitive complexity (CC) [37] of the generated Rust code by 65.3% on average. Third, to evaluate the correctness of RUSTY, we conducted regression testings with the test cases distributed with the benchmarks, and experimental results demonstrate that RUSTY guarantees the functional correctness of the generated Rust code. Fourth, to evaluate the performance of RUSTY, we measured the time RUSTY spent in processing C code, and experimental results showed that RUSTY can process 817 lines of C code per second. Finally, to evaluate the usefulness of RUSTY, we conducted a developer study, and the survey results demonstrated that RUSTY is helpful to end-users in converting C to Rust in a fully automated manner.

Contributions. To the best of our knowledge, this work is the *first* study for effective C to Rust code migration via unstructured control specialization. To summarize, this work makes the following contributions:

- **A systematic study of existing limitations and root cause investigation.** We conducted a systematic study of limitations of prior studies and tools and investigated the root causes, which we believe will also benefit future studies in this direction.
- **A novel infrastructure RUSTY.** We presented the design of RUSTY, the first effective transpiler from C to Rust via unstructured control specializations.
- **A prototype implementation and extensive evaluations.** We implemented a software prototype for RUSTY, for automated code migration and conducted systematic experiments to evaluate RUSTY.

Outline. The rest of this paper is organized as follows. Section II presents the necessary background. Section III analyzes the limitations of existing studies and investigates the root causes. Section IV and V present the design of RUSTY and its prototype implementation, respectively. Section VI presents the experiments we performed to evaluate RUSTY. Section VII discusses possible improvements and directions for future work. Section VIII discusses related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work: the Rust programming language (Section II-A), and the transpilers for language conversions (Section II-B).

A. Rust

Capsule history. Rust is an emerging programming language dedicated to building reliable and efficient system software. It grew out of a personal project in 2006 by Graydon

Hoare, and was subsequently sponsored by Mozilla [38] officially since 2009. The first public version was released in 2015, and its latest stable version is 1.66.1 (as of this study). Rust is becoming mature and of production quality, after more than 15 years of active development.

Advantages. Rust emphasizes performance, reliability, and productivity. First, Rust provides high performance by utilizing an ownership-based explicit memory management [39] as well as a lifetime model, without any runtimes or garbage collectors. Both the ownership and lifetime are checked and enforced at compile-time, without incurring runtime overhead. Second, Rust guarantees reliability via its sound type system. Rust's type system are based on linear logic [40] [41] and alias types [42] [43]. These advanced types not only rule out memory vulnerabilities such as dangling pointers, memory leaking, and double frees, but also enforce thread safety by preventing data races and deadlocks. Third, Rust provides productivity via advanced programming features for object-oriented programming such as traits and generics, as well as for functional programming such as pattern matching and closures. These programming features not only provide productivity, but also make Rust's performance on par with C/C++ due to their zero-cost design rationals.

Wide adoptions. Rust is gaining more adoptions and becoming an increasingly important language due to its technical advantages. First, Rust continues to attract more developers. For example, according to Stack Overflow, Rust has been ranked the most loved language for six consecutive years [44]. Second, Rust has an active community with rich resources. According to the official statistics, Rust's central repository (i.e., `crates.io`) contains more than 90 thousand crates (Rust's terminology for libraries), with more than 20 billion downloads [45]. Third, academia has conducted a significant number of studies on Rust [46] [47] and the industry are deploying Rust in production environments [48] (e.g., Amazon, Microsoft, Facebook, and Google). Moreover, in the next decade, a desire to secure cloud or edge computing infrastructures will offer more opportunities for Rust.

B. Transpiler for Language Conversion

Concept. A *transpiler* (translating compiler) converts programs in one programming language into semantic equivalent programs in another language [49]. Unlike compilers, which compile programs in high-level languages (e.g., C or Java) to programs in low-level languages (e.g., x86 or ARM), a transpiler translates programs between languages at (roughly) the same abstraction level. For example, the C2Rust transpiler [30] converts C programs to functionally equivalent Rust programs, in which both C and Rust are system programming languages. Transpilers have a relatively long history of studies, dating back at least to 1998 [50] [51], for converting C/C++ programs to Java programs. Recently, transpilers have been extensively studied with many tool proposed such as C to Rust [27] [28] [29] [30], Python to JavaScript [52] [53], and JavaScript to Python [54]. In the future, with more adoptions of domain-specific languages, transpilers will be used more widely for program conversion between different languages.

Workflow. A transpiler's workflow, from a conceptual point of view, typically consists of three consecutive stages: 1) source processing; 2) translation; and 3) target generation. First, the input source is processed into an internal program representation, which are often abstract syntax trees (ASTs) as they convey almost all source information that subsequent translation needed. Second, these internal program representations are translated to another language's internal representations with different syntax but equivalent semantics. Third, target programs are generated from the converted internal representations. Notably, although transpilers look simple from a conceptual point of view, they are actually large and complex software, especially those for converting general purpose languages such as C and Rust. For example, the C2Rust transpiler consists of more than 80,000 lines of Rust code, even excluding the supporting libraries.

III. Problem Analysis

In this section, we conduct a problem analysis for existing studies to motivate our insights and solutions. We first discuss challenges in language conversions (Section III-A), as well as existing solutions (Section III-B). We then discuss the limitations of existing solutions and investigate the root causes leading to these limitations (Section III-C). Finally, we propose requirements for ideal source migrations (Section III-D).

A. Challenges for Language Conversions

In designing and implementing a transpiler, the key challenge one must address is the syntactic discrepancies between the source and target languages. Technically, as the Venn diagram in Fig. 1 presents, in the context of C to Rust conversion, language features in the source and target languages can be classified into three categories: 1) common features (region B, in the middle); 2) source language features absent from the target (region A, on the left); and 3) target language features absent from the source (region C, on the right).

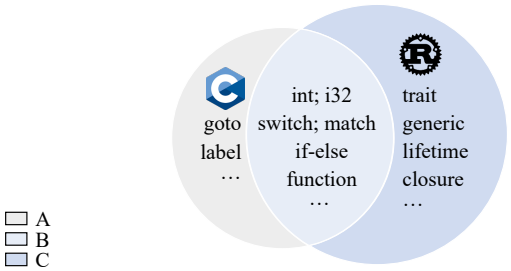
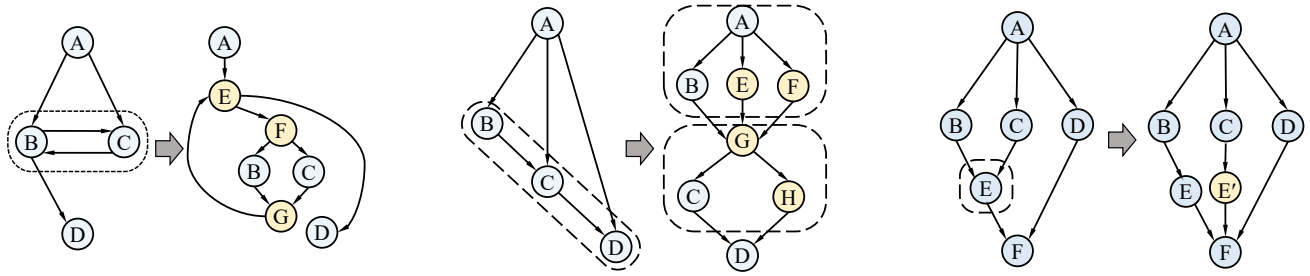


Fig. 1: A Venn diagram illustrating the syntactic discrepancies between C and Rust.

Features in regions B and C do not bring any difficulties: first, features in region C do not affect the source language translation, as they only belong to the target language. For example, while traits are important object-oriented programming features in Rust, they do not affect C to Rust conversion



(a) Complex structures (b) Code explosion (c) Poor performance

Fig. 2: Three sample CFG conversions to illustrate the three limitations: complex structures, code explosion, and poor performance.

simply because they are not features of the C language, and thus do not appear in any legal C programs. Second, features in region B can be translated straightforwardly via a syntax-directed fashion. For example, most structured control flows (e.g., `if` or `while` in C) can be converted *as is* to their corresponding counterparts in Rust.

Features in region A, unfortunately, pose challenges for language conversion because they do not appear in the target languages. Conceptually, to convert such a feature F in region A, one must encode the feature F with other features of the target language to mimic its functionality. Depending on the semantic complexity of the target feature F, the process of encoding might be rather complex, and thus the results are unsatisfactory. For example, while C has unstructured controls such as `goto` statement supporting arbitrary jumps, Rust does not support such syntactic features. Prior studies [31] have demonstrated that it is challenging to convert C's `goto` statement to languages with only structured control flows such as Rust.

It should be noted that, although we have focused on C to Rust conversion in this work and made use of a specific transpiler, C2Rust, as our running example, the aforementioned challenge is ubiquitous in transpilers for other languages, and even in compilers as well (for target languages with structured control flows). For example, in C to WebAssembly compilation [55] [56], it is challenging to convert C's unstructured control statements to WebAssembly, as the latter one has only structured control flows. This challenge even caused GCC, one of the leading C compilers, to abandon its project to build a WebAssembly code generator [57], after several's efforts.

B. Existing Solutions

To address the aforementioned challenge of unstructured control discrepancy, prior studies have proposed the Relooper algorithm [31], which is the de facto algorithm in state-of-the-art transpilers or compilers (e.g., [58]). Space limit prevents us from presenting all details of this algorithm, instead, we present a high-level overview of this algorithm which is enough to discuss its limitations.

Technically, the Relooper algorithm takes as input a Control Flow Graph (CFG) representation of a program, and generates as output a set of structured blocks. Based on the conversion strategies, blocks in the CFG are classified into three categories: 1) *Simple*; 2) *Loop*; and 3) *Multiple*; corresponding to three kinds of control structures: 1) sequence; 2) selection; and 3) iteration; respectively. Relooper is essentially a greedy algorithm constructing the aforementioned three categories of blocks based on pattern matching subgraphs in the input CFG. If the target subgraph being matched is irreducible, a *Loop* construct (that is, the second category) is constructed.

C. Limitations and Root Cause Analysis

Although the Relooper algorithm is general to process arbitrary CFG, it suffers from three limitations: 1) complex structures; 2) code explosion; and 3) poor performance.

First, the Relooper algorithm often generates Rust code exhibiting more complex structures than the original C code. We studied the source of Relooper algorithm and identified the root cause leading to complex structures in the generated Rust code is irreducibility of the CFG. For example, as presented by Fig. 2a, the subgraph (on the left) consisting of nodes B and C is irreducible. To make the subgraph reducible, Relooper algorithm adds three new nodes E, F, and G (on the right), and corresponding edges between them. The addition of such new nodes and edges complicated the structure of the programs considerably.

Second, the Relooper algorithm often generates Rust code of undesired sizes. We explored the root causes and identified that the irregularity of the CFG leads to such code explosions. For example, in Fig. 2b, the CFG on the left cannot be converted to a *Multiple* construct directly as nodes B, C and D have different successors. To convert this CFG, Relooper adds four new nodes E, F, G, and H, as shown on the right of Fig. 2b. Although the addition of the four new nodes makes the two subgraphs in the dash boxes on the right in Fig. 2b easily convertible, it does increase the code size considerably (by 2X).

Third, the Relooper algorithm often generates Rust code with low performance. We explored the root cause and identified the key reason is code duplication. For example, in Fig. 2c, to convert the left CFG, Relooper will duplicate the node E as a new node E', thus forming the CFG on the right of Fig. 2c. The duplication and addition of nodes affect the performance

of the program by incurring more cache misses, especially for large code blocks E with many instructions.

In summary, while existing solutions might solve the program migration problem, they are far from a satisfying solution due to the three limitations.

D. Requirements

We propose that: a transpiler, to be practically useful, should satisfy the following three important requirements: 1) idiomatic style; 2) competitive code size; and 3) high performance. First, the target programs generated by a transpiler should be of idiomatic style of the target language and conforms to the developer programming habits. Ideally, the generated programs should be indistinguishable from the programs manually crafted by experienced developers (in the target languages). This requirement is important not only for diagnosing potential problems during conversion but also for subsequent target program evolutions and maintenance. Second, the generated target programs should be of competitive code size with the original source programs. Ideally, the generated programs should be of the same code size as the corresponding source programs. Otherwise, the usefulness of a transpiler is significantly reduced if the generated programs explode. However, it is difficult to satisfy this requirement due to the syntactic discrepancies between the source and target languages. Third, the generated target programs should be performant, especially by exploiting the most appropriate features of the target languages.

We argue that it is challenging for a transpiler to meet all three requirements simultaneously. For example, idiomatic target programs generated by a transpiler may not be the most performant. Hence, transpiler design involves a trade-off between these goals, in which heuristics must be employed.

IV. Design

In this section, we present the design of RUSTY. First, we introduce the design goals of RUSTY and its architecture. Next, we present the design of each component of RUSTY, respectively.

A. Design Goals

We have three goals in designing RUSTY: 1) completeness; 2) ideal Rust code generation; and 3) modularity. First, RUSTY should support the *complete* syntax of C; otherwise, it is difficult if not impossible for RUSTY to process large C projects with nontrivial C features. Second, RUSTY should generate *ideal Rust code* for the input C source with respect to the requirements we discussed in Section III-D: idiomatic code style, competitive code size, and high performance. Third, the architecture of RUSTY should be *modular* so that each module can be easily tested, extended, or replaced separately.

B. Architecture

With these design goals, in Fig. 3, we present the architecture of RUSTY, consisting of six primary components: 1) the C frontend; 2) control-flow graph generation; 3) Rust code generator; 4) Rust rewriter; 5) Rust syntax extensions; and 6) validation.

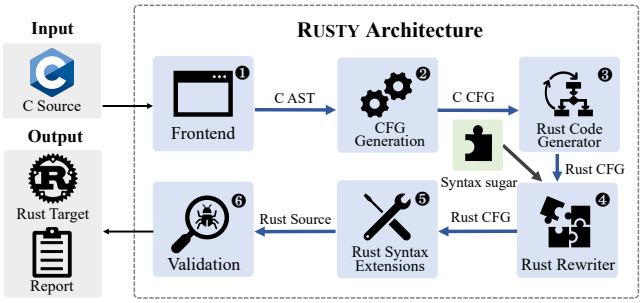


Fig. 3: The architecture of RUSTY.

First, the C frontend (❶) takes as input the C program sources and parses them into abstract syntax trees (ASTs) representations. Second, the CFG generation (❷) takes as input the C ASTs and generates annotated CFGs, in which the goto and label statements have special syntactic annotations. Third, the Rust code generator (❸) translates the C CFG and generate as output a Rust CFG without unstructured control flows. Fourth, the Rust rewriter (❹) takes as input the Rust CFG and rewrites it into a semantically equivalent Rust CFG. Fifth, the Rust syntax extensions (❺) take as input the Rust CFG, expands the syntax extensions and then outputs the Rust code. Finally, the validation (❻) takes as input the generated Rust code, validates its normal functionality and performs measurements, to generate a final report for subsequent analysis.

In the following sections, we discuss the design details of each module, respectively.

C. Frontend

The C frontend parses the C source files into abstract syntax trees, and consists of classical components of preprocessors, scanners, parsers, and abstract syntax tree builders.

To eliminate the syntax discrepancies between the two languages C and Rust, we designed an algorithm to rewrite the corresponding C syntax, which do not have counterparts in Rust (i.e., syntactic features in region A of Fig. 1). Algorithm 1 shows how we rewrite the C ASTs. The function REWRITE-AST takes as input the original C AST t generated by the frontend and rewrites it to a new C AST by eliminating the syntax discrepancies. The algorithm iterates over each statement s in the AST t (line 2), then rewrite the statement s (line 4) to a new syntax form $\mathcal{M}(s)$, if s is a rewriting candidate (line 3). The function ISCANDIDATE returns whether or not a statement s is a rewriting candidate, by determining whether the syntax form of s belongs to C but not Rust. For example, if statement s is a `goto` statement (line 7), it is determined and returned to be a rewriting candidate (line 8).

Three key properties of Algorithm 1 deserve further explanations: first, the algorithm, as presented here, is of imperative style, that is, tree nodes are modified and returned in a destructive manner (line 5). This is not an intrinsic limitation of this

Algorithm 1 : Syntax Discrepancy Elimination

Input: t: an AST representation of the C Source
Output: t: the converted C AST

```
1:  procedure REWRITE-AST(t)
2:      for each statement s ∈ t do
3:          if isCandidate(s) then
4:              Rewrite s into M(s);
5:      return t
6:  procedure ISCANDIDATE(s)
7:      if s is a goto L then
8:          return True
9:      else if s is a label L then
10:         return True
11:     else if s is an declaration then
12:         return True
                                  ▷ Omit similar syntax forms
13:     return False
```

algorithm, as we might modify this algorithm to a functional one by constructing new tree nodes instead of destroying or updating older ones in place. Although functional updating may be more concise, it might incur more execution overhead and might not be compatible with state-of-the-art frontend infrastructures such as LLVM, which makes extensive use of imperative styles.

Second, the rewriting strategy (line 4) in this algorithm is intentionally kept abstract, leaving spaces for practical implementations. In practice, any implementation technique can be leveraged. For example, an arguably simplest implementation strategy is to introduce a new node by boxing all relevant statements.

Third, the algorithm is extensible and scalable in its encapsulation the rewriting candidate functionality into a separate function ISCANDIDATE. In addition to the conditions listed (line 7 to 12), other judgment conditions can be added as well without affecting the tree nodes rewriting strategies. Hence, the modularity design goal is achieved by separating this module.

D. CFG Generation

The CFG generation takes as input the C AST and builds control-flow graphs (CFGs), which are graph-based compiler intermediate representations with statements as graph nodes and branches as directed edges between nodes.

The CFG generation is based on the linear scan algorithm [59], with two key novelties RUSTY brings: first, the graph construction algorithm should account for the new syntax form $\mathcal{M}(s)$. This is straightforward, as $\mathcal{M}(s)$ is treated as a normal statement without any control-flow semantics.

Second, and more importantly, the generated control-flow graphs by RUSTY are always reducible [60], as only structured control flows are preserved after AST rewriting based on Algorithm 1. This nice property of graph reducibility addressed the aforementioned challenges and limitations (Section III-C).

E. Rust Code Generator and Rust Rewriter

The Rust code generator module takes as input the C CFG, and generates as output a corresponding Rust CFG by compiling C statements into equivalent Rust statements (i.e., features in region B of Fig. 1).

The compilation of most C statements to Rust is performed in a syntax-directed manner using a pattern matching algorithm. For example, as the result of compiling a C `if` statement, a corresponding Rust `if` statement is generated.

The compilation of the unstructured control statements deserves further discussion. Recall that, from Section IV-C, such statements are represented specially on the C AST, as Rust does not support unstructured control statements directly. Hence, we designed novel Rust syntax extensions to represent unstructured controls, whose internal design details will be discussed next (Section IV-F). For example, to represent arbitrary jumps and code labels, we introduced two syntax forms: `rust_goto ``L''` and `rust_label ``L''`, where `` ``L'' `` is a Rust string. With these Rust extensions, the compilation from C to Rust can be performed in a direct and intuitive manner:

$$\mathcal{L}(\texttt{goto L}) = \texttt{rust\_goto "L"}$$
$$\mathcal{L}(\texttt{L}) = \texttt{rust\_label "L"}$$

in which $\mathcal{L}$ is the compilation function compiling a C statement into a corresponding Rust statement.

F. Rust Syntax Extensions

To eliminate the discrepancies between C and Rust, we designed special syntax extensions as Rust syntactic sugars, as exemplified by `rust_goto` and `rust_label` in the preceding section (IV-E). Although these syntax extensions make the compilation straightforward and intuitive, they are not valid Rust syntax, and thus must be implemented in the first place.

The Rust syntax extension module takes as input the Rust code with syntax extensions and generate as output an equivalent Rust code by expanding the syntax extensions. As a result, the generated Rust code conforms to Rust syntax standard and thus compiles without any errors. Algorithm 2 shows how syntax extensions are performed on Rust AST. The function EXPAND-STATEMENTS takes as input a Rust syntax extension generated by the Rust code generator, and generate as output an equivalent Rust code by expanding the syntax extension. First, the function iterates over each statement s in the AST t (line 3), and collects s into G or L, according to whether s is `rust_goto ``L''` or `rust_label ``L''`, by the function COLLECT-STATEMENT, respectively. Then, the function iterates each statement l in the L (line 5) and expands each g in G (line 6) if its destination is l (line 7). The statement g is replaced by a new break statement (line 9), when g is forward goto (line 8). In contrast, g is replaced by a new continue statement (line 11), when g is a backward goto (line 10). Then, the statement l is replaced by a new loop, after all statement g with the destination l have been processed (line 12).

Algorithm 2 : Syntax Extension Expansion

Input: t: a Rust AST with syntax extensions
Output: t: the Rust AST with extension expanded

```
1:  procedure EXPAND-STATEMENTS(t)
2:      G, L = ∅
3:      for each statement s ∈ t do
4:          G, L = COLLECT-STATEMENT(s, G, L)
5:      for each statement l ∈ L do
6:          for each statement g ∈ G do
7:              if ISDESTINATION(g, l) then
8:                  if ISFORWARD(g, l) then
9:                      t = NEW-BREAK(g, t)
10:                 else if ISBACKWARD(g, l) then
11:                     t = NEW-CONTINUE(g, t)
12:         t = NEW-LOOP(l, t)
13:     return t
14: procedure COLLECT-STATEMENT(s, G, L)
15:     if s is rust_goto then
16:         G ∪ = s;
17:     if s is rust_label then
18:         L ∪ = s;
19:     return G, L
```

The runtime complexity of this algorithm is $O(n^2)$, for a function with n statements. Although the complexity is not linear, it is efficient in practice, as unstructured controls occur infrequently, even in large C programs. Our evaluation results (Section VI) demonstrated that this algorithm is efficient even for large benchmarks.

G. Validation

Design flaws in RUSTY or bugs in its implementation will lead to the generation of noncompilable or buggy Rust code. Furthermore, even correct Rust code may be not efficient as the original C code, thus violating the three requirements for transpilers (Section III-D). To this end, the validator validates the generated Rust code, with respect to two specific goals: 1) validating the normal functionality of the generated Rust code; and 2) evaluating the Rust code in terms of complexity, and performance. First, to guarantee the normal functionalities of the generated Rust code, we perform regression testing against the original C code, using the test cases distributed with benchmarks. Second, to evaluate the generated Rust code, we measure its cognitive complexities as well as performance, whose results are discussed in detail in Section VI.

H. Program and Report Generation

After validation, RUSTY generates as outputs the final Rust code, as well as the corresponding final report for subsequent analysis.

V. IMPLEMENTATION

We have implemented a software prototype for RUSTY. We leveraged the `clang` C compiler's frontend [61] to parse C sources. We extended Clang's abstract syntax trees to represent the C syntax extensions we introduced. We leveraged and extended the Relooper algorithm [31] to convert the C control-flow graphs to its corresponding Rust control-flow graphs. In particular, we extended the Rust control-flow graphs to incorporate the new Rust syntax extensions we defined. We leveraged two off-the-shelf Rust libraries: `forward_goto` and `goto`, to implement the syntax extension of the Rust syntax. These two libraries are both open source and publicly available on Rust's central registry `crates.io` [45]. We used the procedural macro and the declarative macro capability provided by Rust compiler to modify the Rust abstract syntax trees internally. Finally, we implemented the validator using bash and Python scripts, which are also used to collect and process evaluation results.

VI. EVALUATION

In this section, we conduct experiments to evaluate RUSTY. We first introduce the research questions guiding the evaluation (Section VI-A), the experimental setup (Section VI-B), and the data sets used (Section VI-C). Then, we evaluate RUSTY in terms of effectiveness, complexity, correctness, performance, and usefulness (Section VI-D to VI-H), respectively.

A. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. As RUSTY is proposed to translate C code to Rust without code explosions, is RUSTY effective in reducing code expansion in C to Rust conversion?

RQ2: Complexity. As RUSTY is proposed to translate C code to Rust without complex structures, is RUSTY effective in simplifying the complexity of the generated Rust code?

RQ3: Correctness. As RUSTY is proposed to translate C code to Rust without affecting the correctness of original C code, does RUSTY guarantee the correctness of the generated code?

RQ4: Performance. As RUSTY is introduced to generate high performance target code, does RUSTY achieve this design goal?

RQ5: Usefulness. As RUSTY is intended to relieve developers from the burden of manual C to Rust conversion, is RUSTY useful to help developers to achieve this goal?

B. Experimental Setup

We execute the Rust compiler version nightly 1.65.0 (2022-08-06 build). All experiments and measurements are performed on a server with one 12 physical Intel i7 core (20 hyperthreads) CPU and 64 GB of RAM running Ubuntu 20.04.

C. Data Sets

We used two data sets to conduct the evaluation: 1) microbenchmarks; and 2) three real-world large C projects.

Microbenchmarks. To verify the effectiveness and correctness of RUSTY, we need to create a microbenchmark of the ground truth. To this end, we built microbenchmarks of ground truth consisting of 10 C programs as shown in Table I. These

TABLE I: The microbenchmarks used in the evaluation.

No.	LoC				Binary DEC[2]	CORR[3]
	C	Baseline	RUSTY	DEC[1]		
1	23	43	37	14.0%	96	✔
2	29	58	43	25.9%	176	✔
3	17	22	27	-22.7%	0	✔
4	19	32	29	9.4%	80	✔
5	19	34	27	20.6%	80	✔
6	20	61	43	29.5%	144	✔
7	18	82	47	42.7%	160	✔
8	17	44	37	15.9%	32	✔
9	18	26	28	-7.7%	0	✔
10	26	48	37	22.9%	128	✔
Total	206	450	355	21.1%	896	✔

[1], [2], and [3] are abbreviations of DECrement, binary size DECrement (in bytes), and CORRectness, respectively.

TABLE II: Macro benchmarks of 3 real-world projects.

Project	Domain	LoC	#Files	Github Stars (k)
Vim	System Tools	358,707	186	28.4
cURL	Web	145,802	535	26.5
Silver searcher	Dev Tools	3,932	12	24.1

benchmark programs are created in two different ways: 1) we selected 6 benchmarks containing C's `goto` statements from the C2Rust project's test cases (the first 6 rows); and 2) we manually developed 4 other C programs that make explicit use of `goto`.

It should be noted that the sole purpose of these microbenchmarks is to verify RUSTY's effectiveness and correctness, but not its complexity or performance, so the size of these benchmarks is irrelevant here. To measure RUSTY's complexity, performance, and usefulness on real-world programs, we created and used the following real-world large C benchmarks.

Real-world projects. To further verify the effectiveness, correctness, complexity, performance, and usefulness of RUSTY, we need to select create macro benchmarks consisting of real-world C projects. Two principles guide our selection of real-world C projects. First, the project we selected should be widely used and publicly available, thus the evaluation in this work can be reproduced on such public projects by others. Second, to cover as many C usage scenarios as possible, we aim to include as many domains in our study as possible. According to the aforementioned project selection criteria, we collected C projects from 3 different domains: system tools, Web, and dev tools. The selected domains and projects are presented in Table II. For each of the projects included, we give the name of the selected projects in the corresponding domain, the sizes of these projects (measured by lines of C source code), the numbers of C source files, and the GitHub stars reflecting their popularity.

D. RQ1: Effectiveness

To answer **RQ1** by investigating the effectiveness of RUSTY, we applied RUSTY to the microbenchmarks in Table I to measure the decrements of code sizes (in terms of both source code and binary code).

Table I presents the experimental results of the microbenchmarks. The second column (**C**) presents the sizes of the original C source code. The third (**Baseline**) and the fourth (**RUSTY**) columns present the sizes of the Rust target code generated by the C2Rust transpiler and RUSTY in this work, respectively. We select C2Rust as our baseline, as it is the state-of-the-art transpiler from C to Rust. The next column (**DEC**) shows the source code size decrement which is calculated by $DEC = 1 - \text{RUSTY}/Baseline$. Similarly, the next column (**Binary DEC**) presents the decrement of binary sizes. It should be noted that to avoid the long-tail effect, we only measure the `.text` section in the target binary, as it contains program binary code.

The results give interesting findings and insights. First, RUSTY successfully reduced the code sizes of 8 out of 10 benchmarks. The decrements of sizes are between -22.7% and 42.7%, with an average of 21.1% for all microbenchmarks. Second, RUSTY successfully reduced the size of the binary sizes of 8 out of the 10 benchmarks. The decrement of binary size is 896B in total and 4.35B per line of C code. Third, although the code sizes of the two cases (the 3rd and the 9th) increases (-22.7% and -7.7%, respectively), their binary sizes do not increase.

We then investigated the root causes for the 2 cases whose code sizes increased, based on a manual code inspection. This inspection reveals the key reason: dead code preservation. In C programs, the code between a `goto` statement and a corresponding label statement is *dead code*, as they are not executed. During conversion, RUSTY retains such dead code, whereas C2Rust does not. Therefore, the code sizes of the Rust target code generated by RUSTY are higher than those generated by C2Rust. However, due to compiler optimization, the dead code is removed when source programs are compiled to binaries. Therefore, the binary sizes of programs generated by RUSTY are the same as those binaries generated by C2Rust. It should be noted that the preservation of dead code in RUSTY does not reflect any limitations or drawbacks, as code structures should be preserved faithfully during the conversion.

Summary: RUSTY is effective in reducing the code sizes of eight microbenchmarks successfully. The decrement is 21.1% of source code sizes on average, and is 4.35B per line for binary code. We investigated the root cause and identified dead code preservation is the key reason.

E. RQ2: Complexity

To answer **RQ2** by investigating the complexity of conversion result, we applied RUSTY to the macro benchmarks in our data set. In particular, to calculate the code structure complexity, we leveraged a metric of cognitive complexity (CC). Cognitive complexity was explicitly designed to measure code understandability, which was not only extensively used in academic studies [37] [62], but also integrated into widely used static code analysis tools [63] [64].

TABLE III: Experimental results on the macro benchmarks.

File	LoC	Cognitive Complexity (CC)			Processing Time (s)			Binary Size (B)			CORR[3]
		Baseline	RUSTY	DEC[1] (%)	Baseline	RUSTY	INC[2] (%)	Baseline	RUSTY	DEC[1]	
alloc	621	20	11	45	1.517	1.729	14.0	20,482,080	20,481,744	336	✔
autocmd	2,295	433	127	70.7	2.267	2.362	4.2	20,848,908	20,848,468	440	✔
cindent	2,864	67	59	11.9	2.767	2.858	3.3	20,823,638	20,823,406	232	✔
diff	2,582	59	27	54.2	2.466	2.429	-1.5	20,998,256	20,997,976	280	✔
findfile	1,858	172	102	40.7	2.072	2.121	2.4	20,805,350	20,805,078	272	✔
xhistogram	267	29	9	69.0	1.463	1.492	2.0	20,492,454	20,492,454	0	✔
conncache	415	12	6	50	1.099	1.291	17.5	20,784,516	20,784,188	328	✔
cookie	1082	21	11	47.6	1.468	1.652	12.5	20,810,174	20,810,198	-24	✔
http	3032	18	10	44.4	2.045	2.018	1.3	20,995,970	20,995,962	8	✔
mqtt	604	75	32	57.3	1.148	1.203	4.8	20,623,278	20,622,998	280	✔
url	2,885	278	71	74.5	2.065	2.118	2.6	21,189,972	21,189,220	752	✔
decompress	207	84	34	59.5	0.638	0.901	41.2	20,458,600	20,458,912	-312	✔
search	576	319	49	84.6	1.081	1.105	2.2	20,630,334	20,629,846	488	✔
zfile	314	23	10	56.5	0.695	0.715	2.9	20,497,832	20,497,776	56	✔
Total	19,602	1,610	558	65.3	22.791	23.994	5.3	290,441,362	290,438,226	3,136	✔

[1], [2], and [3] are abbreviations of DECrement, INCrement and CORRectness, respectively.

Table III presents the experimental results. The first column lists the specific **File** from the macro benchmarks for which the results are recorded, whereas the second column (**LoC**) gives the file sizes (in line of code). The three columns in (**Cognitive Complexity (CC)**) present the CC metric of the Rust code generated by C2Rust, RUSTY, and the decrements, respectively. The decrement is calculated by $DEC = 1 - \text{RUSTY}/Baseline$.

The results give interesting findings and insights. First, RUSTY successfully reduced the CC metrics of all the benchmarks. Second, the decrements of CC are between 8% and 84.6%, with an average of 65.3%; furthermore, the most significant decrease is 84.6%, demonstrating the effectiveness of RUSTY in reducing the complexity of the generated Rust code.

We then investigated the reasons that RUSTY could reduce cognitive complexity of generated Rust target code, and found that RUSTY successfully hides the complex control structures caused by the `goto` statement with the syntactic extensions, thus leaving syntactic sugars to end users.

Summary: RUSTY successfully reduces the complexity of the generated Rust code by reducing the cognitive complexity metrics of all the macro benchmarks, with an average of 65.3%.

F. RQ3: Correctness

To answer **RQ3** by investigating the correctness of conversion, we applied RUSTY to all the microbenchmarks as well as macro benchmarks, and evaluated the correctness of the generated Rust code.

We leveraged a strategy of regression testing and applied retest-all technique for correctness checking. Although regression testing is not program verification, it is well-established and widely used software engineering practice to guarantee functional correctness of program transformations. In particular, for any test case T distributed with the benchmark, we first compile and execute the original C program, then compile and run the Rust target code generated by RUSTY. Then, we compared the two outputs to determine that the results are same, to guarantee the correctness RUSTY's conversion.

The last columns in Table I and III present the results of microbenchmarks and macro benchmarks real-world projects, respectively. The results demonstrated that RUSTY passed all the regression tests distributed with these benchmarks, thus illustrating RUSTY's code conversion is semantics-preserving.

Summary: RUSTY guarantees the correctness of conversion by preserving program semantics and functionalities.

G. RQ4: Performance

To answer **RQ4** by investigating the performance of the conversion, we applied RUSTY to the macro benchmarks in our data set (Table II). Each program is processed in 10 rounds, to calculate an average processing time.

The three columns (under **Processing Time (s)**) in Table III present the processing times of the baseline, RUSTY, and the time increment, respectively. The increment of processing time (INC) is calculated by $INC = \text{RUSTY}/Baseline - 1$.

The results give interesting findings and insights. First, RUSTY spent more processing time than the baseline except for one test case, with an average time increase of 5.3% compared to the baseline. Second, RUSTY is performant to process 817 line of C source code per second.

Recall that the theoretical time complexity of the conversion algorithm is $O(n^2)$ (Section IV-F), whereas the real time increases are considerably lower than the theoretical calculation. We then investigated the root causes by a manual inspection of the relevant source code. This inspection revealed one key reason: the goto statements are rarely used even in practical large C projects.

Summary: RUSTY is performant in processing 817 line of C per second, with the time increase of 5.3% compared to the baseline. We investigated the root causes and identified the reason for this insignificant time increase is the rareness of goto statements, even in practical large C projects.

TABLE IV: Time and difficulty results for the tasks, and satisfaction with their conversion as reported by the developers.

Task No.	Time (m)			Difficulty	Satisfaction
	Median	Min	Max	Median (1-7)	Median (1-7)
1 (Manual)	24.5	23	27	2.5	6.2
2 (Baseline)	32.7	29	39	5.2	4.5
3 (RUSTY)	4.7	4	5	1.2	6.5

H. RQ5: Usefulness

To answer **RQ5** by investigating the practical usefulness of RUSTY, we conducted a developer study to compare the efforts needed to migrate C to Rust, manually and automatically by RUSTY.

To quantify the manual effort needed to migrate C to Rust and evaluate the usefulness of RUSTY, we conducted a developer study with six Rust developers. All developers have extensive experience in using C and Rust. In particular, they have been using Rust as their primary developing language for more than 3 years. However, they are not very familiar with migrating C to Rust. None of them had ever migrated a real-world C project to Rust before. Therefore, we can quantify the effort required for a Rust developer to learn and apply an automatic conversion pattern.

Throughout our study, we asked the developers to finish three tasks, without or with the aid of RUSTY: 1) migrating a C program containing goto statement to Rust by manual rewriting (task 1); 2) migrating a C program to Rust with existing transpilers (i.e., C2Rust in this study), and then adjust the control structures of the generated Rust code to improve understandability (task 2); and 3) migrating a C program to Rust with the aid of RUSTY (task 3). The first task measured the manual effort required to migrate C program to Rust. The second task measured the usefulness of transpilers in conversion from C to Rust and the effort required to adjust control structure in Rust target code. The third task measured the usefulness of RUSTY in conversion from C to Rust, which could simplify the control structures and reduce the size of the generated Rust code compared with existing transpilers, in a fully automated manner.

As shown in Table IV, we measured the time required by the developer to perform the task (excluding the time required to read and understand the task's description) for all tasks. We also asked the developers to rate the difficulty of performing the tasks and their satisfaction levels for their conversions on a 7-point Likert scale.

The results give interesting findings and insights. First, the median time needed to migrate C to Rust manually (i.e., task 1) is 24.5 minutes, whereas the median time for RUSTY-aided migration (task 3) is only 4.7 minutes. Interestingly, the median time used in task 2 (with the aid of baseline tool C2Rust) is the highest at 32.7 minutes. Second, the highest difficulty is task 2 with a score of 5.2 (the higher, the more difficult), whereas the lowest one is task 3 with a score of 1.2. Third, the highest satisfaction score is task 3 with a score of 6.5 (the higher, the more satisfying with the migration), whereas the lowest one is task 2 with a score of 4.5.

We then investigated the root causes leading to these results, and identified three key reasons. First, although manually migrating the original C code (task 1) require a lot of time and efforts, understanding, adjusting and tuning the automatically generated Rust code required more time and efforts (task 2), due to the complex control-flow structures in the generated Rust code by existing transpilers (as we discussed in Section III). On the contrary, RUSTY-aided migration is easy and straightforward to take considerably less time and efforts. Second, task 2 is most difficult as the generated code from C2Rust is not only complex but also of undesirable sizes. Third, the developers are most satisfied with RUSTY, as it is not only fully automated, but also incurs no effort for manual code rewriting.

Summary: The developer study showed that RUSTY is practically useful to end developers, with the shortest processing time, lowest usage difficulties, and highest satisfactions.

VII. DISCUSSION

In this section, we discuss some possible enhancements of this work along with directions for future work. It should be noted that this work represents the first step towards effective C to Rust conversion via unstructured control transfer specialization.

C++ support. While this study has made considerable progress on effective migration from real-world C projects to Rust, it does not currently support the C++ language. To support C++, RUSTY can process more realistic projects. Fortunately, the design of the RUSTY architecture (Section IV-B) is neutral to C or C++, thus it is possible to add C++ support by extending other modules. However, C++ language is more complex than C by containing many novel features, thus more effort is required to study the conversion strategy for these C++ language features. To the best of our knowledge, none of the existing transpiler to Rust supports C++ (including the state-of-the-art C2Rust). We leave this an important future work.

Compilers. We have focused on the study of effective conversion strategies for transpilers in this work. However, it should be noted that the techniques studies here can also be applied to compilers as well without any intrinsically technical difficulties. For example, recently, there have been considerable studies on WebAssembly (WASM) [**?**] and C/C++ to WebAssembly compilations. As WebAssembly only supports structured control flows, thus compilation from C/C++ to WebAssembly also faces the same challenges as we discussed in this work (Section III). In particular, the state-of-the-art Emscripten compiler [**?**] from C/C++ to WebAssembly also leveraged the Relooper algorithm to support unstructured control flows. We believe the techniques proposed in this work might also benefit compilers such as Emscripten. This is an important direction for our future work.

VIII. Related Work

There has been a significant amount of work on transpilers in general, and on automated C to Rust conversion particularly. However, this work in this study stands for a novel contribution to these fields.

Transpilers. There have been several transpilers converting C to Rust. Bindgen [27] generates Rust FFI bindings for C libraries automatically. Corrode [28] is semantics-preserving transpiler intending for partial automation. Similar to Bindgen, Citrus [29] generates function bodies but without preserving C semantics. As the successor of Corrode, C2Rust [30] supports large-scale automatic conversion while preserving semantics.

However, a major limitation of existing work is that they only considered the reliability of the conversion from C to Rust, but ignored the quality of the generated Rust code. In contrast, in this work, we conducted a systematic problem analysis for existing work and proposed an effective C to Rust conversion via a strategy of unstructured control specialization.

Automated C to Rust Conversion. There have been several studies on the optimization of Rust target code generated by the C to Rust conversion. Emre et al. [15] first analyzed the sources of unsafety in Rust code generated by C2Rust and proposed a technique to convert raw pointers into references in translated programs, which hooks into the `rustc` compiler to extract type and borrow checker results. Hong et al. [16] proposed an approach to lift raw pointers to arrays with type constraints. Ling et al. [17] presented CRustS, which eliminates non-mandatory unsafe keywords in function signatures and refines unsafe block scopes inside safe functions using code structure pattern matching and transformation.

However, all aforementioned studies on C to Rust automated conversion focused on safety. In contrast, this work, for the first time, optimizes Rust target code via unstructured control specialization. Alternatively, RUSTY can be deployed along with existing works and tools, as RUSTY is orthogonal to these works.

IX. Conclusion

This paper presents RUSTY, the first framework for effective C to Rust code conversion via unstructured control specialization. The key novelty of RUSTY is to implement C-oriented syntactic on top of Rust as syntactic sugars. We have implemented a software prototype for RUSTY and conducted extensive experiments to evaluate it. Experimental results demonstrated that RUSTY is effective in generating ideal target code with idiomatic code structures, while remaining sufficiently efficient and retaining correctness. This work stands for the first step towards effective C to Rust conversion, thus making the promise of Rust as a practical safe system language a reality.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

References

[1] N. D. Matsakis and F. S. Klock, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, New York, NY, USA: Association for Computing Machinery, pp. 103–104, October 2014.

[2] "Understanding ownership," https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[3] "The borrow checker," https:// rustc-dev-guide .rust-lang.org /borrow _check. html.

[4] "Type checking," https:// rustc-dev-guide .rust-lang.org /type-checking.html.

[5] "Fearless concurrency," https://doc.rust-lang.org/book/ch16-00-concurrency.html.

[6] "Rust 2015," https://doc.rust-lang.org/edition-guide/rust-2015/index.html.

[7] Tock, https://www.tockos.org/.

[8] D. Peter, "Fd," https://github.com/sharkdp/fd.

[9] Cloudflare, "Quiche," https://github.com/cloudflare/quiche.

[10] "Speedy Web Compiler," https://github.com/swc-project/swc.

[11] Deno, https://deno.land/.

[12] Meilisearch, https://www.meilisearch.com/.

[13] P. Cohen, "Aws pitches rust for sustainable cloud programming," February. 2022.

[14] Diem blockchain, https://www.diem.com/en-us/.

[15] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 121:1–121:29, October. 2021.

[16] T. Y. Hong and Bryan, "From C towards idiomatic & safer Rust through constraints-guided refactoring," Doctoral dissertation, National University of Singapore, 2022.

[17] 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 *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 354–355.

[18] "Quantum," https://wiki.mozilla.org/Quantum.

[19] M. Team, "A proactive approach to more secure code – Microsoft security response center."

[20] T. Saito, R. Watanabe, S. Kondo, S. Sugawara, and M. Yokoyama, "A survey of prevention/mitigation against memory corruption attacks," in *2016 19th International Conference on Network-Based Information Systems*, Sep. 2016, pp. 500–505.

[21] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," in *2017 IEEE Security and Privacy Workshops*, 2017, pp. 80–92.

[22] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," https://arxiv.org/abs/2206.05503v1, Jun. 2022.

[23] "Mitigating memory safety issues in open source software."

[24] C. Wiltz, "A brief history of rust at facebook," Apr. 2021.

[25] S. Bhartiya, "Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd," Jan. 2020.

[26] "MtSystems" https://www.mtsystems.com/.

[27] "Bindgen," https://github.com/rust-lang/rust-bindgen.

[28] "Corrode," https://github.com/jameysharp/corrode

[29] "Citrus," https://gitlab.com/citrus-rs/citrus.

[30] "C2rust," https://c2rust.com/.

[31] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, Association for Computing Machinery, New York, NY, USA, 2011, pp. 301–312.

[32] "Examples," https://gitee.com/pnext/examples.

[33] "Translate ftplistparser.c into Rust," https://gitee.com/openeuler/curl-rust/commit/fb90c6feae3c8f3fade387c12338d367e77273b8.

[34] N. Ramsey, "Beyond relooper: Recursive translation of unstructured control flow to structured control flow (functional pearl)," in *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 90:1–90:22, Aug. 2022.

[35] "Forward_goto," https://crates.io/crates/forward_goto.

[36] "Goto," https://crates.io/crates/goto.

[37] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18, New York, NY, USA: Association for Computing Machinery, pp. 57–58, May 2018,

[38] "Internet for people, not profit," https://www.mozilla.org/en-US/.

[39] D. J. Pearce, "A lightweight formalism for reference lifetimes and borrowing in rust," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, pp. 3:1–3:73, April 2021.

[40] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, January 1987.

[41] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, vol. 3, no. 4, 1990, p. 5.

[42] J. Boyland, "Alias burying: Unique variables without destructive reads," *Software: Practice and Experience*, vol. 31, no. 6, 2001, pp. 533–553.

[43] D. Clarke and T. Wrigstad, "External uniqueness is unique enough," in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 176–200.

[44] "Stack overflow developer survey 2021," https://insights.stackoverflow.com/survey/2021.

[45] "Crates.io: Rust package registry," https://crates.io/.

[46] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: finding memory safety bugs in Rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, pp. 84–99, October 2021

[47] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: detecting bugs in Rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, pp. 2183–2196, November 2021.

[48] O. F. Anthonia, "Top 10 big companies using Rust," https://careerkarma.com/blog/companies-that-use-rust/, February 2022.

[49] R. Kulkarni, A. Chavan, and A. Hardikar, "Transpiler and it's advantages," in *International Journal of Computer Science and Information Technologies*, vol. 6, 2015, pp.1629-1631.

[50] F. Buddrus and J. Schödel, "Cappuccino — a C++ to Java translator," in *Proceedings of the 1998 ACM Symposium on Applied Computing*, ser. SAC '98. New York, NY, USA: Association for Computing Machinery, pp. 660–665, February 1998.

[51] S. Malabarba, P. Devanbu, and A. Stearns, "MoHCA-Java: a tool for C++ to Java conversion support," in *Proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 650–653.

[52] "Javascripthon", https://pypi.org/project/javascripthon/.

[53] "Jiphy," https://github.com/timothycrosley/jiphy.

[54] "Js2Py," https://github.com/PiotrDabkowski/Js2Py.

[55] "Crash in WebAssembly CFG Stackify," https://github.com/llvm/llvm-project/issues/40652.

[56] "Cannot handle irreducible CFGs caused by unwind edges," https://github.com/llvm/llvm-project/issues/49292.

[57] "Re: more compatibility it's possible?" https://gcc.gnu.org/legacy-ml/gcc/2019-11/msg00095.html.

[58] WebAssembly, "Binaryen," https://github.com/WebAssembly/binaryen.

[59] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[60] Appel, *Modern Compiler Implementation*. USA: Foundation Press, Inc., 2007.

[61] Clang, https://clang.llvm.org/.

[62] J. Bogner and M. Merkel, "To type or not to type? a systematic comparison of the software quality of JavaScript and TypeScript applications on GitHub," *2022 IEEE/ACM 19th International Conference on Mining Software Repositories*, Pittsburgh, PA, USA, 2022, pp. 658-669.

[63] SonarCloud, https://sonarcloud.io/.

[64] M. Koch, "Cognitive complexity - IntelliJ IDEs plugin," https://plugins.jetbrains.com/plugin/12024-cognitivecomplexity.