Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs

Jaemin Hong
School of Computing
KAIST
Daejeon, South Korea
jaemin.hong@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

Abstract—Concurrent programs suffer from data races. To prevent data races, programmers use locks. However, programs can eliminate data races only when they acquire and release correct locks at correct timing. The lock API of C, in which people have developed a large portion of legacy system programs, does not validate the correct use of locks. On the other hand, Rust, a recently developed system programming language, provides a lock API that guarantees the correct use of locks via type checking. This makes rewriting legacy system programs in Rust a promising way to retrofit safety into them. Unfortunately, manual C-to-Rust translation is extremely laborious due to the discrepancies between their lock APIs. Even the state-of-the-art automatic C-to-Rust translator retains the C lock API, expecting developers to replace them with the Rust lock API. In this work, we propose an automatic tool to replace the C lock API with the Rust lock API. It facilitates C-to-Rust translation of concurrent programs with less human effort than the current practice. Our tool consists of a Rust code transformer that takes a lock summary as an input and a static analyzer that efficiently generates precise lock summaries. We show that the transformer is scalable and widely applicable while preserving the semantics; it transforms 66 KLOC in 2.6 seconds and successfully handles 74% of real-world programs. We also show that the analyzer is scalable and precise; it analyzes 66 KLOC in 4.3 seconds.

I. INTRODUCTION

In system programming, concurrency is important yet notoriously difficult to get right. System software reduces execution time by spawning multiple threads and splitting tasks. As a drawback, it suffers from various bugs not existing in the sequential setting: data races, deadlock, starvation, etc [18].

Data races are the most common category of concurrency bugs [18]. It happens when multiple threads read and write the same memory address simultaneously. Data races lead system programs to exhibit not only unpleasant malfunctions but also critical security vulnerabilities [22].

Among synchronization mechanisms to avoid data races, locks are the most widely-used one. Each thread acquires and releases a lock before and after accessing shared data. This simplicity has facilitated the adoption of locks in diverse system software. Unfortunately, locks prevent data races only when they are used correctly. Programmers may acquire wrong locks, acquire locks too late, or release locks too early, thereby failing to eliminate data races.

C, in which people have developed a significant portion of system programs, burdens programmers with the valida-

tion of correct lock use. The most popular lock API of C, pthreads [16], does not automatically check whether programs use locks correctly. Developers often fail to recognize incorrectly used locks in their programs, and C programs thus have suffered from data races.

Rust [17], [42], a recently developed system programming language, provides a lock API guaranteeing *thread safety*, *i.e.*, the absence of data races, in `std::sync` of its standard library [8]. The combination of the ownership type system of Rust and the carefully designed API allows the type checker to validate the correct use of locks at compile time [35]. The API is different from the C lock API not only syntactically, *e.g.*, in names of functions, but also semantically. For instance, the Rust lock API requires programs to explicitly describe which lock protects which data, while the C lock API does not.

Thanks to the thread-safe lock API of Rust, rewriting legacy concurrent C programs in Rust is a promising approach to secure safety. It can reveal unknown data races and allow developers to make fixes. In addition, rewriting in Rust prevents introducing new data races while adding new features.

Noticing this potential, programmers have begun to rewrite concurrent programs in Rust. Mozilla developed Servo, a web browser written in Rust, and has replaced modules of Firefox with those of Servo. Its developers said that Rust's thread safety significantly helped implement concurrent renderers correctly [32]. People also adopt Rust into operating systems, in which concurrency is extremely important. The next release of the Linux kernel will support Rust code [23]. Android and Fuchsia implementations also use Rust [1], [53].

Reimplementing concurrent programs in Rust is, however, labor-intensive if done manually. The discrepancies between the lock APIs of C and Rust hinder programmers from mechanical rewriting. They have to understand the use of the C lock API in original programs and restructure the programs to express the intended logic with the Rust lock API. It poses the necessity for a tool for automatic C-to-Rust translation.

The state-of-the-art C-to-Rust translator, C2Rust [58], is still far from alleviating the burden on programmers. The translation of C2Rust is completely syntactic and generates Rust code using the C lock API. Programmers are expected to replace the C lock API in C2Rust-generated code with the Rust lock API for thread safety. While being better than nothing,

the benefit from C2Rust's syntactic translation is marginal.

In this work, we propose an automatic tool to replace the C lock API with the Rust lock API. Specifically, we make the following contributions:

- We identify the problem of replacing the C lock API with the Rust lock API as the first step toward automatic C-to-Rust translation of concurrent programs (§III).
- We propose an automatic Rust code transformer that takes C2Rust-generated code and its lock summary as inputs and replaces the C lock API with the Rust lock API (§IV).
- We propose a static analyzer that efficiently generates a precise lock summary by combining bottom-up dataflow analysis and top-down data fact propagation (§V).
- We build Concrat (**Con**current-**C** to **R**ust **A**utomatic **T**ranslator) by combining C2Rust, the static analyzer, and the transformer. Our evaluation shows that the transformer efficiently and correctly transforms real-world programs and the analyzer outperforms the state-of-the-art static analyzer in terms of both speed and precision. Specifically, they transform and analyze 66 KLOC in 2.6 seconds and 4.3 seconds, respectively, and translate 74% of real-world programs to compilable code (§VI).

We discuss related work (§VII) and conclude (§VIII). Our implementation and evaluation data are publicly available [34].

II. BACKGROUND: LOCK APIS OF C AND RUST

A. Lock API of C

The most widely used C lock API, pthreads [16], provides three types of lock: mutexes, read-write locks, and spin locks. While all of them are within the scope of this work, we mainly discuss mutexes throughout the paper. The others are similar to mutexes and briefly discussed in §VI-A.

The API provides the `pthread_mutex_lock` and `pthread_mutex_unlock` functions, each of which takes a pointer to a lock; the former acquires the lock, and the latter releases the lock. Locks are used together with shared data. C programs have two common patterns to organize locks and shared data: *global* and *struct* [55].

1) Global: Both data and lock are global variables.

```
int n = ...; pthread_mutex_t m = ...;
void inc() {
    pthread_mutex_lock(&m); n += 1;
    pthread_mutex_unlock(&m);
}
```

The global variable `n` is a shared integer and `m` is a lock. Each thread must hold `m` when accessing `n`, *i.e.*, `m` protects `n`. Thus, `inc` acquires and releases `m` before and after increasing `n`.

2) Struct: Both data and lock are fields of the same struct.

```
struct s { int n; pthread_mutex_t m; };
void inc(struct s *x) {
    pthread_mutex_lock(&x->m); x->n += 1;
    pthread_mutex_unlock(&x->m);
}
```

The lock stored in the field `m` of a struct `s` value protects the integer stored in the field `n` of the same struct value. Each thread must hold `x->m` when accessing `x->n`.

B. Data Races in C

The C lock API does not guarantee whether programs use locks correctly. Data races may occur by mistake despite the use of locks. There are two major reasons for data races: *data-lock mismatches* and *flow-lock mismatches*.

1) Data-Lock Mismatch: A data-lock mismatch is an acquisition of an incorrect lock when accessing shared data. See the following where `m1` protects `n1` and `m2` protects `n2`:

```
pthread_mutex_lock(&m2); n1 += 1;
pthread_mutex_unlock(&m2);
```

All the other parts of the program acquire `m1` when accessing `n1`. However, the above code has a bug: it acquires `m2`, instead of `m1`, when accessing `n1`. This allows multiple threads to access `n1` simultaneously, thereby incurring a data race.

2) Flow-Lock Mismatch: A flow-lock mismatch is an acquisition of a lock at an incorrect program point. Consider the following program, where `m` protects `n`:

```
void f1() { n += 1; pthread_mutex_lock(&m); ... }
void f2() { ... pthread_mutex_unlock(&m); n += 1; }
```

The function `f1` accesses `n` before acquiring `m`, and `f2` accesses `n` after releasing `m`. Both functions are buggy as they allow accesses to the shared data when the lock is not held.

C. Lock API of Rust

The Rust lock API guarantees the correct use of locks [35]. The API makes two kinds of relation explicit: *data-lock relations*, *i.e.*, which lock protects which data; *flow-lock relations*, *i.e.*, which lock is held at which program point. It naturally prevents both data-lock mismatches and flow-lock mismatches.

Rust makes the data-lock relation explicit by coupling a lock with shared data. A Rust lock is a C lock plus shared data; it can be considered a protected container for shared data. The type of a lock is `Mutex<T>`, where `T` is the type of the protected data [14]. A program can create a lock as follows:

```
static m: Mutex<i32> = Mutex::new(0);
```

making `m` a lock initially containing `0`. The coupling of data and a lock prevents data-lock mismatches. When accessing shared data, threads acquire the lock coupled with the data.

Rust makes the flow-lock relations explicit by introducing the notion of a *guard*. Threads need a guard to access the in-lock data. A guard is a special kind of pointer to the in-lock data. The only way to create a guard is to acquire a lock. The `lock` method of a lock produces a guard as a return value. Threads can access the protected data by dereferencing the guard. When a thread wants to release a lock, it drops, *i.e.*, deallocates, the guard by calling `drop`. A predefined drop handler attached to the guard automatically releases the lock. The following shows the process from construction to destruction of a guard:

```
let mut g = m.lock().unwrap(); *g += 1; drop(g);
```

Because `lock` returns a wrapped guard, `unwrap()` is required. The `unwrap` call fails and makes the current thread panic when a thread previously holding the lock has panicked before releasing it. Otherwise, `unwrap` returns the guard.
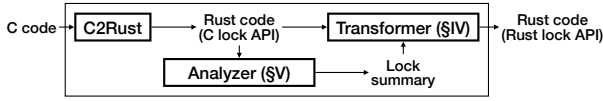
Fig. 1: Workflow of Concrat

With the help of Rust's ownership type system [35], guards prevent flow-lock mismatches. In Rust, each function can use only the variables it owns. A function owns a variable after initializing it and loses the ownership after passing the variable to another function as an argument. The type checker detects every flow-lock mismatch at compile time by tracking the ownership of guards. Consider the following buggy code:

```
fn f1() {let mut g; *g += 1; g = m.lock().unwrap();}
fn f2() { let mut g; ... drop(g); *g += 1; }
```

Because `f1` uses `g` before owning it and `f2` uses `g` after losing the ownership, the type checker rejects both functions.

III. MOTIVATION

C-to-Rust translation should ideally replace the C lock API with the Rust lock API because implementing concurrent programs in Rust benefits mostly from the thread safety guaranteed by the Rust lock API. From this perspective, C2Rust [58], the state-of-the-art C-to-Rust automatic translator, is unsatisfactory. It syntactically translates C code to Rust code. As a result, the C lock API remains in C2Rust-generated code. For example, the result of translating the first C code snippet in §II-A with C2Rust is as follows:

```
static mut n: i32 = ...;
static mut m: pthread_mutex_t = ...;
fn inc() {
    pthread_mutex_lock(&mut m); n += 1;
    pthread_mutex_unlock(&mut m);
}
```

To narrow the gap between C2Rust-generated code and desirable Rust code, we present the problem of *replacing the C lock API in C2Rust-generated code with the Rust lock API*. Solving this problem will significantly reduce programmers' burden when rewriting legacy concurrent programs in Rust.

Our solution is Concrat, an automatic C-to-Rust translator for concurrent programs. Fig. 1 shows the workflow of Concrat. It takes C code and translates it using C2Rust. A static analyzer generates a lock summary of the Rust code (§V). A transformer converts the C lock API to the Rust lock API using the summary (§IV). Concrat outputs the transformed code.

IV. TRANSFORMATION OF C2RUST-GENERATED CODE

This section proposes an automatic Rust code transformer that takes C2Rust-generated code and its lock summary as inputs and replaces the C lock API with the Rust lock API. We first describe the contents of a lock summary (§IV-A) and then show how the transformer replaces the C lock API with the Rust lock API using the summary (§IV-B).

A. Lock Summary

A program's lock summary abstracts its data-lock and flow-lock relations. A lock summary is a JSON file containing three maps: `global_lock_map`, `struct_lock_map`, and

```
1 static mut n: i32 = 0;
2 static mut m: pthread_mutex_t = ...;
3 struct s { n: i32, m: pthread_mutex_t }
4 fn f() {
5     pthread_mutex_lock(&mut m);
6     n += 1; pthread_mutex_unlock(&mut m); }
7 fn unlock() { pthread_mutex_unlock(&mut m); }
8 fn lock() {
9     pthread_mutex_lock(&mut m); }
10 fn g() {
11     lock();
12     n += 1; unlock(); }
13 fn foo() {
14     n += 1; // safe for some reason
15     pthread_mutex_lock(&mut m);
16     n += 1;
17     pthread_mutex_unlock(&mut m); }
```
Listing 1: Before transformation

```
{ "global_lock_map": { "n": "m" },
  "struct_lock_map": { "s": { "n": "m" } },
  "function_map": {
    "unlock": {
      "entry_lock": ["m"], "return_lock": [], ... },
    "lock": {
      "entry_lock": [], "return_lock": ["m"], ... },
    "foo": { ... "lock_line": { "m": [16, 17] }, },
    ... }}
```
Listing 2: Lock summary

```
1 struct mData { n: i32 }
2 static mut m: Mutex<mData> = Mutex::new(mData{n:0});
3 struct smData {n:i32}   struct s {m:Mutex<smData>}
4 fn f() { let mut m_guard;
5     m_guard = m.lock().unwrap();
6     (*m_guard).n += 1; drop(m_guard); }
7 fn unlock(m_guard: MutexGuard<i32>) {drop(m_guard);}
8 fn lock() -> MutexGuard<i32> { let mut m_guard;
9     m_guard = m.lock().unwrap(); m_guard }
10 fn g() { let mut m_guard;
11     m_guard = lock();
12     (*m_guard).n += 1; unlock(m_guard); }
13 fn foo() { let mut m_guard;
14     m.get_mut().n += 1; // safe for some reason
15     m_guard = m.lock().unwrap();
16     (*m_guard).n += 1;
17     drop(m_guard); }
```
Listing 3: After transformation

`function_map`. The first two represent data-lock relations, and the last one represents flow-lock relations. Listing 1 is C2Rust-generated code, and Listing 2 is its lock summary. We describe each component of the summary in detail.

1) Global Lock Map: `global_lock_map` expresses data-lock relations of the global pattern (§II-A1). It is a map from a global variable to the lock variable protecting it. The summary states that `n` is protected by `m`.

2) Struct Lock Map: `struct_lock_map` keeps data-lock relations of the struct pattern (§II-A2). It maps a struct type name to its summary, which maps a field to the lock field protecting it. The summary states that the field `n` of the struct type `s` is protected by the field `m`.

3) Function Map: `function_map` expresses the flow-lock relation. It maps a function name to its summary, which consists of `entry_lock`, `return_lock`, and `lock_line`. Locks are represented by symbolic paths. For example, `m` and `x.m` are locks, where the type of the variable `x` is `s`.

a) Entry Lock: `entry_lock` is a list of locks that are always held at the entry of the function. The summary states that `m` is always held at the entry of `unlock`.

b) Return Lock: `return_lock` is a list of locks that are always held at the return of the function. The summary states that `m` is always held at the return of `lock`.

c) Lock Line: `lock_line` is a map from a lock to a list of lines in the function where the lock is held. The summary states that `m` is held in lines 16 and 17.

B. Transformation

The transformer produces Listing 3 by replacing the C lock API in Listing 1 with the Rust lock API. Note that each line of Listing 3 corresponds to the same line of Listing 1. We explain the transformation line-by-line.

- Lines 1 to 2: We check `global_lock_map` to identify locks and variables they protect. We define a new struct containing variables protected by a certain lock and replace the original C lock with a Rust lock containing a struct value.
- Line 3: Similar to the above, but using `struct_lock_map`, instead of `global_lock_map`.
- Lines 4 to 6: We define an uninitialized guard variable at the beginning of each function using the guard. Each `pthread_mutex_lock` and `pthread_mutex_unlock` call is syntactically transformed into a `lock` method call and a `drop` function call, respectively. Note that the name of a guard is syntactically determined from the name of the lock according to a predefined rule. We replace each expression accessing protected data with an expression dereferencing a guard, whose lock name is found in `global_lock_map` or `struct_lock_map`, depending on the access path.
- Line 7: We make a function take a guard as an argument if its `entry_lock` is nonempty.
- Lines 8 to 9: We make a function return a guard if its `return_lock` is nonempty. If there are multiple return guards or the original return value, tuples are constructed.
- Lines 10 to 12: We add a guard as an argument to a call to a function with nonempty `entry_lock`. We assign the return value of a function with nonempty `return_lock` to a guard variable.
- Lines 13 to 17: Even when `m` protects `n`, some accesses to `n` may not hold `m` because the developer thinks that `n` is never concurrently accessed by other threads in those specific lines. For this reason, we cannot blindly replace all the accesses to protected data with guard dereference. We need to figure out whether a certain guard exists in each line by checking `lock_line`. Since the summary states that `m` is held only in lines 16 and 17, the access in line 14 uses the `get_mut` method, instead of the guard. The method returns a pointer to the in-lock data.

  Note that the use of `get_mut` relies on that `m` is defined mutable. If `m` is immutable, the type checker disallows calling `get_mut`. In Rust, mutable global variables are discouraged [10]. Reference-counted types, `Rc` [11] (in the sequential setting) and `Arc` [12] (in the concurrent setting), should replace mutable global variables. This makes

`get_mut` succeed if the reference count equals one and panic otherwise, consequently preventing data races at run time even when the developer's assumption is wrong. Automatically replacing mutable global variables with `Rc` and `Arc` is beyond the scope of this work.

The transformed code looks similar to human-written code because the transformer utilizes code patterns that real-world Rust programmers use, e.g., putting fields into structs protected by locks, storing guards in variables, passing guards as arguments, and returning guards from functions. Still, there are some discrepancies: humans may prefer wrapping guards in structs and defining their methods instead of functions taking guards; they often omit `drop` calls at the end of a function, which can be automatically inserted by the compiler.

V. SUMMARY GENERATION VIA STATIC ANALYSIS

In this section, we propose a static analysis to automatically generate lock summaries required by the transformer. The analysis must precisely determine the flow-lock relation to lead the transformer to produce compilable code. If a summary contains an imprecise flow-lock relation, the transformed code may be uncompilable due to the use of unowned guards. On the other hand, an imprecise data-lock relation does not hinder the transformed code from being compiled. If the analysis fails to find that `m` protects `n`, `n` will not be a field of a struct protected by `m`. If the analysis incorrectly concludes that `m` protects `n`, `n` will be accessed via `get_mut`. Both kinds of code are unideal but compilable and preserve the original semantics. We thus focus on designing an analysis that precisely computes the flow-lock relation.

The key intuition behind our analysis design is that the precision of the analysis does not need to exceed that of the type checker. Consider the following example:

```
if b { pthread_mutex_lock(&m); } ...
if b { pthread_mutex_unlock(&m); }
```

Even when the analysis is precise enough to track the path-sensitive use of locks, the transformed code is uncompilable:

```
let mut m_guard;
if b { m_guard = m.lock().unwrap(); } ...
if b { drop(m_guard); }
```

Since type checking is path-insensitive, it considers `m_guard` possibly uninitialized in the last line. This shows that a path-insensitive analysis is enough. Similarly, our analysis can be context-insensitive as the type checker is context-insensitive.

This intuition makes our analysis distinct from existing techniques: it is tailored to efficiently generate precise summaries for the code transformation by aiming the same precision as the type checker. Existing ones are either too imprecise or too precise. Some overapproximate the behavior of a program too much, so using their results as summaries would make the transformed code uncompilable. Some unnecessarily adopt rich techniques to make their results precise, thereby failing to finish the analyses in a reasonable amount of time.

Note that aiming the same precision as the type checker does not mean that we repeat the work of the type checker.

While the goal of the type checker is to validate the use of guards, our goal is to infer the use of guards, which is more difficult. Specifically, the type checker takes code that already has guards and checks whether it uses guards properly in terms of ownership, but our analyzer takes code without any guards and reconstructs the flow of guards to determine whether each function needs to take or return certain guards.

Since guards are more concrete than information that certain locks are held, guards often make our explanation intuitive. Thus, we sometimes use guards in the explanation although the code being analyzed does not have any guards. The existence of a guard at a certain program point is equivalent to the corresponding lock always being held at the program point, and the term guard is exchangeable for the term *held lock*.

Our analysis consists of four phases: call graph construction (§V-A), bottom-up dataflow analysis (§V-B), top-down data fact propagation (§V-C), and data-lock relation identification (§V-D). The call graph is required for both bottom-up analysis and top-down propagation. The bottom-up analysis and the top-down propagation collectively compute the flow-lock relation. Using the flow-lock relation, the last phase computes the data-lock relation.

A. Call Graph Construction

We draw call graphs by collecting the function names called in each function, without expensive control flow analysis. The drawback is that the call graph misses edges created by function pointers. However, the number of such edges is usually small because function pointers are rarely used in practice, and the subsequent analyses remain precise enough. Each node is a user-defined function; all the library functions are excluded from the graph. Therefore, each leaf node calls zero or more library functions but no user-defined functions.

We identify all the strongly connected components in the call graph to find mutually recursive functions, which need special treatment during the bottom-up analysis. We create a *merged* version of the call graph by merging each strongly connected component into a single node. We keep both original and merged call graphs to use the former for the top-down propagation and the latter for the bottom-up analysis.

B. Bottom-Up Dataflow Analysis

The goal of the bottom-up analysis is to identify the *minimum entry lock set* (MELS) and the *minimum return lock set* (MRLS) of each function. They are locks that must be held at the entry and the return, respectively. To compute the MELS and MRLS of each function, we perform two dataflow analyses on each function: *live guard analysis* (LGA) and *available guard analysis* (AGA). LGA computes MELSs, and AGA computes MRLSs. We need the control flow graph of each function for the analyses. The nodes are statements of the function, with two special nodes, entry and ret, which denote the entry and the return, respectively.

We traverse the merged call graph in post order to find the analysis target. It allows us to analyze leaf nodes first (§V-B1) and then use their results to analyze internal nodes (§V-B2). Each node contains a single function or a set of mutually

recursive functions. We discuss the analysis of non-recursive functions first and recursive functions afterward (§V-B3).

1) Leaf Node:

a) Live Guard Analysis: The goal of LGA is to compute MELSs. It is similar to the well-known live variable analysis [37]. Just like that the live variable analysis computes variables to be used in the future, LGA computes guards to be consumed by `pthread_mutex_unlock` in the future. Live guards at the entry of a function are the MELS of the function.

The analysis is a backward may analysis. Each `pthread_mutex_unlock` call, which consumes a guard, generates a guard. Each `pthread_mutex_lock` call, which produces a guard, kills a guard. The dataflow equations are defined as follows:

$$\text{In}^L_s = (\text{Out}^L_s - \text{Kill}^L_s) \cup \text{Gen}^L_s$$
$$\text{Out}^L_s = \begin{cases} \emptyset & \text{if } s = \text{ret} \\ \bigcup_{t \in \text{Succ}_s} \text{In}^L_t & \text{otherwise} \end{cases}$$
$$\text{Gen}^L_s = \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_unlock}(p) \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{Kill}^L_s = \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_lock}(p) \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{MELS} = \text{In}^L_{\text{entry}}$$

where s and t range over statements; p ranges over paths; Succ_s denotes the set of every successor of s.

Example 1. The MELS of the following function is $\{\texttt{m}\}$:

```
fn unlock() { pthread_mutex_unlock(&mut m); }
```

Example 2. The MELS of the following function is $\{\texttt{m}\}$:

```
fn may_unlock() {
    if ... { pthread_mutex_unlock(&mut m); }
}
```

We get $\{\texttt{m}\}$ by $\{\texttt{m}\} \cup \emptyset$ because LGA is a may analysis. If it was a must analysis, MELS would be $\emptyset$, making the function take no guard after the transformation. Then, the function is uncompilable as it drops an unexisting guard.

b) Available Guard Analysis: The goal of AGA is to compute MRLSs. It is similar to the well-known available expression analysis [37]. Just like that the available expression analysis identifies expressions whose values have been computed in the past, AGA identifies guards constructed by `pthread_mutex_lock` in the past. Available guards at the return of a function are the MRLS of the function.

The analysis is a forward must analysis. Each `pthread_mutex_lock` call generates a guard, and each `pthread_mutex_unlock` call kills a guard. The dataflow equations are defined as follows:

$$\text{Out}^A_s = (\text{In}^A_s - \text{Kill}^A_s) \cup \text{Gen}^A_s$$
$$\text{In}^A_s = \begin{cases} \text{MELS} & \text{if } s = \text{entry} \\ \bigcap_{t \in \text{Pred}_s} \text{Out}^A_t & \text{otherwise} \end{cases}$$
$$\text{Gen}^A_s = \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_lock}(p) \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{Kill}^A_s = \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_unlock}(p) \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{MRLS} = \text{Out}^A_{\text{ret}}$$

where Pred_s denotes the set of every predecessor of s.

Example 3. The MRLS of the following function is $\{m\}$:

```
fn lock() { pthread_mutex_lock(&mut m); }
```

Example 4. The MRLS of the following function is $\emptyset$:

```
fn may_lock() {
    if ... { pthread_mutex_lock(&mut m); }
}
```

We get $\emptyset$ by $\{m\} \cap \emptyset$ because AGA is a must analysis. If it was a may analysis, MRLS would be $\{m\}$, making the function return a guard after the transformation, which is uncompilable as it returns a possibly uninitialized guard.

Example 5. Both MELS and MRLS of the following are $\{m\}$:

```
fn unlock_and_lock() {
    if ... {
        pthread_mutex_unlock(&mut m); ...
        pthread_mutex_lock(&mut m);
    }
}
```

We get $\{m\}$ as the MRLS by intersecting $\{m\}$ and $\{m\}$. It is because setting $\mathsf{In}^{A}_{\mathsf{entry}}$ to MELS allows m to be available even in the path where the condition is false. If $\mathsf{In}^{A}_{\mathsf{entry}}$ was $\emptyset$, the MRLS would be $\emptyset$, making the transformed function not return an existing guard.

2) Internal Node: Analysis of internal nodes should consider the MELSs and MRLSs of callees. A function with a nonempty MELS consumes guards and acts like `pthread_mutex_unlock`. A function with a nonempty MRLS produces guards, like `pthread_mutex_lock`. Thus, during LGA, calling f kills MRLS_f and generates MELS_f, and during AGA, calling f kills MELS_f and generates MRLS_f.

Example 6. The MELS of `unlock2` is $\{m\}$.

```
fn unlock() { pthread_mutex_unlock(&mut m); }
fn unlock2() { unlock(); }
```

When structs are involved, we need to consider aliasing through argument passing. A caller and a callee represent the same lock with different paths if the path being an argument is different from the name of the corresponding parameter. Unless we recompute paths to reflect aliasing, the analyses produce incorrect results.

In this regard, we define *alias*, which recomputes paths:

$$alias(p, [x_1, \cdots, x_n], [e_1, \cdots, e_n]) = \begin{cases} e_i.p' & \text{if } p = x_i.p' \\ p & \text{otherwise} \end{cases}$$

It takes a path, a parameter list, and an argument list. If the path has one of the parameters as a prefix, *alias* replaces the prefix with the corresponding argument. Otherwise, the path remains the same. For example, $alias(\texttt{a.m}, [\texttt{a}], [\texttt{b}])$ equals $\texttt{b.m}$. Since we have a set of paths, we extend the definition of *alias* to recompute each path in a given set:

$$alias(\{\cdots, p, \cdots\}, \overline{x}, \overline{e}) = \{\cdots, alias(p, \overline{x}, \overline{e}), \cdots\}$$

An overlined symbol denotes a list. We revise our dataflow equations to handle user-defined function calls correctly:

$$\begin{aligned} \text{If } s = f(\overline{e}), \ \mathsf{Gen}^L_s = \mathsf{Kill}^A_s &= alias(\mathsf{MELS}_f, \mathsf{Params}_f, \overline{e}) \\ \mathsf{Kill}^L_s = \mathsf{Gen}^A_s &= alias(\mathsf{MRLS}_f, \mathsf{Params}_f, \overline{e}) \end{aligned}$$

where Params_f denotes the parameter list of f.

Example 7. The MELS of `lock_and_unlock` is $\emptyset$.

```
fn unlock(a: *mut s) {
    pthread_mutex_unlock(&mut (*a).m);
}
fn lock_and_unlock(b: *mut s) {
    pthread_mutex_lock(&mut (*b).m); unlock(b);
}
```

While the MELS of `unlock` is $\{a.m\}$, the `unlock` call in `lock_and_unlock` generates `b.m`, which is killed by the preceding `pthread_mutex_lock` call.

3) Recursive Function: Analysis of a recursive function is challenging because it requires the MELS and MRLS of the function being analyzed. Our solution is an iterative analysis.

In the beginning, we have no information and set the MELS and MRLS to the bottom values: $\mathsf{MELS} = \emptyset$ and $\mathsf{MRLS} = \mathcal{L}$, the set of every possible lock path. For the MELS, $\emptyset$ is the bottom because LGA is a may analysis. On the other hand, $\mathcal{L}$ is the bottom for the MRLS because AGA is a must analysis.

We iteratively find a fixed point to compute the correct MELS and MRLS. We analyze the function with the MELS and MRLS we have. After the analysis, we update them with the result of the analysis. We repeat this until no change.

Example 8. The MELS of the following function is $\{m\}$.

```
fn unlock(n: i32) {
    if n <= 0 { pthread_mutex_unlock(&mut m); }
    else { unlock(n - 1); }
}
```

The first iteration gives us $\mathsf{MELS} = \{m\}$ by $\{m\} \cup \emptyset$. The second iteration produces the same result by $\{m\} \cup \{m\}$ and reaches a fixed point.

Example 9. The MRLS of the following function is $\{m\}$.

```
fn lock(n: i32) {
    if n <= 0 { pthread_mutex_lock(&mut m); }
    else { lock(n - 1); }
}
```

The first iteration makes $\mathsf{MRLS} = \{m\}$ by $\{m\} \cap \mathcal{L}$. The second iteration computes $\{m\} \cap \{m\}$, reaching a fixed point.

The iteration is guaranteed to terminate if $\mathcal{L}$ is finite. During the iteration, MELS can only grow, and MRLS can only shrink. Thus, the number of iterations is bounded by the size of $\mathcal{L}$. The iteration terminates almost always in practice. In most programs, $\mathcal{L}$ is finite, and the termination is guaranteed. However, some programs have a recursive data structure with locks, which makes $\mathcal{L}$ infinite. That said, a recursive function interacting with an unbounded number of locks is rare in practice, so the iteration can terminate despite $\mathcal{L}$ being infinite.

We can easily generalize this approach to mutually recursive functions. Given a set of mutually recursive functions, $f_1, \cdots, f_n$, we set all the MELSs and MRLSs to the bottom values: $\mathsf{MELS}_{f_i} = \emptyset$ and $\mathsf{MRLS}_{f_i} = \mathcal{L}$. We then analyze each function and update them with the results. We repeat the analysis until none of them change.

C. Top-Down Data Fact Propagation

A function summary for the transformation has to contain the *entry lock set* (ELS) and the *return lock set* (RLS) of the function. They are locks that can be always held at the entry and the return, respectively. It is important that they are different from the MELS and MRLS. The ELS contains guards

given to a function by its caller, and the MELS contains some of them, which are dropped by the function. Consequently, the MELS is always a subset of the ELS. Similarly, the RLS contains guards returned by a function to its caller, and the MRLS contains some of them, which are constructed in the function. The MRLS is always a subset of the RLS. In addition, the following equation holds: ELS − MELS = RLS − MRLS. We call this common difference the *propagated lock set* (PLS). The PLS of a function is the set of guards given from and returned to its caller.

Example 10. Both MELS and MRLS of inc are $\emptyset$, but both ELS and RLS of inc are $\{m\}$.

```
fn safe_inc() {
    pthread_mutex_lock(&mut m); inc();
    pthread_mutex_unlock(&mut m);
}
fn inc() { n += 1; }
```

We need the ELS and RLS of inc to identify the data-lock relation correctly. If we consider only the MELS and MRLS, we incorrectly conclude that m does not protect n.

The goal of the top-down data fact propagation is to compute the ELS and RLS of each function. We first compute the ELS of each function. It allows us to find the PLS by subtracting the MELS from the ELS. Then, the union of the PLS and the MRLS is the RLS.

We first collect all the available guards at each function call. The arguments of the function call are collected together to recompute paths according to aliasing. $\mathsf{Call}_{f,g}$ is the set of pairs, each of which consists of the set of available guards and the list of arguments when f calls g. Because f may call g multiple times, multiple pairs may exist. Available guards at each call are already computed during AGA. Thus, $\mathsf{Call}_{f,g}$ is:

$$\mathsf{Call}_{f,g} = \{(\mathsf{In}_s^{\mathsf{A}}, \overline{e}) \mid s = g(\overline{e}) \ \wedge \ s \text{ is in } f\}$$

We then perform a top-down dataflow analysis to compute the ELS of each function. The analysis is cheap because it does not analyze function bodies and simply propagates data facts through call edges. If a function does not have any callers, its ELS is the same as its MELS. Otherwise, its callers propagate available guards. Each caller propagates not only the available guards identified by AGA, but also the guards propagated from its own callers. We want always-propagated guards, so we compute the intersection of the guards from each caller. The dataflow equations are as follows:

$$\mathsf{ELS}_g = \begin{cases} \mathsf{MELS}_g & \text{if } \mathsf{Pred}_g = \emptyset \\ \bigcap_{f \in \mathsf{Pred}_g} \mathsf{Prop}_{f,g} & \text{otherwise} \end{cases}$$
$$\mathsf{Prop}_{f,g} = \bigcap_{(P,\overline{e}) \in \mathsf{Call}_{f,g}} alias(P \cup \mathsf{ELS}_f, \overline{e}, \mathsf{Params}_g)$$

where $\mathsf{Prop}_{f,g}$ is the set of guards propagated from f to g. We finally compute the PLS and RLS:

$$\mathsf{PLS} = \mathsf{ELS} - \mathsf{MELS} \qquad \mathsf{RLS} = \mathsf{MRLS} \cup \mathsf{PLS}$$

Example 10. (*continued*)
- $\mathsf{Call}_{\mathtt{safe\_inc},\mathtt{inc}} = \{(\{m\}, [])\}$
- $\mathsf{Prop}_{\mathtt{safe\_inc},\mathtt{inc}} = \{m\}$
- $\mathsf{ELS}_{\mathtt{inc}} = \mathsf{PLS}_{\mathtt{inc}} = \mathsf{RLS}_{\mathtt{inc}} = \{m\}$

After finishing the top-down propagation, we can generate a function summary of each function. entry_lock and return_lock are the same as the ELS and RLS, respectively. The set of locks held in each line, required by lock_line, is determined by combining available guards identified by AGA and the PLS of the function.

D. Data-Lock Relation Identification

We identify the data-lock relation from the flow-lock relation computed by the preceding analysis. The key idea is to find the lock held at each access to a certain path. Since the global pattern (§II-A1) and the struct pattern (§II-A2) require different treatments, we split paths into global variables and struct fields and compute the data-lock relation of each.

We first discuss the global pattern. The first step is to collect every access to each global variable. We record all the available guards and whether the access is read or write. The available guards are the union of those found by AGA and the PLS of the function where the access happens. Acc_x is the set of accesses to a global variable x:

$$\mathsf{Acc}_x = \{(s, \mathsf{In}_s^{\mathsf{A}} \cup \mathsf{PLS}_f, a) \mid access(s, x, a) \ \wedge \ s \text{ is in } f\}$$

where $access(s, x, \mathsf{r})$ and $access(s, x, \mathsf{w})$ hold when s reads x and s modifies x, respectively. We then find a candidate lock for each global variable. A candidate lock is a lock that is held most frequently when accessing the variable:

$$\mathsf{Cand}_x = \mathrm{argmax}_y |\{(s, P, a) \in \mathsf{Acc}_x \mid y \in P\}|$$

We split accesses into safe and unsafe ones according to the existence of the candidate lock. We consider an access safe if it happens when the candidate is held, and unsafe otherwise:

$$\mathsf{Safe}_x = \{(s, P, a) \in \mathsf{Acc}_x \mid \mathsf{Cand}_x \in P\}$$
$$\mathsf{Unsafe}_x = \{(s, P, a) \in \mathsf{Acc}_x \mid \mathsf{Cand}_x \notin P\}$$

To determine the data-lock relation, we need to check whether each statement is *concurrent*, *i.e.*, can run concurrently with other threads. The existence of an unsafe access does not necessarily mean that the candidate lock does not protect the global variable. If a statement is non-concurrent, it can safely access a global variable without holding a lock. Therefore, the precise identification of the data-lock relation requires a precise thread analysis. Since a precise thread analysis is expensive, we instead propose a simple heuristic. We consider a statement concurrent only if it belongs to a function reachable from an argument to pthread_create, the thread-spawning function.

Using the heuristic, we determine whether a candidate lock really protects the global variable. The candidate protects the variable if a safe write access exists and every unsafe access happens in a non-concurrent statement.

$$\mathsf{Cand}_x \text{ protects } x \text{ iff}$$
$$(\exists (s, P, a) \in \mathsf{Safe}_x, a = \mathsf{w}) \ \wedge$$
$$(\forall (s, P, a) \in \mathsf{Unsafe}_x, s \text{ is non-concurrent})$$

For the struct pattern, we collect accesses to each field of a struct. A candidate lock for a field must be a field in the same struct. $\mathsf{Acc}_{T,l}$ is the set of accesses to a field l in a type T, and $\mathsf{Cand}_{T,l}$ is a candidate for it:

$$\mathsf{Acc}_{T,l} = \{(s, \mathsf{In}_s^{\mathsf{A}} \cup \mathsf{PLS}_f, p, a) \mid$$
$$access(s, p.l, a) \ \wedge \ type(p) = T \ \wedge \ s \text{ is in } f\}$$
$$\mathsf{Cand}_{T,l} = \mathrm{argmax}_{l'} |\{(s, P, p, a) \in \mathsf{Acc}_x \mid p.l' \in P\}|$$
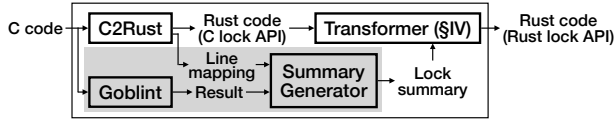
Fig. 2: Workflow of Concrat$_G$ (gray: differences from Concrat)

Fig. 3: Transformation time according to Rust LOC

We split accesses into safe and unsafe ones and check whether the candidate protects the field, just as we do for the global pattern. The only difference is that the condition for a statement to be considered concurrent is stricter than before. A struct value is not accessible from other threads right after its creation. It becomes accessible only after the function shares it with other threads by storing it in a global data structure or passing it as a thread argument. Determining when a value is shared requires a precise thread analysis as well, so we propose a heuristic. For a given struct value containing a lock field l, we consider a statement non-concurrent not only when its function is unreachable from an argument to `pthread_create`, but also when the function initializes l by calling `pthread_mutex_init`. Such a function is usually where the struct value is created and uniquely accessed.

VI. EVALUATION

Our experiments are on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM.

A. Implementation

We implemented Concrat on top of the Rust compiler [3]. The transformer lowers given code to the compiler's high-level intermediate representation [5] and walks it to replace the C lock API. The analyzer uses the compiler's dataflow analysis framework [4] for its mid-level intermediate representation [6].

Concrat handles not only mutexes, but also read-write locks, spin locks, and condition variables. Since Rust recommends using mutexes instead of spin locks [38], we replace them with `RwLock` [15], `Mutex`, and `Condvar` [13] of `std::sync`. Concrat does not support re-entrant locks because the Rust standard library does not provide them.

To compare our analyzer with the state-of-the-art static analyzer for concurrent programs, we additionally built Concrat$_G$. Fig. 2 illustrates the workflow of Concrat$_G$. It uses Goblint [52], [55] to analyze C code. Goblint computes both data-lock and flow-lock relations using abstract interpretation [26]. Since Goblint's result contains line numbers for C code, our summary generator replaces them with those for Rust code using the C-to-Rust line mappings generated by C2Rust.

Note that we can change the implementation of Concrat to analyze C code, instead of Rust code, just like Concrat$_G$, because the proposed analysis is language-agnostic. Our choice of analyzing Rust code eases implementation as we can utilize the Rust compiler's dataflow analysis framework.

B. Test Set Collection

We collected 46 real-world concurrent C programs, all of the public GitHub repositories satisfying the following conditions: 1) more than 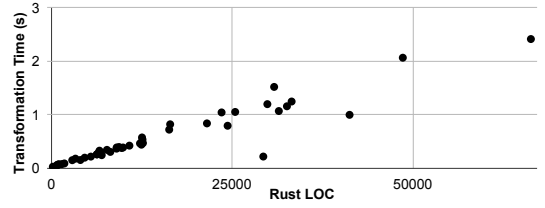1,000 stars, 2) not a study material, 3) using the pthread lock API at least once, 4) C code less than 500,000 bytes, and 5) translatable with C2Rust. Two projects satisfied the first four conditions but not the last; they use C11 Atomics, but C2Rust supports only C99-compliant code. When C2Rust generates uncompilable code due to wrong type casts, we included such projects after manually fixing them.

The first seven columns of Table I show the collected programs and the code size of each. The second and third columns show the lines of C code and C2Rust-generated code, respectively; the fourth to seventh columns show the numbers of `pthread_mutex_*`, `pthread_rwlock_*`, `pthread_spin_*`, and `pthread_cond_*` calls, respectively.

C. Transformation

We evaluate our transformer with the following questions:
- RQ1. Scalability: Does it transform large programs in a reasonable amount of time?
- RQ2. Applicability: Does it handle most code patterns found in real-world programs?
- RQ3. Correctness: Does it preserve the semantics of the original program?

1) Scalability: We translate the programs with Concrat to evaluate the transformer's scalability. In Table I, the eighth column shows the transformation time; the ninth and tenth show the inserted and deleted lines, measured with `diff`.

The result shows that the transformer is scalable. As Fig. 3 shows, the transformation time is proportional to the Rust LOC, and it takes less than 2.5 seconds to transform 66 KLOC by inserting and deleting hundreds of lines.

2) Applicability: We check whether the transformer handles diverse code patterns in real-world programs to evaluate its applicability. We consider that the transformer successfully handles a certain pattern if the transformed code is compilable. In Table I, the eleventh column shows compilability; the twelfth shows the reason for a failure; the thirteenth shows our manual fix for the original code to make compilation succeed.

The transformer has high applicability. Among 46, 29 are compilable, 5 are compilable requiring manual fixes, and 12 are not. Overall, 74% of the programs are compilable.

a) Failures: We manually investigated the reasons for compilation failures and found three code patterns.
- *Conditional acquisitions (cond acq)*: A function conditionally acquires a lock. Consider the following code:

```
fn may_lock() -> i32 {
    if b {pthread_mutex_lock(&mut m); 0} else {1}
}
if may_lock() == 0 {
```

TABLE I: Test set and experimental results

Project	Size						Transformation									Analysis		
	C LOC	Rust LOC	Mutex	Rwlock	Spin	Cond	Time	Insertion	Deletion	Succ	Reason	Fix	Test	Test$_C$	Test$_O$	Time$_O$	Time$_G$	Succ$_G$
AirConnect	17516	32565	88	0	0	14	1.159	870	874	✗	cond acq					1.701	timeout	
axel	5848	7685	16	0	0	0	0.344	78	112	✓						0.409	error	
brubeck	5635	6769	15	0	17	0	0.289	99	91	✓			✓	✓	✓	0.362	80.393	✗
C-Thread-Pool	710	791	23	0	0	7	0.059	106	113	✓			✓	✓	✓	0.060	0.728	✓
cava	4768	6538	10	0	0	0	0.289	29	27	✓						0.334	72.908	✓
Cello	20885	30796	5	0	0	0	1.521	27	15	✗	func ptr					3.103	error	
Chipmunk2D	16053	21509	12	0	0	10	0.839	52	60	✓						2.074	error	
clib	25073	66287	38	0	0	0	2.416	193	207	✗	func ptr					4.234	timeout	
dnspod-sr	9259	12596	0	0	99	0	0.537	272	234	✗→✓	dead					0.696	1667.203	✗
dump1090	4646	6281	9	0	0	6	0.259	44	77	✓			✓	✓	✓	0.307	timeout	
EasyLogger	2011	29298	4	0	0	0	0.219	428	415	✗	cond acq					0.234	6.458	✗
fzy	2621	4013	4	0	0	0	0.154	16	16	✓			✓	✓	✓	0.168	error	
klib	716	1016	14	0	0	14	0.076	62	100	✓			✓	✓	✓	0.078	error	
kona	38850	48583	10	0	0	0	2.067	37	31	✓			✓	✓	✓	3.223	timeout	
level-ip	5414	6651	36	23	0	4	0.329	232	317	✗	func ptr					0.442	timeout	
libaco	1282	1800	6	0	0	0	0.090	22	33	✓			✓	✓	✓	0.105	timeout	
libfaketime	521	806	6	0	0	6	0.059	43	85	✓			✓	✓	✗	0.059	0.147	✓
libfreenect	627	962	10	0	0	4	0.066	87	116	✓						0.069	3.439	✓
libqrencode	6670	9013	4	0	0	0	0.367	28	40	✓			✓	✓	✓	0.447	error	
lmdb	10827	16290	27	0	0	6	0.722	266	292	✗	lock arg					0.910	error	
minimap2	17279	23531	6	0	0	4	1.044	26	56	✓			✓	✓	✓	1.438	73.902	✓
Mirai-Source-Code	1839	2889	14	0	0	0	0.151	118	135	✗→✓		goto				0.164	43.736	✗→✓
neural-redis	3645	6312	12	0	0	0	0.261	51	59	✓						0.310	186.593	✓
nnn	12091	16424	7	0	0	0	0.822	37	66	✓						1.056	2264.742	✓
pg_repack	7420	8152	10	0	0	0	0.306	36	35	✓			✓	✓	✓	0.384	error	
phpspy	19390	29860	8	0	0	10	1.199	1441	1479	✗	cond acq					1.580	error	
pianobar	11452	33212	45	0	0	17	1.248	132	182	✓						1.624	timeout	
pigz	9118	12660	5	0	0	7	0.471	176	178	✗→✓		no ret	✓	✓	✓	0.615	error	
pingfs	2318	3332	26	0	0	6	0.180	110	137	✗	cond acq					0.198	9.278	✗
ProcDump-for-Linux	4152	6961	31	0	0	11	0.245	171	438	✓			✗			0.286	79.284	✗
proxychains	2686	5460	6	0	0	0	0.218	44	52	✓						0.223	49.203	✓
proxychains-ng	5203	9031	8	0	0	0	0.389	24	32	✓						0.444	1058.823	✓
Remotery	7212	9361	4	0	0	0	0.397	34	24	✗	cond acq					0.567	error	
sc	142	206	7	0	0	0	0.029	34	54	✓			✓	✓	✓	0.030	0.049	✓
shairport	8605	12533	37	0	0	0	0.576	1093	1046	✗	cond acq					0.736	error	
siege	19281	25412	21	0	0	16	1.053	202	293	✗	lock arg					1.882	error	
snoopy	2262	4605	4	0	0	0	0.198	23	38	✓			✓	✓	✓	0.234	3.855	✓
sshfs	7193	9914	75	0	0	13	0.388	277	302	✓			✓	✓	✓	0.517	2028.390	✗
streem	20169	31444	36	0	0	0	1.070	115	114	✗→✓		bug fix	✓	✓	✓	2.084	error	
stud	7931	10789	12	0	0	0	0.423	61	54	✓						0.505	error	
sysbench	16020	41222	19	10	0	9	0.999	110	244	✗→✓		goto	✓	✗		1.149	error	
the_silver_searcher	7242	12453	23	0	0	5	0.441	112	177	✓			✓	✓	✓	0.548	error	
uthash	817	1450	0	6	0	0	0.078	56	57	✓			✓	✓	✓	0.091	0.100	✓
vanitygen	10919	9710	23	0	0	11	0.379	121	224	✓						0.454	error	
wrk	8658	12255	12	0	0	0	0.465	35	42	✓						0.525	error	
zmap	17435	24366	12	0	0	0	0.795	42	71	✗	lock arg					1.257	timeout	

(Times are in seconds. Subscript C means C2Rust, O means ours, and G means Goblint.)

```
    n += 1; pthread_mutex_unlock(&mut m);
}
```

The function `may_lock` acquires a lock and returns `0` if a certain condition is satisfied, and otherwise returns `1` without acquiring the lock. Its caller accesses the shared data only when the return value is `0`. The transformer cannot handle this pattern. Since `may_lock` has empty `return_lock`, it does not return any guards after the transformation. Its caller drops an unowned guard, thereby being uncompilable. To solve this, we have to make `may_lock` return `Option` [7] of a guard. Conditionally-succeeding functions appear in most C programs, not only in concurrent ones. Translating them to functions returning `Option` will be promising future work.

- *Function pointers (func ptr)*: A function that takes or returns guards is used as a function pointer. Since adding guards changes the type of the function pointer, the transformed code is uncompilable. To address this pattern, we need to transform functions that take function pointers. Such functions are often in a library; if it is the case, the library also should be rewritten in Rust.

- *Lock arguments (lock arg)*: A function takes only a pointer to a lock as an argument. It would be an interesting

improvement to handle this pattern with generic functions. If the function does not access any data protected by a specific lock, we can make the function take an argument of `Mutex<T>`, where `T` is a type parameter.

b) Manual Fixes: We made four kinds of manual fix. We first explain two code patterns requiring manual fixes.

- *Removing `goto` (goto)*: A function uses `goto`.

```
if (b) { pthread_mutex_unlock(&m); goto err; }
n += 1; ... return 0;
err: return 1;
```

For the above code, C2Rust replaces `goto` with `if`:

```
if b { pthread_mutex_unlock(&mut m); x = 123; }
if x != 123 { n += 1; ... return 0; }
return 1;
```

Then, the transformer generates uncompilable code:

```
if b { drop(m_guard); x = 123; }
if x != 123 { *m_guard += 1; ... return 0; }
return 1;
```

Since type checking is path-insensitive, it considers `m_guard` possibly unowned in the second line. We replaced `goto` with the statements after the jump target:

```
if (b) { pthread_mutex_unlock(&m); return 1; }
n += 1; ... return 0;
```

which Concrat translates to compilable code.

- *Changing return type of no-return function (no ret)*: A function does not return. Consider the following code:

```
fn err() { exit(-1); }
if b { pthread_mutex_unlock(&mut m); err(); }
n += 1; ...
```

Since `exit` does not return, `err` does not either and accessing `n` is safe. But, the type checker does not know that `err` never returns, and the transformed code is uncompilable:

```
if b { drop(m_guard); err(); }
*m_guard += 1; ...
```

The type checker considers `m_guard` possibly unowned in the last line. We changed the return type of `err` to `!` [9]:

```
fn err() -> ! { ... exit(-1); }
```

indicating no return. The transformed code is compilable.

Both manual fixes can be avoided by improving C2Rust. We now explain program-specific fixes.

- *Removing dead code (dead)*: We deleted a function taking a pointer to a lock because it is never called.
- *Bug fix (bug fix)*: streem has the following code (simplified):

```
pthread_mutex_unlock(&m);
if (...) { pthread_mutex_unlock(&m); return; }
```

Due to the second `pthread_mutex_unlock` call, the transformed code is uncompilable. We believe it is a bug and removed it. We contacted the developer but have not received a response yet. This confirms the common belief that rewriting legacy programs in Rust can reveal unknown bugs.

3) Correctness: We ran the test cases of each program whose transformation succeeds to evaluate the correctness of the transformer. A correct transformer must preserve the semantics of the original program. We consider the transformer correct if the transformed program passes all of its test cases. The fourteenth to sixteenth columns show whether the original C program, the C2Rust-generated program, and the transformed program pass the test cases, respectively.

The result shows that our approach transforms most programs correctly. Among 34 programs compilable after the transformation possibly with manual fixes, 14 have no test cases or only those covering no lock API calls, so we performed the evaluation with the remaining 20. One fails even before C2Rust's translation. One fails after C2Rust's translation because it incorrectly translates some inline assembly. After the transformation, 17 still pass their tests, but 1 fails.

The failing program does not reveal an inherent limitation of our approach. The reason for the incorrect transformation is the imprecise `timespec` tracking of our transformer implementation. While `pthread_cond_timedwait` of the C lock API takes what time to wait for, its Rust counterpart takes how long to wait. To address this discrepancy, the transformer syntactically finds a `clock_gettime` call, which sets a given `timespec` to the current time, and how

many seconds are added to the `timespec` before calling `pthread_cond_timedwait`. However, the failing program uses multiple `timespec` values, whose relation cannot be found by our syntactic analysis. We can easily resolve this issue by adopting intraprocedural value analysis for `timespec`.

It is not surprising that the transformer is correct in most cases as far as the transformed code is compilable because the design of the transformation justifies the correctness. It transforms a `lock` function call to a `lock` method call and an `unlock` function call to the drop of a guard whose finalizer unlocks the connected lock. Since the names of lock and guard variables are syntactically related, the dropped guard always unlocks the correct lock. It transforms each lock-protected data access to field access through a guard. Again, the lock and guard names are syntactically related, so the transformed code always accesses the correct data. The only possible threat is guards being used before initialization or after destruction, but the compiler detects it.

D. Analysis

We evaluate our analyzer with the following questions:

- RQ4. Scalability: Does it analyze large programs quickly, compared to the state-of-the-art static analyzer?
- RQ5. Precision: Does it produce precise lock summaries, compared to the state-of-the-art static analyzer?

1) Scalability: We translate the collected programs with both Concrat and Concrat$_G$ and measure the analysis time to compare the scalability of our analyzer and Goblint. For Goblint, we use the `medium-program.json` configuration [2] and additionally enable the `allfuns` option to analyze every function for programs without `main`. It makes Goblint perform flow-, context-, path-sensitive analysis. We set a 24-hour time limit. The seventeenth and eighteenth columns of Table I show the time required by ours and Goblint.

The result shows that our analyzer is more scalable than Goblint. Goblint fails to analyze 19 programs due to internal errors. Our analyzer processes all the remaining 27 programs faster than Goblint. It takes less than 4.3 seconds to analyze 66 KLOC. On the other hand, Goblint does not even finish the analysis of eight programs in the time limit and takes $1.1\times$ to $3923\times$ more than ours to analyze the other 19 programs.

2) Precision: We measure the precision of lock summaries generated by our analyzer and Goblint to compare their precision. We use the compilability of code translated by Concrat and Concrat$_G$ as a proxy for assessing the analyzers' precision because an imprecise flow-lock relation makes transformed code uncompilable, as discussed in §V. The eleventh and nineteenth columns of Table I show compilability after Concrat's and Concrat$_G$'s translation, respectively.

The result shows that our analyzer is more precise than Goblint for four programs, generating summaries that made the transformed code compilable. Our manual investigation confirms that those translated by Concrat$_G$ are uncompilable due to imprecision in the flow-lock relations. Goblint's imprecision mainly stems from the imprecise lock-aliasing information. It knows locks `a` and `b` are aliased when `a` is locked, but this

information is sometimes lost due to overapproximation, and `b` is considered still locked even after `a` is unlocked, leading to imprecise flow-lock relations.

E. Threats to Validity

The primary threats to internal validity lie in the implementation of our tool. We implemented it with the dataflow analysis framework of the Rust compiler, which is already massively used and tested by the compiler.

The threats to external validity concern the choice of the translated C projects, each of which has more than 1,000 stars and C code of fewer than 500,000 bytes. Less popular or larger projects may have code patterns unseen in the selected projects. Further experiments with more C projects can give more confidence in the generalizability of our approach.

The threats to construct validity include evaluation metrics. We used whether a compilation succeeds or not as a proxy for the applicability of the transformer and the precision of the analyzer. We ran test cases to evaluate the correctness of the transformer. While test cases cannot guarantee the semantics preservation of the transformation, in practice, running test cases is the most popular way to check whether a certain program has the intended semantics.

VII. Related Work

1) Transforming C2Rust-Generated Code: A few tools have been proposed to replace C features in C2Rust-Generated code with their counterparts in Rust, but they do not target the lock API. Emre et al. [28] tackled replacing C pointers in Rust with Rust pointers. Their work also requires semantic information, but they focus on pointers and sequential programs. Ling et al. [40] proposed CRustS, which contains 220 syntactic replacement rules written in the TXL transformation language [25]. It targets simpler features than this work.

2) C to Safe Languages: Many safe substitutes for C and (semi-) automatic translation to those languages have been proposed [19], [24], [27], [29]–[31], [33], [41], [48], [51]. However, they guarantee only some form of memory safety and do not provide any safe lock API. Necula et al. [48] proposed CCured, a language extending C with new kinds of pointer and the type system to statically verify or dynamically enforce the safety of each pointer. Grossman et al. [33] proposed Cyclone, which extends C with region-based memory management [54]. Cyclone's type system prevents dangling pointer dereference by annotating each pointer with a region to use the pointer. Machiry et al.'s **3C** tool [41] automatically translates C to Checked C [27], an extension of C with checked pointer types. Checked pointers guarantee the absence of null pointer dereference and out-of-bound accesses.

3) Static Analysis for Concurrent Programs: Various static analyzers for concurrent C programs have been proposed. Our analyzer analyzes concurrent Rust programs but assumes only C2Rust-generated code. While the goal of existing analyzers is to detect concurrency bugs, our goal is to efficiently generate lock summaries for our transformation. This different goal makes our design completely distinct from the others. Our analyzer targets the same precision as the Rust type checker by using context- and path-insensitive dataflow analyses. However, other analyzers perform context- or path-sensitive analyses to reduce false alarms. Goblint [52], [55] and others [45]– [47] are based on abstract interpretation [26]. RELAY [56] and SDRacer [57] utilize symbolic execution [20]. Locksmith [50] and Kahlon et al's tool [36] perform context-sensitive analyses.

A few static analyzers for concurrent Java programs [21], [39] exist. Because Java provides a syntactic `synchronized` block, a lock acquired by a certain function cannot be released by another function, which frequently happens in C.

4) API Mapping Mining: Researchers have proposed API mapping mining [44], [49], [60], which automatically discovers syntactic discrepancies, e.g., type/function/method names and receiver/parameter positions, between the APIs of different languages. Such information can be utilized by syntactic translators. It is useful when there are many API functions, and their mappings are unknown.

Unfortunately, API mapping mining is ineffective in dealing with our problem. We focus on translating concurrent programs, which significantly rely on the lock API. The lock API provides only a few functions, and their mappings can be constructed with low manual effort. Our challenge rather resides in the semantic discrepancies between C and Rust's lock APIs. The Rust lock API requires information not written in C code to be explicitly written. This work proposes a dataflow analysis technique to find the required information.

5) Learning from Code Edit Examples: To facilitate automatic code transformation, researchers have proposed techniques to learn code edit rules from code edit examples [43], [59]. However, these example-based approaches do not fit our problem, which requires code transformation that cannot be syntactically generalized. For example, to transform a lock-initializing API call, we need the name of every variable protected by the lock. While code edit examples contain variable names, the names originate from arbitrary source code locations, and we cannot syntactically generalize the examples to code edit rules regardless of the number of examples.

VIII. Conclusion

We tackle the automatic C-to-Rust translation of concurrent programs by replacing the C lock API with the Rust lock API utilizing data-lock and flow-lock relations. Our solution is a transformer replacing the lock API using a lock summary efficiently and precisely generated by our static analyzer. Our evaluation shows that the transformer is scalable, widely applicable, and correct, and the analyzer is scalable and precise.

REFERENCES

[1] Fuchsia guides: Rust. https://fuchsia.dev/fuchsia-src/development/languages/rust.

[2] Goblint documentation: configuring. https://goblint.readthedocs.io/en/latest/user-guide/configuring.

[3] Guide to Rustc development. https://rustc-dev-guide.rust-lang.org/.

[4] Guide to Rustc development: Dataflow analysis. https://rustc-dev-guide.rust-lang.org/mir/dataflow.html.

[5] Guide to Rustc development: The HIR. https://rustc-dev-guide.rust-lang.org/hir.html.

[6] Guide to Rustc development: The MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html.

[7] Module std::option. https://doc.rust-lang.org/std/option/.

[8] Module std::sync. https://doc.rust-lang.org/stable/std/sync/index.html.

[9] The Rust programming language: The never type that never returns. https://doc.rust-lang.org/book/ch19-04-advanced-types.html#the-never-type-that-never-returns.

[10] The Rust reference: Static items—mutable statics. https://doc.rust-lang.org/reference/items/static-items.html#mutable-statics.

[11] Struct std::rc::Rc. https://doc.rust-lang.org/std/rc/struct.Rc.html.

[12] Struct std::sync::Arc. https://doc.rust-lang.org/std/sync/struct.Arc.html.

[13] Struct std::sync::Condvar. https://doc.rust-lang.org/stable/std/sync/struct.Condvar.html.

[14] Struct std::sync::Mutex. https://doc.rust-lang.org/stable/std/sync/struct.Mutex.html.

[15] Struct std::sync::RwLock. https://doc.rust-lang.org/stable/std/sync/struct.RwLock.html.

[16] IEEE standard for information technology—portable operating system interface (POSIX). *IEEE Std. 1003.1-2017*, 2017.

[17] The Rust programming language. http://rust-lang.org/, 2022.

[18] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1):4, 2017.

[19] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 290–301, New York, NY, USA, 1994. Association for Computing Machinery.

[20] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018.

[21] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[22] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[23] Thomas Claburn. Linus Torvalds says Rust is coming to the Linux kernel 'real soon now'. https://www.theregister.com/2022/06/23/linus_torvalds_rust_linux_kernel/, 2022.

[24] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 520–535, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[25] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[26] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[27] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018.

[28] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[29] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, nov 2006.

[30] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 313–323, New York, NY, USA, 1998. Association for Computing Machinery.

[31] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 70–80, New York, NY, USA, 2001. Association for Computing Machinery.

[32] Manish Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html, 2017.

[33] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 282–293, New York, NY, USA, 2002. Association for Computing Machinery.

[34] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs (artifact). https://doi.org/10.5281/zenodo.7573490, January 2023.

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[36] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 13–22, New York, NY, USA, 2009. Association for Computing Machinery.

[37] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[38] Aleksey Kladov. Spinlocks considered harmful. https://matklad.github.io/2020/01/02/spinlocks-considered-harmful.html, 2020.

[39] Yanze Li, Bozhen Liu, and Jeff Huang. SWORD: A scalable whole program race detector for Java. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 75–78. IEEE Press, 2019.

[40] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed 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)*, pages 354–355, 2022.

[41] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to Checked C by 3C. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[42] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511, 2013.

[44] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, 2012.

[45] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 398–418, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[46] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 39–58, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[47] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking static analysis to the next level: proving the absence of run-time errors and data races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[48] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.

[49] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 457–468, New York, NY, USA, 2014. Association for Computing Machinery.

[50] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 320–331, New York, NY, USA, 2006. Association for Computing Machinery.

[51] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing.

[52] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving thread-modular abstract interpretation. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings*, page 359–383, Berlin, Heidelberg, 2021. Springer-Verlag.

[53] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2021.

[54] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[55] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: The Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 205–214, New York, NY, USA, 2007. Association for Computing Machinery.

[57] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 113–124, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Frances Wingerter. C2Rust is back. https://immunant.com/blog/2022/06/back/, 2022.

[59] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 335–346, 2019.

[60] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 195–204, New York, NY, USA, 2010. Association for Computing Machinery.