GenC2Rust: Towards Generating Generic Rust Code from C

Xiafa Wu
University of California, Irvine
Irvine, California, U.S.A
xiafaw@uci.edu

Brian Demsky
University of California, Irvine
Irvine, California, U.S.A
bdemsky@uci.edu

***Abstract*—Rust provides an exciting combination of strong safety guarantees and high performance. Many new systems are being implemented in Rust. Nevertheless, there is a large body of existing C code that could greatly benefit from Rust's safety guarantees. Unfortunately, the manual effort required to rewrite C code into Rust is often prohibitively expensive.**

Researchers have explored tools to assist developers in translating legacy C code into Rust code. However, the mismatch between C abstractions and idiomatic Rust abstractions makes it challenging to automatically utilize Rust's language features, resulting in non-idiomatic Rust code that requires extensive manual effort to further refactor. For example, existing tools often fail to map polymorphic uses of void pointers in C to Rust's generic pointers. In this paper, we present a translation tool, GenC2Rust, that translates non-generic C code into generic Rust code. GenC2Rust statically analyzes the use of `void` pointers in the C program to compute the typing constraints and then retypes the parametric polymorphic `void` pointers into generic pointers. We conducted an evaluation of GenC2Rust across 42 C programs that vary in size and span multiple domains to demonstrate its scalability as well as correctness. We discovered GenC2Rust has translated 4,572 `void` pointers to use generics. We also discuss the limiting factors encountered in the translation process.

I. Introduction

The Rust programming language has attracted significant adoption for new software development due to its combination of static safety guarantees with performance. Rust combines a strong type system with a practical design that allows users to escape the type system via *unsafe* code when the type system is overly restrictive. A common usage pattern for unsafe Rust code is to implement a component's key internal functionality that cannot be efficiently expressed in safe Rust using unsafe code, and then provide an external interface that can be checked by the Rust type system. There have been efforts to verify key unsafe components using other techniques [1]. Despite the limitations of unsafe Rust, Rust represents a major advancement for developing reliable system code.

One issue that has limited the use of the Rust programming language is that while there are many legacy code bases written in C that could benefit from the safety assurances provided by Rust, the development overhead of porting legacy code bases from C to Rust often makes this impractical. To address this issue, researchers and practitioners have developed tools such as C2Rust to automatically translate C programs to Rust [2].

Researchers also have explored tools to assist in translating C programs into more idiomatic Rust code. One challenge is handling C-style raw pointers generated by tools like C2Rust. Since C2Rust does not natively retype raw pointers into Rust smart pointers such as `Box<T>` or `&T`, the translated code is not idiomatic Rust code. Researchers have explored replacing raw pointers with Rust's smart pointers using various approaches [3], [4]. The C2Rust team is also developing support for refactoring raw pointers [5]. In addition, C and Rust utilize different library APIs for various operations. Concrat [6] automatically removes the C lock APIs with Rust lock APIs to help users refactor their concurrent C programs. Different idioms in C and Rust also create challenges if high quality translated Rust code is desired. Nopcrat [7], for instance, tackles the problem of translating C code to utilize Rust's Algebraic Data Types such as `Option` and `Result`.

The problem of mapping `void` pointers from C into more idiomatic Rust remains largely untouched. Often, due to C's lack of first class support for polymorphism, developers use `void` pointers to implement this feature. C2Rust ports this use of `void` pointer into `*mut libc::c_void` Rust pointers. At this time, to refactor the polymorphic uses of `*mut libc::c_void` pointers in the Rust program to generic pointers `*mut T`, one has to first manually review the source C program to understand which `*mut libc::c_void` pointers are used as generics and instantiate the type parameters properly to produce programs that type check.

We introduce GenC2Rust, a tool that can automatically translate polymorphic C `void` pointers into Rust generics. This paper makes the following contributions:

- **Analysis for Extracting Typing Constraints from C Code**: The paper presents a static analysis to collect typing constraints introduced by the operations in the functions.
- **Algorithm to Map C Polymorphism to Rust**: We introduce an algorithm that uses the typing constraints to map polymorphic `void` pointers to generic pointers.
- **Implementation of GenC2Rust**: We developed GenC2Rust to implement the analysis and rewriting algorithm. GenC2Rust takes in C code and generates Rust code that makes appropriate use of generics.
- **Evaluation**: We have evaluated GenC2Rust on a set of benchmarks taken from Laertes [3], Crown [4], and Concrat [6]. GenC2Rust translates 424,110 lines of code

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00127

in 351.55 seconds with a average of 8.37 seconds per program. It is able to retype in total 4,572 `void` pointers, among which 3,831 pointers are type parameters.

II. Rust Generic Pointers and C `void` Pointers

We next discuss the use of polymorphic `void` pointers and Rust generics in more detail. Figure 1a presents a generic `Node struct` that can store references to arbitrary data types using a `void` pointer. For simplicity, we will temporarily ignore the second field `Node *next` since it introduces the complexity of recursive data structures, but we will return to it later when we explain how we handle self-referencing types. For the generic field, line 8 stores a `Cat` struct in a `Node` by upcasting the `Cat` struct to a `void` pointer. The corresponding downcast at line 10 allows us to retrieve the stored data from the generic container. Passing this C code to C2Rust will result in the code shown in Figure 1b. We simplified the resulting code for readability and conciseness. The key observation is that the transcompiled code does not replace the `void` pointer uses with idiomatic Rust constructs like generics.

(a) Polymorphic C Node

```c
typedef struct Node {
    void* data;
} Node;

int main() {
    Cat *cat = /* Cat alloc */;
    Node w;
    w.data = (void *) cat;
    // Some operations
    cat = (Cat *) w.data;
    return 0;
}
```

(b) C2Rust Translated Code

```rust
#[derive(Copy, Clone)]
#[repr(C)]
pub struct Node {
    pub data: *mut libc::c_void,
}
unsafe fn main_0() -> libc::c_int {
    let mut cat: *mut Cat = /* Cat alloc */;
    let mut w: Node = /* Node init */
    w.data = cat as *mut libc::c_void;
    cat = w.data as *mut Cat;
    return 0 as libc::c_int;
}
```

Fig. 1: Polymorphic Linked List Node

This is an error-prone approach and the C compiler will not guarantee the type-safety of such usage since the type information is lost by the upcast operation. Suppose we introduce a new `struct` called `Dog`, which is incompatible with the `Cat` struct. Figure 2 demonstrates a contrived example where we can upcast a `Cat` pointer to `void *`, and then downcast it to a `Dog` pointer. This can be an issue since we are now interpreting a `Cat` struct as a `Dog`, and we can quickly run into undefined behavior once we dereference the `Dog` pointer. While this issue can be easily spotted in a small example, keeping track of such types can be error-prone in a large code base.

```c
Cat *cat = /* Cat allocation */;
Node w;
w.data = (void *) c;
// We are reinterpreting Cat as Dog
Dog *dog = (Dog *) w.data;
```

Fig. 2: Type Mismatch by Cast Example

If we were to rewrite the code from Figure 1a in Rust, we would ideally utilize Rust's support for generics and produce the code shown in Figure 3. This approach has the following benefits. First, we can utilize Rust's type system to check type safety of the translated code. The Rust compiler differentiates the parameterized types such that a `Node` becomes `Node<T>`, where `T` is the type of contained data. Consequently, a `Node<Cat>` and a `Node<Dog>` are no longer the same type in Rust's type system, which is not the case in C. Additionally, we can remove the now unnecessary cast operations since we are no longer accessing void pointers. While raw pointer cast operations are not `unsafe` in Rust, they can lead to issues related to type safety. This would help us in further refactoring (either manually or automatically) raw pointers into safer Rust constructs such as `Box<T>` or `&mut T` where the cast operations are invalid.

```rust
pub struct Node<T> { pub data: *mut T, }
fn main() {
    let mut cat: *mut Cat = /* Cat allocation */;
    let mut w: Node<Cat> = Node { data: 0 as *mut Cat, };
    // Upcast from Cat * to void * removed
    w.data = cat;
    // Some operations
    // Downcast from void * to Cat * removed
    cat = w.data;
}
```

Fig. 3: Generic Rust Node

III. Translation Overview

A. GenC2Rust Architecture

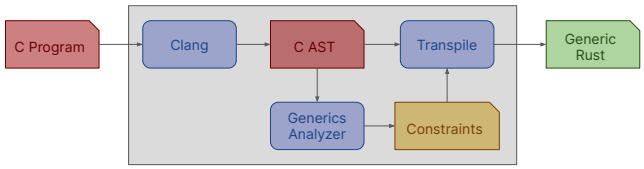


Fig. 4: GenC2Rust Architecture

We present a new tool, GenC2Rust, to retype polymorphic C `void` pointers into uses of Rust generics. GenC2Rust is developed as an extension of C2Rust [2], the state of the art C to Rust transcompiler developed by Immuant and Galois. GenC2Rust combines an analysis that can extract generics from C-style `void` pointer polymorphism and an extension of the C2Rust transpiler module that uses the analysis result to retype the `void` pointers with generic pointers. GenC2Rust also removes upcast and downcast operations when accessing polymorphic `void` pointers when appropriate.

Figure 4 provides a high-level overview of GenC2Rust's architecture. GenC2Rust has three key components: the Clang module to produce C AST, the Generic Analyzer module to extract typing constraints, and the Transpile module that processes the C AST and the typing constraints to produce Generic Rust code. GenC2Rust uses the Clang frontend from

the C2Rust translator. GenC2Rust introduces the Generics Analyzer as a new component and modifies the C2Rust transpiler module to utilize the type constraints to produce generic Rust translation.

B. Example

The process of retyping `void` pointers effectively considers each `void *` declaration to be a free type variable and then merges free type variables as needed to satisfy the typing constraints imposed by Rust's type system. Renaming a free type variable can either rewrite the `void *` type into a type parameter `T` or assign it to a concrete type, e.g., `i32`. GenC2Rust models the constraints placed by the Rust type system using a set of equality-based constraints.

Supposed we extend the linked list example by providing the `create` function in Figure 5. Observe that there are two `void` pointers in this function—the parameter and the `data` field of the `Node` object.

```
Node* create(void* data) {
    Node* node=malloc(sizeof(Node));
    node->data=data;
    return node;
}
```

Fig. 5: Linked List Node Create Function

If we assigned each of these `void` pointers to a different unique type parameter as shown in Figure 6, the Rust compiler will reject the code because the assignment operation `(*node).data=data` requires that the `data` field and the local `data` variable have the same type. However, in the rewritten code, the type for `(*node).data` is `*mut A`, and the local variable `data` has the type `*mut B`, which is not compatible. The key observation here is that the type assignment for the `create` method must be sufficiently strict such that the `create` method can type check.

```
pub unsafe
fn create<A>(mut data: *mut A) -> *mut Node::<B> {
    let mut node: *mut Node::<B> = malloc...;
    (*node).data = data;
    return node;
}
```

Fig. 6: Overly Relaxed Type Assignment

On the other hand, the analysis must also avoid type assignments that are too strict. In the example in Figure 7, suppose that we infer that both `x` and `y` have the type `*mut i32` due to the caller `bar`. This assignment may be too strict since it is possible that another function also calls `foo` with a type for `x` that is different than `y`'s type. The key observation here is that we should not constrain the type of the `foo` method beyond what is needed for it to type check.

```
 /* function does not perform anything */
fn foo<T0>(x: *mut T0, y: *mut T0) {...}
fn bar() { let mut a: i32 = 0; let mut b: i32 = 1;
    foo(&mut a, &mut b); }
```

Fig. 7: Overly Strict Type Assignment

For instance, a different function `baz` in Figure 8 may call the `foo` function at line 3. If we had previously assigned all parameters of `foo` to the same type `T0`, any caller's arguments must respect this constraint. However, the `baz` function calls `foo` with different type arguments. Argument `i` is an `i32`, but `j` is an `f64`. A practical example of this problem is a `HashMap<K, V>`. The analysis might be reasonably conclude from its knowledge that `K` and `V` are the same (e.g., both `int`s), but it is likely not the case and later uses might assign different types to `K` and `V`.

```
fn baz() {
    let mut i: i32 = 0; let mut j: f64 = 3.14;
    foo(&mut i, &mut j);
}
```

Fig. 8: Non-compilable Caller

Ideally, we want to encode constraints that allow us to produce the desired code in Figure 9, in which `x` and `y` are `*mut T0` and `*mut T1`, respectively, because there's no constraint from `foo` itself that requires the types for `x` and `y` to match. Therefore, for a function, we *only* need to encode the constraints within the function and its callees.

```
/* function does not perform anything */
fn foo<T0, T1>(x: *mut T0, y: *mut T1) { }
```

Fig. 9: Desired Translation

1) Equivalence Groups: We encode the constraint through equivalence group $\mathcal{E}$. An equivalence group captures the equality constraint as a set of types that can be either free type variables or concrete types. Solving the constraint involves assigning the free type variables such that all types in the set will match. If we can name the free type variable successfully, GenC2Rust will replace the `void` pointer with the solution type. Otherwise, we will not rewrite the `void` pointer's type.

In the `create` function, we introduce two free type variables `t1` and `t2` for the field `(*node).data` and local variable `data`, respectively. The assignment operation `(*node).data = data` in the function body introduces the equality constraint {`t1, t2`}, indicating that `t1` and `t2` must have the same type. Therefore, GenC2Rust will use the same name `A` for the two free type variables as shown in Figure 10.

```
pub unsafe
fn create<A>(mut data: *mut A) -> *mut Node::<A> {
    let mut node: *mut Node::<A> = malloc...;
    (*node).data = data;
    return node;
}
```

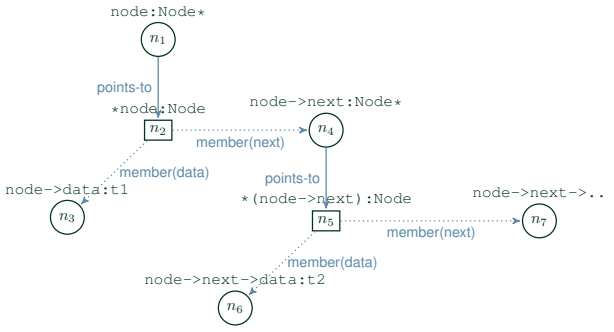
Fig. 10: Correctly Typed `create` function

2) Type Graphs: Now that we have the type variables and have placed constraints on them, we need a mechanism to identify which type variables belong to which memory locations. The type variable `t0` in `create` belongs to the field `(*node).data`, and `t1` belongs to the local variable `data`. We use type graph to track the correspondence between free type variables and data structure fields by modeling the structures of memory objects. Each variable's graph is constructed purely based on the type declarations. Essentially, it allows us to locate the type variable from an access path such that our equality constraints can be placed on the type variables. It is important to note that our type graphs are very different from typical graphs used to model heap structures such as points-to analyses graphs where the nodes provide aliasing information. In our type graph, nodes encode objects and their types referenced by variables, and each variable might refer

to several nodes. Additionally, the equivalence groups do not capture alias relations but a set of type constraints.

To complete our library implementation, we introduce a missing `Node *next` field as well as an additional `append` function in Figure 12a. The additional field can be a problem since it presents a self-referencing `Node` struct and the type graph could be expanded infinitely down the access path `node->next->next->next->...`. Figure 11a shows such infinite graph for variable `node:*Node`. To address this issue, we require that a variable's type graph only have a single node for a given type and we refer back to the previous nodes as shown in Figure 11b. The major difference is instead of producing a new node for `node->next`, we reuse the node for `node` which stores the same type.

(a) Infinitely Expanding Type Graph



(b) `Node*` Type Graph

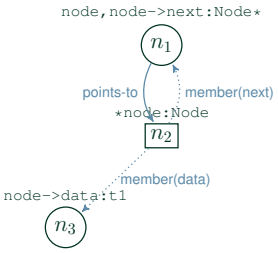


Fig. 11: Node Definition and Type Graph

3) Analysis Example: Figure 12a introduces an additional function `append`, which takes the node to append as well as the appending data. We have three type variables `data:t2`, `last->data:t3`, `node->data:t4`. We can easily derive constraints from the operations `last->next = node->next` and `node->next = last` such that `last->next`, `node->next`, `last` are in the same equivalence group. We would get the following constraint {`t3,t4`} over the free type variables by propagating the equivalence relation along the access paths. However, this is not the complete view of type equality. Notice the statement `Node *last = create(data)`. Since we called the `create` function, we should also include its constraints in the current function. GenC2Rust propagates the callee constraints to their callers to achieve this goal. In this case, because the equality of parameters `t0` and `t1` in `create`, we have a propagated effect for the arguments `t2` and `t3` resulting in a new constraint {`t2,t3`}. Now we have two equivalence groups regarding the free type variables: {`t3,t4`} and

(a) Polymorphic Linked List Implementation

```
typedef
struct Node {
    void* data;
    struct Node* next;
} Node;

Node* create(void* data) {
    Node* node = malloc(sizeof(Node));
    node->data = data;
    node->next = 0;
    return node;
}

void append(Node* node, void *data) {
    Node* last = create(data);
    if (node->next != 0) {
        last->next = node->next;
    }
    node->next = last;
}
```

(b) Generic Linked List Implementation

```
struct Node<A> {
    pub data: *mut A,
    pub next: *mut Node::<A>,
}
pub unsafe
fn create<A>(mut data: *mut A) -> *mut Node::<A> {
    let mut node: *mut Node::<A> = malloc...;
    (*node).data = data;
    (*node).next = 0 as *mut Node::<A>;
    return node;
}
pub unsafe
fn append<A>(mut node: *mut Node::<A>, mut data: *mut A) {
    let mut last: *mut Node::<A> = create(data);
    if !(*node).next.is_null() {
        (*last).next = (*node).next;
    }
    (*node).next = last;
}
```

Fig. 12: Linked List Library

{`t2,t3`}. Since equivalence is transitive, we would merge the two groups to form a new group {`t2,t3,t4`}. In summary, by analyzing the two functions `create` and `append`, one would obtain the constraints in Figure 13.

```
create: {
    { data:t0, node->data:t1 },
    { node->next:*Node, node:*Node }
}
append: {
    { node:*Node, last:*Node, node->next:*Node,
      last->next:*Node },
    { data:t2, last->data:t3, node->data:t4 }
}
```

Fig. 13: Equality Constraints for `create` and `append`

The equivalence between free type variables allows us to rename them into type parameters similar to translating `create` function. We give the type variables {`t2,t3,t4`} the same name "`A`" that is unique to their scope. At the end of the translation, we would have generic Rust program shown in Figure 12b.

C. Naming Type Variables

So far we have seen naming for free type variables in the library code `append` and `create`. Satisfying the equality constraint requires us to assign a unique name to the current function such that all types in the equivalence group can

match. Looking at the equivalence group {t2,t3,t4} from append, we do not see any concrete type in the equality constraint over the free type variables. Therefore, satisfying the equality constraint is relatively straightforward: we name {t2,t3,t4} unanimously as "A", the equivalence group becomes {A,A,A} and the type equality is satisfied.

This is not always the case since we don't only translate library code. GenC2Rust also handles client code such that it provides proper instantiations for the type parameters. Let's introduce some client code in Figure 14a. The client code uses the library code we just encountered in Figure 12a, and essentially, it encapsulates a major challenge of type parameter instantiation in translating user code. node_i:Node is a Node storing int values, and node_f:Node is a Node storing float values. We want to provide the concrete type for instantiations correctly so we have node_i: Node<i32> and node_f: Node<f64>. If the analysis is not precise enough, we might observe that node_i is a node containing either i32 or f64. This result is not acceptable as it will produce the equality constraint {...,t,i32,f64}. No type assignment to the type variable t can satisfy this equality constraint since the two existing types i32 and f64 do not match. If the types do not agree, GenC2Rust chooses the simple approach to default back to void pointers as Node<libc::c_void>.

(a) Polymorphic Linked List Client

```
int main() {
    int* a = malloc(sizeof(int));
    int* b = malloc(sizeof(int));

    float* c = malloc(sizeof(float));
    float* d = malloc(sizeof(float));

    // Init for a, b, c, d

    Node* node_i = create(a);
    append(node_i, b);

    Node* node_f = create(c);
    append(node_f, d);

    return 0;
}
```

(b) Generic Linked List Client

```
fn main() -> libc::c_int {
    let mut a: *mut i32 = malloc...;
    let mut b: *mut i32 = malloc...;

    let mut c: *mut f64 = malloc...;
    let mut d: *mut f64 = malloc...;

    // Init for a, b, c, d

    let mut node_i: *mut Node::<i32> = create(a);
    append(node_i, b);

    let mut node_f: *mut Node::<f64> = create(c);
    append(node_f, d);

    return 0 as libc::c_int;
}
```

Fig. 14: GenC2Rust Transcompiled Rust program.

Fortunately, we are able to obtain precise enough type information from equality constraints. We provide type instantiation exactly the same way as we name the free type variables previously. In the main function, given that we have the argument type variables a:int* and node_i->data:t5 at line 10, c:float* and node_f->data:t6 at line 13, and the parameter to the argument type mapping [t0->int*, t1->t5] for the call at line 10, and [t0->float*, t1->t6] for the call at line 13, we use the existing constraint on t0 and t1 as essentially a summary to their argument types such that we obtain two new equality groups {int*,t5} and {float*,t6}. GenC2Rust does not generate a single equality constraint {int*,t5,float*,t6} since our analysis applies function summaries in a context sensitive manner.

In the append function, we have the equivalence group {data:t2,last->data:t3,node->data:t4}. Using it as function summary, and apply it to the function calls append(node_i,b) and append(node_f,d), we obtain equivalence relation where node_i->data:t5,b:int*, as well as node_f->data:t6,d:float*. Eventually, we obtain the equality constraints in Figure 15.

```
main: {
    {a: int*, b: int*, node_i->data:t5},
    {c: float*, d: float*, node_i->data:t6}, ...
}
```

Fig. 15: Equality Constraints for Client Code Figure shows equality constraints for type variables t5 and t6. Rest of the constraints are skipped for simplicity.

As mentioned before, the instantiation problem is effectively substituting type variables to satisfy equality constraints. For equivalence group {a:int*,b:int*,node_i->data:t5}, we name t5 to int* to satisfy the constraint. Similarly, we can substitute the type t6 with float* in {c:float*,d:float*,node_i->data:t6}. Replacing the type variables will give us the translation result in Figure 14b. For node_i: Node<t5>, since we have the substitution t5->int*, the declaration becomes node_i: Node<int>. Applying the second substitution t6->float*, node_f is transformed to Node<float>.

IV. Equality Constraint Generation

A. Type Graph & Equivalence Groups

Formally, we denote a type graph as $\mathbb{G} = \{\mathbb{N}, \mathbb{E}\}$. It is composed of a set of type nodes $\mathbb{N}$ and type edges $\mathbb{E}$. It models the data structures of all memory locations. If two variables of the same type are declared, $\mathbb{G}$ would contain two disjoint subgraphs, one for each declared variable. A type node provides a name that can be used in equivalence groups to express that type nodes must have the same type. We use the node $n \in \mathbb{N}$ to represent either a pointer T* or a C struct. The edge $e \in \mathbb{E}$ represents a points-to or a struct field between nodes. Since the type graph only models the data structure, we can construct a variable's type graph purely from its type declarations. This *Type Graph Construction (TGC)* procedure is listed in Algorithm 1.

Algorithm 1 Type Graph Construction

```
1:  function TGC(T, iterated)                                   ▷ T: construction type
2:      if T ∈ dom(iterated) then
3:          return ⟨iterated(T), ∅, ∅⟩
4:      ▷ Make a new node n of type T and equivalence group ℰ containing node n
5:      ⟨n, ℰ⟩ ← MkNode(T)
6:      ⟨𝔼, ℕ, 𝕊⟩ ← ⟨∅, {n}, {ℰ}⟩
7:      iterated(T) ← node
8:      if T is pointer to T' then
9:          ⟨n', 𝔼', ℕ', 𝕊'⟩ ← TGC(T')
10:         ⟨𝔼, ℕ, 𝕊⟩ ← ⟨𝔼 ∪ 𝔼', ℕ ∪ ℕ', 𝕊 ∪ 𝕊'⟩
11:         𝔼 ← 𝔼' ∪ {MkPtrEdge(n, n')}                         ▷ Make a points-to edge
12:     else if T is struct with fields then
13:         for all f_ident: f_T ∈ fields do
14:             ⟨f_n, 𝔼', ℕ'⟩ ← TGC(f_T, iterated)
15:             ⟨𝔼, ℕ, 𝕊⟩ ← ⟨𝔼 ∪ 𝔼', ℕ ∪ ℕ', 𝕊 ∪ 𝕊'⟩
16:             𝔼 ← 𝔼 ∪ {MkFldEdge(n, f_n, f_ident)}            ▷ Make a field edge
17:     return ⟨node, 𝔼, ℕ, 𝕊⟩                                  ▷ Returns the type graph as nodes and edges
```

Algorithm 2 Merge Constraint

```
1:  function MergeConstraint(n1, n2, 𝔾, 𝕊, iterated)
2:      if n1 ∈ iterated or n2 ∈ iterated then
3:          return 𝕊
4:      iterated ← iterated ∪ {n1, n2}
5:      𝕊' ← MergeGroup(n1, n2, 𝕊)                              ▷ Merges equality groups of arg nodes
6:      T1 ← TypeOf(n1)
7:      T2 ← TypeOf(n2)
8:      if T1 == T2 and T1 is pointer to T' then
9:          n'_1 ← pointTo(𝔾, n_1)                              ▷ Finds pointed to node from pointer
10:         n'_2 ← pointTo(𝔾, n_2)
11:         𝕊'' ← MergeConstraint(n'_1, n'_2, 𝔾, 𝕊')
12:     else if T1 == T2 and T1 is struct with fields then
13:         for all f_ident ∈ fields do
14:             m_1 ← FldNode(𝔾, n_1, field_i) phantomsection   ▷ Finds field node
15:             m_2 ← FldNode(𝔾, n_2, field_i)
16:             𝕊' ← MergeConstraint(m_1, m_2, G, 𝕊')
17:     return 𝕊'
```

It constructs the subgraph for the variable of type `T` and returns it as nodes $\mathbb{N}$ and edges $\mathbb{E}$. Essentially, the procedure recursively traverses the type definition of the argument `T`. Depending on whether `T` is a pointer type or a `struct` type, the call to *MkNode* at line 5 creates a new node `n` and assigns it a unique equivalence group $\mathcal{E}$. If `T` is a pointer to some type `T'`, *TGC* then recursively constructs the type graph for `T'`. Afterward, it uses *MkPtrEdge* at line 11 to connect the newly created pointer node `n` to the node `n'` representing `T'`. *TGC* also maintains an `iterated` mapping of previously processed types and their corresponding nodes, enabling it to handle recursive data structures. In the case of a C `struct`, *TGC* similarly traverses each field's type `f_T` and, as with pointers, connects them using *MkFldEdge* at line 16.

Each equivalence group created by *MkNode* only contains one node after the type graph is constructed. This is only the initial creation of the equivalence groups, and later during analysis we might discover that two distinct types are equivalent.

An equivalence group encodes equality constraints on type variables. It essentially is a set of type nodes in the form $\mathcal{E} = \{n_0 : t_0, n_1 : t_1, ..., n_m : t_m\}$. Each node n_i stores the type information t_i as part of the type graph generation process. Since equality constraints are transitive, we apply a unification operation to satisfy this property. We name this operation *MergeConstraint* in Algorithm 2 that takes two type graph nodes as input to recursively walk down the access paths and merge the reachable nodes (thus the type variables).

Lines 9 to 11 demonstrate pointer lookups using the helper function *PointTo*, which retrieves the pointed-to nodes from the type graph $\mathbb{G}$ for a given pointer node `n`, and these nodes are then merged recursively. Similarly, lines 14 to 16 perform field lookups with the helper function *FldNode*, which retrieves field nodes from $\mathbb{G}$ given a `struct` node `n` and a field declaration `field`, with the retrieved nodes also merged recursively. Helper function *MergeGroup* merges the equality groups corresponding two argument type graph nodes. The merging operation does not alter the type graph as it only serves as a place to hold type variables. It updates the equivalence groups to reflect the introduced equality constraints.

B. Per Function Constraints

Constraint generation begins with per function constraints that are generated through a static analysis that can be written as a set of inference rules. We list several in Figure 16 to demonstrate the idea. Essentially, we carry the inference environment of $\mathtt{n}, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S}$. The first element `n` is the type node $\in \mathbb{G}$ of the current AST node. We also introduce a new map $\mathcal{N}$ that allows us to map from the identifier `ident` to its corresponding type node `n`, and this mapping is maintained within each function. This allows us to identify the correct type node from an access path. $\mathbb{N}, \mathbb{E}, \mathbb{S}$ are the type graph nodes and edges, and the equivalence groups we discussed earlier. By applying the inference rules, we gradually build up the whole constraints for the function.

(a)
$$\frac{\langle \mathtt{n}, \mathbb{E}_1, \mathbb{N}_1, \mathbb{S}_1\rangle := \textsc{TypeGraphConstruction}(\mathtt{T}, \emptyset) \quad \mathbb{N}_2 := \mathbb{N} \cup \mathbb{N}_1 \quad \mathbb{E}_2 := \mathbb{E} \cup \mathbb{E}_1 \quad \mathbb{S}_3 := \mathbb{S} \cup \mathbb{S}_1}{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{T\ ident} : \emptyset, \mathcal{N} \cup [\mathtt{ident} \mapsto \mathtt{n}], \mathbb{N}_2, \mathbb{E}_2, \mathbb{S}_2}$$

(b)
$$\frac{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{expr} : \mathtt{n}_{expr}, \mathcal{N}_1, \mathbb{N}_1, \mathbb{E}_1, \mathbb{S}_1 \quad \emptyset, \mathcal{N}_1, \mathbb{N}_1, \mathbb{E}_1, \mathbb{S}_1 \vdash \mathtt{ident} : \mathtt{n}_{ident}, \mathcal{N}_2, \mathbb{N}_2, \mathbb{E}_2, \mathbb{S}_2 \quad \mathbb{S}_3 := \textsc{MergeConstraint}(\mathtt{n}_{ident}, \mathtt{n}_{expr}, \langle \mathbb{N}_2, \mathbb{E}_2\rangle, \mathbb{S}_2, \emptyset)}{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{ident=expr} : \emptyset, \mathcal{N}_2, \mathbb{N}_2, \mathbb{E}_2, \mathbb{S}_3}$$

(c)
$$\frac{\mathtt{n} := \mathcal{N}(\mathtt{ident})}{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{ident} : \mathtt{n}, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S}}$$

(d)
$$\frac{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{expr} : \mathtt{n}_{expr}, \mathcal{N}_1, \mathbb{N}_1, \mathbb{E}_1, \mathbb{S}_1 \quad \textsc{FldNode}(\langle \mathbb{N}_1, \mathbb{E}_1\rangle, \mathtt{n}, \mathtt{field})}{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{expr.field} : \mathtt{n}_{field}, \mathcal{N}_1, \mathbb{N}_1, \mathbb{E}_1, \mathbb{S}_1}$$

(e)
$$\frac{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{expr} : \mathtt{n}_{expr}, \mathcal{N}_1, \mathbb{N}_1, \mathbb{E}_1, \mathbb{S}_1 \quad \langle \mathtt{n}_{cast}, \mathbb{E}_2, \mathbb{N}_2, \mathbb{S}_2\rangle := \textsc{TGCOrReuse}(\mathtt{n}_{cast}, \mathtt{n}_{expr}, \mathtt{T}, \emptyset) \quad \mathbb{S}_3 := \textsc{MergeConstraint}(\mathtt{n}_{expr}, \mathbb{S}_1 \cup \mathbb{S}_2, \emptyset)}{\emptyset, \mathcal{N}, \mathbb{N}, \mathbb{E}, \mathbb{S} \vdash \mathtt{(T)expr} : \mathtt{n}_{cast}, \mathcal{N}, \mathbb{N}_1 \cup \mathbb{N}_2, \mathbb{E}_1 \cup \mathbb{E}_2, \mathbb{S}_3}$$

Fig. 16: Inference Rules

Figure 16a demonstrates generating type graph during a variable declaration in the form of `T ident`. It calls the *TGC* algorithm in Algorithm 1 and obtains the type nodes $\mathbb{N}_1$ and edges $\mathbb{E}_1$. We introduce the updated type graph through unions of the node sets $\mathbb{N} \cup \mathbb{N}_1$ and the edge sets $\mathbb{E} \cup \mathbb{E}_1$. Figure 16b shows constraints introduced from assignment operations in the form `ident=expr`. Since the environment carries the nodes for both `ident` and `expr` as $\mathtt{n}_{ident}$ and $\mathtt{n}_{expr}$, we effectively merge the type node of the lhs `ident` and rhs `expr` by calling the union operation function `MergeConstraint`, which produces updated equivalence groups $\mathbb{S}_3$. Figure 16c describes acquiring the type graph node of a variable `ident`

through a lookup from $\mathcal{N}$. We pass the node as part of the environment so that we can look up subsequent nodes from field access operations, as shown in Figure 16d. The look up for pointed to node would be similar. Cast operation `(T) expr` imposes constraints described in Figure 16e. It calls a special function *TGCOrReuse* that checks if there is already a type graph for type `T` in the equivalence group for node $\mathtt{n}_{expr}$. It will reuse the node if such a node exist otherwise it calls the *TGC* algorithm to build the type graph for `T`.

C. Function Summaries

Per function constraints introduce constraints directly imposed by the operations within the function. However, it is not complete. Recall the function call `Node* node_i=create(a)` from the `main` function in Figure 14a. We will not be able to obtain the equality constraint {`node_i->data,a`} merely through the operation within the function itself. We apply a bottom-up propagation of function summaries to pass the constraints over the parameter and return values upwards so that the constraints are imposed on the corresponding caller arguments.

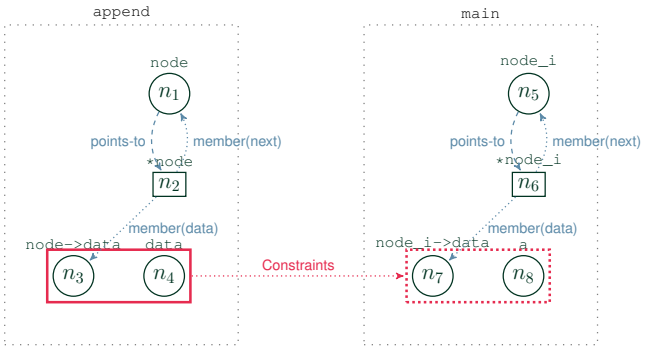


Fig. 17: Create Function Summary Propagation

Propagating function summaries relies on having a call graph. We construct the call graph by scanning all the callsites in the source program. For direct calls, we can establish the call graph by recording the callsite information as a mapping of the form $\langle \mathtt{n}_{ret}, \langle \mathtt{n}_{param1}, \mathtt{n}_{param2}, ..., \mathtt{n}_{paramn} \rangle \rangle \mapsto \langle \mathtt{n}_{call}, \langle \mathtt{n}_{arg1}, \mathtt{n}_{arg2}, ..., \mathtt{n}_{argn} \rangle \rangle$. It maps the callee parameter nodes $\langle \mathtt{n}_{param1}, \mathtt{n}_{param2}, ..., \mathtt{n}_{paramn} \rangle$ as well as the return value node $\mathtt{n}_{ret}$ to the argument nodes $\langle \mathtt{n}_{arg1}, \mathtt{n}_{arg2}, ..., \mathtt{n}_{argn} \rangle$ and the call expression node $\mathtt{n}_{call}$. If any pair of parameter node $\langle \mathtt{n}_{param\_i}, \mathtt{n}_{param\_j} \rangle$ are in the same equivalence group, we would call *MergeConstraint* on their mapped argument nodes $\langle \mathtt{n}_{arg\_i}, \mathtt{n}_{arg\_j} \rangle$. Since the summaries are propagated in a context-sensitive manner, we do not want summaries of different callsites to affect each other. Therefore, we generate each callsite a distinct callsite node $\mathtt{n}_{call}$. In the `main` function, we called `append` with arguments `node_i->data` and `a`. Since there is an equality constraint on `node->data` and `data` in the `create` function, we propagate this constraint as summary and apply a new constraint on `node_i->data` and `a`, as shown in Figure 17.

Since the analysis is not a points-to analysis, GenC2Rust handles indirect calls conservatively by providing a superset of possible callee functions. A possible function at an indirect call is a function whose address has been taken and the type and number of the parameters matches the callsite arguments. This inevitably introduces imprecision. We might introduce false equality constraints that lead to the rewriting not being able to infer a type that satisfies the constraint. However, it is a trade off to avoid generating Rust code with type errors.

To handle recursive calls, GenC2Rust propagates function summaries using a fixed-point algorithm. The fixed-point algorithm stops when no new constraints are introduced throughout all functions. Termination is guaranteed because the propagation only performs merges in the caller and there is a finite number of equivalence groups that can be merged.

The analysis handles global variables by passing them down as arguments in function calls such that the global related constraints are properly pushed upwards to the callers. This limits the analysis to only seeing partial typing constraints of the global variables for each function. To retype global variables, GenC2Rust deploys a top-down pass that pushes the argument constraint's type decision downwards, which allows the analysis to acquire a complete view. We would like to point out that the top-down pass neither introduce nor modify any constraints as it only propagates the callers' type decisions to the callees.

V. Program Rewrite

Rewriting the program requires three steps. We first need to update the `struct` declarations if they contain `void` pointers. This step is very similar to the *TGC* algorithm. We rename each `void` to a unique type parameter $\mathtt{T}_i$ and reuse the type parameters if we run into iterated data structures.

We also want to replace the declarations of `void` pointers with the substituted types. For parameter and return values, GenC2Rust examines their equivalence groups and attempts to solve the equality constraint. We can choose to replace the original `void` pointer with either a fresh type parameter or a concrete type, depending on the equality constraints. To infer a concrete type for the parameter, GenC2Rust first attempts to compute a concrete type `t` to satisfy the equality constraint. It then checks the pushed down type decision from the top down pass. If `t` exists and matches all pushed down types, GenC2Rust retypes the parameter `void` pointer with the computed type `t`. If the constraint is satisfiable but does not match any pushed down type, GenC2Rust will replace the free type variable with a fresh type parameter unique to the function scope. Global pointers are handled similarly. GenC2Rust would use the top-down type decision and the equality constraint to learn if the type is consistent throughout the program, in which case we replace the `void` pointers with the consistent type.

For the remaining declarations, we attempt to name the free type variables with concrete types. When there are conflicting types in the constraint, or the constraint is not named with a type parameter and contains only free type variables, we do not retype the `void` pointers at all for soundness. This ensures that the translated program preserves the source program behavior.

Once the `void` pointer declarations are removed, we remove redundant casts in the program. This occurs mainly as upcasts or downcasts when accessing the `void` pointers. Since cast expressions are typed with type nodes, as described in Figure 10, we are able to reason which cast operations are redundant by checking if the cast's equality constraint is satisfied. GenC2Rust removes the cast operation and leaves the inner expression in-place if there is a type assignment that satisfies the equality constraint, otherwise GenC2Rust will not attempt to remove the cast.

VI. Evaluation

A. Experiment Setup

GenC2Rust is implemented on top of C2Rust 0.17.0. It is composed of an analyzer extension with inference rules that examine the program's AST. We ran our experiment on a system with a Intel(R) Core(TM) i7-12700K CPU and 128 GB RAM on Ubuntu 22.04.3. Since C2Rust transpiles source programs per compilation unit, we first merged the program using Cilly similar to Hong *et al.* [6]'s approach. It generates a single merged C file as GenC2Rust's input. A merged single file allows GenC2Rust to obtain a whole program view of the AST and observe functions from different modules.

We took our benchmarks from previous work [3], [4], [6] subject to the following requirements: First, Cilly must successfully merge the program. Since GenC2Rust is built on C2Rust, the baseline C2Rust must successfully generate compilable code. We also eliminated programs that do not contain void pointers since no generic pointers will be generated. Table I list the benchmark programs.

B. Research Questions

We would like to understand how well GenC2Rust handles the patterns in various C programs in different domains and sizes. Our evaluation aims to answer the following questions:

- RQ1. Performance: Is the runtime performance of GenC2Rust acceptable? How long does GenC2Rust take to process a whole program from the analysis to rewrite?
- RQ2. Usefulness: How many `void` pointers does GenC2Rust identify and retype. Does the rewrite preserve the intended abstractions and parametric polymorphism present in the original C code?
- RQ3. Limitations: What are the limiting factors preventing translation to generics. Are there any patterns the analysis does not capture well?
- RQ4. Soundness & Correctness: Is GenC2Rust sound? Does GenC2Rust produce correct translation?

1) RQ1. Performance.: Table I presents the runtime performance of each benchmark program measured in seconds. Column *Runtime* shows the complete translation time from parsing the C input programs into C AST to outputting transcompiled Rust files. To further decompose the runtime performance of our analysis, we also introduced analysis time (*ATime*) and rewrite time (*RTime*) in the table. *LOC* shows the lines of code of the input C programs. Since the analysis applies inferences rules based on expressions as well as propagating constraint summaries over functions, expressions counts (*Exprs*) and function counts (*Funcs*) are also included.

Table II provides a holistic view across the programs. We computed the sums and averages of programs from Table I. The sum/average row provides the combined/average results for all 42 programs. Our analysis overall achieves good performance. On overage, each program consists of approximately 10,097.86 lines of code, 45,464.4 expressions and 152.76 functions. For each program, GenC2Rust finishes the complete translation in around 8.37 seconds, where the analysis takes 7.15 seconds, and the rewrite takes 0.51 seconds. We believe this is very reasonable given this is a one time process.

2) RQ2. Usefulness: We designed our analysis to model the Rust type checker using equality constraints. Our goal is to not produce any type errors. When the analysis decides a type to substitute, we should be certain that the new type is the correct representation of the memory location assuming that the program is memory safe.

We evaluate the usefulness of GenC2Rust by measuring the number of `void` pointers it has retyped. Retyping those pointers either results in a type parameter or a concrete non-`void` pointer type. The *Retyped* column in Table I and Table II shows the number of `void` pointers replaced. We would like to point out that it also includes `void` pointer fields within `structs`. The *Percentage* column shows number of `void` pointers translated compared with the total number of `void` pointers. For the Sum row in Table II, we compute the percentage by the total number of retyped pointers in all programs and the total number of `void` pointers in all programs. The *TP* column includes the number of retyped pointers that are now type parameters. Table II shows that it has retyped 4,572 void pointers in benchmarks, where on average each benchmark has around 108.86 `void` pointers removed. Among the 4,572 retyped `void` pointers, approximately 3,831 pointers are retyped into type parameters, and on average around 91.21 `void` pointers are retyped into type parameters per benchmark.

We also randomly examined functions in the benchmark programs to understand the translation quality-the translation retypes polymorphic uses of `void` pointers with generics without compromising the program's intended abstraction. GenC2Rust achieves good results with programs with heavy use of parametric polymorphic uses of `void` pointers. For instance, GenC2Rust is able to replace all `void` pointers with generic pointers in ht, a generic hash table implementation in C. For the `ht_next` function, which advances the iterator to the next item in the hash table, GenC2Rust successfully identifies that the `value` field of the iterator has the same type as the `value` field of the hash table entry. However, sometimes the lack of information might produce a less natural outcome. GenC2Rust concludes that the `value` of the iterator has a type parameter `T1`, while the `value` of the hash table has a type parameter `T2` in function `ht_iterator`, which essentially creates an iterator and returns it to the client code. However, this is not a failed translation, as giving them two

different type parameters does not mean that their type has to be distinct. It would only allow the instantiation of `T1` and `T2` to be different. In the client code, we can provide the correct type arguments when calling the `ht_iterator` function.

Since the analysis propagates the type information upwards, even though the caller might only see a `void` pointer, it will now see the callee types once the code is transformed. Normally this is not an issue, but if `void *` is used as an opaque pointer, the transformed code might leak the implementation detail. In bzip2, it uses `typedef void BZFILE` as an opaque pointer returned from the implementation to the user. GenC2Rust will replace the `void *` in the user code and replace it with the inferred structure `bzFile{FILE *handle; ...}`. The internal typing constraints exposed by the analysis can introduce potential issues as it negates the intention of the original API design. Nonetheless, this is not how opaque pointers are used in idiomatic Rust programs. One could alternatively use `trait` objects as suggested by [3] and methods in an `impl` block to hide the details.

3) RQ3. Limitations: We conducted studies of GenC2Rust's limitations with two perspectives. We first want to understand how constructs and operations that our analysis does not handle affect potential `void` pointers. We also studied cases where the equality constraints cannot be solved, because the analysis might fail to retype real generic pointers due to its imprecision or from patterns our analysis does not handle.

There are some constructs and operations our analysis does not handle. Specifically, we do not handle pointers interacting with extern functions and structures, unions, and integer to pointer casts.

We show the numbers of affecting pointers in Table I and Table II in columns *Extern*, *Union*, and *IntCast*. We merged the pointers for extern functions and `structs` in a single *Extern* column for conciseness. We introduce a *NoType* column where we cannot infer types for local `void` pointers due to missing concrete type information. *Conflict* column shows number of pointers in unsatisfied constraints. Figure 18 shows the causes as a percentage of all `void` pointers in the program. The x-axis represents the programs and the cause, and the y-axis indicates the percentage of `void` pointers affected. One thing to point out is that since a `void` pointer might interact with more than one cause, e.g., accessing `unions` and then passed to some extern functions, a pointer might appear in more than one bar in the percentage chart.

Among the listed causes, we see that extern functions/structures are the leading factors the analysis does not handle. We do not translate pointers from/to extern functions and structures because the analysis does not see the implementation details of those functions, resulting in missing constraints. We prepared a whitelist for the extern functions from common C libraries such as `malloc` and `free`. For functions such as `memcpy`, we modeled their equality constraints manually so calling such functions will not result in missing constraints. We don't consider external functions a limitation of the analysis. We could always provide the library implementation to GenC2Rust for it to produce generics. Alternatively, one could replace C libraries with their Rust counterparts. However, due to API differences, implementing an automated tool to swap them is not trivial. *Union*s and *IntCast*s are typically used to bypass the type system. We believe refactoring those `void` pointers would be better left to the users if needed.

We examined `void` pointers due to failed equality constraints to understand if there are generics our analysis does not capture. In json-h, for instance, we see this is due to their use of generic structure similar to tagged unions. The `struct json_value_s {void *payload; size_t type;}` uses one void pointer field for storing the data and an assisting field to encode the data type. Effectively, the analysis cannot translate such usage since there are multiple incompatible types in the type node's equivalence group. Translating such idiom to plain generic `structures` does not suffice, and one could use `enums` instead.

A good enhancement of the current rewrite strategy is to handle data inheritance. GenC2Rust will produce `void` pointers as it only sees a set of distinct types in the equality constraint, instead of computing the lowest upper bound. We made a design choice to allow inference for the field pointers instead of keeping them as `void` pointers if their `structs` do not match. We believe this pattern is not common because C programs mainly utilize structures as heap objects. One would use Rust `trait`s and encode this subtyping relation among types. We consider this as future work. Additionally, due to incompatibility between C2Rust's Bitfields macro and generic structures, we do not retype bitfield structures at this point.

4) RQ4. Soundness & Correctness: GenC2Rust aims to be sound. Its soundness follows from the fact that GenC2Rust computes a set of constraints for each function to ensure that the LHS and RHS of all assignment operations involving `void` pointers have the same type as well as that the type of cast operation matches its argument.

Since GenC2Rust is outside of the trusted code base (TCB), the Rust type checker rejects translation with any errors. We manually evaluated GenC2Rust's correctness by conducting spot checks to examine if the rust compiler accepts the translation and if its behavior is equivalent to the original program. We consider the behavior to be equivalent if it's test results matches the original C program's output. The spot check randomly selected 12 programs with test cases in the benchmark suite. For all our 12 spot check programs, GenC2Rust produced translations that match the C programs' results. We marked the spot checked programs with a $\star$ symbol in Table I.

To summarize, we designed the analysis to not introduce type errors. By analyzing the translated programs, we believe that the analysis itself is useful in lifting generic `void` pointers in C to Rust translated programs. Overall it has retyped 4,572 `void` pointers to Rust's generics. There exist cases GenC2Rust does not handle such as extern functions due to lack of implementation detail or operations to intentionally bypass the type system (and consequently our analysis), so we do not consider them as a limitation of GenC2Rust. We also

examined unsatisfied equivalence groups in translated code to show the classes of `void` pointers GenC2Rust does not intend to translate.

C. Threats to Validity

One aspect of concern rises in the benchmark programs. We picked benchmark programs from previous work in the domain of C to Rust translations. Even though those programs span different domains and sizes, we might still miss programs that exhibit unique patterns of generics.

We built GenC2Rust on top of C2Rust. C2Rust provides a C AST to our analyzer. We also modified its transpile module to produce generics directly. It is possible to encounter bugs on the pipeline and produce incorrect results. However, C2Rust is actively maintained and well tested. We believe the possibility of C2Rust bugs affecting our translation is very slim.

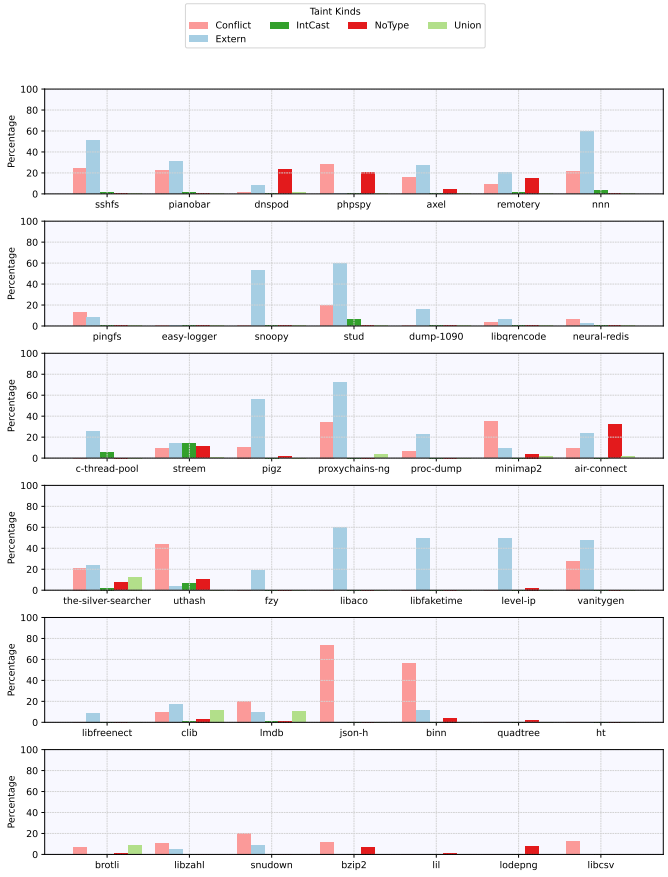


Fig. 18: Cause Percentage Per Program

VII. Related Work

Researchers proposed refactor solutions to generalize programs. Subtype inference based approaches [8], [9] are proposed to convert non-generic Java code to use generics. Duggan *et al.* [8] proposed a translation of a monomorphic Java-like language called MiniJava to its polymorphic variant PolyJava. However, they did not provide any experiments or implementation to demonstrate the usefulness of their approach. Dincklage *et al.* [9] constructs subtyping constraints on free type variables. The constraints are applied across call-sites. It uses "Per Method Polymorphism", which constructs copies of type variables to handle static functions.

Donovan *et al.* [10] presents context-sensitive points-to analysis to instantiate generic types at client site. It presupposes already parameterized libraries, and requires modeling the API methods of the generic classes as part of the analysis. Our method generates generic Rust structures as well as updating their instantiations at client sites. GenC2Rust also does not require a full context sensitive points-to analysis. Siff *et al.* [11] transforms C into C++ templates through type inference, allowing generalizing non-polymorphic data. This contrasts with our goal of generalizing only polymorphic data in the source program without extending its functionality.

There are multiple tools developed to translate C programs to Rust such as Crust [12], Citrus [13], Corrode [14], and C2Rust [2]. However, at this point, most projects are no longer actively maintained except for C2Rust. Bindgen [15] focuses on generating Rust FFI bindings of C libraries. Xia *et al.* [16] conducts study of C to Rust transcompilers to understand their reliability. Researchers also proposed tools to produce more idiomatic C to Rust translated code. Laertes [3] replaces raw pointers with safe references based on compiler-derived information. To understand the limitation of Laertes, Emre *et al.* [17] replaces unsafe sources without original semantics as following work. Crown [4] retypes raw pointers with `Box` type through ownership analysis. CRustS [18] removes unnecessary unsafe qualifiers and blocks from translated Rust programs. Hong *et al.* [19] proposes unsafe features, mainly lock APIs and output parameters, introduced in C2Rust translated code. Concrat [6] automatically replaces C lock APIs with Rust lock APIs by lock summaries from data flow analysis. Nopcrat [7] replaces output parameters with Rust's `Option` and `Result` type through abstract interpretation. Urcrat [20] translates C unions to Rust's `enum`s through points-to analysis. Larson *et al.* [21] proposes pointer derivation graph based on a hybrid of static and dynamic analysis to translate unsafe Rust to safer Rust. Flourine [22] and Vert [23] utilizes large language models to translate Rust programs. Han *et al.* [24] proposed Rusty that transpiles more compact Rust code through unstructured control specializations. Tripuramallu *et al.* [25] conducted manual translations of C++ to Rust and concludes a translation mapping to help generating better Rust.

Rust's unsafe is not limited to the domain of translations. Astrauska *et al.* [26] develops queries to understand how unsafe Rust is used in practice and summarizes a classification of purpose for unsafe Rust. Höltervennhof *et al.* [27] conducts interviews with experienced Rust programmers to understand their experience and decision process with unsafe Rust. Evans *et al.* [28] develops an automated technique to uncover functions using unsafe and characterize the uses. RustBelt [1] provides a formal safety proof for λ_{Rust} that encapsulates core subsets of Rust. Qin *et al.* [29] manually examined the unsafe uses and bugs in Rust programs and implemented two Rust static bug detectors based on the result of examination. SafeNet [30] combines machine learning techniques and

TABLE I: Benchmark Program Statistics and Evaluation Result

Program	Runtime	ATime	RTime	LOC	Exprs	Funcs	Retyped	TP	Percentage	Extern	Union	IntCast	Conflict	NoType
air-connect	13.99	12.62	0.85	17588	65466	292	214	174	36.0%	138	10	0	72	189
axel	0.99	0.45	0.32	6838	19867	95	37	26	42.5%	24	0	0	29	4
c-thread-pool⋆	0.16	0.03	0.02	712	1319	23	27	25	75.0%	9	0	2	2	0
clib	47.38	44.93	1.64	25083	103300	594	493	460	73.4%	126	79	6	77	10
dnspod	4.35	3.64	0.41	9248	33596	201	234	210	62.2%	30	4	0	17	95
dump-1090⋆	0.65	0.21	0.24	4654	15022	89	10	3	76.9%	2	0	0	1	0
easy-logger	0.42	0.18	0.1	2370	5450	61	7	6	100.0%	0	0	0	0	0
fzy⋆	0.42	0.2	0.07	2631	7349	79	12	5	75.0%	3	0	0	1	0
level-ip	0.18	0.04	0.03	722	2033	13	18	11	40.9%	22	0	0	3	1
libaco⋆	0.38	0.08	0.16	1286	6275	14	2	2	40.0%	3	0	0	0	0
libfaketime⋆	0.13	0.01	0.01	523	943	3	3	3	50.0%	3	0	0	0	0
libfreenect	0.15	0.01	0.02	629	1594	10	10	5	90.9%	1	0	0	0	0
libqrencode⋆	1.39	0.89	0.26	6672	23456	189	27	0	81.8%	2	0	0	4	0
lmdb	9.59	8.49	0.74	10829	46505	159	876	866	75.5%	112	118	16	237	16
minimap2⋆	5.45	3.61	1.32	17291	72307	258	269	231	52.7%	51	10	0	188	16
neural-redis	0.56	0.19	0.18	3649	15140	65	44	6	91.7%	1	0	0	3	0
nnn	3.20	1.92	0.92	12095	44180	169	22	13	40.0%	33	0	2	12	0
phpspy	7.80	6.14	1.11	19400	68097	239	345	311	43.8%	7	0	0	275	160
pianobar	35.33	33.51	1.38	11460	34499	146	68	53	50.0%	42	0	2	30	0
pigz	4.80	3.81	0.7	9130	30121	86	25	18	37.9%	37	0	0	10	1
pingfs	0.73	0.51	0.07	2320	5675	65	17	9	70.8%	2	0	0	7	0
proc-dump	0.71	0.3	0.21	4160	14902	53	24	16	77.4%	7	0	0	2	0
proxychains-ng	1.16	0.57	0.39	5209	17351	78	5	5	17.2%	21	1	0	10	0
remotery	3.84	3.31	0.28	7218	24189	255	116	93	63.7%	38	0	3	17	27
snoopy⋆	0.58	0.25	0.15	3727	12015	82	7	5	46.7%	8	0	0	0	0
sshfs⋆	3.97	3.45	0.24	7197	20468	187	30	19	46.2%	33	0	1	15	0
streem	91.81	90.33	0.87	20184	73203	542	657	486	57.9%	166	8	154	109	127
stud	61.67	60.93	0.4	7941	21222	102	5	1	33.3%	9	0	1	4	0
the-silver-searcher⋆	1.76	1.09	0.41	7252	25779	97	42	20	44.7%	22	12	2	28	4
uthash⋆	0.22	0.02	0.08	819	3329	3	8	2	26.7%	1	0	2	16	3
vanitygen	2.51	1.2	1.01	10952	28957	138	10	1	40.0%	12	0	0	7	0
binn	0.49	0.2	0.09	3453	11207	168	126	126	36.0%	44	0	0	209	15
brotli	27.72	5.52	3.51	127793	858330	892	294	164	82.8%	0	30	0	26	5
ht	0.13	0.0	0.01	207	741	10	22	22	100.0%	0	0	0	0	0
json-h	0.43	0.13	0.14	2861	9590	53	17	17	23.3%	0	0	0	56	0
quadtree	0.19	0.04	0.01	461	1884	27	56	56	98.2%	0	0	0	0	1
bzip2⋆	2.77	0.96	1.46	11798	46864	109	100	90	71.4%	0	0	0	30	10
libcsv	0.58	0.05	0.35	3435	15903	31	14	8	87.5%	0	0	0	2	0
libzahl	2.02	1.39	0.32	11867	35435	230	16	6	84.2%	1	0	0	2	0
lil	1.42	1.04	0.16	2907	14427	128	90	90	98.9%	0	0	0	0	1
lodepng	1.09	0.21	0.55	9633	38829	235	58	56	92.1%	0	0	0	0	5
snudown	8.43	7.77	0.35	9906	32686	146	115	111	90.6%	11	0	0	1	0

TABLE II: Evaluation Summary

Program	Runtime	ATime	RTime	LOC	Exprs	Funcs	Retyped	TP	Percentage	Extern	Union	IntCast	Conflict	NoType
Sum	351.55	300.23	21.54	424110	1909505	6416	4572	3831	60.0%	1021	272	191	1502	690
Average	8.37	7.15	0.51	10097.86	45464.4	152.76	108.86	91.21	-	24.31	6.48	4.55	35.76	16.43

dataflow analysis to assist replacing unnecessary unsafe API uses. Thy *et al.* [31] proposes an Extract Method refactoring tool with a static intra-procedural ownership analysis. Rustlancet *et al.* [32] automatically fixes Rust's ownership rule violations by applying patches. Kirth *et al.* [33] proposed PKRU-Safe to prevent unsafe external code from corrupting the memory of safe languages such as Rust. Hind *et al.* [34] and Choi *et al.* [35] use pairs to represent alias information. This is similar to our equivalence groups. However, alias pairs answer if two pointers may alias at a given program location while equivalence groups encode equality constraints introduced by operations.

There is a substantial body of research on type inference in C. Johnson *et al.* [36] leverage type-qualifier inference to identify exploitable user/kernel pointer bugs in the Linux kernel. Type inference techniques have also been applied to translating C into safer dialects. Necula *et al.* [37] introduced CCured, which extends the C type system and uses type inference to ensure correct pointer types. Machiry *et al.* [38] employed type inference to convert legacy C pointers into checked pointers in Checked C, enhancing spatial safety. Melo *et al.* [39] explored type inference to reconstruct partial programs, demonstrating its utility in dealing with incomplete codebases.

VIII. Conclusion

One challenge in translating C code to Rust code is handling `void` pointers. GenC2Rust addresses this by converting generic `void` pointers in C programs into Rust generic pointers, making the code more idiomatic. It introduces type parameters to replace `void *` types and extracts type constraints from C programs to properly constrain these parameters. We present a static analysis that extracts these constraints and a strategy for using them to generate Rust generics. Our implementation of GenC2Rust has been tested on 42 C programs and retyped 4,572 pointers to demonstrate its usefulness.

Acknowledgments

We would like to thank the anonymous reviewers for their thorough and insightful comments that helped us improve the paper. This work is supported by National Science Foundation grants OAC-1740210, CCF-2006948, CCF-2102940, and CCF-2220410.

REFERENCES

[1] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158154

[2] C. Team, "C2Rust," https://c2rust.com/, 2023.

[3] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485498

[4] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership guided C to Rust translation," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds., vol. 13966. Springer, 2023, pp. 459–482. [Online]. Available: https://doi.org/10.1007/978-3-031-37709-9_22

[5] P. Larsen and E. Westbrook, "Emitting safer Rust with C2Rust," https://immunant.com/blog/2023/03/lifting/, 2023.

[6] J. Hong and S. Ryu, "Concrat: An automatic C-to-Rust lock API translator for concurrent programs," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 716–728. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00069

[7] ——, "Don't write, but return: Replacing output parameters with algebraic data types in C-to-Rust translation," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, jun 2024. [Online]. Available: https://doi.org/10.1145/3656406

[8] D. Duggan, "Modular type-based reverse engineering of parameterized types in Java code," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 97–113. [Online]. Available: https://doi.org/10.1145/320384.320393

[9] D. von Dincklage and A. Diwan, "Converting Java classes to use generics," *SIGPLAN Not.*, vol. 39, no. 10, p. 1–14, oct 2004. [Online]. Available: https://doi.org/10.1145/1035292.1028978

[10] A. Donovan, A. Kiežun, M. S. Tschantz, and M. D. Ernst, "Converting Java programs to use generic libraries," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 15–34. [Online]. Available: https://doi.org/10.1145/1028976.1028979

[11] M. Siff and T. Reps, "Program generalization for software reuse: From C to C++," in *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 135–146. [Online]. Available: https://doi.org/10.1145/239098.239121

[12] N. Shetty, "C/C++ to Rust transpiler," https://nishanthspshetty.github.io/crust/.

[13] K. Lesinski, "Citrus: C to Rust converted," https://users.rust-lang.org/t/citrus-c-to-rust-converter/12441, 2017.

[14] J. Sharp, "Corrode: Automatic semantics-preserving translation from C to Rust," https://github.com/jameysharp/corrode, 2017.

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

[16] L. Xia, B. Hua, and Z. Peng, "An empirical study of C to Rust transpilers," *School of Software Engineering, University of Science and Technology of China, and Suzhou Institute for Advanced Research, University of Science and Technology of China-04/27*, 2023.

[17] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, "Aliasing limits on translating C to safe Rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586046

[18] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In Rust we trust: A transpiler from unsafe C to safer Rust," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 354–355. [Online]. Available: https://doi.org/10.1145/3510454.3528640

[19] J. Hong, "Improving automatic C-to-Rust translation with static analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 273–277.

[20] J. Hong and S. Ryu, "To tag, or not to tag: Translating c's unions to rust's tagged unions," 2024. [Online]. Available: https://arxiv.org/abs/2408.11418

[21] P. Larsen, "Migrating C to Rust for memory safety," *IEEE Security & Privacy*, pp. 2–9, 2024.

[22] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with LLMs: A study of translating to Rust," 2024. [Online]. Available: https://arxiv.org/abs/2405.11514

[23] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "VERT: Verified equivalent Rust transpilation with large language models as few-shot learners," 2024. [Online]. Available: https://arxiv.org/abs/2404.18852

[24] X. Han, B. Hua, Y. Wang, and Z. Zhang, "RUSTY: Effective C to Rust conversion via unstructured control specialization," in *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022, pp. 760–761.

[25] D. Tripuramallu, S. Singh, S. Deshmukh, S. Pinisetty, S. A. Shivaji, R. Balusamy, and A. Bandeppa, "Towards a transpiler for C/C++ to safer Rust," 2024. [Online]. Available: https://arxiv.org/abs/2401.08264

[26] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe Rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428204

[27] S. Höltervennhoff, P. Klostermeyer, N. Wöhler, Y. Acar, and S. Fahl, ""I wouldn't want my unsafe code to run my pacemaker": An interview study on the use, comprehension, and perceived risks of unsafe Rust," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2509–2525. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/holtervennhoff

[28] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 246–257. [Online]. Available: https://doi.org/10.1145/3377811.3380413

[29] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 763–779. [Online]. Available: https://doi.org/10.1145/3385412.3386036

[30] Y. Dong, Z. Zhang, M. Cui, and H. Xu, "SafeNet: Towards mitigating replaceable unsafe Rust code via a recommendation-based approach," *Software Testing, Verification and Reliability*, p. e1875, 2024.

[31] S. Thy, A. Costea, K. Gopinathan, and I. Sergey, "Adventure of a lifetime: Extract method refactoring for Rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622821

[32] W. Yang, L. Song, and Y. Xue, "Rust-lancet: Automated ownership-rule-violation fixing with behavior preservation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639103

[33] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "PKRU-safe: automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–148. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[34] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, p. 848–894, jul 1999. [Online]. Available: https://doi.org/10.1145/325478.325519

[35] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 232–245. [Online]. Available: https://doi.org/10.1145/158511.158639

[36] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference," in *Proceedings of the 13th Conference on USENIX Security*

Symposium - Volume 13, ser. SSYM'04. USA: USENIX Association, 2004, p. 9.

[37] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, p. 477–526, May 2005. [Online]. Available: https://doi.org/10.1145/1065887.1065892

[38] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, "C to checked c by 3c," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3527322

[39] L. T. C. Melo, R. G. Ribeiro, B. C. F. Guimarães, and F. M. Q. a. Pereira, "Type inference for c: Applications to the static analysis of incomplete programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 3, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3421472