Enhancing C-to-Rust Translation with Subtype Inference for Enum Types

Minwook Lee
ryan-lee@kaist.ac.kr

***Abstract*—Automatic C-to-Rust translation is a promising way to improve the reliability of legacy C software. However, the existing C2Rust translator primarily performs direct syntactic conversion, which often fails to produce idiomatic Rust code and thus undermines the goal of safer, more reliable software.**

For example, in C an `enum` is effectively just an integer type and does not ensure that a variable only holds one of its defined variant values. Consequently, C2Rust translates C enums as integers in Rust, using type aliases and constants for the variants. In contrast, Rust's `enum` type enforces that only valid variant values can be assigned and supports pattern matching for variant-specific behavior. Ideally, a translation tool should convert C enums into real Rust enums to take advantage of these safety features.

In this report, we introduce a subtype inference technique to enhance C2Rust's handling of enum types. This approach analyzes the usage of an enum in the C code to determine if it can safely be represented as a Rust enum. If so, the translator replaces the integer alias with a proper Rust enum definition. This enhancement leverages Rust's safety and expressiveness, yielding translated code that is more idiomatic and type-safe.

I. Background and Motivation

A. Enumeration Types in C and Rust

In C, an `enum` is essentially a set of named integer constants; the language does not enforce that an enum variable holds only the defined variant values. Each enumerator is assigned an integer (either explicitly or by default in sequence), and different enum types can even share the same underlying values. This means a program can assign an integer value to an enum variable even if that value doesn't correspond to any defined variant. Consider the following code example:

```c
enum Color {
    r, // 0
    g, // 1
    b = 3
};

int main() {
    enum Color color = r;
    color = 4; // no error
    return 0;
}
```

In this C code, the `enum Color` defines three variants, where `r` and `g` are implicitly mapped to 0 and 1, and `b` is explicitly mapped to 3. In the `main` function, the variable `color` is declared as type `Color` and initialized to `r`. However, the code then assigns `color = 4`, which is not one of the defined variants, and C allows this without any error. This illustrates that C enums do not provide value safety.

In contrast, Rust's `enum` type ensures value safety and is more expressive. Each variant in a Rust enum is distinct and cannot be conflated with an unrelated integer value. Moreover, Rust allows pattern matching on enums, enabling clear and type-safe differentiation of variants in code. The following Rust example demonstrates these properties:

```rust
#[repr(C)]
enum Color {
    R, // 0
    G, // 1
    B = 3,
}

fn main() {
    use std::mem::transmute;

    assert_eq!(Color::R as i32, 0);
    assert!(matches!(
        unsafe { transmute::<i32, Color>(0) },
        Color::R
    ));
    // let color: Color = 4; // error
    // let color: Color = 4 as Color; // error
    let color: Color = unsafe { transmute::<i32,
    Color>(4) }; // Undefined behavior
}
```

Note that Rust enums are a form of algebraic data type and can be more expressive (for example, variants can hold associated data), but here we restrict our focus to simple C-like enums with integer values. In the Rust code above, the enum `Color` has three variants (`R`, `G`, and `B`) corresponding to 0, 1, and 3 respectively. In `main`, the variable `color` is of type `Color` and is initialized to `Color::R`. Any attempt to assign an out-of-range value (such as `4`) to `color` results in a compile-time error. This demonstrates that Rust enums guarantee an enum variable cannot hold values outside its defined variants.

B. C2Rust's Enum Translation

C2Rust is a tool for translating C code to Rust. Although it can handle enums, in practice C2Rust translates each C enum into a Rust integer type alias (with constants for the variants) rather than using a proper Rust enum type. This means it fails to leverage Rust's value safety and expressiveness for enums. The limitation stems from C2Rust's reliance on straightforward syntactic transformations.

For example, consider the C enumeration below and how C2Rust translates it:

```c
enum Color {
    r, g, b
};
```

C2Rust produces the following Rust code:

```rust
pub type Color = libc::c_uint;
pub const b: Color = 2;
pub const g: Color = 1;
pub const r: Color = 0;
```

In this output, the enum is essentially lost: `Color` is defined as an alias for `libc::c_uint`, and each variant is a constant of that type. The translated code behaves like an untyped integer, not leveraging Rust's enum features or value checking.

In C, an enum type is effectively interchangeable with an integer type (it is just a labeled alias), so C2Rust's current translation is technically faithful as a direct transformation. However, if a developer explicitly uses an enum type name in a variable or function declaration, it implies an intention that the variable should only take on values from that enum's set of variants. If we can verify from the code that this intention is respected (i.e., only valid variant values are ever assigned to that variable), then it would be more appropriate for the translated Rust code to use a real enum type instead of a plain integer.

For these reasons, we propose to augment C2Rust with a subtype inference mechanism for enums. This mechanism determines if a given C enum's usage is consistent with a Rust enum's constraints, and if so, translates it into an actual Rust enum (replacing the integer alias). The goal is to produce translated code that fully benefits from Rust's safety features and clarity.

II. Approach

A. Naïve Translation

A straightforward idea would be to always translate a C enum into a Rust enum in the output, as long as doing so does not break the code's correctness. For example, consider the following C code and how C2Rust currently translates it:

```c
enum Color {
    r, g, b
};

int ctoi(const enum Color color) {
    switch (color) {
        case r:
            return 0x0000ff;
        case g:
            return 0x00ff00;
        case b:
            return 0xff0000;
    }
    return -1;
}
```

```rust
pub type Color = libc::c_uint;
pub const b: Color = 2;
pub const g: Color = 1;
pub const r: Color = 0;

unsafe fn ctoi(color: Color) -> libc::c_int {
    match color as libc::c_uint {
        0 => return 0xff as libc::c_int,
        1 => return 0xff00 as libc::c_int,
        2 => return 0xff0000 as libc::c_int,
        _ => {}
    }
    return -(1 as libc::c_int);
}
```

C2Rust's translation above leaves `Color` as an alias for `c_uint`. Suppose instead we manually alter the translation to use a Rust enum for `Color`:

```rust
#[repr(C)]
enum Color {
    R, G, B
}

unsafe fn ctoi(color: Color) -> libc::c_int {
    match color {
        Color::R => return 0xff as libc::c_int,
        Color::G => return 0xff00 as libc::c_int,
        Color::B => return 0xff0000 as libc::c_int,
    }
    return -(1 as libc::c_int);
}
```

In this instance, replacing the alias with a real Rust enum works correctly. As long as the C code never introduces invalid values for `Color`, translating it directly to a Rust enum is sound. However, now consider an additional piece of code added to this program:

```rust
// ...

int foo() {
    const int green = 1;
    return ctoi(green);
}
```

```rust
// ...

fn foo() -> libc::c_int {
    use std::mem::transmute;
    let green: libc::c_uint = 1;
    // return ctoi(green); // error
    // return ctoi(green as Color); // error
    return ctoi(unsafe {
        transmute::<libc::c_uint, Color>(green)
    }); // only workaround, but unsafe
}
```

In the `foo` function above, C allows an implicit conversion: the constant integer `green` (value 1) is passed to `ctoi`, which expects an enum `Color`. Rust, however, does not permit implicitly or explicitly casting an integer to an enum type. As a result, calling `ctoi(green)` would not compile in Rust, even an attempt like `ctoi(green as Color)` is disallowed. Only a workaround using `transmute` allows passing `green` to `ctoi`, but this is unsafe and can lead to undefined behavior if `green` does not correspond to a valid `Color` variant. This scenario highlights why C2Rust sticks to integer types for enums; a naïve direct conversion would break here. We need a more nuanced approach to handle such cases. To address this, we propose a technique called *subtype inference*.

B. Subtype Inference

Our approach introduces the concept of treating the C enum type as a subtype of a general integer type. In other words, we assume a C enum ϵ should allow only the values defined by its variants (making it, conceptually, a subset of $\mathbb{Z}$). Under this interpretation, we enforce that any variable declared with type ϵ can only be assigned one of the enum's

variant values. In C, this property is not enforced by the compiler (any integer can be assigned), whereas in Rust the compiler does enforce it (an enum variable can only be one of its variants).

With subtype inference, we analyze the program to infer whether an enum type is consistently used in a restricted way (only holding its variant values). If so, we consider that enum to be a subtype of the integer type, meaning it effectively behaves like an integer with a limited range. We then translate it as a Rust enum. This analysis requires examining every usage of the enum in the C code and checking that any value assigned to a variable of that type is indeed one of the defined variant values.

For a concrete illustration, revisit the earlier `foo` example. We assume that whenever the code explicitly uses the enum's name in a type annotation, it is intended to truly be of that enum type. Under this assumption:

- The parameter `c` of function `ctoi` is declared as type `Color`, so we treat it as genuinely meant to be a `Color`.
- In the body of `foo`, the variable `green` is passed to `ctoi`. This usage suggests that `green` should be considered a `Color` as well (since it is being used where a `Color` is expected).
- We accordingly change `green`'s declaration from `const int green = 1;` to `const Color green = 1;`.
- We then verify that this change does not introduce any type errors. In this case, the initializer value `1` is indeed one of the defined `Color` variants (assuming `r=0`, `g=1`, `b=3` as earlier), so the assignment is valid under the new type.

If we can apply such transformations for all instances where the enum is used (and none of them produce a type error), then the C enum in question can safely be translated into a Rust enum. In the next section, we formalize this approach with a formal grammar and inference rules.

III. Formal Definitions and Assumptions

A. Notation

We introduce the following notation for our formalization:

1) Let ϵ denote the particular enum type under consideration for translation.
2) Let V_ϵ be the set of all integer values corresponding to the variants of ϵ (i.e., the values that ϵ can take).

B. Assumptions

Our approach relies on several assumptions:

1) The C code is well-typed (we assume no type errors in the input program), and the type of every variable can be determined.
2) If the enum type's name ϵ is explicitly used in a variable or parameter declaration in the C code, we assume the type of that variable is intended to be ϵ.
3) For simplicity, assume that all parameter and variable identifier names are distinct (no naming collisions).
4) We maintain an associated type environment Γ (with elements stored in a set $TEnv$) that records variables and their associated required types.

C. Type Checking and Requirement Functions

We define two functions to formalize the checking of expressions and enforcement of required types:

- $check(\Gamma, e)$: Performs standard type checking on expression e under environment Γ. It returns the inferred type τ of e and an updated associated type environment Γ. This function is used in the general case where there are no special developer requirements on types.
- $require(\Gamma, e, \tau)$: Checks expression e under environment Γ with the requirement that e's type must be exactly τ. It returns the associated type environment Γ after enforcing this requirement. This function is used when a specific type is expected for an expression (for example, due to an explicit annotation or context).

IV. Formal Rules for Subtype Inference

A. Basic Expression Language (No Functions)

1) Abstract Syntax: We first define an abstract syntax for a simple language of expressions (without functions) to model our approach:

$$
\begin{array}{rcl}
\tau & ::= & \epsilon \\
 & | & \mathbb{Z} \\
e & ::= & \mathsf{let}\ x{:}\tau = e\ \mathsf{in}\ e \\
 & | & x \\
 & | & n \\
 & | & e \oplus e \\
 & | & \mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e
\end{array}
$$

2) Subtype and Required Type: We treat the enum type ϵ as a subtype of the integer type $\mathbb{Z}$. We also introduce a notion of a "required" type for enforcing enum types in certain contexts. The formal rules for subtyping and required types are given in Figure 1.

$$
\begin{array}{rl}
\textsc{Sub-Epsilon}: & \epsilon <: \mathbb{Z} \\
\textsc{Sub-Refl}: & \tau <: \tau \\
\textsc{Req-Enum}: & \dfrac{\tau = \epsilon}{\tau \text{ is required}} \\
\textsc{Force-Req}: & \dfrac{\tau_2 \text{ is required} \quad \tau_2 <: \tau_1}{\tau_1 \Rightarrow \tau_2}
\end{array}
$$

Fig. 1. Subtype relation and required-type rules (for the base language). Rule Sub-Epsilon declares the enum type ϵ to be a subtype of the integer type $\mathbb{Z}$, and Sub-Refl allows any type to be a subtype of itself. Rule Req-Enum tags ϵ as a required type (indicating that contexts expecting this type must enforce it), and Force-Req states that if τ_2 is required and $\tau_2 <: \tau_1$, then τ_1 can be forced to a required subtype τ_2.

3) Type Checking: The type checking rules for expressions (without functions) are given in Figure 2.

$$\textsc{Check-Var}: \quad check(\Gamma,\, x) = (\Gamma(x),\, \emptyset)$$

$$\textsc{Check-ConstNonEnum}: \quad \frac{n \notin V_\epsilon}{check(\Gamma,\, n) = (\mathbb{Z},\, \emptyset)}$$

$$\textsc{Check-ConstEnum}: \quad \frac{n \in V_\epsilon}{check(\Gamma,\, n) = (\epsilon,\, \emptyset)}$$

$$\textsc{Check-Subsumption}: \quad \frac{check(\Gamma,\, e) = (\tau',\, \Gamma_1) \quad \tau' <: \tau}{check(\Gamma,\, e) = (\tau,\, \Gamma_1)}$$

$$\textsc{Check-BinOp}: \quad \frac{check(\Gamma,\, e_1) = (\mathbb{Z},\, \Gamma_1) \quad check(\Gamma,\, e_2) = (\mathbb{Z},\, \Gamma_2)}{check(\Gamma,\, e_1 \oplus e_2) = (\mathbb{Z},\, \Gamma_1 \cup \Gamma_2)}$$

$$\textsc{Check-Ternary}: \quad \frac{\begin{array}{c} check(\Gamma,\, e_1) = (\mathbb{Z},\, \Gamma_1) \\ check(\Gamma,\, e_2) = (\tau,\, \Gamma_2) \quad check(\Gamma,\, e_3) = (\tau,\, \Gamma_3) \end{array}}{check(\Gamma,\, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) = (\tau,\, \Gamma_1 \cup \Gamma_2 \cup \Gamma 3)}$$

$$\textsc{Check-Let-Req}: \quad \frac{\begin{array}{c} \tau_1 \text{ is required} \quad require(\Gamma,\, e_1,\, \tau_1) = \Gamma_1 \\ check(\Gamma[x : \tau_1],\, e_2) = (\tau_2,\, \Gamma_2) \end{array}}{check(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2) = (\tau_2,\, \Gamma_1 \cup \Gamma_2)}$$

$$\textsc{Check-Let-NoReq}: \quad \frac{\begin{array}{c} \tau_1 \text{ is not required} \quad check(\Gamma,\, e_1) = (\tau_1,\, \Gamma_1) \\ check(\Gamma[x : \tau_1],\, e_2) = (\tau_2,\, \Gamma_2) \quad x \notin \text{Domain}(\Gamma_2) \end{array}}{check(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2) = (\tau_2,\, \Gamma_1 \cup \Gamma_2)}$$

$$\textsc{Check-Let-Force}: \quad \frac{\begin{array}{c} \tau_1 \Rightarrow \tau \quad require(\Gamma,\, e_1,\, \tau) = \Gamma_1 \\ check(\Gamma[x : \tau],\, e_2) = (\tau_2,\, \Gamma_2) \quad \Gamma_2(x) = \tau \end{array}}{check(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2) = (\tau_2,\, \Gamma_1 \cup \Gamma_2)}$$

Fig. 2. Type checking rules for the base language (no functions). CHECK-VAR retrieves a variable's type from the environment. CHECK-CONSTNONENUM and CHECK-CONSTENUM classify a numeric literal as type $\mathbb{Z}$ or ϵ depending on whether its value belongs to V_ϵ. CHECK-SUBSUMPTION allows an expression of a subtype to be used as a supertype. CHECK-BINOP and CHECK-TERNARY ensure that binary operations use integer operands and that a ternary conditional has an integer condition and consistent branch result types. Finally, CHECK-LET-REQ, CHECK-LET-NOREQ, and CHECK-LET-FORCE handle let-bindings: enforcing required types, handling the absence of a requirement, and forcing a required subtype, respectively.

4) Type Requiring: The rules for requiring (asserting) specific types are given in Figure 3.

B. Extension: First-Class Functions

1) Abstract Syntax: We now extend the language syntax to include first-class functions:

$$\begin{array}{rcl} \tau & ::= & \cdots \\ & | & \tau \to \tau \\ e & ::= & \cdots \\ & | & \lambda x{:}\,\tau.e \\ & | & e\ e \end{array}$$

This extension adds function types ($\tau_1 \to \tau_2$), lambda abstractions (anonymous functions), and function application to our language.

2) Subtype and Required Type: The additional rules for subtyping and required types with function types are shown in Figure 4.

3) Type Checking: The type checking rules for function abstractions and application are shown in Figure 5.

4) Type Requiring: Finally, the requirement rules for function-related expressions are given in Figure 6.

V. Applying to Translation

A. Propositions

Based on the above rules, we formulate two propositions about translating an enum type using this approach:

Let e be a well-typed C program, and let e' be the program obtained by changing the declared type of every variable (and parameter) in e that explicitly uses the enum type ϵ to ϵ (if it isn't already). Then:

1) If $check(\emptyset,\, e')$ succeeds (i.e., all the type-checking rules can be applied without failure on e'), it verifies that the enum type ϵ can be treated as a subtype of the integer type (meaning that ϵ can be safely translated as a Rust enum type).
2) Suppose $check(\emptyset,\, e') = (\tau,\, \Gamma_1)$ (the type checker infers type τ for e' and produces environment Γ_1). If we then modify the declared type of every variable x found in Γ_1 to $\Gamma_1(x)$ in the original program (producing a new program e''), then e'' is well-typed. In other words, after inferring the appropriate enum types via Γ_1, applying

$$\textsc{Require-Var-Env:}\quad \frac{\Gamma(x) = \tau' \quad \tau <: \tau'}{require(\Gamma,\, x,\, \tau) = [x : \tau]}$$

$$\textsc{Require-ConstEnum:}\quad \frac{n \in V_\epsilon}{require(\Gamma,\, n,\, \epsilon) = \emptyset}$$

$$\textsc{Require-Expr-NoReq:}\quad \frac{\tau \text{ is not required} \quad check(\Gamma,\, e) = (\tau,\, \Gamma_1)}{require(\Gamma,\, e,\, \tau) = \Gamma_1}$$

$$\textsc{Require-Var-Expr:}\quad \frac{check(\Gamma,\, x) = (\tau',\, \Gamma_1) \quad \tau' <: \tau}{require(\Gamma,\, x,\, \tau) = \Gamma_1}$$

$$\textsc{Require-Ternary:}\quad \frac{\begin{array}{c} check(\Gamma,\, e_1) = (\mathbb{Z},\, \Gamma_1) \\ require(\Gamma,\, e_2,\, \tau) = \Gamma_2 \quad require(\Gamma,\, e_3,\, \tau) = \Gamma_3 \end{array}}{require(\Gamma,\, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,\, \tau) = \Gamma_1}$$

$$\textsc{Require-Let-Req:}\quad \frac{\begin{array}{c} \tau_1 \text{ is required} \quad require(\Gamma,\, e_1,\, \tau_1) = \Gamma_1 \\ require(\Gamma[x : \tau_1],\, e_2,\, \tau_2) = \Gamma_2 \end{array}}{require(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2,\, \tau_2) = \Gamma_1 \cup \Gamma_2}$$

$$\textsc{Require-Let-NoReq:}\quad \frac{\begin{array}{c} \tau_1 \text{ is not required} \quad check(\Gamma,\, e_1) = (\tau_1,\, \Gamma_1) \\ require(\Gamma[x : \tau_1],\, e_2,\, \tau_2) = \Gamma_2 \quad x \notin \text{Domain}(\Gamma_2) \end{array}}{require(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2,\, \tau_2) = \Gamma_1 \cup \Gamma_2}$$

$$\textsc{Require-Let-Force:}\quad \frac{\begin{array}{c} \tau_1 \Rightarrow \tau \quad require(\Gamma,\, e_1,\, \tau) = \Gamma_1 \\ require(\Gamma[x : \tau],\, e_2,\, \tau_2) = \Gamma_2 \quad \Gamma_2(x) = \tau \end{array}}{require(\Gamma,\, \mathsf{let}\ x{:}\,\tau_1 = e_1\ \mathsf{in}\ e_2,\, \tau_2) = \Gamma_1 \cup \Gamma_2}$$

Fig. 3. Type requirement ("require") rules for the base language. REQUIRE-VAR-ENV adds a variable to the environment with a required type (if that required type τ is a subtype of the variable's original type τ'). REQUIRE-CONSTENUM asserts that a numeric literal has type ϵ if its value is in V_ϵ. REQUIRE-EXPR-NOREQ processes an expression normally when no special requirement applies. REQUIRE-VAR-EXPR handles requiring a variable expression to have a certain type by checking it and ensuring the actual type is a subtype of the required type. REQUIRE-TERNARY enforces that both branches of a conditional expression meet the required type. Finally, REQUIRE-LET-REQ, REQUIRE-LET-NOREQ, and REQUIRE-LET-FORCE impose type requirements in let-binding contexts analogously to the type checking rules.

$$\textsc{Req-Arrow:}\quad \frac{\tau_2 \text{ is required}}{\tau_1 \to \tau_2 \text{ is required}}$$

$$\textsc{Sub-Arrow:}\quad \frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4}$$

Fig. 4. Additional subtyping and requirement rules for function types. REQ-ARROW marks a function type as required if its return type is required (i.e., if τ_2 must be ϵ, then the function type $\tau_1 \to \tau_2$ is considered required). SUB-ARROW is the subtyping rule for function types: $\tau_1 \to \tau_2$ is a subtype of $\tau_3 \to \tau_4$ if τ_3 is a subtype of τ_1 (parameter types are contravariant) and τ_2 is a subtype of τ_4 (return types are covariant).

those types back to the program yields a consistent, type-correct program.

We do not present a formal proof of these propositions here. Instead, we illustrate them with examples to show that the formal rules behave as intended.

B. Examples

We first consider a simple code fragment (assuming that $0 \in V_\epsilon$):

$$\mathsf{let}\ x{:}\,\mathbb{Z} = 0\ \mathsf{in}\ \{\ \mathsf{let}\ y{:}\,\epsilon = x\ \mathsf{in}\ \{\ \mathsf{let}\ z{:}\,\mathbb{Z} = 0\ \mathsf{in}\ z\ \}\ \}$$

In this code, y is explicitly declared with type ϵ, so we interpret that as the developer intending y to be of the enum type. To avoid any type errors, our system should infer that x also has type ϵ because x is initialized to 0 (and we assumed $0 \in V_\epsilon$). Inferring $x : \epsilon$ is valid here since 0 is one of ϵ's variant values. In contrast, z is declared as an $\mathbb{Z}$ (integer) and has no connection to ϵ apart from coincidentally being assigned 0. Therefore, z should *not* be inferred as type ϵ.

By applying the subtype inference rules, we find that only x gets associated with the enum type ϵ (while z remains an integer).

If we then change the declaration of the associated variable x to use type ϵ (as inferred), the resulting code becomes well-typed with the enum type correctly applied:

$$\mathsf{let}\ x{:}\,\epsilon = 0\ \mathsf{in}\ \{\ \mathsf{let}\ y{:}\,\epsilon = x\ \mathsf{in}\ \{\ \mathsf{let}\ z{:}\,\mathbb{Z} = 0\ \mathsf{in}\ z\ \}\ \}$$

$$\textsc{Check-Lambda-Req:}\quad \frac{\tau_1 \text{ is required} \quad \mathit{check}(\Gamma[x:\tau_1],\, e) = (\tau_2,\, \Gamma_1)}{\mathit{check}(\Gamma,\, \lambda x{:}\,\tau_1.e) = (\tau_1 \to \tau_2,\, \Gamma_1)}$$

$$\textsc{Check-Lambda-NoReq:}\quad \frac{\tau_1 \text{ is not required} \quad \mathit{check}(\Gamma[x:\tau_1],\, e) = (\tau_2,\, \Gamma_1) \quad x \notin \text{Domain}(\Gamma_1)}{\mathit{check}(\Gamma,\, \lambda x{:}\,\tau_1.e) = (\tau_1 \to \tau_2,\, \Gamma_1)}$$

$$\textsc{Check-Lambda-Force:}\quad \frac{\tau \Rightarrow \tau_1 \quad \mathit{check}(\Gamma[x:\tau_1],\, e) = (\tau_2,\, \Gamma_1) \quad \Gamma_1(x) = \tau_1}{\mathit{check}(\Gamma,\, \lambda x{:}\,\tau.e) = (\tau_1 \to \tau_2,\, \Gamma_1)}$$

$$\textsc{Check-Apply:}\quad \frac{\mathit{require}(\Gamma,\, e_1,\, \tau_1 \to \tau_2) = \Gamma_1 \quad \mathit{require}(\Gamma,\, e_2,\, \tau_1) = \Gamma_2}{\mathit{check}(\Gamma,\, e_1\ e_2) = (\tau_2,\, \Gamma_1 \cup \Gamma_2)}$$

Fig. 5. Extended type checking rules to support first-class functions. The CHECK-LAMBDA-* rules handle type checking of lambda (anonymous function) expressions under three scenarios: when the parameter's type is required (CHECK-LAMBDA-REQ), when the parameter has no special requirement (CHECK-LAMBDA-NOREQ), and when a required subtype must be enforced for the parameter (CHECK-LAMBDA-FORCE). CHECK-APPLY checks a function application by first ensuring the function expression has an arrow type and then requiring the argument expression to have the function's parameter type.

$$\textsc{Require-Lambda-Req:}\quad \frac{\tau_1 \text{ is required} \quad \mathit{require}(\Gamma[x:\tau_1],\, e,\, \tau_2) = \Gamma_1}{\mathit{require}(\Gamma,\, \lambda x{:}\,\tau_1.e,\, \tau_1 \to \tau_2) = \Gamma_1}$$

$$\textsc{Require-Lambda-NoReq:}\quad \frac{\tau_1 \text{ is not required} \quad \mathit{require}(\Gamma[x:\tau_1],\, e,\, \tau_2) = \Gamma_1 \quad x \notin \text{Domain}(\Gamma_1)}{\mathit{require}(\Gamma,\, \lambda x{:}\,\tau_1.e,\, \tau_1 \to \tau_2) = \Gamma_1}$$

$$\textsc{Require-Lambda-Force:}\quad \frac{\tau \Rightarrow \tau_1 \quad \mathit{require}(\Gamma[x:\tau_1],\, e,\, \tau_2) = \Gamma_1 \quad \Gamma_1(x) = \tau_1}{\mathit{require}(\Gamma,\, \lambda x{:}\,\tau.e,\, \tau_1 \to \tau_2) = \Gamma_1}$$

$$\textsc{Require-Apply:}\quad \frac{\mathit{check}(\Gamma,\, e_1) = (\tau_3 \to \tau_4,\, \Gamma_1) \quad \tau_1 <: \tau_3 \quad \mathit{require}(\Gamma,\, e_1,\, \tau_1 \to \tau_2) = \Gamma_2 \quad \mathit{require}(\Gamma,\, e_2,\, \tau_1) = \Gamma_3}{\mathit{require}(\Gamma,\, e_1\ e_2,\, \tau_2) = \Gamma_2 \cup \Gamma_3}$$

Fig. 6. Extended requirement rules for first-class functions. REQUIRE-LAMBDA-REQ enforces an expected parameter type on a lambda expression (if the parameter's type is required to be ϵ). REQUIRE-LAMBDA-NOREQ handles a lambda with no special parameter requirement (ensuring the parameter is not inadvertently inferred to a required type elsewhere). REQUIRE-LAMBDA-FORCE forces a lambda's parameter to have a specific subtype if needed. REQUIRE-APPLY imposes requirements when a function is applied: it first checks that the function has a suitable arrow type (and may refine the expected parameter type τ_1 if it was too general), and then requires the argument to have that parameter type τ_1.

We can also confirm that this approach works when functions are involved. Consider the following example:

$$\begin{array}{l}\mathsf{let}\ \mathit{ctoi}{:}\,\epsilon \to \mathbb{Z} = \lambda c{:}\,\epsilon.(3 \times c)\ \mathsf{in}\\ \quad \mathsf{let}\ \mathit{green}{:}\,\mathbb{Z} = 1\ \mathsf{in}\\ \quad\quad \mathit{ctoi}\ \mathit{green}\end{array}$$

In this code, the function `ctoi` expects a parameter of type ϵ and returns an integer (the value multiplied by 3). The variable `green` is declared with type $\mathbb{Z}$ and assigned 1, but when we pass `green` to `ctoi`, that usage effectively associates `green` with the enum type ϵ (since `ctoi` requires an ϵ argument). Therefore, `green` should be inferred to have type ϵ. Applying the formal rules confirms this inference, resulting in the following well-typed code (with `green` now declared as type ϵ):

$$\begin{array}{l}\mathsf{let}\ \mathit{ctoi}{:}\,\epsilon \to \mathbb{Z} = \lambda c{:}\,\epsilon.(3 \times c)\ \mathsf{in}\\ \quad \mathsf{let}\ \mathit{green}{:}\,\epsilon = 1\ \mathsf{in}\\ \quad\quad \mathit{ctoi}\ \mathit{green}\end{array}$$

These examples illustrate that the formal rules correctly identify when a C enum type can be translated to a Rust enum type and guide the necessary transformations. The subtype inference approach helps determine when a C enum can be upgraded to a Rust enum in translation, and it carries out the replacement of the integer alias with a Rust enum where it is valid to do so.

VI. Conclusion

In this report, we presented an approach to enhance C2Rust's enum translation by introducing subtype inference. Our approach determines when a given C enum can be safely translated into a Rust enum and, if feasible, replaces the integer type alias with a proper Rust enum. By leveraging Rust's value safety and expressiveness for enums, the translation process produces output code that is more idiomatic and maintains type correctness.

This approach offers several benefits. First, it improves the accuracy of the translation by inferring and enforcing the intended enum types, which leads to fewer type errors and a smoother migration from C to Rust. Second, by replacing opaque integer aliases with meaningful Rust enums, it enhances the readability and maintainability of the code. Developers can more easily understand the intent of the code when enums are used instead of generic integers.

However, our approach in its current form relies on a strict static type analysis and may fail in certain cases that a human might still consider valid enum usage. For example, consider a common C idiom:

```c
enum Day {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

enum Day next_day(enum Day day) {
    return (day + 1) % 7;
}
```

This code uses an enum to represent days of the week and calculates the next day. For a successful translation to a Rust enum, we would need to deduce that the return value of `next_day` is always between 0 and 6 (i.e., a valid `Day`). However, the type system alone cannot guarantee this property, so our formal rules would not allow translating the return type of `next_day` as an enum. Handling such cases would require additional analysis, such as range analysis or value constraint inference. Instead of automatically concluding that an expression involving arithmetic cannot correspond to an enum, the translator could attempt to deduce constraints on the expression's possible values and determine whether it stays within the valid range of an enum's variants. Incorporating such analysis could enable us to capture more cases where a C enum type can be translated to a Rust enum type.