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 enhance the reliability of legacy system software. However, C2Rust, an industrially developed translator, generates Rust code with syntactic transformations, failing to produce idiomatic Rust code and undermining the translation's objectives.**

A representative example is that in C, enums are treated as aliases for integer types, which do not enforce the values of specified variants. Consequently, C2Rust generates Rust code with integer type aliases. However, Rust's enums provide value safety and expressiveness, and the translation process can leverage these features by converting C enums to Rust enums.

This report proposes a novel approach to improve C2Rust's enum translation. The approach determines whether a C enum can be translated to a Rust enum and, if feasible, replaces the integer type alias with a Rust enum. This enhancement aims to maximize Rust's safety and expressiveness in the translation process.

I. Background

A. Enumeration Types in C and Rust

In C, enums are used for storing constant integer values, but do not enforce the values of specified variants. For example, when defining an enum in C, each variant is mapped to an integer value, which can lead to collisions between different variants. This allows for the assignment of values that are not specific to any variant in C code. 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 the above C code, the enum Color defines three variants. However, `r` and `g` are mapped to 0 and 1, respectively, while `b` is mapped to 3. In the main function, the color variable is initialized to `r`, but can later be assigned the value 4. This demonstrates that C enums do not provide value safety.

In contrast, Rust's enums provide value safety and expressiveness. When defining an enum in Rust, each variant has a unique value that does not collide with other variants. Additionally, Rust's enums allow for easy differentiation of variants through pattern matching. Consider the following code example:

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

fn main() {
    let mut color = Color::R;
    assert!(color as i32 == 0);
    // color = 4; // error: mismatched types
    // color = 4 as Color; // error: can't cast
    integer to enum
}
```

In Rust, the enum type is an Abstract Data Type with a broader meaning, but this study focuses on translating C enums, so we only address enums in the form similar to C. In the above Rust code, the enum type `Color` defines three variants. Each variant has a unique value: R is mapped to 0, G to 1, and B to 3. In the `main` function, the variable `color` is initialized to `Color::R`, and attempting to assign the value 4 later results in a compile-time error. This demonstrates that Rust's enums provide value safety.

B. C2Rust's Enum Translation

C2Rust is a tool that translates C code to Rust, providing functionality to convert C enums to Rust enums. However, C2Rust translates enums as integer type aliases, failing to leverage the value safety and expressiveness offered by Rust's enums. This limitation arises because C2Rust primarily performs syntactic transformations.

For example, consider the following C code that we would like to translate using C2Rust:

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

C2Rust translates the above C code as follows:

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

In the above Rust code, C2Rust translates the enum as an integer type alias and defines each variant as a constant. However, this does not utilize the value safety and expressiveness provided by Rust's enums.

In C, the relationship between enumeration types and integer types is merely type aliasing. Therefore, C2Rust's approach of translating C's enumeration types to Rust's constant integer values is appropriate from the perspective of performing syntactic transformations. However, since enumeration types can be used to specify variable or function types, if a developer has used the enumeration name in

another variable declaration, it can be inferred that they intended for the value to exist among the enum variants. If this fact can be proven in the code, then using an enum type in the translated idiomatic Rust code would be appropriate.

This report proposes a novel approach to improve C2Rust's enum translation. The approach determines whether a C enum can be translated to a Rust enum and, if feasible, replaces the integer type alias with a Rust enum. This enhancement aims to maximize Rust's safety and expressiveness in the translation process.

II. Approach

A. Naïve Translation

The simplest approach is to translate any enumeration type defined in C to a Rust enum type, but only adopt it when the expression is safe and well-typed. For example, consider the following C code and its translation via C2Rust:

```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 translates the enum type as an integer type alias, but let's assume we force it to be an enum type:

```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 0xff00000 as libc::c_int
    }
    return -(1 as libc::c_int);
}
```

In this specific case, C2Rust's translation of the enum to a Rust enum works well. If the C developer has ideally written the code to prevent incorrect values from entering the enumeration type, this approach will work consistently. However, consider the following additional code:

```c
// ...

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

In this case, an implicit conversion occurs in C, converting `green` to the `Color` type. Rust's enums do not allow such conversions, so calling `ctoi(green)` (even with a casting syntax like `ctoi(green as Color)`) results in a compile-time error. This is one of the reasons why C2Rust translates enums as integer type aliases. To correctly translate such cases, a new approach is needed. The next section proposes a novel approach called subtype inference to address these issues.

B. Subtype Inference

The core idea of this approach is to view enum types as subtypes of integer types. This ensures that variables declared with an enum type can only be assigned values that are one of the variants of that enum type. From this perspective, C does not guarantee safety because it allows any integer value to be assigned to a variable of an enum type, while Rust guarantees safety by allowing only values that are one of the variants of the enum type.

Subtype inference involves deducing whether the developer intended to declare an enum type as a subtype of an integer type. If the developer did intend for the enum type to be a subtype of an integer type, it can be translated to a Rust enum type. To achieve this, all occurrences of the enum type in the C code must be analyzed. This analysis checks whether values assigned at these occurrences are one of the variants of the enum type.

For example, let's reconsider the `foo` function mentioned earlier. We assume that the developer intended to declare the type as "Color" only when explicitly specified in the variable or parameter declaration. We then check only the variables associated with that declaration to see if they can be translated to an enum type. In the example code:

- The parameter type of `ctoi` is explicitly declared as "Color". Therefore, we assume that the parameter is intended to be of type `Color`.
- In the body of the `foo` function, `green` is applied as a parameter to `ctoi`, so we associate the type of `green` with `Color`.
- We modify the declaration of the associated variable from `int` to `Color`. It becomes `const Color green = 1;`.
- We check if the modified type does not cause a type error. In the definition statement, the right-hand side of the assignment operator is an expression that evaluates to 1, which can be immediately verified. Since 1 is one of the variants of `Color`, this is valid.

If this transformation can be applied to all associated variables without issues, we can safely translate the enumeration type to a Rust enum type.

The next section presents a formal grammar to define this approach.

III. Definitions

A. Notation

We define the following notations to formalize our approach:

1) We define the target enum type that we want to verify for translation as ϵ.
2) We denote the set of all values associated with the target enum type ϵ as V_ϵ.

B. Assumptions

We make the following assumptions for applying this approach:

1) The C code is always well-typed, and the type of every variable declaration has been inferred.
2) If the name of the enum type is explicitly used in the variable declaration in the C code, we assume that the type of that variable is intended to be ϵ.
3) For convenience, we assume that all identifier names used in parameter specifications or variable declarations are distinct.
4) The "associated type environment variable" stores all pairs of associated variables and their required types as elements of $TEnv$.

C. Define Functions

We define the following functions to formalize our approach:

- $check(\Gamma, e)$: Returns the type τ obtained through type checking and the associated type environment variable Γ. This is executed in the general case where there are no specific developer requirements.
- $require(\Gamma, e, \tau)$: Requires the current expression to have the exact type τ, and returns the associated type environment variable Γ at that time. This is called when the developer has specific type requirements.

IV. Formal Rule of Subtype Inference

A. No Functions, Single Expression

1) Abstract Syntax: We define the abstract syntax for types and expressions to formalize this approach:

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

2) Subtype and Required Type:

$$\epsilon <: \mathbb{Z} \quad \tau <: \tau$$

$$\frac{\tau = \epsilon}{\tau \text{ is required}}$$

$$\frac{\tau_2 \text{ is required} \quad \tau_2 <: \tau_1}{\tau_1 \Rightarrow \tau_2}$$

3) Type Checking:

$$check(\Gamma, x) = (\Gamma(x), \emptyset)$$

$$\frac{n \notin V_\epsilon}{check(\Gamma, n) = (\mathbb{Z}, \emptyset)}$$

$$\frac{n \in V_\epsilon}{check(\Gamma, n) = (\epsilon, \emptyset)}$$

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

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

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

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

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

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

4) Type Requiring:

$$\frac{\Gamma(x) = \tau' \quad \tau <: \tau'}{require(\Gamma,\, x,\, \tau) = [x : \tau]}$$

$$\frac{n \in V_\epsilon}{require(\Gamma,\, n,\, \epsilon) = \emptyset}$$

$$\frac{\tau \text{ is not required} \quad check(\Gamma,\, e) = (\tau,\, \Gamma_1)}{require(\Gamma,\, e,\, \tau) = \Gamma_1}$$

$$\frac{check(\Gamma,\, x) = (\tau',\, \Gamma_1) \quad \tau' <: \tau}{require(\Gamma,\, x,\, \tau) = \Gamma_1}$$

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

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

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

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

B. With First-Class Functions

1) Abstract Syntax: We extend the abstract syntax to include functions:

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

2) Subtype and Required Type:

$$\frac{\tau_2 \text{ is required}}{\tau_1 \to \tau_2 \text{ is required}}$$

$$\frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4}$$

3) Type Checking:

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

$$\frac{\begin{array}{c} \tau_1 \text{ is not required} \\ check(\Gamma[x : \tau_1],\, e) = (\tau_2,\, \Gamma_1) \quad x \notin Domain(\Gamma_1) \end{array}}{check(\Gamma,\, \lambda x{:}\,\tau_1.e) = (\tau_1 \to \tau_2,\, \Gamma_1)}$$

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

$$\frac{require(\Gamma,\, e_1,\, \tau_1 \to \tau_2) = \Gamma_1 \quad require(\Gamma,\, e_2,\, \tau_1) = \Gamma_2}{check(\Gamma,\, e_1\ e_2) = (\tau_2,\, \Gamma_1 \cup \Gamma_2)}$$

4) Type Requiring:

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

$$\frac{\begin{array}{c} \tau_1 \text{ is not required} \\ require(\Gamma[x : \tau_1],\, e,\, \tau_2) = \Gamma_1 \quad x \notin Domain(\Gamma_1) \end{array}}{require(\Gamma,\, \lambda x{:}\,\tau_1.e,\, \tau_1 \to \tau_2) = \Gamma_1}$$

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

$$\frac{\begin{array}{c} check(\Gamma,\, e_1) = (\tau_3 \to \tau_4,\, \Gamma_1) \quad \tau_1 <: \tau_3 \\ require(\Gamma,\, e_1,\, \tau_1 \to \tau_2) = \Gamma_2 \quad require(\Gamma,\, e_2,\, \tau_1) = \Gamma_3 \end{array}}{require(\Gamma,\, e_1\ e_2,\, \tau_2) = \Gamma_2 \cup \Gamma_3}$$

V. Applying to Translation

A. Propositions

We propose the following propositions based on the rules defined above:

Let e be a well-typed C code. Let e' be the code obtained by modifying the declaration of all variables in e where the name of the enum type is used in the type declaration to ϵ. Then:

1) If $check(\emptyset, e')$ passes all rules, it is verified that the enum type can be translated as a subtype of the integer type.
2) Let $check(\emptyset,\, e') = (\tau,\, \Gamma_1)$. If we modify the declaration of the type of every variable x in Γ to $\Gamma(x)$ to obtain the expression e'', then e'' is well-typed.

This report does not present a rigorous proof of the propositions. Instead, we provide examples to verify that these propositions work correctly. These examples serve to confirm that the formal rules defined earlier function as intended.

B. Examples

We consider the following code, assuming that ϵ includes the value 0:

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

Consider the goal we need to achieve for normal enum translation: since the type of y is specified as ϵ, we should assume that the developer intended it to be an enum type. To ensure that this code works without type errors, we need to perform subtype inference on x to be of type ϵ. This inference is valid since x is assigned the value 0. However, z is not specified as an ϵ type and has no other associations with ϵ, so it should not be inferred as an ϵ type even though it is declared as 0.

Applying the formal rules defined earlier, we can easily determine that only x is associated with the ϵ type. By modifying the declaration of the associated variable, we obtain a well-typed expression with the enum type applied correctly:

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

We can confirm that this approach works well even when functions are applied, as shown in the following example:

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

In this code, the ctoi function takes a parameter of type ϵ and returns the value multiplied by 3. Although green is declared as type $\mathbb{Z}$, it is associated with the ϵ type when applied to the ctoi function. Therefore, green should be inferred as type ϵ. Again, applying the formal rules allows us to make the correct inference, resulting in the following well-typed expression:

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

These examples demonstrate that the formal rules work as intended. This approach is useful for determining whether a C code's enum type can be translated to a Rust enum type and, if feasible, replacing the integer type alias with a Rust enum.

VI. Conclusion

In this report, we have proposed a novel approach to improve C2Rust's enum translation by introducing subtype inference. This approach allows us to determine whether a C enum can be translated to a Rust enum and, if feasible, replaces the integer type alias with a Rust enum. By leveraging the value safety and expressiveness of Rust's enums, we enhance the translation process and ensure that the resulting code is idiomatic and well-typed.

The proposed approach has several benefits. First, it increases the accuracy of the translation process by ensuring that the types are correctly inferred and associated. This leads to fewer type errors and a smoother migration from C to Rust. Second, by replacing integer type aliases with Rust enums, we improve the readability and maintainability of the code. Developers can more easily understand the intent behind the code when enums are used instead of generic integer types.

However, this research is based on a strict type system, so it may fail to translate even in cases where it can be interpreted as an enum type. For example, consider the following common C code pattern:

```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 type in C to represent days of the week and calculates the next day. For a successful translation to an enum, it must be deduced that the return value is always between 0 and 6. However, the type system alone cannot guarantee this. Therefore, the formal rules proposed in this research will fail to translate this case to an enum type.

To address such cases, additional analysis is needed to infer the range of integer types. Instead of concluding that an expression with arithmetic operations cannot be an enum type, we should deduce the constraints on the values that the expression can take and determine whether it can be translated to an enum type. This analysis would allow us to capture more cases where a C code's enum type can be translated to a Rust enum type.