

# 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:

```
1 enum Color {  
2     r, // 0  
3     g, // 1  
4     b = 3  
5 };  
6  
7 int main() {  
8     enum Color color = r;  
9     color = 4; // no error  
10    return 0;  
11 }
```

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:

```
1 #[repr(C)]  
2 enum Color {  
3     R, // 0  
4     G, // 1  
5     B = 3,  
6 }  
7  
8 fn main() {  
9     use std::mem::transmute;  
10  
11    assert_eq!(Color::R as i32, 0);  
12    assert!(matches!(  
13        unsafe { transmute::<i32, Color>(0) },  
14        Color::R  
15    ));  
16    // let color: Color = 4; // error  
17    // let color: Color = 4 as Color; // error  
18    let color: Color = unsafe { transmute::<i32, Color>(4) }; // Undefined behavior  
19 }
```

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:

```
1 enum Color {
2     r, g, b
3 };
```

C2Rust produces the following Rust code:

```
1 pub type Color = libc::c_uint;
2 pub const b: Color = 2;
3 pub const g: Color = 1;
4 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:

```
1 enum Color {
2     r, g, b
3 };
4
5 int ctoi(const enum Color color) {
6     switch (color) {
7         case r:
8             return 0x0000ff;
9         case g:
10            return 0x00ff00;
11        case b:
12            return 0xff0000;
13    }
14    return -1;
15 }
```

```
1 pub type Color = libc::c_uint;
2 pub const b: Color = 2;
3 pub const g: Color = 1;
4 pub const r: Color = 0;
5
```

```

6 unsafe fn ctoi(color: Color) -> libc::c_int {
7     match color as libc::c_uint {
8         0 => return 0xff as libc::c_int,
9         1 => return 0xff00 as libc::c_int,
10        2 => return 0xff0000 as libc::c_int,
11        _ => {}
12    }
13    return -(1 as libc::c_int);
14 }

```

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`:

```

1 #[repr(C)]
2 enum Color {
3     R, G, B
4 }
5
6 unsafe fn ctoi(color: Color) -> libc::c_int {
7     match color {
8         Color::R => return 0xff as libc::c_int,
9         Color::G => return 0xff00 as libc::c_int,
10        Color::B => return 0xff0000 as libc::c_int,
11    }
12    return -(1 as libc::c_int);
13 }

```

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:

```

1 // ...
2
3 int foo() {
4     const int green = 1;
5     return ctoi(green);
6 }

```

  

```

1 // ...
2
3 fn foo() -> libc::c_int {
4     use std::mem::transmute;
5     let green: libc::c_uint = 1;
6     // return ctoi(green); // error
7     // return ctoi(green as Color); // error
8     return ctoi(unsafe {
9         transmute::<libc::c_uint, Color>(green)
10    }); // only workaround, but unsafe
11 }

```

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  $\epsilon$  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  $\epsilon$  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  $\epsilon$  denote the particular enum type under consideration for translation.
- 2) Let  $V_\epsilon$  be the set of all integer values corresponding to the variants of  $\epsilon$  (i.e., the values that  $\epsilon$  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  $\epsilon$  is explicitly used in a variable or parameter declaration in the C code, we assume the type of that variable is intended to be  $\epsilon$ .
- 3) For simplicity, assume that all parameter and variable identifier names are distinct (no naming collisions).
- 4) We maintain an associated type environment  $\Gamma$  (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  $\Gamma$ . It returns the inferred type  $\tau$  of  $e$  and an updated associated type environment  $\Gamma$ . 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  $\Gamma$  with the requirement that  $e$ ’s type must be exactly  $\tau$ . It returns the associated type environment  $\Gamma$  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:

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

2) *Subtype, Required Type, and Type Environment Rules*: We treat the enum type  $\epsilon$  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, required types, and type environments are given in Figure 1. The definition of the lower bound type,  $\sqcap$ , is in Appendix.

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

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

$$\begin{array}{ll}
\text{SUB-EPSILON:} & \epsilon <: \mathbb{Z} \quad \text{REQ-ENUM:} \quad \frac{\tau = \epsilon}{\tau \text{ is required}} \\
\text{SUB-REFL:} & \tau <: \tau \quad \text{FORCE-REQ:} \quad \frac{\tau_2 \text{ is required} \quad \tau_2 <: \tau_1}{\tau_1 \Rightarrow \tau_2} \\
\text{TENV-UNION:} & \Gamma_1 \cup \Gamma_2 = \Gamma \text{ where } \Gamma(x) = \begin{cases} \Gamma_1(x) \sqcap \Gamma_2(x) & \text{if } x \in \text{Domain}(\Gamma_1) \cap \text{Domain}(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in \text{Domain}(\Gamma_1) \\ \Gamma_2(x) & \text{if } x \in \text{Domain}(\Gamma_2) \end{cases}
\end{array}$$

Fig. 1. Subtype relation and required-type rules (for the base language). Rule SUB-EPSILON declares the enum type  $\epsilon$  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  $\epsilon$  as a required type (indicating that contexts expecting this type must enforce it), and FORCE-REQ states that if  $\tau_2$  is required and  $\tau_2 <: \tau_1$ , then  $\tau_1$  can be forced to a required subtype  $\tau_2$ . Finally, TENV-UNION defines how to combine two type environments  $\Gamma_1$  and  $\Gamma_2$  into a new environment  $\Gamma$ , resolving conflicts by taking the more specific type when both environments define the same variable.

$$\begin{array}{ll}
\text{CHECK-VAR:} & \text{check}(\Gamma, x) = (\Gamma(x), \emptyset) \\
\text{CHECK-CONSTNONENUM:} & \frac{n \notin V_\epsilon}{\text{check}(\Gamma, n) = (\mathbb{Z}, \emptyset)} \\
\text{CHECK-CONSTENUM:} & \frac{n \in V_\epsilon}{\text{check}(\Gamma, n) = (\epsilon, \emptyset)} \\
\text{CHECK-SUBSUMPTION:} & \frac{\text{check}(\Gamma, e) = (\tau', \Gamma_1) \quad \tau' <: \tau}{\text{check}(\Gamma, e) = (\tau, \Gamma_1)} \\
\text{CHECK-BINOP:} & \frac{\text{check}(\Gamma, e_1) = (\mathbb{Z}, \Gamma_1) \quad \text{check}(\Gamma, e_2) = (\mathbb{Z}, \Gamma_2)}{\text{check}(\Gamma, e_1 \oplus e_2) = (\mathbb{Z}, \Gamma_1 \cup \Gamma_2)} \\
\text{CHECK-TERNARY:} & \frac{\text{check}(\Gamma, e_1) = (\mathbb{Z}, \Gamma_1) \quad \text{check}(\Gamma, e_2) = (\tau, \Gamma_2) \quad \text{check}(\Gamma, e_3) = (\tau, \Gamma_3)}{\text{check}(\Gamma, \text{if0 } e_1 \text{ then } e_2 \text{ else } e_3) = (\tau, \Gamma_1 \cup \Gamma_2 \cup \Gamma_3)} \\
\text{CHECK-LET-REQ:} & \frac{\tau_1 \text{ is required} \quad \text{require}(\Gamma, e_1, \tau_1) = \Gamma_1 \quad \text{check}(\Gamma[x : \tau_1], e_2) = (\tau_2, \Gamma_2) \quad x \notin \text{Domain}(\Gamma_2) \text{ or } \tau_1 <: \Gamma_2(x)}{\text{check}(\Gamma, \text{let } x: \tau_1 = e_1 \text{ in } e_2) = (\tau_2, \Gamma_1 \cup \Gamma_2)} \\
\text{CHECK-LET-NOREQ:} & \frac{\tau_1 \text{ is not required} \quad \text{check}(\Gamma, e_1) = (\tau_1, \Gamma_1) \quad \text{check}(\Gamma[x : \tau_1], e_2) = (\tau_2, \Gamma_2) \quad x \notin \text{Domain}(\Gamma_2)}{\text{check}(\Gamma, \text{let } x: \tau_1 = e_1 \text{ in } e_2) = (\tau_2, \Gamma_1 \cup \Gamma_2)} \\
\text{CHECK-LET-FORCE:} & \frac{\tau_1 \Rightarrow \tau \quad \text{require}(\Gamma, e_1, \tau) = \Gamma_1 \quad \text{check}(\Gamma[x : \tau], e_2) = (\tau_2, \Gamma_2) \quad \Gamma_2(x) = \tau}{\text{check}(\Gamma, \text{let } x: \tau_1 = e_1 \text{ in } e_2) = (\tau_2, \Gamma_1 \cup \Gamma_2)}
\end{array}$$

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  $\epsilon$  depending on whether its value belongs to  $V_\epsilon$ . 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.

## B. Extension: First-Class Functions

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

$$\begin{array}{lcl}
\tau & ::= & \dots \\
& | & \tau \rightarrow \tau
\end{array}$$

REQUIRE-VAR-ENV:	$require(\Gamma, x, \tau) = [x : \tau]$
REQUIRE-CONSTENUM:	$\frac{n \in V_\epsilon}{require(\Gamma, n, \epsilon) = \emptyset}$
REQUIRE-SUBSUMPTION:	$\frac{check(\Gamma, e) = (\tau', \Gamma_1) \quad \tau' <: \tau}{require(\Gamma, e, \tau) = \Gamma_1}$
REQUIRE-EXPR-NOREQ:	$\frac{\tau \text{ is not required} \quad check(\Gamma, e) = (\tau, \Gamma_1)}{require(\Gamma, e, \tau) = \Gamma_1}$
REQUIRE-TERNARY:	$\frac{check(\Gamma, e_1) = (\mathbb{Z}, \Gamma_1) \quad require(\Gamma, e_2, \tau) = \Gamma_2 \quad require(\Gamma, e_3, \tau) = \Gamma_3}{require(\Gamma, \text{if0 } e_1 \text{ then } e_2 \text{ else } e_3, \tau) = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3}$
REQUIRE-LET-REQ:	$\frac{\tau_1 \text{ is required} \quad require(\Gamma, e_1, \tau_1) = \Gamma_1 \quad require(\Gamma[x : \tau_1], e_2, \tau_2) = \Gamma_2 \quad x \notin \text{Domain}(\Gamma_2) \text{ or } \tau_1 <: \Gamma_2(x)}{require(\Gamma, \text{let } x : \tau_1 = e_1 \text{ in } e_2, \tau_2) = \Gamma_1 \cup \Gamma_2}$
REQUIRE-LET-NOREQ:	$\frac{\tau_1 \text{ is not required} \quad check(\Gamma, e_1) = (\tau_1, \Gamma_1) \quad require(\Gamma[x : \tau_1], e_2, \tau_2) = \Gamma_2 \quad x \notin \text{Domain}(\Gamma_2)}{require(\Gamma, \text{let } x : \tau_1 = e_1 \text{ in } e_2, \tau_2) = \Gamma_1 \cup \Gamma_2}$
REQUIRE-LET-FORCE:	$\frac{\tau_1 \Rightarrow \tau \quad require(\Gamma, e_1, \tau) = \Gamma_1 \quad require(\Gamma[x : \tau], e_2, \tau_2) = \Gamma_2 \quad \Gamma_2(x) = \tau}{require(\Gamma, \text{let } x : \tau_1 = e_1 \text{ in } e_2, \tau_2) = \Gamma_1 \cup \Gamma_2}$

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  $\tau$  is a subtype of the variable’s original type  $\tau'$ ). REQUIRE-CONSTENUM asserts that a numeric literal has type  $\epsilon$  if its value is in  $V_\epsilon$ . REQUIRE-EXPR-NOREQ processes an expression normally when no special requirement applies. REQUIRE-SUBSUMPTION 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.

$$e ::= \dots$$

$$\quad | \quad \lambda x : \tau. e$$

$$\quad | \quad e \ e$$

This extension adds function types ( $\tau_1 \rightarrow \tau_2$ ), 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.

REQ-ARROW:	$\frac{\tau_2 \text{ is required}}{\tau_1 \rightarrow \tau_2 \text{ is required}}$
SUB-ARROW:	$\frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4}$

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  $\tau_2$  must be  $\epsilon$ , then the function type  $\tau_1 \rightarrow \tau_2$  is considered required). SUB-ARROW is the subtyping rule for function types:  $\tau_1 \rightarrow \tau_2$  is a subtype of  $\tau_3 \rightarrow \tau_4$  if  $\tau_3$  is a subtype of  $\tau_1$  (parameter types are contravariant) and  $\tau_2$  is a subtype of  $\tau_4$  (return types are covariant).

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.

	$\frac{\tau_1 \text{ is required } \quad check(\Gamma[x : \tau_1], e) = (\tau_2, \Gamma_1) \quad x \notin \text{Domain}(\Gamma_1) \text{ or } \tau_1 <: \Gamma_1(x)}{check(\Gamma, \lambda x: \tau_1. e) = (\tau_1 \rightarrow \tau_2, \Gamma_1)}$
CHECK-LAMBDA-REQ:	
	$\frac{\tau_1 \text{ is not required } \quad check(\Gamma[x : \tau_1], e) = (\tau_2, \Gamma_1) \quad x \notin \text{Domain}(\Gamma_1)}{check(\Gamma, \lambda x: \tau_1. e) = (\tau_1 \rightarrow \tau_2, \Gamma_1)}$
CHECK-LAMBDA-NOREQ:	
	$\frac{\tau \Rightarrow \tau_1 \quad check(\Gamma[x : \tau_1], e) = (\tau_2, \Gamma_1) \quad \Gamma_1(x) = \tau_1}{check(\Gamma, \lambda x: \tau. e) = (\tau_1 \rightarrow \tau_2, \Gamma_1)}$
CHECK-LAMBDA-FORCE:	
	$\frac{check(\Gamma, e_1) = (\tau_1 \rightarrow \tau_2, \Gamma_1) \quad require(\Gamma, e_2, \tau_1) = \Gamma_2}{check(\Gamma, e_1 e_2) = (\tau_2, \Gamma_1 \cup \Gamma_2)}$
CHECK-APPLY:	

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.

	$\frac{\tau_1 \text{ is required } \quad require(\Gamma[x : \tau_1], e, \tau_2) = \Gamma_1 \quad x \notin \text{Domain}(\Gamma_1) \text{ or } \tau_1 <: \Gamma_1(x)}{require(\Gamma, \lambda x: \tau_1. e, \tau_1 \rightarrow \tau_2) = \Gamma_1}$
REQUIRE-LAMBDA-REQ:	
	$\frac{\tau_1 \text{ is not required } \quad require(\Gamma[x : \tau_1], e, \tau_2) = \Gamma_1 \quad x \notin \text{Domain}(\Gamma_1)}{require(\Gamma, \lambda x: \tau_1. e, \tau_1 \rightarrow \tau_2) = \Gamma_1}$
REQUIRE-LAMBDA-NOREQ:	
	$\frac{\tau \Rightarrow \tau_1 \quad require(\Gamma[x : \tau_1], e, \tau_2) = \Gamma_1 \quad \Gamma_1(x) = \tau_1}{require(\Gamma, \lambda x: \tau. e, \tau_1 \rightarrow \tau_2) = \Gamma_1}$
REQUIRE-LAMBDA-FORCE:	
	$\frac{check(\Gamma, e_1) = (\tau_1 \rightarrow \tau_3, \Gamma_1) \quad \tau_3 \Rightarrow \tau_2 \quad require(\Gamma, e_1, \tau_1 \rightarrow \tau_2) = \Gamma_2 \quad require(\Gamma, e_2, \tau_1) = \Gamma_3}{require(\Gamma, e_1 e_2, \tau_2) = \Gamma_2 \cup \Gamma_3}$
REQUIRE-APPLY:	

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  $\epsilon$ ). 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  $\tau_1$  if it was too general), and then requires the argument to have that parameter type  $\tau_1$ .

## 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  $\epsilon$  to  $\epsilon$  (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  $\epsilon$  can be treated as a subtype of the integer type (meaning that  $\epsilon$  can be safely translated as a Rust enum type).
- 2) Suppose  $check(\emptyset, e') = (\tau, \Gamma_1)$  (the type checker infers type  $\tau$  for  $e'$  and produces environment  $\Gamma_1$ ). If we then modify the declared type of every variable  $x$  found in  $\Gamma_1$  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  $\Gamma_1$ , applying those types back to the program yields a consistent, type-correct program.
- 3) Only necessary changes are made to the program  $e'$  to produce  $e''$ . In other words, reverting any change of type annotation in  $e''$  as in  $e'$  will result in a type error.

*Proof.* See Appendix. □

## B. Examples

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

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

In this code,  $y$  is explicitly declared with type  $\epsilon$ , 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  $\epsilon$  because  $x$  is initialized to 0 (and we assumed  $0 \in V_\epsilon$ ). Inferring  $x:\epsilon$  is valid here since 0 is one of  $\epsilon$ 's variant values. In contrast,  $z$  is declared as an  $\mathbb{Z}$  (integer) and has no connection to  $\epsilon$  apart from coincidentally being assigned 0. Therefore,  $z$  should *not* be inferred as type  $\epsilon$ .

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

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

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

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

$$\begin{aligned} &\text{let } \text{ctoi}:\epsilon \rightarrow \mathbb{Z} = \lambda c:\epsilon.(3 \times c) \text{ in} \\ &\quad \text{let } \text{green}:\mathbb{Z} = 1 \text{ in} \\ &\quad \text{ctoi } \text{green} \end{aligned}$$

In this code, the function `ctoi` expects a parameter of type  $\epsilon$  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  $\epsilon$  (since `ctoi` requires an  $\epsilon$  argument). Therefore, `green` should be inferred to have type  $\epsilon$ . Applying the formal rules confirms this inference, resulting in the following well-typed code (with `green` now declared as type  $\epsilon$ ):

$$\begin{aligned} &\text{let } \text{ctoi}:\epsilon \rightarrow \mathbb{Z} = \lambda c:\epsilon.(3 \times c) \text{ in} \\ &\quad \text{let } \text{green}:\epsilon = 1 \text{ in} \\ &\quad \text{ctoi } \text{green} \end{aligned}$$

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. Finally, by eliminating the need for downcasting in the translated code, our approach reduces the reliance on unsafe Rust constructs, making the resulting code safer and more idiomatic.

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:

```
1 enum Day {
2     Mon, Tue, Wed, Thu, Fri, Sat, Sun
3 };
4
5 enum Day next_day(enum Day day) {
6     return (day + 1) % 7;
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.

## APPENDIX PRELIMINARIES

We define the several notations and rules for the formal proof:

- **Forcing rules:** The **Forcing Rules** are the set of rules that force a type to be a subtype of another type, including CHECK-LAMBDA-FORCE, REQUIRE-LAMBDA-FORCE, CHECK-LET-FORCE, and REQUIRE-LET-FORCE.
- **Type kinds:** We say the types  $\tau_1$  and  $\tau_2$  are of the same kind, denoted as  $\tau_1 \sim \tau_2$ , by the following rules:

$$\tau \sim \tau \quad \frac{\tau_1 \sim \tau_2 \quad \tau_2 \sim \tau_3}{\tau_1 \sim \tau_3} \quad \frac{\tau_2 \sim \tau_1}{\tau_1 \sim \tau_2} \quad \frac{\tau_1 <: \tau_2}{\tau_1 \sim \tau_2} \quad \frac{\tau_1 \sim \tau_3 \quad \tau_2 \sim \tau_4}{\tau_1 \rightarrow \tau_2 \sim \tau_3 \rightarrow \tau_4}$$

- **Type Environment kinds:** We say the type environments  $\Gamma_1$  and  $\Gamma_2$  are of the same kind, denoted as  $\Gamma_1 \sim \Gamma_2$ , if  $\text{Domain}(\Gamma_1) = \text{Domain}(\Gamma_2)$  and for all  $x \in \text{Domain}(\Gamma_1)$ ,  $\Gamma_1(x) \sim \Gamma_2(x)$ .
- **Bounds:** We say the type  $\tau = \tau_1 \sqcup \tau_2$  is the upper bound of a types  $\tau_1$  and  $\tau_2$  if  $\tau_1 <: \tau$  and  $\tau_2 <: \tau$ . We say the type  $\tau = \tau_1 \sqcap \tau_2$  is the lower bound of a types  $\tau_1$  and  $\tau_2$  if  $\tau <: \tau_1$  and  $\tau <: \tau_2$ .
- **Expression cases:** For the inductive steps, we name the cases of expressions as follows:
  - LET: let  $x: \tau = e$  in  $e$
  - VAR:  $x$
  - CONST:  $n$
  - BINOP:  $e \oplus e$
  - TERNARY: if0  $e$  then  $e$  else  $e$
  - LAMBDA:  $\lambda x: \tau. e$
  - APPLY:  $e e$

The base cases of the expression are VAR and CONST. Other cases are inductive.

- **Type cases:** For the type, we name the cases as follows:
  - EPSILON:  $\epsilon$
  - INTEGER:  $\mathbb{Z}$
  - ARROW:  $\tau \rightarrow \tau$

The base cases of the type are EPSILON and INTEGER. The ARROW type is inductive.

- **Environments:** To avoid confusion, we call  $\Gamma$  the **type environment** if it is used in the first argument of the *check* or *require* functions, and we call it the **associated type environment** if it is returned by the *check* or *require* functions.
- **Applying an associated type environment:** We call the process of applying an associated type environment  $\Gamma$  to an expression  $e$  as **applying the environment** to  $e$ , and denote it as  $\Gamma[e]$ . This is the process on the proposition 2, where we apply the associated type environment  $\Gamma_1$  to the expression  $e'$  to produce a new expression  $e''$  (i.e.,  $e'' = \Gamma_1[e']$ ). Following is the formal definition of applying an associated type environment to an expression:
  - VAR:  $\Gamma[x]$  is  $x$ .
  - CONST:  $\Gamma[n]$  is  $n$ .
  - LET: If  $x \in \text{Domain}(\Gamma)$  and  $\Gamma(x) <: \tau$ ,  $\Gamma[\text{let } x: \tau = e_1 \text{ in } e_2]$  is  $\text{let } x: \Gamma(x) = \Gamma[e_1] \text{ in } \Gamma[e_2]$ . Otherwise,  $\Gamma[\text{let } x: \tau = e_1 \text{ in } e_2]$  is  $\text{let } x: \tau = \Gamma[e_1] \text{ in } \Gamma[e_2]$ .
  - BINOP:  $\Gamma[e_1 \oplus e_2]$  is  $\Gamma[e_1] \oplus \Gamma[e_2]$ .
  - TERNARY:  $\Gamma[\text{if0 } e_1 \text{ then } e_2 \text{ else } e_3]$  is  $\text{if0 } \Gamma[e_1] \text{ then } \Gamma[e_2] \text{ else } \Gamma[e_3]$ .
  - LAMBDA: If  $x \in \text{Domain}(\Gamma)$  and  $\Gamma(x) <: \tau$ ,  $\Gamma[\lambda x: \tau. e]$  is  $\lambda x: \Gamma(x). \Gamma[e]$ . Otherwise,  $\Gamma[\lambda x: \tau. e]$  is  $\lambda x: \tau. \Gamma[e]$ .
  - APPLY:  $\Gamma[e_1 e_2]$  is  $\Gamma[e_1] \Gamma[e_2]$ .
- **Declaration on expression:** If an expression  $e$  contains a subexpression  $\text{let } x: \tau = e_1 \text{ in } e_2$  or  $\lambda x: \tau. e$ , we say that  $e$  has a **declaration** of  $x$ .

## PROPERTIES

**Lemma 1.** *If  $\tau_1 \sim \tau_2$ , then  $\tau_1 \sqcup \tau_2$  and  $\tau_1 \sqcap \tau_2$  exists.*

*Proof.* We use induction on the type  $\tau_1$ .

**Base Cases:**

- EPSILON:  $\epsilon \sqcup \tau_2 = \tau_2$  and  $\epsilon \sqcap \tau_2 = \epsilon$ .
- INTEGER:  $\mathbb{Z} \sqcup \tau_2 = \tau_2$  and  $\mathbb{Z} \sqcap \tau_2 = \mathbb{Z}$ .

### Inductive Cases:

- **ARROW:** By the inductive hypothesis, we have  $\tau_1 \sim \tau_3$  and  $\tau_2 \sim \tau_4$ . Then, we have  $(\tau_1 \rightarrow \tau_2) \sqcup (\tau_3 \rightarrow \tau_4) = (\tau_1 \sqcap \tau_3) \rightarrow (\tau_2 \sqcup \tau_4)$  and  $(\tau_1 \rightarrow \tau_2) \sqcap (\tau_3 \rightarrow \tau_4) = (\tau_1 \sqcup \tau_3) \rightarrow (\tau_2 \sqcap \tau_4)$ .

Therefore, we conclude that  $\tau_1 \sqcup \tau_2$  and  $\tau_1 \sqcap \tau_2$  always exist.  $\square$

**Lemma 2.** For all  $\Gamma_1, \Gamma_2$  and  $e$  where  $\Gamma_1 \sim \Gamma_2$ , if  $\text{check}(\Gamma_1, e) = (\tau_1, \Gamma'_1)$  and  $\text{check}(\Gamma_2, e) = (\tau_2, \Gamma'_2)$ , then  $\tau_1 \sim \tau_2$ . Also, if  $\text{require}(\Gamma, e, \tau_1)$  and  $\text{require}(\Gamma, e, \tau_2)$  both succeed, then  $\tau_1 \sim \tau_2$ .

*Proof.* Easy.  $\square$

**Lemma 3.** If  $\Gamma_1 \sim \Gamma_2$  and  $\tau_1 \sim \tau_2$ , then  $\Gamma_1[x : \tau_1] \sim \Gamma_2[x : \tau_2]$ .

*Proof.* Easy.  $\square$

**Theorem 1.** For all  $\Gamma_1, \Gamma_2$  and  $e$  where  $\Gamma_1 \sim \Gamma_2$ , suppose there exists a type  $\tau$  where  $\text{require}(\Gamma_1, e, \tau)$  and  $\text{require}(\Gamma_2, e, \tau)$  both succeed with  $\text{require}(\Gamma_1, e, \tau) = \Gamma'_1$  and  $\text{require}(\Gamma_2, e, \tau) = \Gamma'_2$ . Then,  $\Gamma'_1 = \Gamma'_2$ . Also, if  $\text{check}(\Gamma_1, e)$  and  $\text{check}(\Gamma_2, e)$  both succeed with  $\text{check}(\Gamma_1, e) = (\tau_1, \Gamma'_1)$  and  $\text{check}(\Gamma_2, e) = (\tau_2, \Gamma'_2)$ , then  $\Gamma'_1 = \Gamma'_2$ .

*Proof.* We use induction on the expression  $e$ .

### Base Cases:

- **CONST  $n$ :** According to the rule REQUIRE-CONSTENUM, CHECK-CONSTENUM and CHECK-CONSTNONENUM, the associated type environment is always  $\emptyset$ .
- **VAR  $x$ :** According to the rule REQUIRE-VAR-ENV, the associated type environment by *require* is always  $[x : \tau]$ . The rule CHECK-VAR always gives the empty environment  $\emptyset$ .

### Inductive Cases:

- **LET let  $x: \tau_1 = e_1$  in  $e_2$ :** For *require*, we have 3 rules: REQUIRE-LET-REQ, REQUIRE-LET-NOREQ, and REQUIRE-LET-FORCE. All of them gets two associated type environments for  $e_1$  and  $e_2$ , and gives a union of the two as the result. Therefore, it is sufficient to show that the two associated type environments are the same for  $e_1$  and  $e_2$ .
  - For  $e_2$ , all the rules use *require* with the same kind of type environment, the same expression, and the same type. Therefore, by the induction hypothesis, we will have the same associated type environment.
  - For  $e_1$ , the given type environment is the same kind for all the rules. If  $\tau_1$  is required, then  $e$  will check the rule REQUIRE-LET-REQ. If  $\tau_1$  is not required, then  $e$  will check the rule REQUIRE-LET-NOREQ only if  $x$  is not in the domain of the associated type environment of  $e_2$ . Otherwise, it will check the rule REQUIRE-LET-FORCE. Therefore, the rule which will be used for *require* depends on the expression  $e$ , not the type environment  $\Gamma_1$  or  $\Gamma_2$ . Using this fact, deriving  $\Gamma'_1 = \Gamma'_2$  is straightforward with the induction hypothesis.

In this case, *check* is similar to *require*.

- **BINOP  $e_1 \oplus e_2$ :** *require* will not succeed if the type  $\tau$  is required. Otherwise,  $\text{check}(\Gamma, e) = (\tau, \Gamma') \Rightarrow \text{require}(\Gamma, e, \tau) = \Gamma'$  by the rule REQUIRE-EXPR-NOREQ. By the induction hypothesis and the rule CHECK-BINOP, we have  $\Gamma'_1 = \Gamma'_2$ .
- **TERNARY:** By the induction hypothesis and the rules REQUIRE-TERNARY and CHECK-TERNARY, we have  $\Gamma'_1 = \Gamma'_2$ .
- **LAMBDA:** It's basically same as LET case without  $e_1$ , thus easier to prove.
- **APPLY  $e_1 e_2$ :** By the induction hypothesis and the rule CHECK-APPLY, we have  $\Gamma'_1 = \Gamma'_2$  for *check*. For *require*:
  - By inductive hypothesis, suppose  $\text{check}(\Gamma_1, e_1) = (\tau_1, \Gamma_a)$  and  $\text{check}(\Gamma_2, e_1) = (\tau_2, \Gamma_b)$ .
  - By Lemma 2, we have  $\tau_1 \sim \tau_2$ .
  - By Lemma 1, let  $\tau = \tau_1 \sqcup \tau_2$ .
  - In order to pass the rule REQUIRE-APPLY,  $\tau_1 = \tau_{1a} \rightarrow \tau_{1b}$ ,  $\tau_2 = \tau_{2a} \rightarrow \tau_{2b}$ , and  $\tau_{1a} \Rightarrow \tau$  and  $\tau_{2a} \Rightarrow \tau$ .
  - Suppose  $\tau_u = \tau_{ua} \rightarrow \tau_{ub}$ . Then,  $\tau_{1b} <: \tau_{ub}$  and  $\tau_{2b} <: \tau_{ub}$ . Therefore, we conclude that  $\tau_{ub} \Rightarrow \tau$ .
  - By the rule CHECK-SUBSUMPTION, we have  $\text{check}(\Gamma_1, e_1) = (\tau_u, \Gamma_c)$  and  $\text{check}(\Gamma_2, e_1) = (\tau_u, \Gamma_d)$ .
  - Using  $\tau_u$  in the rule REQUIRE-APPLY and by the induction hypothesis, we have  $\Gamma'_1 = \Gamma'_2$ .

$\square$

**Theorem 2.** For some  $\tau$ , if  $\text{check}(\Gamma, e)$  or  $\text{require}(\Gamma, e, \tau)$  succeeds without any **Forcing Rules** applied, then  $e$  is well-typed under the type environment  $\Gamma$ .

*Proof.* Easy.  $\square$

**Theorem 3.** For some  $\tau$ , if  $\text{check}(\Gamma, e)$  or  $\text{require}(\Gamma, e, \tau)$  succeeds with some **Forcing Rules** applied, then  $e$  is not well-typed under the type environment  $\Gamma$ .

*Proof.* Easy.  $\square$

**Lemma 4.** Suppose  $check(\Gamma, e) = (\tau, \Gamma')$  or  $require(\Gamma, e, \tau) = \Gamma'$ . For  $x \in \text{Domain}(\Gamma')$ , either  $e$  contains a declaration of  $x$  or  $x \in \text{Domain}(\Gamma)$ .

*Proof.* Easy. □

**Lemma 5.** If there's no such  $x \in \text{Domain}(\Gamma_1)$  such that  $e$  contains a declaration of  $x$ , then  $\Gamma_2[e] = (\Gamma_1 \cup \Gamma_2)[e]$  for all  $\Gamma_1, \Gamma_2$  and  $e$ .

**Theorem 4.** If  $check(\Gamma, e) = (\tau, \Gamma')$ , then  $check(\Gamma, \Gamma'[e])$  produces the same result (both type and associated type environment) without any **Forcing Rules** applied. Also, if  $require(\Gamma, e, \tau) = \Gamma'$ , then  $require(\Gamma, \Gamma'[e], \tau)$  produces the same result without any **Forcing Rules** applied.

*Proof.* We use induction on the expression  $e$ .

**Base Cases:** Easy.

**Inductive Cases:**

- BINOP  $e_1 \oplus e_2$ : No *require* rule for this case, so we only consider *check*.
  - By the rule CHECK-BINOP, we have  $check(\Gamma, e_1) = (\mathbb{Z}, \Gamma_1)$  and  $check(\Gamma, e_2) = (\mathbb{Z}, \Gamma_2)$ .
  - If  $e_1$  has a declaration of  $x$ , then we get  $x \notin \text{Domain}(\Gamma)$  by Lemma 4. Also, by the no-naming-collision assumption,  $e_2$  does not have a declaration of  $x$ .
  - By Lemma 4, we have  $x \notin \text{Domain}(\Gamma_2)$ . Therefore, by Lemma 5, we have  $\Gamma_1[e_1] = (\Gamma_1 \cup \Gamma_2)[e_1]$ .
  - In the same way, we have  $\Gamma_2[e_2] = (\Gamma_1 \cup \Gamma_2)[e_2]$ .
  - By inductive hypothesis, we have  $check(\Gamma, \Gamma_1[e_1]) = (\mathbb{Z}, \Gamma_1)$  and  $check(\Gamma, \Gamma_2[e_2]) = (\mathbb{Z}, \Gamma_2)$ .
  - Using the fact that  $\Gamma_1[e_1] = (\Gamma_1 \cup \Gamma_2)[e_1]$  and  $\Gamma_2[e_2] = (\Gamma_1 \cup \Gamma_2)[e_2]$ , we have  $check(\Gamma, \Gamma'[e]) = (\mathbb{Z}, \Gamma')$ .
- TERNARY: Both *require* and *check* cases are similar to BINOP case.
- APPLY: Similar to TERNARY case.
- LET let  $x: \tau_1 = e_1$  in  $e_2$ : First of all, in the same way as in the BINOP case, we can show that  $\Gamma_1[e_1] = (\Gamma_1 \cup \Gamma_2)[e_1]$  and  $\Gamma_2[e_2] = (\Gamma_1 \cup \Gamma_2)[e_2]$ .

For *require*, we have 3 rules: REQUIRE-LET-REQ, REQUIRE-LET-NOREQ, and REQUIRE-LET-FORCE. Our objective is to show that  $require(\Gamma, \Gamma'[e], \tau)$  passes either REQUIRE-LET-REQ or REQUIRE-LET-NOREQ.

We consider the following cases of which rule is used for  $require(\Gamma, e, \tau)$ :

- REQUIRE-LET-REQ: If  $e$  passed this rule, then type annotation of  $x$  (which is  $\tau_1$ ) is unchanged in  $\Gamma'[e]$ . Therefore, the statement holds by the inductive hypothesis.
- REQUIRE-LET-NOREQ: If  $e$  passed this rule, then type annotation of  $x$  (which is  $\tau_1$ ) is unchanged in  $\Gamma'[e]$ . Therefore, the statement holds by the inductive hypothesis.
- REQUIRE-LET-FORCE: Suppose  $e$  passed this rule by forcing the type  $\tau_1$  to be  $\tau'$ . Then, for the associated type environment  $\Gamma_2$  produced by  $require(\Gamma[x : \tau'], e_2, \tau)$ , we have  $\Gamma_2(x) = \tau'$ . The type annotation of  $x$  in  $\Gamma'[e]$  is  $\tau'$ , so  $require(\Gamma, \Gamma'[e], \tau)$  now passes the rule REQUIRE-LET-REQ since  $\tau'$  is required,  $require(\Gamma, \Gamma_1[e_1], \tau') = \Gamma_1$  and  $require(\Gamma[x : \tau'], \Gamma_2[e_2], \tau) = \Gamma_2$  (by inductive hypothesis).

Therefore, we conclude that  $require(\Gamma, \Gamma'[e], \tau)$  passes either REQUIRE-LET-REQ or REQUIRE-LET-NOREQ, and  $require(\Gamma, \Gamma'[e], \tau) = \Gamma'$ .

For *check*, we can show that  $check(\Gamma, \Gamma'[e])$  doesn't pass any **Forcing Rules**, and  $check(\Gamma, \Gamma'[e]) = (\tau, \Gamma')$  in the same way as above.

- LAMBDA: Easier than LET case.

□

**Lemma 6.** The associated type environment  $\Gamma$  produced by *check* or *require* can only have the required types. i.e., for all  $x \in \text{Domain}(\Gamma)$ ,  $\Gamma(x)$  is required.

*Proof.* Easy. □

**Theorem 5.** If  $check(\Gamma, e) = (\tau, \Gamma')$ , and for any  $\Gamma'' \subset \Gamma'$  where  $\Gamma''[e] \neq \Gamma'[e]$ ,  $check(\Gamma, \Gamma''[e])$  produces the same result, but still uses **Forcing Rules**. Also, if  $require(\Gamma, e, \tau) = \Gamma'$ , and for any  $\Gamma'' \subset \Gamma'$  where  $\Gamma''[e] \neq \Gamma'[e]$ ,  $require(\Gamma, \Gamma''[e], \tau)$  produces the same result, but still uses **Forcing Rules**.

*Proof.* Similar induction as in Theorem 4.

**Base Cases:** Can't happen since  $e = \Gamma'[e] = \Gamma''[e]$ . Therefore, we only consider the inductive cases.

**Inductive Cases:**

- BINOP  $e_1 \oplus e_2$ : By the assumption, we have  $\Gamma''[e_1] \neq \Gamma'[e_1]$  or  $\Gamma''[e_2] \neq \Gamma'[e_2]$ . We use inductive hypothesis on the subexpression where the difference occurs. Therefore, we conclude that  $check(\Gamma, \Gamma''[e])$  or  $require(\Gamma, \Gamma''[e], \tau)$

uses **Forcing Rules**. Also, by using inductive hypothesis and Theorem 4, we have  $check(\Gamma, \Gamma''[e]) = (\tau, \Gamma')$  or  $require(\Gamma, \Gamma''[e], \tau) = \Gamma'$ , therefore the same result is produced.

- TERNARY: Similar to BINOP case.
- APPLY: Similar to TERNARY case.
- LET let  $x: \tau_1 = e_1$  in  $e_2$ : For *require*,
  - **Case 1:**  $\Gamma'[e]$  and  $\Gamma''[e]$  has different type annotation for  $x$ . i.e.,  $x \in \text{Domain}(\Gamma') \setminus \text{Domain}(\Gamma'')$ , and  $\Gamma'(x) <: \tau_1$ . Since  $\Gamma'[e]$  changed the type annotation on  $x$ . According to the LET case in the proof of Theorem 4,  $require(\Gamma, e, \tau)$  passes the rule REQUIRE-LET-FORCE. Suppose the type of  $x$  has been forced to  $\tau'$ . This indicates  $require(\Gamma[x : \tau'], e_2, \tau) = \Gamma_2$  for some  $\Gamma_2$  and  $\Gamma_2(x) = \tau'$ . Assume  $require(\Gamma, \Gamma''[e], \tau)$  passes the rule:
    - \* REQUIRE-LET-REQ:
      - By Theorem 1, we have  $require(\Gamma[x : \tau_1], e_2, \tau) = \Gamma_2$  since  $\Gamma[x : \tau'] \sim \Gamma[x : \tau_1]$ .
      - By the inductive hypothesis and Theorem 4, we have  $require(\Gamma[x : \tau_1], \Gamma_2[e_2], \tau) = \Gamma_2$ .
      - In the same way as above, we also have  $require(\Gamma[x : \tau_1], \Gamma''[e_2], \tau) = \Gamma_2$ .
      - Therefore, the rule REQUIRE-LET-REQ fails, which is a contradiction.
    - \* REQUIRE-LET-NOREQ: In the same way as above, we get a contradiction on  $x \notin \text{Domain}(\Gamma_2)$ .
    - \* REQUIRE-LET-FORCE: This implies that  $require(\Gamma, \Gamma''[e], \tau)$  passes the **Forcing Rules**. It can be shown that  $require(\Gamma, \Gamma''[e], \tau)$  produces the same result as  $require(\Gamma, \Gamma'[e], \tau)$  by the inductive hypothesis, and the Theorem 4.
  - **Case 2:**  $\Gamma'[e]$  and  $\Gamma''[e]$  has the same type annotation for  $x$ : similar to the BINOP case.
- LAMBDA: Similar to LET case.

□

#### PROOF OF PROPOSITIONS

*Proof.* Proving each propositions are straightforward using the above theorems.

- 1) By the proposition 2.
- 2) Case where  $\Gamma = \emptyset$  in the Theorem 4 and use Theorem 2.
- 3) Case where  $\Gamma = \emptyset$  in the Theorem 5 and use Theorem 3.

□