Formal Proofs for the Subtype–Inference Report

Minwook Lee
ryan-lee@kaist.ac.kr

I. Preliminaries

We work with the type system, subtyping relation, and `check`/`require` rules introduced in the report — reproduced in *Figures 1–6* of the original manuscript. We recall only the facts needed for the proofs.

- The distinguished enum type is denoted by ϵ and its set of variant values by $V_\epsilon \subseteq \mathbb{Z}$.
- Subtyping contains a single proper rule for ϵ:

$$\epsilon <: \mathbb{Z}.$$

- A successful judgement $check(\Gamma, e) = (\tau, \Gamma_1)$ certifies that e is well-typed, returns type τ, and yields the additional environment Γ_1 (new "requirements").
- The companion judgement $require(\Gamma, e, \tau) = \Gamma_1$ demands that e have *exactly* type τ; it may refine the environment when the demand forces a narrower type.

The proofs below proceed by structural induction on the successful derivations returned by the `check` algorithm.

II. Main Results

Proposition 1 (Safety of ϵ as a Subtype)**.** *If $check(\emptyset, e') = (\tau, \Gamma_1)$ succeeds, then every run-time value inhabiting an expression that the derivation deems to have type ϵ is drawn from V_ϵ; furthermore, any occurrence of an ϵ-typed expression in a $\mathbb{Z}$-context is mediated solely by the subtyping rule $\epsilon <: \mathbb{Z}$. Consequently, replacing ϵ with a real Rust enum is type-safe.*

Proof. Induct on the depth of the successful `check` derivation for e'.

Base cases. When e' is a constant n or a variable x:

- For a constant n, the derivation uses either Check-ConstEnum (when $n \in V_\epsilon$) or Check-ConstNonEnum. In the first subcase n is literally a variant value; in the second subcase n is assigned the super-type $\mathbb{Z}$, never ϵ.
- For a variable x, the only applicable rule is Check-Var. The premise reads $\Gamma(x) = \tau$ for some τ. If $\tau = \epsilon$ the induction claim holds vacuously because the same environment already records that x must be a value of V_ϵ. If $\tau \neq \epsilon$ the variable is typed as $\mathbb{Z}$ (or another supertype) and never misclassified as ϵ.

Inductive cases. Representative examples follow; all other syntactic forms are analogous.

Binary operation $e_1 \oplus e_2$.: The only typing rule is Check-Binop, whose premises require e_1 and e_2 to each have type $\mathbb{Z}$ (up to subsumption). By the IH, if either sub-expression owns type ϵ it will be immediately upcast via $\epsilon <: \mathbb{Z}$ before the rule is applied. Thus no $\mathbb{Z}$-context ever contains a run-time value outside V_ϵ masquerading as ϵ.

Let expression let $x:, \tau_1 = e_1$ in e_2.: We discuss the most delicate variant, Check-Let-Force. The premises state (i) $\tau_1 \Rightarrow \tau$ with τ required; (ii) $require(\Gamma, e_1, \tau) = \Gamma_1$; and (iii) $check(\Gamma[x{:}\tau], e_2) = (\tau_2, \Gamma_2)$ with $\Gamma_2(x) = \tau$. When τ is ϵ the requirement judgement enforces e_1 to produce only values in V_ϵ. The third premise guarantees that x remains ϵ throughout e_2, hence every subsequent use satisfies the IH. Thus x never leaves the safe range.

All remaining constructs (if0, lambdas, application) observe similar reasoning: the unique subtyping rule $\epsilon <: \mathbb{Z}$ is the only gateway through which an ϵ value enters an integer context, while the converse direction is blocked by the absence of a rule $\mathbb{Z} <: \epsilon$. Consequently, no ill-formed integer can flow into an ϵ slot, and the proposition holds universally. □

Proposition 2 (Soundness of Type Reannotation)**.** *Let $check(\emptyset, e') = (\tau, \Gamma_1)$ succeed and form a new program e'' by updating every variable declaration x to the (possibly stricter) type $\Gamma_1(x)$. Then e'' is well typed, i.e. $check(\emptyset, e'') = (\tau, \Gamma_2)$ succeeds for some Γ_2.*

Proof. Proceed again by induction on the structure of e', reusing the derivation for e'. The crux is that e'' merely *adds* the refinements discovered during the original run; all premises needed by the typing rules are therefore already satisfied.

Variables.: If $x \notin \mathrm{dom}(\Gamma_1)$ its declaration in e'' is unchanged; the original premise $\Gamma(x) = \tau$ remains valid. If $x \in \mathrm{dom}(\Gamma_1)$ the declaration becomes $x{:}\Gamma_1(x)$, which is exactly the type the previous derivation assumed whenever x was used. Hence every instance of Check-Var stays valid.

Let expressions.: Consider the forced case analysed above. In e'' the declaration is already the required subtype τ; consequently the verifier can apply the simpler rule Check-Let-Req, recycling the original subderivations for e_1 and e_2. The normal and required cases are even more direct.

Other constructs.: All remaining rules depend only on the types of subexpressions. By the IH each subexpression remains typable with the same (or more specific) type, so the outer rule still applies.

Thus the whole derivation for e' can be replayed for e'' without failure. □

III. Consequences for Translation

By Propositions 1 and 2 the subtype inference algorithm described in the report is *sound*: whenever it elects to translate a C enum as a proper Rust enum, the resulting program is type safe; moreover, the algorithm can materialise its inferences as explicit annotations while preserving well-typedness. These two facts jointly justify the proposed enhancement to the C2Rust pipeline.

References

[1] M. Lee, "Enhancing C–to–Rust Translation with Subtype Inference for Enum Types," 2025.