Interoperation Without an Interface: C++ and Rust in One Translation Unit, and One Toolchain

B.S. Hartshorn https://github.com/brentharts/crust

August 17, 2026

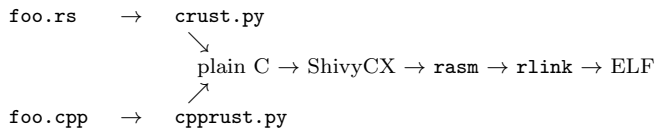
Abstract

Cross-language interoperation is normally a matter of agreeing on a *boundary*: a foreign function interface, a generated shim, a marshalling layer, or a shared intermediate representation. We describe a system in which no boundary exists. Subsets of C++ and of Rust are each lowered to plain C source, and one C compiler consumes the result, so a Rust function calling a C++ method is a C function calling a C function — same translation unit, same intermediate representation, same register allocator. The two lowerings are chosen to converge on identical symbols, with the consequence that a Rust `impl Drop` and a C++ destructor over the same data emit the *same function*, and a C++ class may hold a Rust value by value without a shim, a wrapper, or an annotation.

Removing the boundary between the languages exposes a second one, below them: the artifact is C, but a C compiler, an assembler, a linker and a runtime still stand between C and a running program, and conventionally those are separate projects. We report a stack in which they are not. The whole path — two source languages to C, C to machine code, machine code to a static ELF executable — is roughly fifty thousand lines in one repository, with no external compiler, assembler, linker or C library involved.

1 Introduction

Two languages that must call each other usually meet at an interface, and that interface is where the optimizer stops. Even a zero-copy foreign function interface is an opaque call: the compiler on either side cannot see through it, cannot inline across it, and cannot reason about lifetimes that cross it. Worse, the two sides disagree about what an object *is*. A C++ class with a destructor and a Rust type with `impl Drop` are the same idea expressed twice, but no FFI lets one hold the other by value, because neither compiler will emit a call to the other's destructor. Lowering a high-level language to C is not new. C++ itself was first implemented this way, Cfront having translated C++ to C for most of the 1980s [1].



(gcc or clang may replace the second stage; nothing below C is required to come from outside.)

By the time anything is compiled there is no C++ and no Rust left. This continues the approach of [2], where multiple front ends share one back end in order to build and boot an operating system; here the same mechanism is turned toward the narrower question of whether two languages with incompatible object models can share one.

Our observation is not that C++ can be lowered to C, which Cfront settled. It is that *Rust can be too*, and that if both lowerings are performed by front ends written in coordination, the naming and calling conventions can be made to *converge* rather than merely coexist. Interoperation then costs nothing because there is nothing to interoperate: after lowering, both languages have produced C functions over C structs, and the ones that mean the same thing have the same name.

Removing the boundary between the two languages, however, only relocates the question of where the seams are. C is not an executable. Between C and a running program lie a C compiler, an assembler, a linker and a runtime, and in the conventional arrangement each of those is a separate project with its own release cycle: `gcc` or `clang`, then `gas` and `ld` from binutils, then glibc or musl. A language front end that lowers to C inherits all of them. The second half of this paper reports what happens when it does not — when the compiler, assembler, linker and runtime are the same codebase as the front ends, small enough to read, and therefore cheap to extend downward as well as upward.

1.1 Contributions

- A taxonomy of interoperation by *where two languages meet* — at an interface, an IR, a successor language, or by elimination — and a fifth point, meeting at the symbol, which this system occupies (§2).
- Coordinated lowerings from subsets of C++ and Rust to C whose symbol conventions agree, so a C++ class may hold a Rust value by value and destroy it correctly with no destructor written (§3), together with the defect classes that cross the shared representation and are detected by construction (§6).
- The limit of that convergence: two constructs both called *move*, sharing a symbol and requiring opposite drop obligations, and an account of why choosing wrongly between them is invisible to testing (§3.1).
- *Rust as oracle and as second target*: a third lowering raising the C++ subset to deliberately restrictive Rust, so `rustc`'s borrow checker independently audits the ownership the C lowering inferred — and so that where the C lowering refuses, a Rust target remains (§8).
- A self-contained stack beneath the seam — C compiler, assembler, linker and freestanding runtime in one repository, about fifty thousand lines — and the measured cost of extending it: two bare-metal ISAs brought to full corpus coverage for roughly 2,400 and 2,000 lines each, over a shared 190-line register allocator, each step validated differentially against the GNU toolchain (§5).

2 Where two languages meet

Interoperation mechanisms by the location of the seam, because the seam determines what is lost.

At an interface. The dominant answer. SWIG [3] generates glue between C/C++ and scripting languages; `bindgen` and `cbindgen` [4] generate Rust declarations from C headers and vice versa; the `cxx` crate [5] and `autocxx` [6] generate checked bidirectional bindings between Rust and C++.

At an intermediate representation. The strongest technical alternative, and the one this work must answer. Both `clang` and `rustc` emit LLVM IR [7], so cross-language link-time optimisation can already inline a Rust function into a C++ caller. The JVM and CLR make the same move at a higher level.

We state the comparison honestly. For *performance*, cross-language LTO obtains much of what we obtain, and we do not claim to beat it. What it does not obtain is convergence of the *object model*. LTO can inline across an FFI boundary [8]; it cannot make `Vec_int_drop` and a C++ destructor be the same symbol. IR-level sharing also pins the toolchain: the artifact is LLVM IR, consumable by LLVM. Our artifact is C, consumable by any C compiler, and auditable by a human.

At a successor language. Rather than joining two languages, introduce a third that subsumes one. Carbon [9] is explicitly designed as a C++ successor; `cppfront` [10] compiles an alternative C++ syntax to standard C++; D offers `extern(C++)`. Recent systems languages — Zig [11], C3 [12], Odin — take C as the thing to succeed.

By elimination. Translate one language away. C2Rust [13] converts C to Rust that runs but sits outside the safe subset, inheriting the memory-safety defects of the original [15]. A substantial literature then *refines* that output toward safety: Crown [16] and Emre et al. [17] infer ownership and lifetimes; Hong and Ryu replace output parameters with algebraic data types [18]. Scylla [19] compiles an alias-free C subset to safe Rust directly. A parallel line uses large language models with repair-and-validate loops [20, 21].

Most directly relevant is Cpp2Rust [14], which translates a C++ subset into *safe* Rust automatically — the first system to do so — by deferring ownership and mutability to run time via `Rc<RefCell<T>>` and a weak-reference pointer type, then recovering performance with a source-to-source optimizer. We return to it in §8, because our third lowering deliberately inverts its central design choice for a different purpose.

Elimination is a coherent strategy, but it is not interoperation. Its goal is that one of the two languages ceases to exist in the codebase.

Seam	Example	Principal cost
Interface	`cxx`, SWIG	opaque call
IR	LLVM LTO, JVM	toolchain lock-in
Successor	Carbon, Zig, C3	a third language
Elimination	C2Rust, Cpp2Rust	a language is removed
Symbol	this work	**subsets; refusals**

Table 1: Interoperation mechanisms by where the languages meet.

At the symbol. The position taken here. Both languages are lowered to a common substrate, and the lowerings are written together so that semantically corresponding constructs produce *textually identical declarations*. There is no boundary to optimise across because there is no boundary; there is no marshalling because both sides already agree on layout; and neither language is eliminated.

The cost is stated in Table 1 and is not small: what can be written is a subset of each language. §7 quantifies that cost on real code and §9 defends it.

3 Convergence at the symbol

The two lowerings are not merely compatible; they are chosen to agree. A Rust `impl` method and a C++ class method produce the same C:

```
impl Counter { fn bump(&mut self, by: i32) }
  ->  void Counter_bump(Counter *self, int by)

class Counter { void bump(int by); }
  ->  void Counter_bump(Counter *this, int by)
```

`&mut self` and `this` are the same pointer; `Type_method` is the same name. Templates and Rust generics both monomorphise to one struct per instantiation, and to the same mangled name.

The sharpest consequence concerns destruction, because it is where the two languages have the strongest and most different opinions. A Rust `impl Drop for T` lowers to `T_drop(T *self)`. A C++ `~T()` lowers to `T_drop(T *)`. They are the same symbol, so a C++ class may hold a Rust type *by value* and its implicitly generated destructor calls the Rust one:

```
class Holder {
public:
    Vec_int nums;   /* a Crust Vec<i32>   */
    Res     r;      /* a Crust impl Drop  */
};                  /* no destructor written */
```

No annotation marks `Vec_int` as foreign, because it is not foreign. The C++ front end is told which types own a resource and which function destroys one; from there the ordinary rules of C++ apply, and a class with an owning member acquires an implicit destructor exactly as it would if the member were a C++ class.

Where the languages genuinely disagree, each side follows its own rule rather than one being bent to the other: members are destroyed in reverse declaration order on the C++ side because that is C++, and fields are dropped in declaration order on the Rust side because that is Rust. The symbol is shared; the order is not. We regard this as the correct resolution — a programmer reading the C++ should be able to reason in C++ — but note in §8 that it has a consequence for tooling.

3.1 Where convergence stops: two verbs called move

Drop order is a mild disagreement: the two languages want the same thing in a different sequence. Move semantics are a sharp one, and they are the case in which convergence at the symbol is most dangerous, because the symbols converge while the obligations do not.

Rust's move transfers ownership. The source is dead; it is not dropped, and the compiler enforces that it is not read. C++'s move does not transfer ownership of the *object*: after `T b(std::move(a))`, `a` is a live object in a valid but unspecified state, and `~T` still runs on it at the end of its scope. What the move constructor does is empty it, so that the destructor which is going to run anyway finds nothing left to release.

The two disciplines therefore impose opposite drop obligations on the source, and the C lowering already contained both. Returning an owning local was implemented long before `std::move` was, and it is a Rust-style move out: the returned local is withheld from that path's destructors, because the caller receives the object itself rather than a copy. The machinery for suppressing a drop was thus already present, correct, and adjacent when `std::move` came to be lowered.

Reusing it would have been wrong, and — this is the part worth recording — wrong invisibly. A class written to be moved from empties its source, so destroying the emptied source is a no-op; suppressing that destructor produces a program indistinguishable from the correct one. The divergence appears only for a class whose destructor still has work to do on a moved-from object, which is precisely the class that idiomatic test inputs do not contain.

The resolution is the one §3 already takes for drop order, applied where it is load-bearing rather than cosmetic: each side follows its own language. A `std::move` in C++ source lowers to `T_move(&dst, &src)` and leaves `src` in its scope's destructor list; a Rust move emits no call and removes the source from that list. Both are spelled *move*; neither is bent toward the other.

3.2 Expression position, and the price of a temporary

A move must construct into something. In statement position there is somewhere to construct — a declaration, an assignment — and the lowering is a call. In expression position there is not: `f(std::move(a))` requires a temporary, declaring a temporary requires a statement, and C has no statement inside an expression. This is the same wall that a by-value parameter, a by-value return, and a chained call on a by-value result all meet, and for the same reason.

The wall is not in C as it is actually deployed. GNU statement expressions — `( stmts ; value ; )` — are exactly a statement in expression position, and are implemented by gcc, by clang, and by the self-hosting compiler this system targets. A materialised temporary is then what a C++ compiler would have built, written out:

```
f(std::move(a));
  ->  f(({ T _t; T_move(&_t, &a); _t; }))
```

We record the cost rather than eliding it, because it qualifies a claim made in §2. The artifact for a program that moves in expression position is GNU C rather than ISO C. This is a considerably weaker restriction than depending on a particular IR — three independent implementations accept it, and it remains readable and auditable — but it is a restriction, and the alternative we rejected is worth naming. A compiler-specific intrinsic understood only by the project's own back end would have bought the same expressiveness at the price of two artifacts: one for the fast portable compiler used during development and one for the real target, with no guarantee that testing the first said anything about the second. Choosing an extension that three compilers already implement keeps the output a single file.

The temporary is deliberately not registered for destruction. It is yielded by value, so the caller receives what it holds; destroying the husk left behind would destroy the resource just handed over. The source, as §3.1 requires, is dropped by its own scope regardless.

4 Complementarity

The subsets are not redundant, though the boundary between them has moved since the system was first built, and the direction it moved in is informative. Rust contributed the ownership discipline around moves — passing an owning value by value is a move, the source is invalidated, and reading it afterwards is rejected — the discipline formalised by RustBelt [22] and Stacked Borrows [23]. C++ contributed a full object lifecycle: arity-selected constructors, copy construction and assignment, member and base construction ordering, inheritance and virtual dispatch.

The C++ subset has since acquired moves of its own, and one might expect that to collapse the division. It does not, because what it acquired is *C++'s* move and not Rust's (§3.1). The system now implements both disciplines over one representation and must choose between them per construct, which is a harder position than having only one and a more honest account of what the two languages are. The division that survives is not that one language supplies moves and the other supplies structure, but that each supplies its own semantics for constructs the other also has.

This complementarity is what distinguishes the present system from the elimination approaches of §2. Cpp2Rust [14] must model C++ inheritance in a language that has traits rather than base classes, and consequently sup-

ports only the subset of OOP idioms that map to traits — excluding, by its own account, base classes with fields or non-virtual methods. We are under no such pressure, because inheritance stays in C++ and is lowered to the vtables C has always been able to express. Symmetrically, the Rust side is not asked to express constructor overloading, and — now — not asked to express a move whose source survives it.

5 The stack below the seam

The taxonomy of §2 judged interoperation mechanisms by where the seam falls. The same question can be asked downward. Lowering to C buys portability precisely because C is the one language every toolchain accepts — but it also means the front ends inherit whatever the toolchain is, and a toolchain is not one thing. It is a compiler, an assembler, a linker and a C library, conventionally from three or more separate projects.

We report a stack in which they are one. Table 2 gives its extent. The figures are line counts of the shipped implementation, excluding tests and excluding a large ported `musl` header and libm tree that is data rather than compiler.

Component	Lines
C front end, IL and code generation (ShivyCX)	27,237
Rust subset front end (`crust.py`)	7,557
C++ subset front end (`cpprust.py`)	6,404
Assembler, three ISAs (`rasm`)	5,441
Static ELF linker (`rlink`)	1,961
Freestanding runtimes (three ISAs) and C library	1,371
Total, source to static executable	49,971

Table 2: The whole path from two source languages to a running static binary, in one repository. No external compiler, assembler, linker or C library participates.

We are not claiming parity with production toolchains, and the comparison that matters is not one of size. GCC and LLVM each support far more of their languages, optimise incomparably better, and target dozens of architectures with decades of accumulated correctness. What we claim is narrower and, we think, more interesting: that a stack small enough for one person to hold in mind has a different *cost structure* for extension.

5.1 What a new architecture costs

The concrete evidence is two instruction set architectures added to a system that previously targeted only x86-64. Both were taken to the point where they compile the entire differential corpus with no unsupported construct, assemble through our own assembler, link through our own linker, and run their own C runtime including `printf`.

	AArch64	RV64
Back-end lowering and framing	947	797
Instruction encoder	1,201	963
Freestanding runtime (assembly)	244	246
Total per architecture	2,392	2,006
Shared register allocator, reused	189, unchanged	

Table 3: The cost of a bare-metal target. The allocator — copy coalescing, live-variable analysis, live-interval construction and a caller/callee linear scan — is architecture-neutral and was not modified for either target. When floating point was added to the RV64 back end it required no allocator change at all, having already been parameterised by register pool for AArch64.

Two details of Table 3 carry the argument. The first is the ratio: roughly two thousand lines per architecture against a hundred and ninety shared, which is what it means for a seam to be in the right place. The second is that the per-architecture figure includes the assembler and the runtime. In the conventional arrangement those are not the compiler's concern at all — a new GCC target is work in GCC, and then separately in binutils for `gas` and `ld`, and then separately again in a C library, three codebases with three review processes and three release cycles. Here the encoder that emits an `adrp`/`add` pair, the linker that applies the `R_AARCH64_ADR_PREL_PG_HI21`

relocation resolving it, and the back end that decided to emit it are edited in one change and tested together in one command.

That is not a claim that the work is easier in some absolute sense; the relocation still has to be right. It is a claim about *coupling*. The three bugs that cost the most during bring-up were each visible only because the layers were adjacent: a relocation type silently dropped between assembler and linker, a store width lost between back end and encoder, and a frame offset that no layer alone could see was out of range.

5.2 Validation without trusting ourselves

A small stack is only worth having if it is correct, and a stack that implements every layer itself cannot check itself against itself. Every layer is therefore validated against the corresponding GNU tool, used as an oracle in the sense of [25]:

- **Compiler.** Each corpus program is compiled by both ShivyCX and the cross `gcc`, run under emulation, and the exit statuses compared: 177 programs on AArch64 and 178 on RV64, with no unsupported construct remaining in either.
- **Assembler.** Each instruction form is assembled by both our encoder and GNU `as` and compared *byte for byte*: 182 forms on AArch64, 223 on RV64. Over the full corpus the encoders handle every instruction the compiler emits.
- **Linker.** The same object file is linked by both `rlink` and GNU `ld` and the results compared by execution, over a corpus organised by relocation type.
- **End to end.** Each program is built twice — once entirely with our tools, once with the GNU assembler and linker — and the two binaries compared.

The corpora were not written to look representative; they were shaped by mutation testing, deliberately breaking a decision in the implementation and requiring that some test notice. That process repeatedly found corpora measuring less than they appeared to. The recurring instance is worth recording because it is a property of the method rather than of this system: a process exit status carries eight bits, so a test that returns a value cannot distinguish a sign-extended byte from a zero-extended one, and neither adding a constant nor halving the result separates them — the two readings differ by a multiple of 256. Only a divisor coprime with 256 makes the difference observable. Several tests divide by three for that reason, and a wrong load width went undetected on one architecture for the whole project until a test read the *neighbouring* bytes rather than the value itself.

5.3 What this does not buy

The stack does no optimisation to speak of: there is a register allocator and some peephole-scale work, and nothing resembling the pass pipelines of [7]. It implements a subset of C, not the standard. It emits static executables only — no dynamic linking, no shared libraries. It has been validated under emulation rather than on physical hardware. And the variadic calling convention it uses between its own components is its own rather than the platform ABI's, which is sound while both sides are ours and would not be if a system C library were reached without the platform toolchain.

These are real limits and we would rather state them than have the size figure read as a claim of equivalence. The point of Table 2 is not that fifty thousand lines is enough to replace a production toolchain. It is that fifty thousand lines is enough to *own* one, and that owning one is what made the two-thousand-line architecture in Table 3 possible.

6 Checking the shared boundary

Sharing a representation means defects can cross languages, and the passes are written to find exactly those. Two examples, both encountered in practice rather than anticipated.

Passing an owned object from C++ to a Rust function *by value* is refused. Rust drops a by-value owning parameter when the callee returns, so the callee frees the buffer and the C++ destructor frees it again — one buffer, two frees. The diagnostic names the remedy, which costs nothing: a Rust `&Vec<i32>` parameter lowers to exactly the pointer the C++ side is told to pass.

This refusal is worth revisiting, because the reason for it has changed while the refusal has not. It was originally the same wall as §3.2: the C++ subset had no moves, so there was no way to hand ownership over at all. The subset now has both moves and by-value owning parameters — a C++ by-value parameter is constructed at the call and destroyed by the callee, and both halves are emitted. Yet the cross-language case stays refused, and now for a sharper reason than before. A C++ by-value parameter is an object the callee destroys and the caller must therefore *construct*; a Rust by-value parameter is an object the callee *takes*, and the caller must therefore withhold its own drop. Constructing a temporary for a Rust callee would free the original twice; withholding the drop for a C++ callee would leak. The two look identical at the call site and require opposite code, so the crossing is refused rather than guessed at.

That the two halves of a by-value parameter must travel together is not a theoretical observation. Lifting the refusal on the callee's side alone — so the parameter is destroyed when the function returns, which is what C++ does — turns every call into a double free, because the argument is still handed over as a struct copy. The error is immediate and total, which is the good case; the corresponding error in §3.1 is silent.

Deriving a bitwise copy of a type that owns something is refused on the Rust side for the same reason C++ refuses to copy a class with a destructor and no copy constructor. This is the Rule of Three, arrived at independently from two directions and enforced once.

Both defects are invisible to a conventional FFI, not because an FFI is careless but because they cannot arise there: no FFI would have permitted the C++ class to hold the Rust value by value in the first place. The class of bug is created by the mechanism, and so the mechanism is responsible for detecting it.

7 The bound on the subset

The C++ subset is not delimited by a fixed feature list but by a single criterion: *what plain C can be made to express*. That is a generous bound. Inheritance and virtual dispatch reduce to vtables; templates and generics to monomorphisation; destructors and `Drop` to functions over a struct pointer; lambdas to functions plus a capture record; move constructors and `operator=(T&&)` to functions like any other. Each of these was implemented rather than avoided, as were out-of-line member definitions, `shared_ptr`, `enable_shared_from_this`, anonymous unions, default arguments, nested classes, and most of C++11's declaration syntax.

It is also a moving bound, and it moves in one direction as the front ends mature. Move semantics are the largest recent instance, and they moved the bound twice over: once directly, and once through the constructs that had been refused only for want of them. `std::vector<std::unique_ptr<T>>` is the clearest case. It was previously listed as legal C++ the subset could not express — not a missing feature but a structural exclusion, since the subset's Rule of Three refusal *was* `unique_ptr`'s move-only semantics and a container had no way to get an element into place. Three smaller decisions, each following C++ rather than inventing, removed it: a container gained a move overload of `push_back`, chosen by whether the call site wrote `std::move` rather than by arity, since the two overloads have the same arity and C++ chooses between them the same way; the element operation it needs resolves per instantiation to a move constructor, a copy constructor, or an assignment; and a container member that copies an element the element type cannot copy is *deleted*, exactly as C++ deletes it, rather than the instantiation being rejected over a member the program never calls. A call to a deleted member is then an error naming the reason — not an undefined symbol, which is what dropping it silently would have produced.

The features that resist are those where C offers no faithful encoding rather than merely an inconvenient one — exceptions being the clearest case, since unwinding is control flow C does not have.

We are careful about a claim we do not make. The subset does not compile unmodified real-world C++, and a codebase built on exceptions or multiple inheritance meets the bound quickly. What we do claim is that real C++ tends to fail against the subset for reasons that are few and enumerable rather than for a long tail, and that a codebase avoiding the three features that most complicate lowering C++ — exceptions, `dynamic_cast`, multiple inheritance — can generally be brought inside it by edits that are semantically neutral and few in proportion to its size. The C++ subset has been exercised in this way against a browser layout engine written without regard for this compiler.

We are careful about a claim we do not make. The subset does not compile unmodified real-world C++, and a codebase built on exceptions or multiple inheritance meets the bound quickly. What we do claim is that real C++ tends to fail against the subset for reasons that are few and enumerable rather than for a long tail, and that a codebase avoiding the three features that most complicate lowering C++ — exceptions, `dynamic_cast`, multiple inheritance — can generally be brought inside it by edits that are semantically neutral and few in proportion to

	Cpp2Rust	`cpp2rust.py`
Goal	a program that runs	a rustc question
Variables	`Rc<RefCell<T>>`	their own types
Pointers	weak ref + offset	`*mut T`
Checks land	at run time	at compile time
Bodies	fully translated	stubs, skeletons
Output is	linked and run	discarded

Table 4: The same translation, inverted for a different purpose.

its size. The C++ subset has been exercised in this way against a browser layout engine written without regard for this compiler.

8 Rust as oracle and as second target

Every system surveyed under *elimination* translates C or C++ into Rust in order to obtain a Rust program that runs. We describe a use of the same translation in which the output is thrown away, and a second in which it is not.

8.1 Motivation

The C++ front end infers ownership: which classes own a resource, which copies would double-free, which by-value parameter crosses a boundary unsafely. These inferences are the load-bearing part of the system, and they are checked only by their own test suite. An independent implementation of the same judgement would be worth having, and one exists: Rust's borrow checker, which decides ownership questions for a living.

So a third lowering, `cpp2rust.py`, raises the C++ subset to Rust, runs `rustc --emit=metadata` on the result, and discards it. Nothing is linked and nothing is executed; the only output consumed is the diagnostic stream. In the terminology of Pnueli et al. [24] this is a translation-validation posture rather than a verification one — we do not prove the lowering correct, we obtain a second opinion on each instance — and it is a relative of differential testing [25], with a type system in place of a second compiler.

CompCert [27] establishes correctness once for all inputs at commensurate proof cost; Alive2 [26] occupies the middle ground we aim at, validating individual transformations by bounded checking. Ours is cruder than either: it does not check the lowering, but re-derives one of its conclusions — ownership — independently, and reports disagreement.

8.2 Inverting Cpp2Rust

The obvious way to build this is to follow Cpp2Rust [14], which is the state of the art in C++-to-safe-Rust translation. We did not, and the reason is instructive.

Cpp2Rust boxes every variable into `Value<T> = Rc<RefCell<T>>` and represents every pointer as a weak reference plus an offset. Its own account is explicit that this is what makes translation total: any program can be translated, because ownership and mutability move to *run time*, where `borrow_mut` panics and a dangling weak reference aborts. The paper's memory-safety guarantee follows from the absence of `unsafe` in the output.

For their purpose — a translated program that runs in production, at a measured cost of between 2% and 6× — this is the right design. For ours it is inert. Boxing everything is precisely what makes `rustc` *accept* everything: the ownership questions were postponed, not answered, so the compiler has nothing to say about them. Their own results argue the point from the other side, since the optimizer that removes 71–87% of the boxing does so for speed, and the residue it cannot remove is exactly the aliasing-heavy code where a static answer would be worth most.

We therefore invert the central choice: emit the *most restrictive* Rust that still models the C++ faithfully, so that borrowck has something to reject. Native ownership rather than `Rc`; `impl Drop` for any class that owns a resource; `Clone` *only* where a copy constructor exists; real `&`/`&mut` for C++ references. Table 4 summarises.

By construction, then, a C++ class with a destructor and no copy constructor becomes a Rust type with `Drop` and no `Clone`. A C++ copy of one is therefore a Rust *move*, and any later use of the source is a borrow of a moved value — so the Rule of Three is rediscovered by a third independent route. Similarly `new`/`delete` lower

to `Box` and `drop`, collapsing double-`delete` and use-after-`delete` into one diagnostic, and `T&operator[]` raises to a method returning `&mut T` with an elided lifetime, placing the borrow under the checker's jurisdiction.

8.3 Ownership skeletons

Bodies are the difficulty, and the checker's purpose licenses an unusual shortcut. Because only ownership is at issue, statements that cannot move, borrow, construct or destroy anything may be *erased*: arithmetic cannot double-free. Each function body is reduced to its ownership skeleton and the remainder discarded.

Erasure is sound in one direction only, and the asymmetry is the design. Dropping a statement can only make the check *quieter*, because `rustc` is asked about the moves that survive and an erased move is a question never posed. The failure that must not occur is a statement surviving whose subject was erased: it then names a variable nothing declared, and the checker reports the tool's own defect as a finding about the user's source. The implementation therefore tracks what it has declared and emits no statement naming anything else — an invariant that the implementation's own test suite pins.

8.4 Two honest asymmetries

The oracle disagrees with the C++ front end in two places, and neither is a bug.

First, *moves and by-value parameters*. This asymmetry has inverted since it was first observed, and the inversion is more interesting than the original.

It was originally that the C++ front end refused an owning class crossing a call boundary by value while the raised Rust accepted it, because Rust's by-value is a move and moves are safe. The C++ subset had no moves at all, so a quiet `rustc` was to be read as the two tools answering different questions rather than as the C++ pass being over-strict.

The C++ subset now has moves, and the oracle does not. That is a defect to be repaired rather than an asymmetry to be preserved, but repairing it is not a matter of catching up, because the obvious repair is wrong. Raising `std::move(a)` to a Rust move would make every subsequent use of `a` — including the destructor call that C++ guarantees — a use of a moved value, and `rustc` would reject a program that is correct C++. The oracle would then report the tool's own model as a finding about the user's source, which is the failure mode §8 identifies for erasure and forbids there.

A C++ move must therefore be raised as something that keeps the source alive: a method taking `&mut self` which empties the receiver and returns the contents, leaving the source a live, borrowck-legal, emptied object. That is a faithful model of what the move constructor does. It is also, notably, more restrictive than the C++ it models, since `&mut` forbids the aliasing C++ would permit — which is the direction this oracle is deliberately biased in. It is not yet implemented.

The general moral is the one §3.1 states from the other side: two languages that share a keyword do not thereby share a semantics, and a tool reading both must name which one it is speaking at every construct where they differ. The list of such constructs is longer than drop order alone.

Second, *drop order*. As §3 notes, C++ destroys members in reverse declaration order and Rust drops fields in declaration order. A field-for-field raising would therefore model the wrong language. Fields are emitted reversed, with the base class last, so that the Rust models the C++ rather than the Rust side of an asymmetry the system elsewhere preserves deliberately. This is a small point with a general moral: a tool that reads two languages must decide, explicitly and per construct, which one it is currently speaking.

8.5 The second exit

The oracle has an architectural consequence beyond auditing. The C++ front end now has two back ends, and they do not fail on the same things.

Where the C lowering refuses — because C offers no faithful encoding of some construct, or because a refusal is the honest answer under §9 — Rust may still be a target, since it possesses natively much of what C must be made to encode: destructors, generics, tagged unions, bounds-checked slices. The subset expressible in C and the subset expressible in Rust are not the same set, and their union is larger than either.

We are careful not to overstate what is built. The raising is presently tuned for checking rather than execution, and a Rust back end producing running code is a direction rather than a finished feature. But the pipeline exists and the symbol conventions are already shared, so a construct outside the C subset need not be outside the system.

9 Discussion

The design has a cost: both front ends *refuse* what they cannot lower faithfully, and refuse rather than approximate. A reference-returning function is rejected because lowering `T&` to `T*` would silently change what every call site means. A conversion operator is applied only where the target type is written, because applying it wherever a conversion is wanted would require knowing the type of every expression — type checking, not the reading of written types these passes do.

We regard this as the load-bearing decision rather than a limitation. A subset that can be trusted is worth more than a superset that is quietly wrong somewhere, and each refusal is a diagnostic naming the remedy. In practice the refusals are frequent enough to be principled and rare enough to be portable around.

The contrast with the elimination literature is sharp on exactly this axis: C2Rust [13] translates everything and produces `unsafe` Rust that inherits the original's defects [15], whereas we translate less and guarantee more about what we translate. The oracle of §8 sharpens the same principle: a refusal there is not merely declined work but a second reading of the same ownership question.

The same preference for less explains the stack of §5. A subset compiler, a subset assembler and a static-only linker are each much less than their production counterparts, and it is exactly that which makes the whole path fit in one repository. The extensibility result follows from the smallness rather than from any cleverness: an architecture costs two thousand lines because there are only so many places a decision about an architecture can be made.

We should be careful about the direction of that claim. It is not that extending GCC or LLVM is hard because they are badly built — they are large because they do vastly more, and much of what they do we simply decline. The narrower observation is that their size is distributed across project boundaries that a new target must cross: the compiler, the assembler and linker, and the C library are separately maintained, and a target is not usable until all three have it. A stack that has no such boundaries has none to cross. Whether that remains true at ten times the size is precisely the question we cannot answer from here.

10 Conclusion

Cross-language interoperation is usually approached by making a boundary cheaper. We have described a system that removes it, by lowering subsets of C++ and Rust to C through front ends written to agree on the symbols they emit. The result is that a C++ class can own a Rust value the way it owns a C++ one, with no shim and no annotation, and that a defect crossing the two languages is caught by the pass that made the crossing possible.

Removing the boundary does not, however, remove the disagreement. It relocates it, from a place where two compilers must be made to agree to a place where one lowering must choose. Both languages have a move; the symbols converge and the obligations do not; and the choice is invisible in the output, invisible to a compiler, and invisible to tests written over code that is well behaved. We take this to be the characteristic hazard of meeting at the symbol, and the reason a system that meets there must document, per construct, which language it is currently speaking.

The same machinery, pointed in a third direction, yields something we did not expect: because the C++ subset can be raised to Rust as well as lowered to C, and because raising it *restrictively* rather than conservatively makes the borrow checker speak, `rustc` can be enlisted as an independent auditor of ownership — The translation literature has consistently treated Rust as a destination. It is also, and perhaps more cheaply, an oracle.

Beneath all of this is a stack we did not have to assemble from other people's projects. Lowering to C is usually a way of borrowing a toolchain; here it is a way of reaching one that is fifty thousand lines long and entirely present. The measured consequence is that a bare-metal instruction set architecture — back end, instruction encoder, relocations, freestanding runtime, everything from C source to a static ELF binary — costs about two thousand lines and shares a register allocator with every other target. Whether a system this small can be trusted is a fair question, and our answer is not that it is obviously correct but that every layer of it is checked, instruction by instruction and program by program, against the tools it replaces.

References

[1] B. Stroustrup, "A History of C++: 1979–1991," in *Proc. 2nd ACM SIGPLAN Conf. on History of Programming Languages (HOPL-II)*, 1993.

[2] B. S. Hartshorn, "Full-stackless Computing: A Multi-language Compiler That Builds and Boots its Own Operating System," 2026. https://dx.doi.org/10.2139/ssrn.7226482

[3] D. M. Beazley, "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++," in *Proc. 4th USENIX Tcl/Tk Workshop*, 1996.

[4] The Rust Project Developers, "rust-bindgen: Automatically generates Rust FFI bindings to C and C++ libraries," 2016–. https://github.com/rust-lang/rust-bindgen

[5] D. Tolnay, "cxx: Safe interop between Rust and C++," 2020–. https://github.com/dtolnay/cxx

[6] Google, "autocxx: Calling C++ from Rust with a safe interface," 2021–. https://github.com/google/autocxx

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, 2004.

[8] O. Braunsdorf, T. Lange, K. Hohentanner, J. Horsch, J. Kinder, "SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications" (2025) https://doi.org/10.48550/arXiv.2510.20688

[9] Google, "Carbon Language: An experimental successor to C++," 2022–. https://github.com/carbon-language/carbon-lang

[10] H. Sutter, "cppfront: A personal experimental C++ Syntax 2 to Syntax 1 compiler," 2022–. https://github.com/hsutter/cppfront

[11] A. Kelley et al., "The Zig Programming Language," 2016–. https://ziglang.org

[12] C. Lerno et al., "C3: A C-like language aiming to be an evolution of C," 2019–. https://c3-lang.org

[13] Galois Inc. and Immunant Inc., "C2Rust: Migrate C code to Rust," 2018–. https://github.com/immunant/c2rust

[14] L. Popescu, F. Gouveia, H. Preto, J. Silveira, D. Hrybenko, J. Fragoso Santos, and N. P. Lopes, "Cpp2Rust: Automatic Translation of C++ to Safe Rust," *Proc. ACM Program. Lang.* 10(PLDI), Article 188, 2026. https://doi.org/10.1145/3808266

[15] S. Wu, H. Yang, and B. Hua, "Security Risks of Transpiling C Programs to Rust," in *Proc. IEEE TrustCom*, 2025.

[16] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership Guided C to Rust Translation," in *Proc. Int. Conf. on Computer Aided Verification (CAV)*, 2023.

[17] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, "Aliasing Limits on Translating C to Safe Rust," *Proc. ACM Program. Lang.* 7(OOPSLA1), Article 94, 2023.

[18] J. Hong and S. Ryu, "Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation," *Proc. ACM Program. Lang.* 8(PLDI), Article 176, 2024.

[19] A. Fromherz and J. Protzenko, "Scylla: Translating an Applicative Subset of C to Safe Rust," *Proc. ACM Program. Lang.* 10(OOPSLA1), Article 121, 2026.

[20] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners," arXiv:2404.18852, 2024.

[21] V. Nitin, R. Krishna, L. Lemos do Valle, and B. Ray, "C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques," arXiv:2501.14257, 2025.

[22] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the Foundations of the Rust Programming Language," *Proc. ACM Program. Lang.* 2(POPL), Article 66, 2018.

[23] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked Borrows: An Aliasing Model for Rust," *Proc. ACM Program. Lang.* 4(POPL), Article 41, 2020.

[24] A. Pnueli, M. Siegel, and E. Singerman, "Translation Validation," in *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998.

[25] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, 10(1), 1998.

[26] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded Translation Validation for LLVM," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2021.

[27] X. Leroy, "Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant," in *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2006.