High-Fidelity C Interoperability in Hylo

A Principled Design for Safe and Idiomatic C Bindings

Ambrus Tóth
Supervisors: Andreea Costea, Jaro Reinders

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfillment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025
Latest revision: July 29, 2025

Name of the student: Ambrus Tóth
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Jaro Reinders, Harm Griffioen

Contact the author at: ping@ambrus.dev

Abstract

Interoperability with C is critical for new systems languages, yet achieving a high-fidelity bridge requires navigating a complex space of trade-offs between usability, portability, and maintainability. Beyond mastering platform-specific ABIs and C dialects, a robust tool must decipher C's "semantic dialects"—programming conventions where syntax alone is insufficient to determine intent, such as an enum representing a set of mutually exclusive cases, a collection of combinable bitflags, or simply a group of named integer constants. These idioms defy rigid, one-size-fits-all translation.

This paper presents a principled technical design for a C interoperability layer for Hylo that addresses these challenges. We propose an architecture that leverages Clang for high-fidelity parsing and correct ABI handling, and a semantic mapping framework based on sensible defaults with developer-driven customization. This approach allows ambiguous C constructs to be mapped to the most appropriate and idiomatic Hylo representation, balancing automation with developer control. The result is a complete roadmap for a powerful and usable interoperability solution.

1 Introduction

The Hylo programming language aims to provide the performance of C++ with a simpler, safe programming model based on mutable value semantics and a powerful generics system. For a new systems language, the ability to leverage the vast ecosystem of existing libraries and operating system APIs can significantly accelerate its adoption by reducing the barrier to entry. This allows new users to solve real-world problems from the start, rather than first needing to implement foundational tools like a GUI toolkit or network protocol. Due to its relatively simple and stable Application Binary Interface (ABI)—the low-level conventions for how functions are called and data is represented in machine code—C has become the de facto bridge language between languages. Consequently, a high-fidelity C interoperability solution represents a critical component for Hylo's practical success.

However, designing such a layer is fraught with complexity. It demands a deep understanding of both the source and target languages, platform-specific ABI conventions, and the nuances of various C dialects and compiler extensions which introduce non-standard features or memory layouts that an interoperability tool must understand, as it's easy to introduce undefined behavior. Existing solutions differ widely in their capabilities, often forcing a trade-off between portability and ease-of-use, and frequently impose a rigid or hard-to-customize mapping for C constructs that may have multiple valid interpretations in the new language. A lack of a comprehensive analysis of the design space creates a barrier for new languages, often leading to ad-hoc solutions that suffer from bugs or limitations.

In this paper, drawing from a methodology that combines a review of industry technologies, academic literature, expert interviews, and targeted prototype implementations, we present a principled technical design for high-fidelity C interoperability for Hylo, and answer the main research question: **How can a modern systems language with an emphasis on safety, simplicity and performance approach seamless C interoperability?**

Our contributions are:

1. **A set of design goals for a high-fidelity C interoperability solution**, derived from an analysis of primary use cases and a review of existing state-of-the-art technologies. (Section 3)
2. **A detailed architectural design for Hylo's C interoperability tooling.** This includes a comparative analysis of C header parsing technologies, a proposed architecture for compiling C function calls by leveraging Clang and LLVM, and a plan for providing transparent cross-language IDE features. (Section 4)
3. **A comprehensive specification for mapping C constructs to Hylo.** This defines the semantic translation of primitive and composite types, pointers, and even advanced constructs not present in Hylo like bit-fields, flexible array members and untagged unions, which were validated by targeted prototypes. Crucially, it includes a design for mapping C's "semantic dialects" (e.g., enums used as bit flags) to stronger, idiomatic Hylo types by flexible customization. (Section 5)

The remainder of this paper is structured as follows. Section 2 describes our research methodology. Section 3 establishes the design goals for a high-fidelity interoperability layer. Section 4 presents our architectural design, followed by section 5, which details the complete C-to-Hylo mapping specification. We then review related work in Section 6 and discuss our findings and future work in Section 8. Finally, we conclude the paper in Section 9.

2 Methodology

This research was conducted using a combination of academic and industry research, expert interviews, implementing prototypes, and feedback from the Hylo developer community. The goal was to gain a deep understanding on the state of the art in C interoperability technologies, and to develop targeted prototypes that demonstrate the feasibility of solving key challenges.

2.1 Industry Research

We started by reviewing the existing C/C++[1] interoperability technologies in various programming languages, including Swift, Rust, D, Carbon, and Zenon. This involved reading documentation, blog

[1] C++ was included for perspective on Hylo's long-term goal of C++ interoperability.

posts, and watching conference talks to understand their features, limitations, and collect desirable properties for an interoperability solution for Hylo.

We also conducted interviews with language experts from the C++, Rust, Zenon and Hylo language communities. These interviews were conducted in an open format, allowing us to explore specific topics in depth while also gathering general insights on the challenges and solutions related to C (and C++) interoperability. We conducted preliminary discussions within the Hylo community, and gathered feedback on proposed solutions.

2.2 Academic Literature Review

We conducted an exploratory literature review using a combination of Scopus queries, snowballing from [1], and AI-assisted tools like Elicit and Undermind. The prompts are presented in Appendix C.

The Scopus based search was done in 3 parts: general papers related to cross-language interoperability, papers related to C interoperability, and papers related to C++ interoperability. While the search initially focused on the last decade to capture advancements following the introduction of Swift and Rust, snowballing revealed foundational earlier works, some dating to 1996 on technologies like SWIG[2]. To investigate a more specific challenge, an additional Undermind-based search was conducted regarding C macro translation and interoperability, whose results we present in section 6

2.3 Prototyping

To validate design decisions and explore technical feasibility, we implemented three proof-of-concept prototypes. Each addressed a specific challenge in C interoperability: the conversion of integer types, the mapping of complex C data structures, and the introspection of a platform's C ABI.

The first prototype for **C integer conversions** implemented the design detailed in section 5.1. With the aim of developing a solution that could be upstreamed to the Hylo compiler, this work involved API design, code generation for the N^2 conversions, extending compiler intrinsics, and implementing the LLVM backend lowering.

We then developed prototypes to validate a strategy for representing C data structures without coupling the Hylo compiler to a C compiler. This involved two interconnected efforts. First, we built the **ABI Explorer**, a standalone Swift tool exposed in a web interface[3] that uses LibClang's C interface to inspect C struct layouts (including field offsets, sizes, and alignment) for any target platform. This tool provided the necessary data to implement a series of **mapping prototypes** for challenging C constructs, namely unions[4], flexible array members[5], and bit-fields[6]. The correctness of these mappings was verified using unit- and property-based tests.

3 Design Goals for High-Fidelity Interoperability

3.1 Validation and Automation

Manual FFI writing is error-prone; we should aim to avoid inconsistencies by design, detect errors early, and automate common error-prone or tedious processes instead of relying solely on the developer.

3.2 High Coverage of C Constructs

Language interoperability is a notorious source of bugs and vulnerabilities, stemming primarily from the error-prone manual glue code written for unsupported features. Therefore, our goal is to reduce the amount of glue code needed as much as possible by maximizing the coverage of C constructs and mitigating the need for glue code, most importantly on the C side. However, this pursuit of high coverage must be principled; direct mapping of inherently error-prone constructs like macros and variadic functions is less desirable. For these specific cases, we could provide specialized tools that help with writing glue code based on some user guidance.

3.3 Portability

The C language has evolved into various dialects over the course of its 50+ year evolution, despite being specified by an ISO standard. For example, the Linux kernel extensively uses extensions provided by the gcc compiler, thus cannot be compiled with MSVC. Dialects pose a challenge for interoperability, as an interoperability tool shall support a diverse range of C projects to be useful for adopting much of the existing ecosystem.

The primary sources of language differences are as follows:

- **Standard versions:** The C standard has evolved over time, and different projects support different standard versions. Since C is largely backwards compatible, many libraries stay with older language version to keep supporting their clients who may not be able to upgrade, while allowing to be used by projects with newer language versions.
- **Compiler-specific extensions:** Major compilers like GCC, Clang and MSVC provide various extensions to the language that are not part of the standard. Some extensions only affect implementation like the embedded assembly blocks, while others affect the ABI of the generated code like the `__attribute((packed))__` which can change the memory layout of structs. When implementation is separately maintained outside of header files, only the ABI-affecting category poses challenges, but inline functions are often defined in headers for potential performance benefits, so an interoperability tool shall ideally handle both cases, at least for code parsing but potentially also for code generation (see section 4.4).
- **Compiler Plugins:** Clang and GCC have a plugin system that developers can use to inject custom compiler behavior. It is impossible for an interoperability tool to account for them all, but since their main uses are for static analysis and runtime instrumentation tools, where the ABI of the code is unaffected, they are less important to support.
- **High-performance computing / GPU programming compilers:** NVidia HPC, Intel oneAPI DPC++ are compilers with specialized extensions. Further exploration is needed to investigate how they affect the ABI.
- **Conditional Features:** Some features in the C standard are conditionally supported, the compiler vendor doesn't have to support them to be standard compliant. Such features include the `_Atomic` qualifier and complex numbers. The interop tool shall be able to parse all these features, and preferably also map them to the other language.

An additional source of differences is conditional compilation, which needs to be handled by the interop tool by allowing to specify C preprocessor variables.

3.4 Scalable Maintenance

Maintaining a library/application that uses C interop for multiple platforms should be easy. We shouldn't try to replicate implementation-defined behavior of various C compilers or specific logic for each ABI's handling, so that we can focus efforts elsewhere.

3.5 Control and Customizability

There is in fact one more kind of dialect: the semantic meaning of standard C code itself. Unlike C++, C doesn't provide many features for type based abstraction: In C, error conditions are often signaled using special return values (like negative numbers or null pointers) or by setting a global variable. Additionally, C enums frequently serve as bit flags, where individual constants are powers of two, allowing values to represent combinations of these flags. Mapping this loose semantics to stronger types requires additional guidance from the user to avoid ambiguity.

4 Architectural Design and Technical Considerations

Existing interop technologies have vastly different architectures and technical choices. This section provides a design for Hylo's interop architecture while comparing it with the surveyed technologies.

4.1 Core Principles

Fidelity through Clang

Generating accurate bindings without writing many platform- and language dialect specific details requires a high-fidelity parser for the C language. While various tools exist, a comparative analysis presented in Appendix B reveals that leveraging the Clang compiler front-end is the most robust and flexible approach, as Clang is architected to handle various C dialects and emit ABI-compatible code for all major compilers and platforms.

Safety and Portability through Abstraction

A source of bugs and unexpected incompatibilities stems from languages importing the platform-dependent C types, like `long` to a simple type alias to a fixed-width integer type (e.g. `Int64`) on the target platform using conditional compilation. This approach forces no consideration for platforms

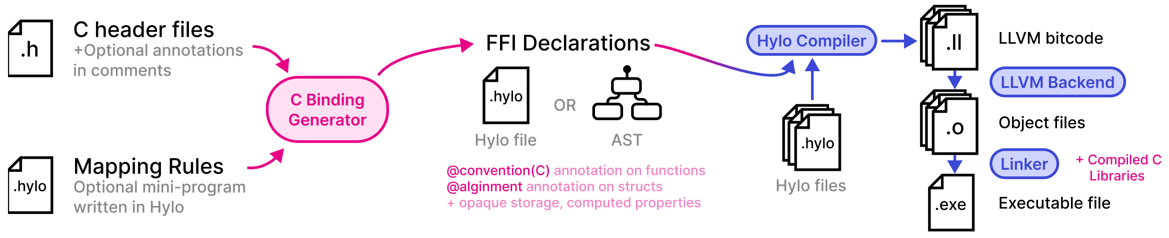


Figure 1: **Architecture Overview**: extending the compilation pipeline with an FFI binding generator.

where the size may differ, leading to fragile code. To counter this in Hylo, platform-dependent C types are imported as distinct, non-aliasing types (e.g., `CLong`). These types cannot be used directly in arithmetic with Hylo's fixed-width integers; they require explicit, **intention-revealing conversions** that force the developer to handle potential narrowing and signedness differences first. This design choice makes portability risks visible at the type level, preventing an entire category of errors.

Idiomatic Hylo through Customization

A rigid, one-size-fits-all mapping is insufficient because C often uses a single syntactic construct to represent multiple semantic ideas. A C enum, for example, could be a set of mutually exclusive cases, a collection of combinable bitflags, or simply a group of named constants. Thus, Hylo must empower developers to map these C idioms to the most appropriate and type-safe construct for ergonomic experience.

This principle is realized through a framework of sensible defaults with developer-driven customization, for which the system provides two primary mechanisms:

- **Rule Files:** External configuration files allow developers to specify mapping rules for C headers they do not own, enabling non-intrusive customization for system libraries or third-party dependencies. Rules can be applied globally, per-header, or use name-based pattern matching. Flexibility of domain-specific languages is often limited, thus we recommend adopting regular Hylo code for specifying the importing rules, with a library of helper primitives. A hypothetical `mapping-rules.hylo` may look like this:

```
importer.for(name_matching: "*Enum", type: .enum).map(as: .option_set)
importer.for("do_work_with_error", type: .function)
          .map_return(as: .throwsOn(negative: true))
```

- **In-Source Annotations:** When developers have ownership of the C source, they can use annotations directly in the header files to improve local reasoning and maintainability. Unlike Swift's attribute-based C code annotations that require compiler extensions to work, Hylo shall adopt Bindgen's approach of using comments that are ignored by the C compilers but can be understood by our external parser. The same technique is used by existing documentation generation tools (usually parseing MarkDown or HTML inside comments), thus to avoid the annotations presented in the generated documentation in unintended ways, Bindgen's solution is to utilize invisible, `div` tags with added classes and special attributes.

4.2 System Overview

The two main challenges every native interoperability tool must solve are understanding the memory representation of C data structures and calling C functions with the correct calling convention according to the C ABI. The memory representation defines struct field ordering, padding, endianness, and so on, while the calling convention defines details such as which parameters to pass in registers or on the stack, how values are returned from a function and what registers are callee or caller saved.

As shown in Figure 1, our design extends Hylo's compilation pipeline with a binding generator to C. This tool parses C header files using LibClang, applies developer-defined mapping rules, and emits corresponding .hylo declarations to either files or directly into Hylo abstract syntax tree (AST). The Hylo compiler then processes these generated declarations alongside the rest of the program's source code. Finally, the resulting object files are linked with the necessary C libraries—either statically or at runtime—to produce the final executable. By leveraging LibClang, a cross-compiler, we can capture the memory layout into the generated Hylo structs precisely, while LLVM allows us to call C functions according to the platform's C ABI.

4.3 Capturing the Memory Layout: Opaque Storage for Composite Types

C structs and unions are challenging to map to other languages because the new language has to replicate the memory layout of the original type, so that field access is done with correct memory offsets. Determining the struct layout involves handling the size and alignment of members, potentially including padding, and distributing bit-fields. The layout may also be influenced by compiler-specific packing attributes in all major C compilers, though their behavior is consistent.[2]

C bit-fields have largely implementation-defined layout rules, which can vary significantly between compilers[3]. Due to the challenging nature of C bit-fields, many C interop technologies do not support interoperating with structs with bit-fields[4][5], and it is generally recommended to avoid using them in separately compiled library code, as interoperability is not even guaranteed between C compilers.

We identified two existing approaches for capturing the memory layout:

1. Rust lets us **annotate Rust structs, unions and enums –that have C-compatible members–** with the `#[repr(C)]` attribute. Rust then lays out these members according to the C ABI by reimplemented logic[7]. Notably, bit-fields are not supported natively.

   However, Bindgen[8], an external binding generator for Rust, attempts to emit Rust `#[repr(C)]` structs from C header files, complementing the layout algorithm with a custom bit-field handling algorithm, and exposing accessor methods that let the user operate with a correctly aligned and padded Rust integer type. This method can often work but has limitations and bugs, a notable problem being that C bit-fields may share storage with other members' padding bits, which is challenging to replicate when generating separate struct fields for each member.[6].

2. Swift imports C declarations by **directly embedding Clang into the compiler**, and transforming declarations from Clang AST directly to Swift AST declarations[7]. It synthesizes computed properties for accessing C bit-fields, which present an API as if the bit-fields were regular Swift fields while performing the necessary bit-masking and shifting to access the individual fields[9]. Swift preserves the references to the original Clang AST nodes, which can be used when compiling member access to find out the exact member offsets, therefore resulting in a robust mapping.

The first approach, decoupling the mapping of implementation-defined C constructs from the main compiler is useful for reducing the compiler's binary size (no need to embed Clang in the general case), simplifies the development environment setup for both the compiler authors and its users, and keeps the bugs easier to fix in the substantially smaller codebase of the binding generator. On the other hand, reimplementing the layouting logic in the new language's compiler requires a significant effort and may be incomplete or inaccurate. Embedding Clang and directly maintaining layout information from the Clang AST for the backend mitigates this maintenance burden, and is robust due to Clang being a production-quality cross-compiler reimplementing all struct layouting rules and compiler-specific behavior in order to interoperate with gcc-/MSVC-produced binaries.

We propose a hybrid approach for Hylo that combines advantages from both methods while reducing coupling, shown in Figure 2. Instead of letting the Hylo compiler take care of C struct layout, we map members of unions and structs to a Hylo struct containing a **contiguous inline byte sequence** with the same length as the original C type's size (including padding bytes). Subsequently, we synthesize computed properties (subscripts) for accessing the individual members, which provide idiomatic access to the members, exactly as if they were stored properties (and due to inlining, with the same efficiency). For struct members, we can leverage LibClang's functions (e.g. `clang_Cursor_getOffsetOfField`) to get the offset, the size in bytes of the underlying type and the bit-width in case of bit-fields. Union members always start at the same memory address, without an offset[10].

To ensure the same data layout in Hylo as in C, we need an additional annotation for the Hylo struct: we need to specify its *alignment*, so that the Hylo compiler can know on which boundary it should allocate instances of the struct.

[2]See the effect of packing attributes in MSVC, Clang and GCC: https://edu.nl/8hg98

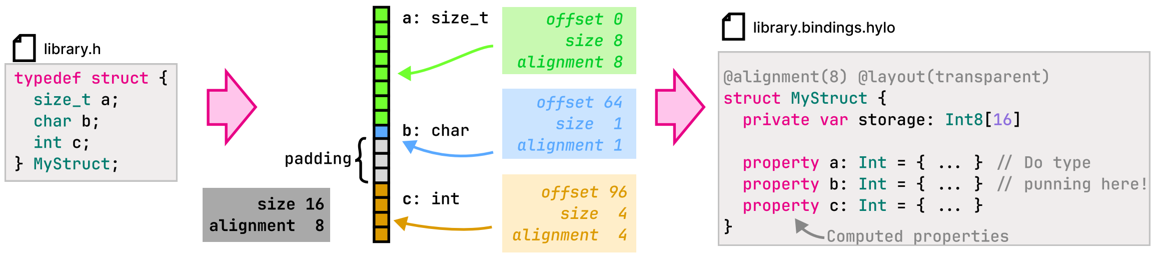
[3]See an example of varying layout of bit-fields: https://godbolt.org/z/xqPEvs6YK

[4]See Rust's open issue about bit-field support in `#[repr(C)]`

[5]See Zig's tracking issue #1499 for implementing bit-field support.

[6]See issue #743 in Bindgen's repository related to the overlapping storage bug.

[7]See structs being imported to Swift in ImportDecl.cpp at line 2079



```c
typedef struct {
  size_t a;
  char b;
  int c;
} MyStruct;
```

```
@alignment(8) @layout(transparent)
struct MyStruct {
  private var storage: Int8[16]

  property a: Int = { ... } // Do type
  property b: Int = { ... } // punning here!
  property c: Int = { ... }
}
```

Figure 2: **Capturing the Memory Layout of C Constructs**

4.4 Function Calls

Reimplementing the complex logic of lowering C-compatible functions and function calls for every supported platform is a monumental and error-prone task. Instead, our proposed architecture delegates the responsibility of ABI-correct lowering to LLVM, the compiler backend also utilized by Clang. The Hylo compiler only needs to remember which functions to lower with Hylo vs. C calling conventions.

Compiling Header-Defined Functions: `static`/`static inline`/`inline`

A significant challenge in C interop is handling functions defined entirely within header files. These functions, declared with the `static`, `inline`, or `static inline` specifiers, lack a stable, externally visible symbol in the object file, making them non-linkable by default. To invoke them from another language, their implementation must be made accessible. Our analysis reveals two primary architectural approaches to this problem:

Generating C Wrappers. This strategy involves creating a new `.c` source file that contains simple, externally visible wrapper functions for each desired header-defined function. This file is then compiled alongside the original library, and the foreign language binds to the newly created symbols with unique names (e.g., suffixed with `__wrapper`). Recovering the lost performance caused by the extra functin call is possible via Link-Time Optimization (LTO) but the technique relies on a shared compiler backend like LLVM, and even then, the inlining is a heuristic optimization that may not be performed.

This technique is employed by Rust's Bindgen, though it currently has (unnecessary) limitations regarding plain `inline` functions[11] that we proposed to mitigate in[12].

Direct LLVM IR Emission: A more deeply integrated approach, employed by Swift-C interop, is to use Clang as a library to request the compilation of the header-defined function directly into LLVM Intermediate Representation. LLVM IR can then be consumed by the new language's backend, given that it uses LLVM. By operating at the compiler level rather than the linker level, this strategy eliminates the cost of function calls.

For Hylo, a language early in its development, we adopt the wrapper-generation approach. This prioritizes compiler simplicity, and maintains the possibility for Hylo to use different backends in the future, which we consider a prudent trade-off at this stage.

4.5 IDE Integration

Language tooling shouldn't stop at developing a working compiler but to encompass the entire developer experience. A seamless workflow where IDE features like documentation previews, semantic symbol renaming and go-to-definition work across language boundaries greatly enhances usability.

Our design proposes the creation of a Hylo Language Server that acts as a proxy, coordinating with `clangd`, the official C/C++ language server for the LLVM project and Hylo's own language server that leverages the Hylo compiler as a library under the hood. This is a standard approach taken by many web technologies, and more relevantly, by Swift's SourceKit-LSP[8].

4.6 Customization and Ambiguous Mappings

A core principle of our design is acknowledging that a single, direct mapping is often insufficient. Many C constructs are used to represent different semantic ideas. A prime example is the C `enum`. It can represent:

1. A set of mutually exclusive cases → translate to a Hylo enum for proper exhaustiveness checking

[8]See SvelteKit's Language Tools, Angular's Language Service, and Swift's SourceKit-LSP for examples.

2. A collection of independent bitflags → translate to an `OptionSet` [13] for easy combination and checking of binary flags.
3. A simple group of named integer constants. → translate to a Hylo namespace with constants.

A rigid mapping to a single Hylo construct would be non-idiomatic in at least two of these cases. Therefore, our design proposes a system of **sensible defaults with user-customizable overrides.** By default, an `enum` might be imported as a struct of static constants, which is always safe. However, the developer can provide an annotation (e.g., in a separate configuration file or directly in the C header if they have ownership) to guide the mapping:

This principle of user-guided mapping extends to other areas, such as specifying that a C function returning an integer error code should be imported as a Hylo function that `throws`. This empowers developers to create bindings that are not only correct but also safe and highly idiomatic.

5 Mapping C Constructs to Hylo

This section provides the detailed mapping rules of C constructs to Hylo. These rules implement the architecture described in section 4, providing reasonable defaults and alternative customization options for capturing the semantic details of C constructs in Hylo's type system.

5.1 Primitive Types

Integer Types

The mapping of primitive types is foundational to interoperability. Mapping fixed-size integers (`uint8_t`, `int32_t`, etc.) and word-sized integers (`intptr_t`/`uintptr_t`) is trivial, as they have an exact corresponding type in Hylo. A key challenge is that the size of C's standard integer types (`char`, `short`, `int`, `long`) are platform-dependent. For example, `char` is defined as an at least 8-bit wide, either signed or unsigned integer type, representing the smallest addressable unit of memory on the target (1 byte). Modern processors generally agree on 8-bit bytes, and the niche use cases regarding digital signal processors fail to support modern C/C++ standards, which lead to the C++ standard proposal [14] to constrain a byte to have exactly 8 bits, which Hylo will deliberately adopt. However, the signedness of `char`, and the exact sizes of standard integer types vary per platform.

Table 1: **Numeric type mapping in Swift[15][16], Muon[17], Zig[18], Rust[19] and TinyGo[20].** Cells colored blue indicate that the language uses a platform-specific types, while white cells indicate a fixed type assignment that limits portability for incompatible platforms. Red cells mark unavailable or experimental mappings, and *?*-s indicate undocumented or unknown mappings.

C Type Equivalent	*Swift Mapping*	*Muon Mapping*	*Zig Mapping*	*Rust Mapping*	*TinyGO*
char	CChar = Int8 (!)	*byte/sbyte (8-bit)*	*c_char = i8/u8*	*c_char = i8/u8*	*int8/uint8*
signed char	CSignedChar = Int8	sbyte (8-bit)	i8	c_schar = i8	int8
unsigned char	CUnsignedChar = UInt8	byte (8-bit)	u8	c_uchar = u8	uint8
(signed) short	CShort = Int16	short (16-bit)	*c_short*	c_short = i16	*int8...int64*
unsigned short	CUnsignedShort = UInt16	ushort (16-bit)	*c_ushort*	c_ushort = u16	*uint8...uint64*
(signed) int	CInt = Int32	int (32-bit)	*c_int*	*c_int = i16/i32*	*int8...int64*
unsigned int	CUnsignedInt = UInt32	uint (32-bit)	*c_uint*	*c_uint = u16/u32*	*uint8...uint64*
(signed) long	*CLong = Int32/Int*	*int/long (32/64-bit)*	*c_long*	*c_long = i32/i64*	*int8...int64*
unsigned long	*CUnsignedLong = UInt32/UInt*	*uint/ulong (32/64-bit)*	*c_ulong*	*c_ulong = u32/u64*	*uint8...uint64*
(signed) long long	CLongLong = Int64	long (64-bit)	*c_longlong*	c_longlong = i64	*int8...int64*
unsigned long long	CUnsignedLongLong = UInt64	ulong (64-bit)	*c_ulonglong*	c_ulonglong = u64	*uint8...uint64*
float	CFloat = Float	float (32-bit)	f32	c_float = f32	*float32/float64*
double	CDouble = Double	double (64-bit)	f64	c_double = f64	*float32/float64*
_Float16	CFloat16 = Float16	-	f16	f16 (Exp.)	-
long double	*CLongDouble = Double/Float80*	-	*c_longdouble*	*c_longdouble*	*float32/float64*
wchar_t	CWideChar = Unicode.Scalar	*platform-defined int*	?	*c_wchar = u16/u32/ i32/c_uint/c_int*	-
ptrdiff_t	Int	*platform-defined int*	?	c_ptrdiff_t (Exp.) = isize	?
size_t	Int (!)	*platform-defined int*	usize	c_size_t (Exp.) = size	?
ssize_t	Int	*platform-defined int*	isize	c_ssize_t (Exp.) = isize	?

A simplified mapping is possible when the target platforms are known. On modern desktop platforms, the ILP32, LLP64, or LP64 data models define standard integer sizes as follows: 8-bit `char`, 16-bit short, 32-bit int, 64-bit long long, and either 32-bit or 64-bit long. Based on this, Swift and Muon map standard C integer types directly to their fixed-size equivalents, except for long, which is mapped to a type alias CLong that resolves to either a 32-bit or 64-bit integer based on the platform[9].

Other languages like Rust and Zig that aim to support diverse platforms, including various embedded and mobile architectures, define more of their types with conditional type aliases. Table 1 presents a comparison of numeric type mapping in Swift, Rust, Zig and Muon.

In Hylo, we propose a conservative mapping by default similar to Zig but with a novel addition: instead of type aliases, we use distinct types and provide the following explicit conversions to clearly express the intent of the programmer:

- **Trap on loss:** when the conversion is narrowing, we insert a runtime assertion that ensures that the particular input can be represented in the target type. We provide this as the default conversion, e.g. Int32(c_value). When the conversion is non-narrowing, the assertion is not needed, and a zero-extend/sign-extend operation is performed based on signedness. Additionally, this default and recommended trapping behavior could be disabled with a compiler flag to sacrifice safety for maximal runtime performance.
- **Truncate if needed:** when the represented value's meaning allows, we can silently drop the most significant bits that don't fit the target type. Note, due to two's complement representation, values that fit in the target type will preserve their sign. This may be spelled as Int32(truncating_if_needed: c_value)
- **Don't narrow:** We can use this when performance is critical but we also want a compile-time guarantee that a conversion is lossless on the target platform. Int32(non_narrowing: c_value) is a conditionally enabled conversion that is only available when the conversion would be lossless.

Using these explicit conversions ensures that we explicitly capture the programmer's intent and the code remains maximally portable. The viability of the approach was validated by implementing the C integer types and conversion functions between them in the Hylo standard library, and testing its usage in Hylo code. The standard library extension and example usage can be found in [22].

However, writing all these explicit conversions by hand may be very inconvenient, especially when we have solid assumptions about the target platform. E.g. when writing a wrapper around the LLVM compiler, we know that it will only be used on modern desktop platforms, where we can define many of the mappings like Swift or Muon did. Therefore, we allow making project-specific explicit assumptions about the target platform, such as c_short is 16-bit. These assumptions can be checked for all the specified target platforms before compilation, so there are no risks introduced.

In addition, if a Hylo project generally uses the Int or Int32 type for indices, it may specify to translate all size_t function parameters to the expected types in all or some methods. Unless the bit-width is the same, this may involve truncation, which could be configured to trap on overflow or signal narrowing as an error *before* compilation. When a runtime conversion is needed, the original C function is not exposed to our Hylo code; instead, a new wrapper function calling it and performing the necessary checks is exposed.

The technique of mapping a C type to its own distinct type in Hylo, and utilizing explicit conversions can not only be applied to standard integer types, but also to other types that have varying valid value ranges on different platforms: (u)int_fastN_t, (u)int_leastN_t, intmax_t, wchar_t, wint_t, size_t, ssize_t[10], (u)intptr_t and ptrdiff_t.

On modern, flat-memory model systems, compilers define size_t, ssize_t, (u)intptr_t, ptrdiff_t generally as pointer-sized integers, but special architectures like capability-based architectures[23][24] or 16-bit segmented memory architectures[25] may have different definitions. Due to this constraint holding true on most modern platforms, we suggest mapping these types to Hylo's pointer-sized Int/UInt types by default for general convenience, and providing options to override this assumption when a project wants to support strange platforms and needs these types.

The rest of the C integer types are more straightforward to map:

[9]Swift maps char to a signed Int8 regardless of whether the platform defines it as signed. Zig originally mapped char to u8, but it was later decided to introduce the c_char alias [21].

Swift extensively uses its signed word-sized Int for indices, and maps C's unsigned size_t to a signed representation for convenience, relying on the assumption that programs don't use size_t's most significant bit. This implies that when importing a size_t constant $2^{64} - 1$ from C, Swift sees it as -1.

[10]ssize_t is available as a POSIX extension

- **`bool`/`_Bool`** is mapped to Hylo's `Bool` type. Before C99, there was no dedicated boolean type, so conversions from `int`/`char` would be needed.
- **`(u)intN_t` –explicit-width integers–** are mapped to Hylo's corresponding fixed-width integer types (`UInt8`, `UInt64`, etc.), which have the same memory layout and bit-width as the C types.
- **`_BitInt(N)`** cannot be directly mapped to Hylo yet due to the lack of arbitrarily sized integers. However, in structs we can map them to an opaque byte sequence, exposing the value through a computed property of the closest sufficient fixed-width integer type. In function calls, a wrapper function shall be exposed that presents fixed-width integer types in parameter and return types, with the necessary conversion performed and optionally bounds-checked.

5.2 Enums

C enums are unlike enums in most other languages. In C, enums are essentially named integer constants, and the enum's type can be implicitly converted to and from the underlying integer type. However, their usage often involves specialized semantics:

1. **Discriminated values:** often each enum case represents a distinct value, and a variable holding an enum value is expected to only hold one of these values. One can perform pattern matching on such values, but exhaustiveness is not checked.
2. **Bit flags:** enums are often used to represent a set of flags, where each case is a power of two, and the enum value can be any combination of these flags. Set operations such as *union*, *intersection*, or membership checking may be performed via bitwise operations.

Assuming Hylo's enums will be similar to Swift's enum type, there are multiple possible mappings, of which the developer shall be able to choose the most suitable one for their declaration:

- **Closed enum**: a closed set of enum cases, usable for exhaustive pattern matching. When mapping libraries, we must be careful with these, as C programmers generally think of adding a new enum case as a non-breaking change. The mapping generator should assert that all enum cases are distinct.
- **Open enum**: an extendable set of enum cases, allowing for extensibility. This mapping is more flexible but requires a default case in pattern matching, handling any unexpected values. The generator should assert that all enum cases are distinct.
- **OptionSet**: a data structure with high-level operations for set operations like union, intersection, and membership checking. This mapping is suitable for bit flags, and the generator should assert that all enum cases are powers of two (not necessarily distinct).
- **Raw enum**: This mapping treats the enum simply as a collection of named integer constants. It is the most conservative approach, best suited for when an enum's purpose is to group a set of related values. While this is the default mapping, we could customize the project-level default.

Floating Point Types

There are 3 universally available **standard floating point types** in the C23 standard: `float`, `double`, `long double`. The standard doesn't mandate their representation, but implementations generally follow[26] binary formats defined by IEEE 754[27], except for `long double`[28][29], thus Rust, Zig and Swift universally map `float` and `double` to IEEE 754's 32- and 64-bit binary formats correspondingly, while mapping `long double` to platform-dependent type-alias when supported. Hylo should follow this pragmatic approach by default but also allow a configuration option to import them as `CFloat`/`CDouble` to be more platform-agnostic, with explicit conversions where needed. These conversions may be narrowing due to rounding, overflow to $\pm\infty$ or underflow to 0. However, since floating point types are generally assumed to be used for imprecise calculations, these conversions aren't handled specially by default. In the unlikely case when a guaranteed non-overflow/underflow is needed, one can perform the checks manually.

The standard also defines two **optional features**: **complex**/imaginary (since C99) and **decimal** (since C23) floating point types. While complex types may be easily implemented as a Hylo product type of the corresponding floating point types (e.g. `Complex<CLongDouble>`), on most architectures decimal types require software emulation of arithmetic operations (such as by a library like: Libdfp), and are not yet supported in LLVM/Clang[30]. Decimal types are also not used commonly in practice, so we also don't support them in Hylo natively.

5.3 Composite Types

As established in the architecture subsection 4.3, composite types like structs and unions are mapped using opaque storage and computed properties to ensure a correct memory layout. Additionally, we describe how flexible array members are mapped.

Flexible Array Member In C, a special array member without a specified size may appear at the end of a struct declaration. Such member can get more storage based on how much space is allocated for the struct, which allows accurate representations of e.g. network packets composed of a header and a variable length payload. This may be implemented in Hylo as a computed property that exposes a typed pointer to the first element of the flexible array member[11]. Structs with a flexible array member shall additionally have a non-copyable zero-sized member that prevents the synthesis of the `Copyable`, `Movable` and `Equatable` traits which then must be implemented explicitly when needed.

5.4 Pointer Types

C's **pointer to data** are mapped to Hylo's `Pointer<T>` and `PointerToMutable<T>` types, depending on the `constness` of the pointee. **Function pointers** can be mapped to Hylo closure types with a `@convention(C)` attribute and an empty environment, which ensures that the correct calling convention is used when the function is called. The details of this are discussed in the proposal [31].

For indicating nullability of pointers, Hylo could adopt a similar approach to Swift[32], where C declarations are annotated with a *nullable*, *non-nullable* or *null-unspecified* attribute. A *nullable* annotation would wrap the imported Hylo pointer/closure within an optional type, which must be guaranteed the same memory layout as the original C pointer type, and where unwrapping the optional involves checking for null under the hood.

Swift already supports a basic scoped customizability using `pragmas` but the mapping should be more flexible, and allow customizability without source modification of an existing library.

5.5 Type Qualifiers

C types may have zero or more type qualifiers that modify their semantics or the way their usage gets compiled. `const`-qualified types indicate that their value cannot be modified, at least not through that particular binding. This is widely applicable to language constructs that need to be mapped, such as global variable bindings, function parameters, struct members, and nested types within pointers. Constant global variables and struct members are mapped to immutable let bindings, function parameters (as always) are mapped to immutable parameters, and pointers to const types are mapped to `Pointer<T>` instead of `PointerToMutable<T>`.

The `volatile`, `restrict` and `_Atomic` qualifiers signal additional information for the compiler backend. Further research is necessary to see how to handle these properly, as many interop technologies ignore them (Swift, Rust Bindgen) while preliminary efforts for support exist in Zig[33].

5.6 Global Variables and Constants

While Hylo deliberately avoids shared mutable state due to its inherent risks [34][35], practical needs in C interoperability and embedded systems, such as sharing data with interrupt service routines [36], require its introduction. We propose declaring mutable globals with `unsafe var <name>: <type>`, mandating that all access occurs within an `unsafe` context to ensure developer vigilance. The `@extern` version, `@extern unsafe var <name>: <CType>`, supports linking with external C code. In contrast, their immutable counterparts are declared using `let` and `@extern let` for global constants, and can always be accessed safely. Support for thread-local variables is not yet proposed and remains an area for future research.

5.7 Function Declarations

C functions are mapped to `@extern` Hylo function declarations with an additional `@convention(C)` attribute so that the compiler can use the correct calling convention when calling the function.

C function signatures are often ambiguous; a pointer parameter might be used for reading, writing, or both. This ambiguity presents a challenge because while Hylo's own parameter conventions (`sink`, `let`, `inout`, `set`) could precisely capture these different intents, automatically and safely inferring the correct one from a C header is infeasible, and often impossible due to less tight semantics.

[11]Even though LLVM allows this type of access without undefined behavior, care should we taken as Hylo adds more optimizations.

Therefore, Hylo adopts a consistent default inspired by Rust and Swift. A C pointer's value is defined as the memory address it points to. In function parameters, pointer arguments are copied by their "value", a memory address. The address itself cannot be changed within a function's body unless taking a pointer to pointer as an argument, thus all C functions are imported with their parameters treated as immutable `let` bindings, pointers being translated to `Pointer<T>` or `PointerToMutable<T>`.

It is then the developer's responsibility to build safe, idiomatic Hylo wrappers around these raw, low-level imports to build abstractions whose values are not merely memory addresses but have proper whole-part relationships.

Developers can also annotate C function parameter and return types to use a more suitable Hylo type where the default mapping is inadequate. If this custom type's memory representation differs from the original, a wrapper function is automatically synthesized and exposed to perform the necessary conversions under the hood, resulting in less verbose or error-prone usage of the C code.

Header-Defined Functions (`static`, `inline`, `static inline`)

Header-defined functions are handled using the wrapper-generation approach described in section 4.4, ensuring they can be called from Hylo without requiring direct inlining.

Variadic Functions

Variadic functions in C can be called with any number and type of arguments without their usage being checked at compile time or runtime. Due to their inherent lack of type safety, they shouldn't be directly imported into Hylo. Instead of adding a special language feature exclusively for C interoperability, as Rust did for its FFI[37], Hylo prioritizes keeping the language simple.

Since variadic functions are common in C code, Hylo supports two primary strategies for interoperability. The first approach is to write a dedicated, type-safe wrapper in C. Alternatively, developers can leverage the common C practice of calling a function variant that accepts a `va_list` pointer[38]. Hylo may facilitate this by providing a mechanism to explicitly construct a `va_list` object[39]. Although this method remains fundamentally type-unsafe, its manual nature makes the associated risks explicit. This design choice enables necessary interoperability without creating a misleading abstraction that would disguise an unsafe C pattern as a conventional, safe Hylo function call.

5.8 Synthesizing Core Trait Conformances

Hylo offers a set of core traits, including `Copyable`, `Movable` and `Deinitializable`, that provide ways to specify and customize a type's lifetime management capabilities. To add these capabilities to a standard Hylo struct, a developer must explicitly declare conformance to the desired traits. The compiler can then automatically synthesize the necessary implementations, provided that all of the struct's members also conform to those traits, or it can be customized by providing an implementation.

While it might seem practical to apply this synthesis to C types—for instance, by using a heuristic such as checking if a C struct contains any pointers—doing so would introduce hidden, implementation-defined promises that could hinder library evolution. A developer must remain conscious of the capabilities assigned to imported C types and determine whether a default implementation is sufficient or if a custom one is required.

This challenge is evident in existing tools. For example, Rust's bindgen attempts to derive `Copy` and `Clone` traits automatically on a best-effort basis, allowing users to opt out as needed[40]. However, this approach can lead to incorrect value semantics[12], such as when a type containing pointers to heap-allocated data is mistakenly marked as copyable[13].

Therefore, we propose a more deliberate approach for Hylo. Developers can either write the trait conformances for C types manually or use an auxiliary tool to generate the declarations based on conservative heuristics. Regardless of the method chosen, we encourage developers to check the resulting code into their repository and take ownership of it. This practice avoids integrating trait derivations into the build process directly. Should subsequent changes to the original C declarations—such as adding a pointer to a struct—invalidate the initial assumptions, the compiler can emit a warning to alert the developer as proposed in [41].

[12]This may seem like an insignificant issue in practice, as most C functions take struct parameters by pointers, but within a Hylo context, we expect that the majority of wrapping code can deal with values, and only expose their pointers underlying memory address for duration of the C function call.

[13]See an example for an incorrectly derived Copy/Clone conformance: https://edu.nl/ghfrm

5.9 Macros

C preprocessor macros present a major challenge for high-fidelity interoperability due to their purely textual nature. Unlike functions, macros operate on token streams before the C compiler performs semantic analysis, leading to well-known issues with operator precedence, unintended duplication of side effects, and a complete lack of type information. While some simple use cases can be handled, the direct translation of more complex use cases is an open research problem. Consequently, devising a macro translation strategy for Hylo is considered out of scope for this paper, however we provide a brief overview of existing industry and research efforts in subsection 6.1.

5.10 Array Types

In C, arrays are contiguous sequences of elements that can appear in several contexts[42], each requiring a specific mapping strategy in Hylo.

Arrays of Constant Known Size: These arrays can be defined as global variables, members of structs, or as part of multi-dimensional array types. This C type directly corresponds to Hylo's `Buffer` type, which represents a fixed-size array with its elements stored inline.

Arrays of Unknown Size: These can be found in a few specific situations: as incomplete types at file scope (a case not currently supported), as flexible array members in structs (see section 5.3), and in function parameters. When used as function arguments, C arrays–whether of known or unknown size–decay into a pointer to their first element. While this process loses compile-time information about the array's length, it preserves the element's type information, which is crucial for calculating the correct offset during pointer indexing. Consequently, arrays in function parameters are mapped to Hylo's `Pointer<T>` or `PointerToMutable<T>`, depending on the `const`-qualification of the element type.

Variable Length Arrays: C supports a highly constrained form of dependent type called variable length arrays (VLAs) that allows the size of an array type to be determined at runtime. This is especially useful for multidimensional data[43] where the length of each dimension can vary but we still want to allocate the data contiguously and use indexing to access the elements:

```
int arr[10][n][m] = malloc(10 * n * m * sizeof(int));
arr[1][2][3] = 42;
```

Concerning imported declarations, VLAs may only appear in function parameters, where their corresponding dimensions are also specified, allowing the implementation to use those to calculate the accurate offsets for indexing. Hylo doesn't have a concept of VLAs in the type system, but for interoperability purposes it's enough to expose VLA parameters as `Pointer<T>/PointerToMutable<T>` parameters, where `T` is the underlying type of the smallest unit element in the VLA (`int` in the example above). This lets the user allocate and set up a VLA parameter in Hylo using manual offset calculations for indexing, and pass the pointer to the C function.

6 Related Work

6.1 C Macro Importing and Translation

Handling the C preprocessor presents a primary challenge for high-fidelity interoperability, creating a significant gap where progress requires analyzing the limitations of mainstream tools alongside state-of-the-art academic and experimental techniques. Despite this complexity, research shows that many macros in C libraries conform to analyzable patterns and can thus be ported more easily [44]. An empirical study has also established a taxonomy of macro types by analyzing large codebases [45], which aids the development of tools that manipulate C programs. Building on these insights, language interoperability technologies used by the industry leverage various heuristics to detect and import different kinds of C macros with varying fidelity.

Rust Bindgen, Zig, Muon and Swift support importing C's `#define` macros that represent constants, including ones that contain simple expressions or depend on other constant-like macros. These are translated into constants whose types are inferred from their values to be a suitable type. Zig also supports function-like macros, which are translated into Zig inline functions.

A significant ongoing effort in Rust Bindgen aims to add support for more complex macros, including function-like macros, token pasting, stringification, and expression evaluation [46]. This approach relies on translating C macros into either functions or Rust macros, which provide the necessary flexibility in many use cases.

Other approaches have emerged in the context of full $C \to Rust$ source-to-source translation efforts. While C2Rust [47] and Corrode [48] only translate already preprocessed C code where macros are already expanded, Oxidize [49] presents a technique to translate most C macros into Rust-compatible constructs, thus minimizing the number of places where expansion is necessary before translation. Another approach is to fully expand macros and then use LLMs to refold them into more idiomatic Rust constructs to reintroduce abstractions. The refolded code is then verified for equivalence to the original C code [50].

When translating function-like macros to regular functions, it is important to establish the correct parameter types. Since these may not be evident from the macro definition, CppSig [51] extracts type signatures by analyzing their usage in an existing codebase. While mainly intended for developers' code comprehension, this work could also be useful for aiding automatic translators.

While existing tools have made significant strides in translating constant and simple function-like macros, handling the full complexity of the C preprocessor remains a major challenge. Achieving truly high-fidelity C interoperability therefore necessitates more sophisticated and robust macro translation mechanisms than are currently available in mainstream language tooling.

7 Responsible Research

7.1 Threats to validity

This research focused specifically on analyzing interop in statically compiled languages, thus we may have overlooked useful techniques that have only been used for interpreted/JIT-compiled languages.

The presented design in this paper is also not fully validated by implementation, thus additional ABI- or type system-related challenges may arise during the follow-up full implementation. To mitigate risks, we aimed at prototyping critical parts of the design and have it reviewed by language experts. Interview with experts were conducted in an open and informal format, and therefore transcripts are not shared.

7.2 Use of AI and knowledge management tools

We found that a significant portion of the knowledge related to cross-language interoperability resides within online documentation, community forums, articles, rather than academic papers. The sheer volume and varying density of relevant information within these resources presented a considerable navigational challenge. To assist in targeted exploration of specific research questions within this landscape, we utilized Gemini 2.5 Pro's Deep Research and documented our findings and sources in Obsidian. In addition, we used Gemini 2.5 Flash for answering questions from the uploaded C23 and C++23 standards, and highlighting relevant sections that we consequently reviewed in detail.

To ensure transparency and enable critical assessment of AI-derived insights, we adopted a systematic recording process. For each significant AI-assisted inquiry, we documented the input prompt, the research methodology generated by the model, and its complete output report, including all cited references. This practice proved crucial when a contradiction in AI-synthesized information was detected. Our records allowed us to trace this discrepancy back to the AI model's reliance on forum discussions where information was partially presented or inaccurate. This incident highlighted the imperative of meticulously validating primary sources before drawing firm conclusions—a principle of heightened importance when AI is employed to navigate and synthesize information.

Our experience affirms that AI tools can significantly accelerate the initial identification of relevant materials within vast datasets, and they fundamentally augment, rather than replace, critical human oversight and rigorous source vetting. It is crucial to acknowledge that AI usage introduces its own set of risks if not managed with transparency and care. However, purely manual research methodologies are not without their own challenges; the sheer scale of available information and the limited personal experience within the field can amplify the effect of cognitive biases[52] influencing the exploration, potentially resulting in a less exhaustive and incomplete understanding of the topic. We found that the responsible and transparent application of AI, far from being merely a high-risk endeavour, can actually bolster research efforts. When used with caution and transparency, AI can help researchers delve deeper and explore more broadly than might be feasible through manual methods alone, thus reducing some inherent human limitations in processing extensive information.

To manage the collected information, resources, references and notes, we employed Obsidian as our knowledge management environment, adapting the Luhmann's Zettelkasten method[53] to organize our findings. This method involves creating atomic, interlinked notes enriched with tags and metadata, facilitating a granular and networked understanding of the subject matter. The DataView plugin for Obsidian further enhanced our organizational capacity by enabling queries based on these tags and metadata, allowing us, for example, to effectively track the progress of reviewing academic articles and conference talks.

This approach to both AI-assisted exploration and overall information management was intended to ensure the integrity of our findings and maximize their utility to the research community. No further ethical concerns were identified during the research.

8 Discussion and Future Work

This paper presents a concrete architectural design for a C interoperability layer in Hylo. The core of the design is to leverage the Clang compiler front-end for its high-fidelity C parsing and ABI handling, which ensures correctness across platforms and C dialects. We propose a decoupled C binding generator that produces Hylo declarations. This approach simplifies maintenance, as changes to the interoperability layer do not need to affect the main compiler, and it allows Hylo's development to focus on language features rather than replicating the complex, platform-specific logic already perfected in Clang.

A known limitation is the handling of complex C preprocessor macros, which are difficult to translate directly and will often require manual C wrappers, and type qualifiers, which, apart from `const`, don't have many precedents for being mapped.

The implementation of this design opens several avenues for future work:

- **Full Implementation:** The immediate priority is to build the C binding generator and fully implement the C-to-Hylo mapping specification detailed in this paper.
- **Macro Translation Research:** A dedicated research effort is needed to expand support for translating complex and function-like C macros into idiomatic Hylo constructs, building upon existing work in the field.
- **Mapping Rule Library:** The design of a Hylo library for defining custom mapping rules needs to established through an iterative process through implementation. This will allow developers to provide the guidance and extra information to the binding generator that one-size-fits-all binding generators lack.
- **Foundation for C++ Interoperability:** This C interoperability layer is a prerequisite for the more complex challenge of C++ interoperability. While this C framework provides a foundation, seamless C++ integration will likely require a more tightly-coupled approach. For instance, the Hylo compiler may need to directly interact with Clang to request C++ template instantiations where required. Zngur[54] is a promising technology building on the C interoperability layer of Rust to interoperate with C++ robustly.

9 Conclusion

Effective C interoperability is critical for the success of a new systems language like Hylo. This paper provides a principled and practical design for a high-fidelity interoperability layer that balances safety, usability, and performance. By leveraging Clang for correctness and providing a flexible semantic mapping framework, our work offers a clear implementation roadmap for Hylo. It also serves as a reference for other language designers facing the essential task of bridging new languages with the existing C ecosystem.

Acknowledgements

I would like to thank the following individuals for generously sharing their time and expertise in interviews that informed this research:

- David Sankel, principle scientist at Adobe, C++ committee member, and contributor to the Zngur Rust-C++ interop project[54].
- Michal Brzozowski, creator of the Zenon language[55].
- Timur Doumler, C++ standards expert and committee member.
- Emilio Cobos Álvarez, contributor to Rust Bindgen[8] and CBindgen[56] at Mozilla.
- Nick Matthijssen, creator of the Muon language.
- Timen Zandbergen, a masters student at TU Delft researching safe separate recompilation in Rust.

I would also like to thank the Hylo contributors for the discussions and feedback on the Hylo C interoperability design, in particular Dimitri Racordon, Lucian Radu Teodorescu, and Dave Abrahams. Last but not least, I would like to thank Dénes Balogh for his invaluable help in deploying the ABI Explorer prototype for public use[3].

References

[1] Tsvi Cherny-Shahar and Amiram Yehudai. *Multi-Lingual Development & Programming Languages Interoperability: An Empirical Study*. 2024. arXiv: 2411.08388 [cs.PL]. URL: https://arxiv.org/abs/2411.08388.

[2] David M. Beazley. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++". In: *Proceedings of the 4th Annual Tcl/Tk Workshop*. Monterey, CA, July 1996. URL: https://www.swig.org/papers/Tcl96/tcl96.html.

[3] Dénes Balogh Ambrus Tóth. *ABI Explorer*. 2025. URL: https://abiexplorer.org/ (visited on 05/30/2025).

[4] Ambrus Tóth. *An example mapping of a C union to Hylo*. June 2025. URL: https://gist.github.com/tothambrus11/fc8d9385ddcb3593f22b1e9ecce5f3b2 (visited on 06/21/2025).

[5] Ambrus Tóth. *Mapping of flexible array members to Hylo*. June 2025. URL: https://gist.github.com/tothambrus11/84b2e44bfa5a1453698505c451a3ef65 (visited on 06/21/2025).

[6] Ambrus Tóth. *Mapping of C bit-fields to Hylo*. June 2025. URL: https://gist.github.com/tothambrus11/79a101d86d787950d7fb3ae15ca4daf8 (visited on 06/21/2025).

[7] Rust Contributors. *Type Layout - The C Representation*. URL: https://doc.rust-lang.org/beta/reference/type-layout.html#r-layout.repr.c.struct (visited on 06/16/2025).

[8] Rust Bindgen Contributors. *Bindgen*. URL: https://github.com/rust-lang/rust-bindgen (visited on 06/13/2025).

[9] Apple Inc. *How Swift Imports C APIs - Structs*. URL: https://github.com/swiftlang/swift/blob/main/docs/HowSwiftImportsCAPIs.md#structs (visited on 06/16/2025).

[10] International Organization for Standardization. *Information technology – Programming languages – C*. Standard ISO/IEC 9899:2024(E). Section 6.7.3.2, "Structure and union specifiers". ISO/IEC, 2024.

[11] Rust Bindgen Contributors. *Why isn't bindgen generating bindings to inline functions? - FAQ - The Bindgen User Guide*. 2025. URL: https://web.archive.org/web/20250621101712/https://rust-lang.github.io/rust-bindgen/faq.html#why-isnt-bindgen-generating-bindings-to-inline-functions (visited on 06/21/2025).

[12] Ambrus Tóth. *Comment on feature request for a flag for wrapping inline functions #3060*. 2025. URL: https://github.com/rust-lang/rust-bindgen/issues/3060#issuecomment-2993451966 (visited on 06/21/2025).

[13] Apple Inc. *OptionSet Protocol*. URL: https://developer.apple.com/documentation/swift/optionset (visited on 06/15/2025).

[14] JF Bastien. *There are exactly 8 bits in a byte*. Tech. rep. P3477R1. C++ Standard Proposal. ISO/IEC JTC1/SC22/WG21, 2024. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3477r1.html.

[15] Apple Inc. *How Swift Imports C APIs - Fundamental Types*. URL: https://github.com/swiftlang/swift/blob/main/docs/HowSwiftImportsCAPIs.md#fundamental-types (visited on 06/18/2025).

[16] Apple Inc. *BuiltinMappedTypes.def*. 2024. URL: https://github.com/swiftlang/swift/blob/3d9e49038b6ef568e79841ec1528cdbda420cc8c/include/swift/ClangImporter/BuiltinMappedTypes.def (visited on 06/18/2025).

[17] Nick Matthijssen. *Muon's FFI Generate Pass - Primitive Types*. 2022. URL: https://github.com/nickmqb/muon/blob/10781443d11fc49e4e5a32550c012407e7d714cf/ffigen/src/generate_pass.mu#L90 (visited on 06/18/2025).

[18] Zig Contributors. *Primitive Types - Zig Documentation*. URL: https://ziglang.org/documentation/0.14.1/#toc-Primitive-Types (visited on 06/18/2025).

[19] Rust Contributors. *Rust Core FFI Primitives*. 2025. URL: https://github.com/rust-lang/rust/blob/1bb335244c311a07cee165c28c553c869e6f64a9/library/core/src/ffi/primitives.rs (visited on 06/18/2025).

[20] TinyGo Contributors. *LibClang Parser in TinyGo CGO*. 2025. URL: https://github.com/tinygo-org/tinygo/blob/release/cgo/libclang.go#L744.

[21] Zig Contributors. *Add c_char type - Issue 875 - Zig Language*. 2018. URL: https://github.com/ziglang/zig/issues/875 (visited on 06/18/2025).

[22] Ambrus Tóth. *Distinct Standard C Integer Types for Hylo and Mapping Examples #1712*. June 2025. URL: https://github.com/hylo-lang/hylo/pull/1712 (visited on 06/21/2025).

[23] Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann. *An Introduction to CHERI*. Technical Report 941. Section 2.1, "A Portable Architectural Protection Model". University of Cambridge Computer Laboratory, 2019.

[24] HackerNews Users. *On Beej's Guide to C Programming*. Forum discussion. 2023. URL: https://news.ycombinator.com/item?id=34950530#:~:text=sizeof(size_t)%20%3D%3D%20sizeof(intptr_t) (visited on 06/18/2025).

[25] Alex Martelli. *size_t vs. uintptr_t - Stack Overflow*. 2009. URL: https://en.wikipedia.org/w/index.php?title=16-bit_computing&oldid=1295239340 (visited on 06/18/2025).

[26] LLVM Contributors. *Floating Point Types - LLVM Language Reference Manual*. 2025. URL: https://llvm.org/docs/LangRef.html#floating-point-types.

[27] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[28] Wikipedia contributors. *Long double — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-June-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Long_double&oldid=1279938517.

[29] Alex Martelli. *Half & Quad Precision Floating Point Support*. Apr. 2025. URL: https://hackmd.io/@8W5l8q6-Qyyn_vKh2-L0RQ/rJpZsmcdh (visited on 06/18/2025).

[30] LLVM Contributors. *GCC Extensions not implemented yet - LLVM Language Reference Manual*. 2025. URL: https://clang.llvm.org/docs/UsersManual.html#:~:text=clang%20does%20not%20support%20decimal%20floating%20point%20types.

[31] Ambrus Tóth. *Function, Function Pointer, Closure and C Function Pointer Interop in Hylo*. URL: https://github.com/orgs/hylo-lang/discussions/1705 (visited on 06/16/2025).

[32] Apple Inc. *How Swift Imports C APIs - Nullable and non-nullable pointers*. URL: https://github.com/swiftlang/swift/blob/main/docs/HowSwiftImportsCAPIs.md#nullable-and-non-nullable-pointers (visited on 06/16/2025).

[33] Zig Contributors. 2025. URL: https://github.com/ziglang/zig/issues/23329 (visited on 06/16/2025).

[34] Cosmin Marsavina. "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems". In: *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. 2020, pp. 000105–000110. DOI: 10.1109/SACI49304.2020.9118816.

[35] Matthieu Cneude. *Global Variables and States: Why So Much Hate?* 2019. URL: https://thevaluable.dev/global-variable-explained/ (visited on 06/18/2025).

[36] Rust Embedded Devices Working Group. *Not Yet Awesome Embedded Rust: Sharing Data with Interrupts*. 2024. URL: https://github.com/rust-embedded/not-yet-awesome-embedded-rust#sharing-data-with-interrupts (visited on 06/18/2025).

[37] Rust Contributors. *Variadic Functions - FFI - The Rustonomicon*. 2025. URL: https://doc.rust-lang.org/nomicon/ffi.html#variadic-functions (visited on 06/20/2025).

[38] 2022. URL: https://bumbershootsoft.wordpress.com/2022/07/24/custom-printf-wrapping-variadic-functions-in-c/ (visited on 06/20/2025).

[39] Apple Inc. *Use a CVaListPointer to Call Variadic Functions - Apple Developer Documentation*. 2025. URL: https://developer.apple.com/documentation/swift/using-imported-c-functions-in-swift#Use-a-CVaListPointer-to-Call-Variadic-Functions (visited on 06/20/2025).

[40] Bindgen Contributors. *Preventing the Derivation of Copy and Clone*. 2025. URL: https://rust-lang.github.io/rust-bindgen/nocopy.html (visited on 06/19/2025).

[41] Ambrus Tóth. *Proposal: Warning for Default Trait Implementations on Structs with Pointers #1711*. 2025. URL: https://github.com/orgs/hylo-lang/discussions/1711 (visited on 06/19/2025).

[42] C++ Reference Contributors. *Array Declaration - cppreference.com*. 2025. URL: https://cppreference.com/w/c/language/array.html (visited on 06/20/2025).

[43] Tstanisl. *Are variable length arrays bad? When or when not should I use them?* 2022. URL: https://www.reddit.com/r/C_Programming/comments/yigtue/are_variable_length_arrays_bad_when_or_when_not/ (visited on 06/20/2025).

[44] Brent Pappas and Paul Gazzillo. "Semantic Analysis of Macro Usage for Portability". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. ACM, Feb. 2024, pp. 1–12. DOI: 10.1145/3597503.3623323. URL: http://dx.doi.org/10.1145/3597503.3623323.

[45] M.D. Ernst, G.J. Badros, and D. Notkin. "An empirical analysis of c preprocessor use". In: *IEEE Transactions on Software Engineering* 28.12 (2002), pp. 1146–1170. DOI: 10.1109/tse.2002.1158288.

[46] Markus Reiter. *Support complex macros. #2369 - Rust Bindgen*. 2024. URL: https://github.com/rust-lang/rust-bindgen/pull/2369 (visited on 06/19/2025).

[47] Per Larsen. *C2Rust: Migrating Legacy Code to Rust*. Presented at RustConf. 2018. URL: https://www.youtube.com/watch?v=WEsR0Vv7jhg (visited on 06/20/2025).

[48] Jamey Sharp. *Corrode: C to Rust Translator*. 2017. (Visited on 06/20/2025).

[49] Robbe De Greef et al. "Towards Macro-Aware C-to-Rust Transpilation (WIP)". In: *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '25. Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 5–61. ISBN: 9798400719219. DOI: 10.1145/3735452.3735535. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3735452.3735535.

[50] Adam Karvonen. *Using GPT-4 to Assist in C to Rust Translation*. 2023. URL: https://www.galois.com/articles/using-gpt-4-to-assist-in-c-to-rust-translation (visited on 06/20/2025).

[51] Christian Dietrich. "CppSig: Extracting Type Information for C-Preprocessor Macro Expansions". In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. PLOS '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 62–68. ISBN: 9781450387071. DOI: 10.1145/3477113.3487268. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3477113.3487268.

[52] Wikipedia Contributors. *List of cognitive biases — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=List_of_cognitive_biases&oldid=1295239340 (visited on 06/13/2025).

[53] Sönke Ahrens. *How to Take Smart Notes: One Simple Technique to Boost Writing, Learning and Thinking - for Students, Academics and Nonfiction Book Writers*. North Charleston: CreateSpace Independent Publishing Platform, 2017.

[54] Hamidreza Kalbasi. *Zngur: A C++/Rust interop tool*. 2023. URL: https://hkalbasi.github.io/zngur/ (visited on 06/13/2025).

[55] Michal Brzozowski. *Zenon language*. URL: https://github.com/miki151/zenon (visited on 06/13/2025).

[56] Mozilla and cbindgen Contributors. *cbindgen*. URL: https://github.com/rust-lang/cbindgen (visited on 06/13/2025).

[57] Wikipedia Contributors. *C standard library — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/wiki/C_standard_library (visited on 06/15/2025).

[58] Hylo Contributors. *The Hylo Programming Language*. 2022. URL: https://hylo-lang.org/ (visited on 05/30/2025).

[59] Bret Brown. *Requirements for C++ Successor Languages*. Presented at C++Now. 2023. URL: https://www.youtube.com/watch?v=VMYVbA2gg0g (visited on 04/26/2025).

[60] Carbon Contributors. *Interoperability Philosophy and Goals*. 2023. URL: https://github.com/carbon-language/carbon-lang/blob/77afd0678baae83da8610dddff286739d271e18e/docs/design/interoperability/philosophy_and_goals.md#carbon-inheritance-from-c-types (visited on 05/30/2025).

[61] Apple Inc. *Mixing Swift and C++: Conforming C++ Type to Swift Protocol*. 2025. URL: https://www.swift.org/documentation/cxx-interop/#conforming-c-type-to-swift-protocol (visited on 05/30/2025).

[62] Apple Inc. *Supported Features and Constraints of C++ Interoperability*. 2025. URL: https://www.swift.org/documentation/cxx-interop/status/ (visited on 07/29/2025).

A Appendix: Use Cases of C/C++ Interoperability for Hylo

This appendix details the primary motivations and applications for C and C++ interoperability within the Hylo language. For any new systems language, understanding why interoperability is needed is as crucial as determining how it should be implemented. The following sections explore the key use cases that guide the design of Hylo's interoperability features, ranging from essential tasks like interfacing with operating system APIs to goals like bootstrapping a rich library ecosystem and enabling gradual adoption within existing projects. This analysis provides the necessary context for the technical design choices presented in this paper and clarifies which applications were prioritized.

A.1 Leveraging Operating System APIs

Interfacing with the operating system is essential for hosted programs to manage hardware, system resources, and communicate with the real world. Major operating systems including Linux, Windows, and macOS offer stable C APIs that enable tight integration. Due to the relative simplicity of the C ABI, C provides the most straightforward means of OS interaction for statically compiled languages.

For many common operations, such as file I/O and memory management, LibC, the **C standard library** [57] offers a portable abstraction layer over OS-specific functionalities, facilitating the development of platform-independent code. While the future Hylo standard library may involve native OS representations, leveraging the ubiquity of LibC allows Hylo to be used in a wide range of environments without requiring extensive platform-specific implementations.

A.2 Bootstrapping a Library Ecosystem

Bootstrapping a library ecosystem is hard. While most programs would be expressible in idiomatic Hylo code, new users typically want to use Hylo to solve their own problems instead of implementing a network protocol or a GUI toolkit. The ability to use C libraries from Hylo greatly **reduces the barrier to entry**, as the first set of Hylo libraries can be lightweight, safe wrappers around existing libraries that expose a C API. This way, Hylo can be used to solve real-world problems right from the start, and the ecosystem can grow organically as more libraries are written in Hylo.

Direct interoperability with other higher-level systems programming languages like Rust, C++ or Swift could further ease the wrapping process, as these languages provide safer abstractions and more static guarantees than C.

A.3 Using Hylo in an existing C project

Translation is also beneficial, see Oxidize, C2Rust, CRust, Corrode. Support function-by-function translation, so that the code can be incrementally translated to Hylo. Also, the user has full ownership of this code for modification.

A.4 Hylo as a C++ Successor

Hylo's goal is to prove that mutable value semantics can provide safety and simplicity without compromising the performance of C and C++[58]. This requires that most problems that C++ is used for can be expressed in Hylo efficiently. While our hypothesis is that Hylo will be great for starting new projects, integrating into an existing project without rearchitecting it to avoid aliasing mutable state is challenging due to Hylo's stricter semantics.

An ideal successor language would allow **unordered, incremental adoption** with bi-directional interoperability[59], meaning that C++ components could be gradually replaced with Hylo components without having to migrate all its dependencies first. Carbon and Swift aim to support this with features like allowing Carbon types to inherit from C++ types[60] or allowing to conform C++ types to Swift protocols[61], however there are many limitations, and this is still an open research area[62]. Zngur's[54] approach is vastly different, it allows exposing a Rust library to C++, which is often a viable way to continue the development of a subset of an application in the new language while still having to write C++ for certain cases.

There is also an apparent **trade-off between the expressiveness of the mapped subset of C++ and the "purity" of the new language**. Mapping foreign constructs may require the new language to be extended with language or library features that circumvent the safety guarantees and value semantics. If the usage of such constructs is only allowed in an unsafe context, much of the Hylo code interacting with the C++ side would need to be marked as unsafe, which would increase the noise without providing much value. It is probably safer to program in the new language even without correctly marking everything unsafe than discouraging people from interop with unnecessarily verbose code and have them end up quitting and keep using C++ only, though developers will need to be cautious to not have false beliefs on the safety of their system.

Integration to the existing developer ecosystem, from IDEs to build is also crucial for commercial adoption in large projects. Debugging, profiling, testing workflows and language server features across boundaries shall be supported.[59] If Hylo will have its own build system, it shall shall support building existing C++ dependencies, but building Hylo code shall be supported in existing C++ build systems like CMake, Bazel and Meson to ease adoption in existing projects.

A.5 Authoring C++ Libraries In Hylo

Hylo offers a more ergonomic and less error-prone approach to generic programming than C++ through its robust, declaration-site-checked generics system. Many of Hylo's features, with a few exceptions like subscripts, could translate to idiomatic C++ code. This opens the door to emitting highly usable C++ libraries from Hylo.

These Hylo-generated libraries would adhere to C++ best practices, including leveraging concepts (derived from Hylo's traits) to ensure misuse is difficult. Using Hylo as a frontend for library development could mitigate common C++ authoring pitfalls, such as the inability to precisely specify function APIs and the reliance on creative template instantiations to expose bugs.

Some limitations arise due to the differences between the generics systems' semantics, e.g. with regards to specializations, whose implications would need further exploration.

Existing languages that transpile to C++ and support generic programming, including Haxe, Felix, Zenon, and Nim, do not rely on C++ templates to express their generics. Instead, they perform monomorphization within their own transpilation process.

A.6 Exposing Hylo Libraries Through a C API

Once a valuable library is written in Hylo and becomes popular, other language ecosystems may want to use it, e.g. for efficiency reasons or because rewriting it in the other language and maintaining the port would be infeasible. To support interfacing with all other languages without dedicated direct interoperability mechanism, it's best to expose Hylo libraries through a C API - which most languages support interfacing with, at least to a practical extent. Other languages can then write wrappers around the C API to provide a more idiomatic interface.

Exposing a C API may be done in various ways:

(a) **Manually**

- You need to make sure that your data structures and functions are marked with C ABI annotations. This may not align with the internals of your library, and you might not want to have C representation in the general case when the library is consumed from Hylo, so some manual wrapping may be necessary. You also need to write the matching C/C++ headers manually.
- See Rust's `reprc(c)` for data structures and `extern "C"` function declarations, and Swift's experimental `@c_decl`.

(b) **Semi-automatically**

- You still need to annotate your exposed data structures and functions as C compatible, but the header files are automatically generated based on your public C-compatible declarations. See Rust's CBindgen.
- Example: C - Go interop (CGO)
- This is a desirable improvement over the manual approach, but it can be added on top of the manual approach without requiring any changes to the language (see Rust's CBindgen).

(c) **Automatically**

- When writing your library, you don't need to think about making your structs and functions C-compatible - everything is handled for you automatically. This is hard for more complex languages, see concerns raised on this forum: https://forums.swift.org/t/formalizing-cdecl/40677.
- Since this C API will likely have many users, it is crucial to not make this a fragile API. The developer should be notified about C ABI breakages when publishing a new release.
- Examples:

- Fusion includes transpilation to C
- Embedded C scripting languages implemented in C

Some technologies also decided to not support declaring C structs in the new language, just import them from a C header file. Despite the loss of flexibility in where we write our APIs, this way, developers can still have fine control over what is being exposed and it also spares implementation effort for the interop technology maintainers, as importing C headers is likely already implemented due to its higher priority. For Hylo, we may implement a subset of the C struct layout rules, like Rust did, which would allow some flexibility for basic cases. Further research is needed to determine if it is feasible to expose a complex arbitrary Hylo library through a C API that is usable from the other languages.

A.7 Authoring Hylo libraries that can be directly used from other languages

- It's the safest and least error-prone to expose a Hylo library without having to rely on C as an intermediary. UniFFI-rs (Rust) provides an example of this.

A.8 Use Cases Discarded

- **Hylo as an embedded scripting language:** multiple embedded scripting languages support great interoperability with C++ such as Hobbes and ChaiScript. However, Hylo is designed to be beneficial for medium to large scale projects, and would provide no significant benefit over existing embedded scripting languages, so we can discard this use case.
- **C/C++ Interop for additional performance:** Hylo is designed as a high-performance system programming language, meaning that unlike managed languages (e.g. Python, NodeJS), it doesn't need to interoperate with C/C++ for maximal performance (except when hardware-specific drivers are written in C/C++ and abstractions are not yet exposed through Hylo).
- **Hylo as a C Successor:** While Hylo aims to leave no room for a lower-level language to achieve additional performance, its design discourages unsafe operations through reduced ergonomics, such as the absence of a dedicated pointer syntax. For problems inherently requiring frequent unsafe operations, this design choice could hinder productivity without offering significant advantages over simpler languages like Zig, C3, or Odin. In such scenarios, these alternatives might prove more suitable, and C interoperability could be used to expose their functionalities to Hylo.
- **Evolving C++**
  C++'s unopinionated design—its greatest strength—lets developers express nearly anything, supporting multiple programming paradigms. This ensures its continued relevance. However, decades of rapid complexity growth have made the language increasingly difficult to manage.

  While efforts like Carbon and Circle aim to evolve C++, Hylo takes a different approach. Instead of directly extending C++, Hylo offers a more constrained programming model, prioritizing safety and simplicity. We believe Hylo can solve the majority of C++'s use cases with greater simplicity, safety, and consistency. Like Rust, certain problems will remain easier to tackle in C++, but our goal is to minimize that subset.

B Appendix: Comparison of C AST Introspection Tools for Binding Generation

Generating high-fidelity Foreign Function Interface (FFI) bindings requires a robust method for parsing C/C++ header files to introspect their Abstract Syntax Trees (AST). This analysis compares the capabilities of major C/C++ compilers and related technologies for this purpose.

B.1 Compiler Introspection Capabilities

Clang Clang provides the most flexible and stable interfaces for building external tools. Its architecture is designed to support deep semantic analysis of C-family languages, making it the preferred choice for many interoperability projects. Key interfaces include:

- **LibClang:** A stable, C-based API that is easier to adopt but can be less performant than more direct interfaces. It is used by major tools like Rust's Bindgen.
- **LibTooling:** A C++-based API that offers more power and performance by exposing Clang's internal representations. However, its API is unstable, often requiring tools to bundle a specific version of Clang.
- **Clang Plugins:** Allow for deep integration but come with the overhead of an unstable API.
- **Forking Clang:** The approach taken by Swift, which involves maintaining a fork of the entire LLVM/Clang project. This offers maximum flexibility, including the ability to add custom attributes for enhancing cross-language IDE features (e.g., `external_source_symbol`), but represents a significant maintenance commitment.

GCC While powerful, GCC's introspection capabilities are primarily exposed through plugins. These plugins provide sufficient information but have an unstable API and rely heavily on macros, making them more challenging to work with than Clang's libraries. There are efforts within the Bindgen community to support GCC for cases where Clang has compatibility issues with specific GCC-dependent libraries.

MSVC Microsoft's C++ compiler has historically offered limited, well-documented public APIs for full AST introspection suitable for FFI generation. While interfaces like the Debug Interface Access (DIA) SDK exist, they are not designed for the kind of source-level analysis required for binding generation.

B.2 Approaches in Existing Interoperability Frameworks

The choice of parsing technology varies widely across different frameworks, reflecting different design philosophies.

Framework	Technology Used	Parsing Strategy
Swift	Fork of Clang/LLVM	Directly embeds and modifies Clang to parse C/C++ headers, enabling deep integration and custom attributes.
Bindgen (Rust)	LibClang	Relies on LibClang to parse C headers. There are ongoing efforts to explore using LibTooling for access to more features.
Crubit (Rust)	Clang/LLVM Libraries	Uses prebuilt Clang and LLVM libraries for its C++ interoperability.
CXX / UniFFI-rs (Rust)	Manual Bridge / IDL	Avoids parsing C/C++ headers. Developers manually declare a "bridge" layer in Rust or use an Interface Definition Language (IDL).
autocxx (Rust)	Fork of Bindgen	Inherits its reliance on LibClang from Bindgen.

Table 2: Parsing technologies used in various FFI frameworks.

B.3 Alternative Technologies

- **Manual Parsers:** Manually implementing a standard-compliant C/C++ parser is a monumental task. While projects like SWIG have their own parser, the effort is generally not justifiable given the quality of existing compiler-based tools.

- **Existing Reflection Libraries (Macro/Template-Based):** Third-party reflection libraries in C++, such as `refl-cpp`, `RTTR`, and `enTT meta`, typically rely on extensive macro or template metaprogramming. This approach has two significant drawbacks for FFI generation: it requires developers to annotate or generate declarations through macros, making it unsuitable for unmodified library headers, and it cannot discover all entities within a translation unit, which is essential for automatic binding generation.
- **C++ Static Reflection:** The upcoming C++26 standard includes static reflection capabilities (`P2996`). While experimental implementations exist, this feature is not yet widely available in major compilers like Apple's Clang. Once mature, it could provide a standardized, powerful alternative to current methods, generating binding code at compile time. The practicality of this approach is unclear.

C Appendix: Academic Literature Review Prompts

C.1 General Interoperability Research from Last 10 years (2015-2025)

Interoperability tools have vastly changed and developed in the last 10 years since the introduction of Swift in 2014 and Rust getting popular for the C and C++ community. We limited the general research to the last 10 years to make reviewing abstracts more feasible.

```
   TITLE-ABS-KEY ( ( ( programming W/2 ( language OR languages ) ) OR "polyglot
programming" OR "multi-language system*" OR "language integration" ) W/10 (
interoperability OR integration OR ffi OR "foreign function interface" OR
"cross-language" OR "inter-language" ) ) AND TITLE-ABS-KEY ( "empirical study"
OR "case study" OR evaluation OR comparison OR guideline* OR challenge* OR
"pain point*" OR trade-off* OR "experience report" OR "lessons learned" OR "best
practice*" OR survey OR limitation* OR obstacle* OR issue* OR qualitative OR
quantitative ) AND NOT TITLE-ABS-KEY ( "large language model" OR "machine learning"
OR "artificial intelligence" OR "natural language processing" OR nlp OR chatbot OR
transformer ) AND ( LIMIT-TO ( DOCTYPE , "ar" ) OR LIMIT-TO ( DOCTYPE , "cp" ) )
AND ( LIMIT-TO ( LANGUAGE , "English" ) ) AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
AND ( LIMIT-TO ( EXACTKEYWORD , "Interoperability" ) OR LIMIT-TO ( EXACTKEYWORD ,
"Computer Programming Languages" ) ) AND ( LIMIT-TO ( PUBYEAR , 2015 ) OR LIMIT-TO
( PUBYEAR , 2016 ) OR LIMIT-TO ( PUBYEAR , 2017 ) OR LIMIT-TO ( PUBYEAR , 2018 ) OR
LIMIT-TO ( PUBYEAR , 2019 ) OR LIMIT-TO ( PUBYEAR , 2020 ) OR LIMIT-TO ( PUBYEAR ,
2021 ) OR LIMIT-TO ( PUBYEAR , 2022 ) OR LIMIT-TO ( PUBYEAR , 2023 ) OR LIMIT-TO (
PUBYEAR , 2024 ) OR LIMIT-TO ( PUBYEAR , 2025 ) )
```

C.2 C Specific Research

```
( "c" W/2 interoperability ) AND NOT "LLM" AND NOT "artificial intelligence" AND NOT
"Large Language model" AND NOT "C-ITS"
```

C.3 C++ Specific Research

```
( "c++" W/2 interoperability ) AND NOT "LLM" AND NOT "artificial intelligence" AND
NOT "Large Language model" AND NOT "C-ITS"
```

C.4 General Review with Undermind AI

TODO redo this again with different query

Full search query: *I want to find a comprehensive collection and analysis of the desirable properties and evaluation criteria for native interoperability technologies between Hylo (as a prospective C++ successor) and C and C++, with a focus on overcoming the greatest challenges in such interop (e.g., code generation for difficult C constructs, macros, ABI compatibility, generics/templates, exceptions, idiomatic stdlib features, asynchrony, build/tooling ecosystem integration, incremental and unordered adoption). The review must include direct/native interop technologies (including Clang-based and other safety-improving FFI approaches, but not IDL/IPC), with concrete examples and architectural mechanisms that address the outlined challenges, emphasizing how they enable practical, safe, and incremental migration or coexistence between C++ and the new language*