Bachelor Thesis

Communication issues: Down the rabbit hole of language interoperability

by

Kai Erik Niermann

(2720905)

First supervisor: Atze Van der Ploeg
Daily supervisor: Atze Van der Ploeg
Second reader: Verano Merino

November 15, 2024

Submitted in partial fulfillment of the requirements for the VU degree of Bachelor of Science in Computer Science

Communication issues: Down the rabbit hole of language interoperability

Kai Erik Niermann
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
k.e.niermann@student.vu.nl

ABSTRACT

Your Master Thesis has to be written in English and should follow the ACM style (see the Overleaf template). We also suggest a *structured abstract* following the structure below.
Context. Write this at the end
Goal. Write this at the end
Method. Write this at the end
Results. Write this at the end
Conclusions. Write this at the end

As a very rough indication, the final thesis report typically entails, excluding appendixes, between 10 and 30 pages, with an average of 15 pages. The large variation depends on many variables including the specific field, project nature, and context. We advise to ask your supervisor if you should consider which number as a reference.

1 INTRODUCTION

It's somewhat hard to properly explain the depth of the rabbit hole I have fallen into with my thesis research. It started somewhat simple, I had this very basic idea in mind, what if you took a part of a program, and rewrote it in different programming languages and kind of like Legos slotted it into place of the original program? Now this isn't by any means a new idea, I honestly just had a mild curiosity about how that would work given that I've never really written and executed a program that was written in two different languages.

So to begin this exploration of language interoperability I had to first think of a piece of software I could use to test this idea. Before doing so I thought it would be important to further refine my idea beyond just wanting to test out my 'lego idea'. As said before the concept of rewriting code is by no means new, I don't believe it would be absurd by any means to suggest that most programmers at one point or another, while writing some piece of code, or even after the fact, recognized ways in which their solution can be improved. Often improvements or iterations on the original piece of code just naturally have to occur to fit the evolving goal for a given piece of software. So it would make sense that there should be some type of reason as to why we want to rewrite a specific piece of code, in our case performance would be a pretty natural choice. While most programming languages of course aim to be performant, respective to their application, due to differences in their design and execution methods they naturally exhibit differences.

Ok, so we want to choose a piece of software, preferably one with a performance-critical component, which we want to rewrite. Looking back at my previous projects one specifically came to mind, my implementation of a simple Ray Tracer in the Julia programming languages based on Peter Shirley's wonderful guide and C++ implementation. My initial goal with this project was to learn Julia as I had heard and seen that if written in a very idiomatic manner it can exhibit performance characteristics close to that of C while offering the dynamically-typed high-level simplicity of Python. Naturally, when benchmarking my final implementation my results were less than desirable. To elaborate on why my program exhibited these performance characteristics it makes sense to briefly go into the core of how Julia's code is run. There is a very nice quote on the Julia forums which I believe effectively conveys this.

> *in short, it's a reasonably good static compiler but a terrible JIT*

The first response some people might have to this is confusion. Julia is a JIT-compiled language so how is its JIT compiler terrible. Well, this comes down in large part to how its JIT compilation works, the core of this is how it performs online partial evaluation (OPE) w.r.t. its type system. OPE w.r.t the type system in this instance refers to the process of, at runtime (online), Julia specializes (partially evaluates) methods for which it can infer the types of every local variable from the types of the arguments or any other constants; in this instance, a method is said to be 'type stable'. Specializing type stable methods simply means it compiles methods for the set of concrete types it can infer. Practically speaking this mechanism can also be seen as a form of **monomorphizing** a set of functions for which the JIT compiler can determine the concrete types. The reason then why Julia's JIT compiler could be called 'terrible' would be that it, currently at least, does not implement profile-guided optimization (PGO). The lack of PGO and its reliance on type stable code for speed means that there is a correlation between idiomatically written Julia and its performance. With this understanding, it might now be apparent why my Ray-Tracer performed so poorly, as I was not fully aware of this strong dependence and I simply wrote the code how I would write something in any other dynamically typed high-level language.

We've now established that we have a piece of performance-critical software written in a very non-idiomatic and by extension inefficient manner. Furthermore, we've seen that its execution method is not only different from ahead-of-time (AOT) compiled languages but also from other JIT-compiled languages by its unique approach to generating machine code. The conventional route, and quite frankly recommended route, one should go here would be to simply rewrite everything with Julia's idiomatic principles in mind, though this would lead to a rather short exploration, so to question the alternative, what if we rewrote the most performance-critical segment of our ray tracer in a language which has less reliance on a unique idiomatic programming style. While at first this seems somewhat nonsensical it has the potential to raise and in turn answer some interesting questions about interoperability and more generally code execution methods including but not limited to:

- How do we design a system of communication between languages that exhibit considerable fundamental differences?
- How do we reconcile different types of systems in language interoperability?
- What are the performance characteristics of language interoperability systems?

2 BACKGROUND & RELATED WORK

To further gain insights into this world of interoperability I thought it best to continue on with the idea I had of rewriting a component in different languages and then applying some system of interoperability. Decomposing this idea into concrete steps we can develop a rough design of how this study will be conducted.

(1) Literature review & Background
  (a) Review current landscape of language interoperability
  (b) Choose suitable libraries to explore our specific use case
(2) Implementation
  (a) Profile the Julia application to extract the most performance intensive piece of code
  (b) Rewrite this piece of code in different languages, preferably ones which have very different execution models
  (c) Write some type of glue layers utilizing the chosen libraries to enable this interoperability
(3) Analysis & Development
  (a) Understand how the type systems of these different languages interact, especially considering types beyond primitives
  (b) Profile the programs for all given languages to understand the performance impacts
    (i) Understand the underlying nature of the performance impacts and how they relate to differences in execution models
  (c) Understand how interop libraries are designed and enable communication between languages
    (i) Attempt to develop certain solutions for any limitations encountered

FFI is a generic term which describes any implementation which enables a piece of code written in one language to call functions written in another. As the name implies these implementations usually provide some interface; that is, abstractions; which enable a programmer to make calls into some binary compiled into some target language differing from the source. In practice FFI can be thought of as existing on two layers, the Application Binary Interface (ABI) level and the Application Programming Interface (API) level. The ABI level describes some common standard for how to compile source code into machine code. The API level then describes the corresponding high-level counterpart which comes in the form of libraries which expose means of working with this underlying shared ABI as to enable communication in a shared low level standard. By far the most common ABI which many languages can compile to and by extension have interoperability with is the C ABI. One slight tangent, naturally when we talk about an ABI we are referring to machine level concepts, which is to say that while its become the de facto standard to say 'C ABI' the underlying standard's we are talking about are the platform specific ABI standards which C compilers correspond to. It's just more straightforward to say there is a universal 'C ABI' as superficially this is more or less accurate.

```c
// mylib.c
#include <stdio.h>

void greet(const char* name) {
    printf("Hello, %s!\n", name);
}
```

```julia
ccall((:greet, "mylib"), Cvoid, (Cstring,), "World")
```

```rust
#[link(name = "mylib")]
extern "C" { fn greet(name: *const c_char); }

fn main() {
    let name = "World";
    let c_name = CString::new(name).expect("CString::new failed");
    unsafe {
        greet(c_name.as_ptr());
    }
}
```

```python
mylib = ctypes.CDLL('./libmylib.so')

mylib.greet.argtypes = [ctypes.c_char_p]
mylib.greet.restype = None

name = b"World"
mylib.greet(name)
```

One critical thing to observe is in all instance we must ensure the FFI boundary method and its arguments conform to the memory layout of the target language as specified by the ABI. Since string types for example generally have different underlying representations depending on the language here we must ensure the string has the form of a C-string, that is its a basic pointer to an array of characters *char**.

This in turn limits FFI boundary methods by the syntax and underlying ABI of our target language. For example, as C has no generics, we cannot utilize a generic method as an FFI boundary despite generics existing in our source language and Rust being compatible with the C ABI.

```rust
extern "C" {
    fn add<T>(a: T, b: T) -> T;
}
```

Stated differently FFI boundary method signatures must mirror those of the target language. We can create some flexibility here by abstracting the boundary utilizing wrappers; commonly macros; which make defining the boundary closer to that of the source language while implicitly ensuring that the actual underlying boundary is valid, an example of this is the `@ccall` macro in Julia.

But this doesn't really address the underlying issue that our FFI boundary is limited by the target language.

2.1 Application Binary Interface (ABI)

ABI stability is a key factor to understanding the current landscape of not only interoperability between languages but also within different versions of languages themselves. As the name implies the stability of an ABI refers to its specification, that being how a compiler generates machine code, being stable or unchanging. An

ABI being 'fully stable' implies that all aspects of its specifications will not change with any future modification of its execution system (JIT compiler, AOT compiler, etc.). One of the primary reasons for so many languages having interop with C is due to the fact that it has full ABI stability, thus there exist essentially a promise that a compiler can always generate the same machine code for a given source language that conforms to the defined C ABI standard knowing this will never some day break due to changes in C's calling conventions, memory layout, etc.

The other side of this is ABI instability, most languages currently have what could be considered partial ABI stability. What this means in practice is that most languages have some execution system which generates machine code, commonly this version or a set of versions have a common ABI. Within this set of versions there exists stability, meaning shared library calls; which are reliant on ABI stability; can function with any other languages capable of producing machine code which conforms to this ABI. Though across different ABI's this communication breaks down as we no longer have a homogenous standard generating the machine code but conflicting and often incompatible standards.

This nature of partial stability in the ABI's of most languages creates one of the primary limitations to language interoperability with FFI, due to the rather complex and to an extent controversial nature of having an ABI which is both fully specified and consistent across versions there are not alot of languages which have this and by extension not alot of compilers which chose to implement an ABI for this. Furthermore there is the point that if for some set of languages there is an existing C ABI already implemented, this in turn implies we have communication between these two languages with the common language taking the form of the C ABI. It should be clarified that while interop in many languages occurs with the C ABI as the basic common ground.

2.2 C-ABI and Interop in practice

To describe how interop utilizing the C ABI actually plays out in practice lets express two arbitrary languages as L_1 and L_2. In the case of primitives the generated machine code usually produces matching layouts, thus little additional work is involved beyond defining the FFI boundaries in both languages. As we start getting to more complex types we start wanting to pass more complex things across the FFI boundary that might not have as straightforward of a representation in C we run into the first issue with pure C ABI based interop. This being that it leads to considerable manual specification and redefining languages specific syntax, otherwise called marshalling or boxing through the FFI boundary and in turn unmarshalling in the target language. Additionally we must ensure that the representation of our type in L_1 matches that in L_2 which for a large library could mean having to manually specific all types such that we establish a mirrored relationship; while this is somewhat obvious that it has to happen the manual nature of it would be less then desirable in the case of very large libraries we wish to interop with.

To address these issues of having to manually box everything such that it is compatible with the C ABI various language features and libraries exist as abstractions to enable more seamless and simplified interop between languages. The common features with these interoperability abstraction systems are usually automatic binding generation through parsing and implicit boxing and unboxing of complex types that cant be trivially passed through the FFI boundary. Examples of this are Rusts CXX library or the way Swift implemented interop as a direct part of its compiler.

Through these abstractions interop between many languages has become a mostly trivial task, in that most of the manual activities are now generally automated in one way or another, though as stated before due to the fact that this is all ultimately still working through the C-ABI (or some slight modification of it) we are still very much limited in the things we can actually pass between languages. Though these limitations are generally not large enough to really discourage the usage of FFI as the primary means of interop.

So to briefly summarize we can see that FFI generally relies on a common denominator ABI between languages; often the C ABI; and a set of tools to automate any manual boxing and unboxing such that the developer can; for the most part; simply define a boundary in both languages and use said boundary as if it were a regular call. An important final consideration with these FFI systems is how optimization works across language boundaries. Since the methods of shared libraries are only resolved at link time no optimizations except LTO can occur. The shared object file from the target language has optimizations applied from its own compiler then depending on if the compiler has Link Time Optimization (LTO) implemented we can optimize the whole program across the language boundary, otherwise no optimizations occurs across the language boundary.

2.3 WebAssembly (WASM)

WASM offers an interesting insights into how language interop is developing on the web. First announced in 2015 and then released in 2017 WASM is a byte-code format designed to be executed within web-browsers using stack-based Virtual Machines (VM) though has now also been implemented to run both via AOT and JIT compilers. The WASM specification was designed by the World Wide Web Consortium (W3C) but has various compiler implementations by different groups. WASM when used as a compilation target for other languages not only allows potentially arbitrary languages to run on the web, but through its ABI also allows for another common denominator of communication. To understand the state of its ABI we have to examine the different compiler toolchains that target WASM. Currently the major ones are

What we can understand from this is that it seems WASI will be the ABI to which wasi-wasm will be the likely target most compilers aim to ultimately achieve, furthermore WASI seems to aim for stability with its canonical ABI, which could suggest that we might see a new common standard of interoperability, especially as wasm compilers evolve. One distinguishing feature of specifically WASI's canonical ABI is development of the **component model**, the component model refers to type definitions written in *WASM Interface Type* (WIT), which is an interface description language (IDL) used to express interface; that is, types and functions; in a language agnostic manner[4]. From the defined WIT code we can generate the corresponding bindings in our language thus mitigating the need for any type-shenanigan's in the case of interop since our generated bindings will by default be compatible with the ABI.

Compiler/Toolchain	Description
Emscripten	This was the first project to define a POSIX compliant ABI for WASM, though as of 2020 at least it is not stable.
Cheerp	Seem to support WASI-WASM as a target for the compiler. Inherently has interop with JS because it literally compiles to JS uses ts2cpp for creating C++ interfaces from TS types.
WASI	A subgroup of the WASM community group which is aiming to define a set of APIs and a canonical WASM ABI, currently the actual implementation of this specification comes in the form of: • Wasienv[1] which can compile Swift and C++ to wasm conforming to the WASI standard. • wasi-sdk[6] which seems to be some more low level compiler for wasi, i think the like backend, so you just need to implement the frontend for specific languages. • wasmtime[5] another wasm runtime which also has WASI as a target specification it seems. • wasix[7] compatibility layer to make WASI POSIX compliant when built with the WASI SDK. • wasm32-wasi[3] seems to be a frontend for Rust again using WASI. WASI with the component model seems to have the most thoroughly documented ABI.
GoJS	Not much info on this, just seems that a go compiler[2] has wasm as a target.

Table 1: Major WASM Compiler Toolchains and Their Descriptions

It should be noted that as the documentation specifies WIT implements contracts, that is similar to how an *extern* block expresses a function which must be ABI compatible so to do the generated bindings from the WIT code all inherently imply compatability with the underlying ABI.

Thus if you have say some library compiled to wasi-wasm which utilizes the component model to specify the API you can in turn use the library specification to generate ABI compliant bindings for your source code, then use a wasi-wasm compiler and have full cross-language interoperability without the need for any complex wrapper libraries. Though once again here we are limited by the expressiveness of the component model and corresponding ABI, meaning that if a library wants to make something available for use externally it has to conform to the component model thus potentially limiting certain language specific features for a cross language setting. Unless similarly to the C ABI we adjust the compiler to support these language specific features and in turn break compatibility with the ABI.

2.4 Binding generator designs

It should be clarified that in the previous instances of interop I for the most part avoided mentioning the exact details of how different type systems are mapped between languages. In the context of FFI based interop there are two key considerations in regards to types one must consider, that being the kind of type we are bridging and the parameter-passing strategy.

To talk about the parameter passing strategy we have to first talk about how types are implemented to begin with. Usually the choice of parameter-passing is closely tied to the underlying execution method of a language. Managed languages and especially high level ones generally make the choice of implementing types through boxing which in turn means they implement the call by sharing parameter passing strategy; which means that types which are cheap to copy are passed by value (boxed or not) and types which arent are passed in a way where reassignments do not affect the caller but mutations are reflected in the caller. Languages generally termed 'safe' which implement some form of memory usually implement both call by value and call by reference leaving it up to the user to decide which to use for what type through certain operator choices. Finally languages which do allow memory management as a native part also usually expose pass by address where you pass a pointer which you can then dereference to use in the calling function.

A common pattern we can spot with the design of languages is the approach that primitives; regardless of implementation; are copied, and usually user-defined types or just anything which would be unreasonably to copy can either be passed by reference by the programmer or is implicitly done so as the implemented parameter-passing strategy. One notable strategy which has gained particular notiaraiety in recent years in the case of pass by value is if you move the type by copying or by destroying it. Rust; similar to most other languages copies small stack allocated types; but in the case of anything heap allocated as a part of the rust borrow checker such values are destructively moved. When we talk about a move as opposed to a copy the simple idea is usually just that as opposed to copying heap allocated data when we pass something by value we instead simply; through some mechanism; invalidate the source object and create a new object which points to this underlying heap allocated data. The destinction being that we in essence only ever create a new owner for the heap data as opposed to creating a new owner with their own copy of the data.

Currently the two prominant implementations of these dynamics come from C++ with move constructors and Rust with its borrow checker. The main distinguishing feature of these implementations is that in principle with C++ the moved-from object is nullified but left in a valid state such that a destructor can be called on it when it goes out of scope, whereas in Rust the old moved-from object is in an invalid state and would raise errors if used after a move, Swift has similarly implemented destructive move since v5.9 with the introduction of the `consume` operator. How long a particular variable exists from the point of creation to it being either destructively or non-destructively moved is generally referred to as the variables lifetime. As Rust has destructive move semantics as a central feature of its borrow-checker and does not use Garbage Collection (GC) it relies on either implicit (via elision) or explicit lifetime annotations as a means of conveying how long a specific variable persists for usage before being destroyed. To then brielfy

summarize, we end up with essentially 3 variants of how languages pass paramaters, we either copy, pass some type of reference (either with shared semantics or not), or we give ownership.

A good question then becomes how do we effectively implement parameter-passing in an interoperability setting. The most obvious starting point would be that; just as we want for programming languages; types which are cheap to copy should be copied. Most languages do this by defining usually some set of ABI (usually C) compatible primitive datatypes. As most languages implement a decent set of primitives in a roughly similar fashion passing these by value across a shared ABI is generally not too complex and usually forms the initial implementation of most FFI based interop systems. Once we move past basic stack allocated primitives we start to get into the realm of some complexity. A common type introduced to new programmers after primitives is the struct. Or more abstractly speaking the user defined type (UDT). UDT's can broadly be described as types which generally group some set of more fundamental types along with sets of operations that can be prefromed on these types. Both of these attributes of UDT's are usually implemented as some form of property of our type.

As FFI based interop generally concerns some low level intermediary an important question which has to be asked is how do different languages represent object properties at a low level. Enumerating all the different object representation models for all languages would be somewhat out of scope and meaningless so I will simply describe the abstract approaches which are generally taken. On a high level we can distinguish broadly between two types of objects; direct objects and indirect objects. Direct objects here refers to objects which are represented directly in memory and whos property access usually translate to basic pointer offset computations. Direct objects usually exist in most statically typed, AOT compiled languages generally denoted with the keyword `struct` or `class` as a convention. When it comes to bridging direct objects the aim is firstly that property accesses resolve to the same memory location and that the type of the property, or more specifically the underlying memory has some analogue in the target language.

In cases where the memory does not align we start to encounter different implementation approaches one can take. The most trivial approach would be to simply expose property getters across the FFI boundary and then handle the property types as necessary. Though if we take this approach at face value its simply not feasable especially for large scale API's in some foreign language we wish to use in a different target.

4 DISCUSSION

Here you put your results in context (possibly grouped by research question). Usually, this section focuses on analyzing the implications of the proposed work for current and future research and for practitioners.

5 CONCLUSION

Briefly summarize your contributions, and share a glimpse of the implications of this work for future research.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

REFERENCES

[1] S. Akbary, M. McCaskey, J. Kruth, *et al.*, *Wasienv/wasienv*, Jun. 6, 2023. [Online]. Available: https://github.com/wasienv/wasienv.

[2] Ayke, R. Evans, D. Gryski, *et al.*, *Tinygo-org/tinygo*, Nov. 14, 2024. [Online]. Available: https://github.com/tinygo-org/tinygo.

[3] Bytecode Alliance, *Cargo-wasi: A lightweight cargo subcommand for the wasm32-wasi target*, Provides tools for building, running, testing, and benchmarking Rust code for the wasm32-wasi target. Licensed under Apache-2.0 with LLVM exception., 2024. [Online]. Available: https://github.com/bytecodealliance/cargo-wasi.

[4] Bytecode Alliance, *The webassembly component model*, https://component-model.bytecodealliance.org/, Accessed: 2024-11-15, 2024. [Online]. Available: https://component-model.bytecodealliance.org/.

[5] Bytecode Alliance, *Wasmtime documentation*, Wasmtime is a standalone runtime for WebAssembly, WASI, and the Component Model. It is designed to be fast, secure, configurable, and standards compliant, and can run WebAssembly outside of the web environment., 2024. [Online]. Available: https://github.com/bytecodealliance/wasmtime.

[6] D. Gohman, A. Crichton, A. Brown, *et al.*, *Webassembly/wasi-sdk*, Oct. 29, 2024. [Online]. Available: https://github.com/WebAssembly/wasi-sdk.

[7] SingleStore Labs, *Wasix: Posix compatibility layer for wasi builds*, Provides stubs for POSIX compatibility in WASI builds, facilitating the compilation of sources expecting POSIX support. Designed for experimental use., 2024. [Online]. Available: https://github.com/singlestore-labs/wasix.