# All the Binaries Together: A Semantic Approach to Application Binary Interfaces

Andrew Wagner
Northeastern University
Boston, MA, USA
ahwagner@ccs.neu.edu

Amal Ahmed
Northeastern University
Boston, MA, USA
amal@ccs.neu.edu

## 1 Introduction

Nearly all modern systems critically depend on interoperability between languages and libraries, but reasoning formally about this interaction has proven to be a serious challenge, as different components maintain drastically different invariants. Recent efforts [13, 14] to tackle the interoperability problem make use of advanced type systems that relate types or behaviors in one language to those in another.

But as we descend down toward the binary layer, the sophisticated type machinery slowly disappears, and all that remains is an unspoken promise between components. This promise is the application binary interface (ABI), which specifies data layouts, calling conventions, and other low-level details required for interaction. But while type systems have grown richer, ABIs have largely remained the same, lacking analogous advances in expressivity and safety guarantees.

However, there is a growing demand for change among newer systems languages that are aiming to gradually replace C. For example, the Swift team has adopted an "ABI Stability Manifesto" [3], which proposes a safe but flexible ABI with dynamic overhead, a decidedly different approach than that of C. Likewise, the Rust team has recently drafted an RFC for "crABI" [16], a higher-level ABI that will provide support for types like references and strings while remaining compatible with C's ABI. As the developer community looks to build these new ABIs, we aim to provide guidance on how to formally specify them and design them to be safe.

## 2 The Approach

In this section, we sketch out our proposed approach and a handful of potential applications over an abstract source language $L_S$ and an abstract target platform $L_T$. The goal of an ABI is to specify precisely the protocol for interacting with a target component $C$ according to a source signature $\Sigma$ (e.g., a header or manifest file). We capture this semantically using a *realizability model*:

**Definition 1** (ABI)**.** An *application binary interface (ABI)* $\mathcal{A}$ is a predicate on target components $C$ that is indexed by source signatures $\Sigma$.

**Definition 2** (Compliance)**.** A component $C$ is *ABI compliant* with $\Sigma$ if $\mathcal{A}[\![\Sigma]\!](C)$.

*PriSC, January 20, 2024, London, United Kingdom*
2024.

The idea of using realizability models to relate high-level types to low-level code is hardly new [6, 7], but here we are treating the *model itself* as an independent artifact, not just a proof device. Usually, one will first be interested in establishing properties of the ABI itself; e.g., that if $\mathcal{A}[\![\Sigma]\!](C)$, then calling any function in $C$ will behave according to its type in $\Sigma$. But perhaps more interesting are properties about *clients* of the ABI (e.g., different compilers, FFIs, shared libraries).

One such property is compliant compilation. A key benefit of a stable semantic ABI is that it canonizes the meaning of source types, so components originating from different compliant compilers can be soundly linked together.

**Application 1** (Compliant Compilation)**.** A compiler $\bullet^+$ is *ABI compliant* if $\vdash C : \Sigma$ implies $\mathcal{A}[\![\Sigma]\!](C^+)$.

This benefit can also be a drawback, since locking into a stable ABI can signifcantly limit the degrees of freedom for compiler writers; e.g., it might disallow niche representation optimizations that could help performance. To mitigate this, compliance is an *extensional property* that is only required at the *boundary* between components: within a component, a compiler might use whatever internal, unstable, ABI it wishes, but it assumes and guarantees complaince at the boundary (i.e., on public imports and exports). This idea need not only apply to compilers from the same language; it can also be used to link code compiled from entirely different languages:

**Application 2** (FFI)**.** A *foreign function interface (FFI)* for a language $L_{S'}$ to language $L_S$ consists of a type translation $\bullet^*$ such that

- whenever one exports a component $C$ at $\Sigma$, the compiler must guarantee that $\mathcal{A}[\![\Sigma^*]\!](C^+)$; and
- whenever one imports a component $C$ at $\Sigma$, the compiler may assume $\mathcal{A}[\![\Sigma^*]\!](C)$.

Another place where stability is crucial is for core shared libraries, where forward and backward compatibility is of the utmost importance. But, just as for compiler writers, rigidity in the ABI can prevent upgrades to libraries, which can force library developers toward leaky abstractions; e.g., reserving space in a struct for future use [5]. Swift, on the other hand, aims to support "library evolution" [2], which permits a certain class of signature updates by specifying layouts to be "resilient" by default. For example, a "fragile"

layout for a struct might hardcode the offsets of its fields based on the declaration order, whereas a resilient one might store the offsets in a lookup table, allowing the declaration order to change without breaking ABI compatibility. With a semantic ABI, we can reason about resilience formally:

**Application 3** (Supported Evolution)**.** A signature $\Sigma$ *supports evolution* to $\Sigma'$ if $\mathcal{A}[\![\Sigma']\!](C)$ implies $\mathcal{A}[\![\Sigma]\!](C)$.

## 3 Case Study: Reference Counting

Our first application of this technique is is a case study that develops a standard functional language with a reference counting ABI over a C-like target. While we plan to explore using a lower-level platform (e.g., WebAssembly [10]) to remove C from the equation, as a first step, we take a cue from Rust's crABI proposal [16]: because so many languages and tools already support C's ABIs, there is practical value in building on top of it rather than outright replacing it. We consider a handful of variations of the ABI, but the core model dictates that all source values are boxed and reference-counted in the target. The model itself is specified in a separation logic with custom ghost state [1, 8, 11, 12].

The heart of the model is the *object predicate* $O$, which specifies how an object of a given type is laid out in memory and what logical resources it owns or shares. The simplest exmaple is the object predicate for integers $\mathbb{Z}$:

$$O[\![\mathbb{Z}]\!](\ell) \triangleq \exists\, n.\ \ell \mapsto n$$

An integer object is just a pointer to an integer in memory, where we use the points-to connective from separation logic to indicate ownership.

A slightly larger example is the object predicate for products $T_1 \times T_2$:

$$O[\![T_1 \times T_2]\!](\ell) \triangleq \exists\, \ell_1, \ell_2.\ \ell \mapsto \ell_1 \star \ell + 1 \mapsto \ell_2 \star \mathcal{R}[\![T_1]\!](\ell_1) \star \mathcal{R}[\![T_2]\!](\ell_2)$$

A product object is a pointer to two adjacent pointers in memory, where the first of these is a *reference* to a $T_1$, and likewise the second is a reference to a $T_2$.

As mentioned earlier, all values are boxed and reference-counted, so the *reference predicate* $\mathcal{R}$ is used pervasively:

$$\mathcal{R}[\![T]\!](\ell) \triangleq @_\ell\, O[\![T]\!](\ell + 1)$$

Physically, a reference is just a pointer to the cell holding the count, but logically it also confers shared permission to the object being reference-counted, which is stored in adjacent memory. We represent this with the *jump modality*[1] $@_\ell\ P$, where $\ell$ is the cell with the reference count and $P$ is the resource it protects. Logically, memory is organized into a graph of resources connected by references, and if we were to "jump across" the edge $\ell$ from the current resource, we would obtain a resource satisfying $P$. Each occurrence of the jump modality at a particular location makes a single contribution to the reference count. The effect that physically incrementing ($++\ell$) or decrementing ($--\ell$) the count has on logical resources can be seen in the following inference rules:

$$\textsc{rc-new}\ \frac{\{P \star @_\ell\, Q\}\ e\ \{R\}}{\{P \star \ell \mapsto 1 \star Q\}\ e\ \{R\}}$$

$$\textsc{rc-incr}\ \{@_\ell\ P\}\ ++\ell\ \{n.\ \ulcorner n > 1 \urcorner \star @_\ell\ P \star @_\ell\ P\}$$

$$\textsc{rc-decr}\ \{@_\ell\ P\}\ --\ell\ \{n.\ \ulcorner n > 0 \urcorner \vee (\ulcorner n = 0 \urcorner \star \ell \mapsto 0 \star P)\}$$

Using these rules to satisfy the counting requirements of the ABI is very reminiscent of working with reference-counted objects in Python's C API [9]: one needs to keep careful track of when a reference may be *stolen*—that is, the reference may be consumed—and when it must be *borrowed*—that is, the reference must be returned. As it relates to function calls, stealing a reference usually requires the caller to increment, whereas borrowing a reference usually requires the callee to increment. To eliminate ambiguity, the ABI dictates a particular *calling convention* for a function $T_1 \to T_2$: it steals its input reference $\mathcal{R}[\![T_1]\!]$ and returns a reference $\mathcal{R}[\![T_2]\!]$:

$$O[\![T_1 \to T_2]\!](\ell) \mathrel{\hat{\approx}} \exists\, f.\ \ell \mapsto f \star \forall\, \ell_1.\ \{\mathcal{R}[\![T_1]\!](\ell_1)\}\ f(\ell_1)\ \{\ell_2.\ \mathcal{R}[\![T_2]\!](\ell_2)\}$$

Roughly, a function object is a pointer to a function pointer that obeys the calling convention. The actual definition of $O[\![T_1 \to T_2]\!]$ is more involved: since we support closures, we must also specify how environments are stored, destroyed, and passed in the calling convention.

To exercise the model, we develop a compiler and prove that it is compliant. The compiler is inspired by the Perceus scheme [15]: internally, the type system tracks variable uses linearly, and explicit applications of contraction and weakening in the typing derivation cause the compiler to insert increment and decrement operations.

## 4 Conclusion

So far, we have formalized an ABI for the reference counting case study and we are in the process of proving compiler compliance. Next, we plan to formalize instances of library evolution and intra-language ABI migration (e.g., two compilers with different representation choices), as well as a foreign function interface to a different source language.

For future work, a natural avenue is scaling the languages of study. For platforms, we will explore using WebAssembly and its Component Model Proposal [17] as building blocks. For sources, Rust is certainly a top priority, as questions

[1] The name and notation is inspired by *hybrid logic* [4].

about its ABI are under active discussion [16], but formalizing the idiosyncracies of Swift's resilient ABI is another compelling option.

## References


[1] Andrew W Appel. 2014. *Program logics for certified compilers*. Cambridge University Press.

[2] Apple. 2015. Library Evolution. https://github.com/apple/swift/blob/main/docs/LibraryEvolution.rst.

[3] Apple. 2017. ABI Stability Manifesto. https://github.com/apple/swift/blob/main/docs/ABIStabilityManifesto.md.

[4] Carlos Areces and Balder ten Cate. 2007. 14 Hybrid logics. In *Studies in Logic and Practical Reasoning*. Vol. 3. Elsevier, 821–868.

[5] Aria Beingessner. 2022. C Isn't A Programming Language Anymore. https://faultlore.com/blah/c-isnt-a-language/#case-study-minidump_handle_data.

[6] Nick Benton and Nicolas Tabareau. 2009. Compiling functional types to relational specifications for low level imperative code. In *Proceedings of the 4th international workshop on Types in language design and implementation*. 3–14.

[7] Nick Benton and Uri Zarfaty. 2007. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 1–12.

[8] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 287–300.

[9] Python Software Foundation. 2023. Python/C API Reference Manual. https://docs.python.org/3/c-api/intro.html#reference-counts.

[10] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[11] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[12] Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 561–574.

[13] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 609–624.

[14] Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation using Linking Types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*. 14–28.

[15] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111.

[16] Rust Language RFCs. 2023. #3470: crABI v1. https://github.com/rust-lang/rfcs/pull/3470.

[17] WebAssembly. 2023. Component Model. https://github.com/WebAssembly/component-model.