Specifying ABIs with Realizability and Type-Preserving Compilation

Brianna Marshall
Northeastern University
Boston, MA, USA

Ryan Doenges
Boston College
Chestnut Hill, MA, USA

Owen Duckham
Northeastern University
Boston, MA, USA

Ari Prakash
Northeastern University
Boston, MA, USA

Maxime Legoupil
Aarhus University
Aarhus, Denmark

Elan Semenova
Northeastern University
Boston, MA, USA

Lars Birkedal
Aarhus University
Aarhus, Denmark

Amal Ahmed
Northeastern University
Boston, MA, USA

1 Background

Despite the widespread use of memory-safe languages in modern software development, C still reigns supreme when programs written in different languages interact. The C Application Binary Interface (ABI) is the lingua franca of language interoperation and is typically the only stable ABI in common between languages. However, the C ABI describes only the *layout* of data and says nothing of the *resources* owned by data. Because of this, users of Foreign Function Interfaces (FFIs) must take special care to avoid memory safety violations such as use after free.

ABIs are typically described with informal prose, which makes them difficult to enforce. Techniques to precisely formalize what an ABI means are an area of active research. Wagner et al. propose that realizability models—essentially, semantic models of source-language types that are inhabited by target-language computations—can be used to rigorously specify ABIs [7]. Target programs that inhabit the model are considered ABI compliant and can be safely linked at a particular source type. If the model is defined so that it not only specifies data layout and calling conventions, but also the use of resources—such as by using separation logic—then ABI compliance can precisely capture memory safety.

WebAssembly [4] is a promising platform for interop research because it has a formally defined semantics and enjoys widespread industrial use. Its type system guarantees sandboxing and module isolation, but sharing memory between modules requires granting unfettered access to the entire address space and the type system is not rich enough to enforce invariants about data in shared memory.

RichWasm [2] eschews WebAssembly's direct access to all of shared memory in favor of typed references—both manually managed and garbage collected—enabling *fine-grained* sharing of memory between modules. Its type system guarantees safe linking between RichWasm modules, but after RichWasm compiles to WebAssembly, there is no means to safely link with further WebAssembly modules. In other words, RichWasm cannot specify an ABI suitable for separate compilation and linking between WebAssembly modules.

PriSC, Rennes, France
2025.

2 ABIs For Free!

Language designers and compiler developers in need of an ABI find themselves caught between two extremes: an ABI can be specified informally with prose and validated by inspection and testing, or it can be specified rigorously with a realizability model and validated by proofs. While the latter addresses the soundness issues of the former, it may be unappealing to software developers because they must work directly with semantic models and program logics using theorem provers.

We propose a middle ground where we define a realizability model for RichWasm, indexed by RichWasm types and inhabited by WebAssembly computations—doing all the work that requires expertise in semantic models and program logics—and source-language compilers can benefit from the memory-safe ABI specified by the model by simply building a type-preserving compiler to RichWasm. To that end, we have redesigned RichWasm to facilitate separate compilation and a stable ABI. When needed for clarity, we refer to the version of RichWasm from [2] as RichWasm 1.0 and the version in this work as RichWasm 2.0.

Now, if there is a type-preserving compiler from a source language to RichWasm 2.0, the ABI for any source type τ is the interpretation $[\![\tau^+]\!]$ in our realizability model, where τ^+ is the type translation for τ. In other words, by writing a type-preserving compiler to RichWasm, a WebAssembly ABI falls out "for free!"

3 Memory and Representations

Our case study compiles core ML and the linear λ-calculus (LLC) with references to RichWasm 2.0. The syntax is standard except for the use of n-ary sums and products; ML includes polymorphism but LLC does not. This demonstrates the interaction between a language with garbage collection and boxed values (ML) and a language with manual memory management and unboxed values (LLC). We are in the process of validating examples.

RichWasm 2.0, in addition to having distinct *types* for Garbage-Collected (GC) and Manually-Managed (MM) references like RichWasm 1.0, also features a *kind* system which

Kinds	κ	::=	VALTYPE $\rho\ \chi\ \delta$ | MEMTYPE $\sigma\ \delta$
Reps	ρ	::=	r | $(+\ \rho^*)$ | $(\times\ \rho^*)$ | ι
Primitives	ι	::=	`ptr` | `i32` | `i64` | `f32` | `f64`
Sizes	σ	::=	s | $(+\ \sigma^*)$ | $(\times\ \sigma^*)$ | $\lvert\rho\rvert$ | n
Copyability	χ	::=	IMCOPY | EXCOPY | NOCOPY
Dropability	δ	::=	IMDROP | EXDROP
Types	τ	::=	t | coderef ϕ | ref $\mu\ \tau$ | $\exists\alpha.\ \tau$ | ...
Function Types	ϕ	::=	$\forall\alpha^*.\ \tau^* \to \tau^*$
Memory	μ	::=	m | gc | mm
Quantifiers	α	::=	m | r | s | $t{:}\kappa$

Figure 1. Syntax of RichWasm 2.0 kinds and select types.

can be used to encode both boxed and unboxed representations in the presence of polymorphism. This kind system is inspired by Downen et al. [1]. As a result, compilers to RichWasm should be both type preserving *and* kind preserving. Since neither ML nor LLC has a kind system, "kind preserving" means that the compilers always produce well-kinded RichWasm types.

Type variables in ML are compiled to RichWasm type variables constrained by the kind that includes garbage collected references, making ML's dependence on a uniform representation for polymorphism explicit in the ABI.

4 The RichWasm 2.0 Language

RichWasm 2.0's type system includes sums and products, references, function pointers, WebAssembly's numeric types, recursive and existential types, and polymorphism over memories, representations, sizes, and types (Figure 1).

The kind system partitions types into VALTYPE and MEMTYPE, reflecting WebAssembly's split type system. Because the operand stack contains typed values and memory contains uninterpreted bytes, VALTYPE is indexed by a representation ρ and MEMTYPE is indexed instead by a size σ.

Kinds are also indexed by a copyability χ and dropability δ, which propagate resource restrictions. GC references are explicitly copyable (EXCOPY) and MM references are not copyable (NOCOPY); both are explicitly droppable (EXDROP). Numeric types, lacking memory resources, are both implicitly copyable (IMCOPY) and droppable (IMDROP). A subkinding rule reflects the ordering IMCOPY $\preceq$ EXCOPY $\preceq$ NOCOPY and IMDROP $\preceq$ EXDROP, similar to System F° [5].

The kind translation for ML's uniform representation is VALTYPE `ptr` EXCOPY EXDROP; the type of GC references has this kind. Because LLC uses unboxed values, it does not use a uniform representation. Instead, its types have the kind VALTYPE ρ NOCOPY EXDROP for some representation ρ.

5 Compilation to WebAssembly

We implement a compiler from RichWasm 2.0 to WebAssembly in Rocq and are in the process of proving it to be ABI compliant. The compiler is both type-directed and kind-directed:

$$[\![\text{ref mm } \tau]\!]_\epsilon(\hat{v}) ::= \exists \ell, f^*, w^*.\ \hat{v} = [\mathbf{ptr}\ \text{mm}\ \ell] \star \ell \overset{\text{rwl}}{\longmapsto} f^* \star \ell \overset{\text{rwh}}{\longmapsto} w^* \star [\![\tau]\!]_\epsilon(w^*)$$

$$[\![\text{ref gc } \tau]\!]_\epsilon(\hat{v}) ::= \exists \ell, f^*.\ \hat{v} = [\mathbf{ptr}\ \text{gc}\ \ell] \star \boxed{\exists w^*.\ \ell \overset{\text{rwl}}{\longmapsto} f^* \star \ell \overset{\text{rwh}}{\longmapsto} w^* \star [\![\tau]\!]_\epsilon(w^*)}$$

Figure 2. Semantic interpretation of references.

types provide high-level structure while kinds guide low-level code generation. A RichWasm type corresponds to a sequence of WebAssembly types; indeed, a closed representation ρ corresponds to a sequence of primitives ι^*. RichWasm instructions may compile to repetitions of the corresponding WebAssembly instruction based on this sequence. The type system faciliates representation-directed code generation by requiring that types used as operands have a monomorphic representation, as in Downen et al. [1].

Runtime. Cycles between RichWasm's two heaps are possible because GC objects can own MM references and MM objects can point to GC objects. When the GC collects an object, it must recursively free any MM references it owns to avoid leaking memory. This requires objects to have a finalizer, so we cannot use the WasmGC proposal—now part of WebAssembly 3.0—to compile RichWasm GC references.

Instead, we rely on a custom GC that is part of the RichWasm runtime, which is itself a WebAssembly module. Since the GC runs within the WebAssembly sandbox, it cannot directly see roots in the operand stack, so roots must be explicitly registered. The compiler generates calls to runtime functions to register and unregister roots when needed. We specify the runtime interface but axiomatize its implementation.

Model. The realizability model is defined using Iris-Wasm [6]. Inspired by Melocoton [3], we use reachability to control the lifetime of heap objects, allowing the runtime to free both GC memory and MM memory owned by the GC. The resources owned by the interpretation of ref $\mu\ \tau$ (Figure 2) survive deallocation of ℓ.

Heap objects contain a pointer bitmap so that the GC can traverse the heap. The bitmap—a sequence of pointer flags f^*—agrees with the positions of pointers in the sequence of words w^*. GC references use an Iris invariant, enabling aliasing, while MM references do not, enabling strong updates.

6 Conclusion

RichWasm 2.0 empowers compiler developers to obtain a memory-safe WebAssembly ABI "for free." Our case study and the proof of ABI compliance of the RichWasm 2.0 compiler are in progress. In the future, we would like to extend the type system to capture additional memory ownership disciplines, such as the call stack and borrowing.

References

[1] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds Are Calling Conventions. *Proc. ACM Program. Lang.* 4, ICFP, Article 104 (Aug. 2020), 29 pages. doi:10.1145/3408986

[2] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly. *Proc. ACM Program. Lang.* 8, PLDI, Article 214 (June 2024), 24 pages. doi:10.1145/3656444

[3] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (Oct. 2023), 29 pages. doi:10.1145/3622823

[4] 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. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. doi:10.1145/3140587.3062363

[5] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/1708016.1708027

[6] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (June 2023), 25 pages. doi:10.1145/3591265

[7] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024), 30 pages. doi:10.1145/3689755