RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly

MICHAEL FITZGIBBONS*, Northeastern University, USA
ZOE PARASKEVOPOULOU*, Ethereum Foundation, Germany and Northeastern University, USA
NOBLE MUSHTAK, Northeastern University, USA
MICHELLE THALAKOTTUR, Northeastern University, USA
JOSE SULAIMAN MANZUR, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

Safe, shared-memory interoperability between languages with different type systems and memory-safety guarantees is an intricate problem as crossing language boundaries may result in memory-safety violations. In this paper, we present RichWasm, a novel richly typed intermediate language designed to serve as a compilation target for typed high-level languages with different memory-safety guarantees. RichWasm is based on WebAssembly and enables safe shared-memory interoperability by incorporating a variety of type features that support fine-grained memory ownership and sharing. RichWasm is rich enough to serve as a typed compilation target for both typed garbage-collected languages and languages with an ownership-based type system and manually managed memory. We demonstrate this by providing compilers from core ML and L^3, a type-safe language with strong updates, to RichWasm. RichWasm is compiled to regular Wasm, allowing for use in existing environments. We formalize RichWasm in Coq and prove type safety.

CCS Concepts: • **Software and its engineering → Formal language definitions**; *Assembly languages.*

Additional Key Words and Phrases: WebAssembly, memory ownership, memory sharing, capability types

ACM Reference Format:
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. https://doi.org/10.1145/3656444

1 INTRODUCTION

WebAssembly [9] (Wasm) is a portable binary format that enables almost-native execution speed for a variety of languages. With around 40 languages compiling to Wasm, the WebAssembly platform has huge potential to serve as a platform for language interoperability. This potential has been recognized for some time, but there are two impediments. The first is that Wasm takes an all-or-nothing approach to sharing memory. Each Wasm module has its own memory. If it wants to share a data structure in its memory with another module, that effectively leaves all of its memory

*Both authors contributed equally to this research.

Authors' addresses: Michael Fitzgibbons, Northeastern University, Boston, USA, michaelfitzgibbons7@gmail.com; Zoe Paraskevopoulou, Ethereum Foundation, Berlin, Germany and Northeastern University, Boston, USA, zoe.paraskevopoulou@gmail.com; Noble Mushtak, Northeastern University, Boston, USA, noble.mushtak@gmail.com; Michelle Thalakottur, Northeastern University, Boston, USA, thalakottur.m@northeastern.edu; Jose Sulaiman Manzur, Northeastern University, Boston, USA, sulaimanmanzur.j@northeastern.edu; Amal Ahmed, Northeastern University, Boston, USA, amal@ccs.neu.edu.

ACM 2475-1421/2024/6-ART214
https://doi.org/10.1145/3656444

exposed to the other module, potentially allowing adversarial code to access and modify arbitrary parts of that memory via pointer arithmetic. The multiple memories proposal [7] addresses this by allowing modules to have multiple independent memories and only share some of them while keeping others private. This mechanism is too coarse-grained, in both a spatial and temporal sense: it does not allow fine-grained sharing of only one data structure in a memory or sharing for only a while and then allowing access to be revoked.

The second impediment is that Wasm has only low-level types (32- and 64-bit integers and floats), so when modules from different languages need to communicate, there is a question of how to exchange high-level values. The Interface Types proposal [6] aimed to address this by introducing a set of high-level "interface types" (e.g., char, list, record, variant, and more can be encoded) that can be used to communicate across modules. However, the Interface Types proposal and its successor, the Component Model proposal [4], support only "shared-nothing" interoperability, requiring that values be copied across language boundaries when necessary. This ensures memory safety but comes with runtime overhead for copying.

In this paper, we propose RichWasm, a richly typed intermediate language (IL) based on WebAssembly. Unlike existing proposals in the Wasm ecosystem, RichWasm supports safe, fine-grained, shared-memory interoperability across Wasm modules, and it is also motivated by a longer-term desire to provide a platform atop which language implementors can easily develop safe FFIs. A particular goal is to enable statically detecting memory safety violations that happen across source-language boundaries and thus cannot be detected by the source type system (though this does not preclude FFI designers from adding constructs that defer interop errors to runtime). For instance, if we mix a type-safe garbage-collected language such as OCaml with a type-safe non-GC'd language—e.g., Rust or L^3 [13, 2]—and allow them to share memory, violations may include the Rust-like language freeing some garbage-collected memory passed to it from OCaml, or OCaml copying a mutable reference from the Rust-like language that the latter considers uniquely owned. We wish to allow source-language designers to add inter-language communication through a simple foreign-function interface that requires no changes to the existing source type systems and minimal changes to the syntax of their languages. Then the source-language modules are compiled separately to RichWasm modules. Typechecking at the RichWasm level can catch potentially problematic interactions between modules, guaranteeing cross-module type and memory safety.

Overview of RichWasm. At the core of RichWasm's type system are capability types in the style of L^3 [13, 2], which is a linear language with locations and safe strong updates (updates that change the type of a location). A key idea in L^3 was that when we allocate a new reference and initialize it with a value of type τ, we get back an existential package $\exists \rho.!\text{Ptr}\ \rho \otimes \text{Cap}\ \rho\ \tau$ that says that there exists some location ρ and we now have an unrestricted (copyable) pointer to that location and we have a linear capability that tells us the type of value currently stored at that location. This linear capability is essentially an ownership token required to access the reference — the capability must be provided to read from, write to, or free the location. In RichWasm, we similarly have *linear* capabilities that represent (allow access to) uniquely owned memory and support strong updates. But we also have *unrestricted* capabilities, which represent (allow access to) garbage-collected memory and are analogous to ML references in that they only support type-preserving updates. Unlike L^3, capability types in RichWasm can provide either read-only access or read-write access.

Formally, RichWasm has a substructural type system, realized via two qualifiers, linear and unrestricted, that annotate pretypes. In addition to capability pretypes, RichWasm supports polymorphism, recursive types, variants, structs, arrays and existential types. Unlike L^3, which assumes a reference cell can hold a value of any size, RichWasm has a low-level memory model like Wasm's, where each memory is simply a sequence of bytes and we must allocate data structures in the "flat"

memory. [1] This leads to an important novelty in the type system, which must keep track of the size of memory slots and disallow strong updates that attempt to store a larger value in that slot. Hence, capability types in RichWasm track the size of the memory slot originally allocated and type variables α must be annotated with a size bound that indicates the maximum size of the type that α can be instantiated with.

We implement compilers from core ML and L^3 to RichWasm, demonstrating that RichWasm can serve as a typed compilation target for both typed garbage-collected languages and languages with an ownership-based type system and manually managed memory. We also compile RichWasm to Wasm, providing a pathway to realistic use in many environments. We have formalized RichWasm in Coq and we have proved type safety (progress and preservation) for the type system.

Situating RichWasm. Before we dive into the details of RichWasm, we would like to make clear where it sits relative to two other pieces of related work. The first is the Wasm Component Model, though we've already discussed the salient difference above, namely "shared-nothing" vs. "fine-grained shared-memory" interoperability. But there are additional similarities and differences. The Component Model is intended to be a part of the WebAssembly ecosystem but not part of the core Wasm spec and the same is true of RichWasm. The Component Model provides a means to organize Wasm modules into components and instrument core Wasm modules so they can take advantage of higher-level types when communicating across components, while RichWasm supports higher-level types and then compiles (or lowers) them to Wasm. On the other hand, while the Component Model implementation employs dynamic techniques to assure safety when it comes to features like resources and handles, RichWasm uses static enforcement.

The second is recent work on safe FFIs by Patterson et al. [20] who proposed a framework for design and verification of safe FFIs. Their main insight is that language designers should build a model of source-level types as sets of target-level terms. Then for all conversions their FFI permits, from a type in one language say τ_A to a type in the other say τ_B, the target-level glue code they write to implement that conversion can be shown to be sound if: given target code that behaves like τ_A, the conversion produces target code that behaves like τ_B. This is a perfectly reasonable recipe for designing and verifying safe FFIs between languages that compile directly to Wasm. However, we would argue that it is a rather heavyweight recipe, one that requires FFI designers to know how to construct semantic models that would then guide their thinking. With RichWasm, we would like to provide support for compiler writers who don't know how to build semantic models and aren't interested in doing formal verification of FFIs. To that end, we've developed a richly typed IR capable of detecting unsafe interoperability via type checking of compiled code. To take advantage of this, compiler writers must implement type-preserving compilers to RichWasm, a much easier task than defining a target-level model of source-language types. Whenever the language designer wants to support additional FFI functionality, they simply have to extend their compiler and see if any additional conversions they allow result in type-checking errors at the RichWasm level.

Contributions. Our central contribution is RichWasm, a typed IL built on top of WebAssembly that is designed to serve as a useful platform for safe FFI design.[2]

- RichWasm supports an advanced substructural type system with capabilities and size tracking that enables static assurance of safe, fine-grained shared memory interoperability in a language with a low-level memory model (i.e., "flat" memory). The type system allows precisely tracking memory ownership and sharing, and avoids memory safety violations even when sharing memory across languages with garbage collection and manual memory management (§2 and §3).

[1] In WebAssembly-speak, what we refer to as "flat" memory, i.e., memory that's a sequence of bytes, is called "linear" memory.
[2] Our technical artifact is available at https://github.com/RichWasm/RichWasm-artifact

- We have formalized RichWasm in Coq and proved type safety (about 75k lines of Coq).
- We have type-preserving compilers from core ML and L^3 to RichWasm and a *simple* FFI between them that allows us to compile interoperating programs to RichWasm (§5).
- We have a compiler from RichWasm to WebAssembly that allows us to run RichWasm programs in all hosts of WebAssembly (§6).

Limitations and Future Work. This paper describes the RichWasm Minimum Viable Product (MVP), which is intended to demonstrate that it can serve as a platform for fine-grained safe interoperability between a type-safe GC'd language with aliasing and a type-safe language with manual memory management and type-enforced memory ownership. The RichWasm MVP has many feature limitations that require further research. In particular, it cannot serve as a target for type-preserving compilation of Rust since the type system is not expressive enough to encode Rust's borrowing and lifetimes. In §8, we discuss future work on extensions of the RichWasm MVP to support source languages with borrowing, concurrency, effect handlers, etc. We also leave the full-fledged design and evaluation of realistic FFIs on top of RichWasm to future work (§8). Each real FFI design ought to involve an investigation of flexibility/performance tradeoffs, along with an evaluation of usabliity.

2 RICHWASM OVERVIEW

In this section, we give an overview of the RichWasm language, presenting its types and syntax (§2.1). Then we sketch a simple FFI between L^3 and ML and show an example of how RichWasm can statically detect unsafe interop between them (§2.2).

First, consider a small example to get a sense for the sort of errors RichWasm can catch. In Fig. 1 we have a GC'd program that provides two functions: an identity function on integer references, which stashes a copy of the reference, and a function which returns the stashed copy. Our manually managed program first creates a reference, passes it to the `stash` function, and frees the returned reference. Next, the linear program retrieves the stashed reference and attempts to free it, resulting in a double free. If compiled naively, RichWasm's type system will first complain that `stash` requires an unrestricted reference, but is called with a linear reference. If `stash` were compiled to take a linear reference, RichWasm would not admit the function since it duplicates a linear value.

```
let c = ref (ref 0) in
fun stash (r : Ref Int) = c := r; r in
fun get_stashed (() : Unit) = !c
```

(a) Garbage-collected program

```
free (ml.stash (new 42));
free (ml.get_stashed ()) (* CRASH *)
```

(b) Manually managed program

Fig. 1. Unsafe interoperability

RichWasm and Wasm. While RichWasm is coherent without comparison to Wasm, readers who are familiar with Wasm will note the parallel structure of the two languages. Existing constructs from Wasm are extended to support RichWasm's new types, while continuing to fulfill their original purpose. For instance, Wasm has "local variables": a location which lives for the duration of a function call and can store one numeric type. RichWasm has an analagous concept. However, RichWasm also has strong tools for reasoning about sizes and linearity. This allows us to use locals even more effectively, strongly updating them while guaranteeing that there will be space for any value we store and that we only duplicate or ignore values which are not linear.

The largest departure from Wasm is in RichWasm's treatment of memory. Because we want to support strong memory invariants, RichWasm supports a series of structured memory types, rather than the raw sequence of numeric types present in Wasm. Wasm's unfettered access to the memory makes it impossible to maintain any invariants about one's memory layout when linking with other code. We no longer support arbitrary memory operations, but thanks to the same size and

linearity reasoning tools that give us greater control over locals, we retain control over memory layouts and sharing, without the burden of losing invariants when linking with other code.

2.1 RichWasm Syntax and Types

In Fig. 2 we give a full account of RichWasm types and programs. The highlighted constructs are new in RichWasm and not present in Wasm. We start explaining the syntax of the language from the top-level structures.

Types

	i	$\in$	$\mathbb{N}$
pretypes	p	::=	**unit** | np | (τ^*) | **ref** $\pi\ \ell\ \psi$ | **ptr** ℓ | **cap** $\pi\ \ell\ \psi$ | **rec** $q \preceq \alpha.\ \tau$ | $\exists \rho.\ \tau$ | **coderef** χ | **own** ℓ | α
numeric pretypes	np	::=	**ui32** | **ui64** | **i32** | **i64** | **f32** | **f64**
types	τ	::=	p^q
qualifiers	q	::=	δ | **unr** | **lin**
memory privilege	π	::=	**rw** | **r**
heap types	ψ	::=	$(\textbf{variant}\ \tau^*)$ | $(\textbf{struct}\ (\tau,\ sz)^*)$ | $(\textbf{array}\ \tau)$ | $(\exists\ q \preceq \alpha \lesssim sz.\ \tau)$
locations	ℓ	::=	ρ | $i_{\textbf{unr}}$ | $i_{\textbf{lin}}$
quantifiers	κ	::=	ρ | $sz^* \leq \sigma \leq sz^*$ | $q^* \preceq \delta \preceq q^*$ | $q \preceq \alpha^{(c^?)} \lesssim sz$
indices	z	::=	ℓ | sz | q | p
arrow types	tf	::=	${\tau_1}^* \to {\tau_2}^*$
function types	χ	::=	$\forall \kappa^*.\ {\tau_1}^* \to {\tau_2}^*$
sizes	sz	::=	σ | $sz + sz$ | i

Terms

heap values	hv	::=	$(\textbf{variant}\ i\ v)$ | $(\textbf{struct}\ v^*)$ | $(\textbf{array}\ i\ v^*)$ | $(\textbf{pack}\ p\ v\ \psi)$
values	v	::=	$()$ | $np.\textbf{const}\ c$ | (v^*) | **ref** ℓ | **ptr** ℓ | **cap** | **fold** v | **mempack** $\ell\ v$ | **coderef** $i\ j\ z^*$ | **own**
instructions	e	::=	v | $np.\textbf{unopt}_{np}$ | $np.\textbf{binop}_{np}$ | $np.\textbf{testop}_{np}$ | $np.\textbf{cvtop}\ np'$ | **unreachable** | **nop** | **drop** | **select** | **block** $tf\ (i,\tau)^*\ e^*$ **end** | **loop** $tf\ e^*$ **end** | **if** $tf\ (i,\tau)^*\ e^*$ **else** e^* **end** | **br** i | **br_if** i | **br_table** $i^*\ j$ | **get_local** $i\ q$ | **set_local** i | **tee_local** i | **get_global** i | **set_global** i | **qualify** q | **return** | **coderef** i | **inst** z^* | **call_indirect** | **call** $i\ z^*$ | **rec.fold** p | **rec.unfold** | **mem.pack** ℓ | **mem.unpack** $tf\ (i,\tau)^*\ \rho.\ e^*$ **seq.group** $i\ q$ | **seq.ungroup** | **cap.split** | **cap.join** | **ref.demote** | **ref.split** | **ref.join** | **struct.malloc** $sz^*\ q$ | **struct.free** | **struct.get** i | **struct.set** i | **struct.swap** i | **variant.malloc** $i\ \tau^*\ q$ | **variant.case** $q\ \psi\ tf\ (i,\tau)^*\ (e^*)^*$ **end** | **array.malloc** q | **array.get** | **array.set** | **array.free** | **exist.pack** $p\ \psi\ q$ | **exist.unpack** $q\ \psi\ tf\ (i,\tau)^*\ p\ e^*$ **end** |

$unop_{\text{iN}}$	::=	**clz** | **cnt** | **popcnt**	
$binop_{\text{iN}}$	::=	**add** | **sub** | **mul** | **div** sx | **rem** sx | **and** | **or** | **xor** | **shl** | **shr** | **rotl** | **rotr**	
$unop_{\text{fN}}$	::=	**abs** | **neq** | **sqrt** | **ceil** | **floor** | **trunc** | **nearest**	
$binop_{\text{fN}}$	::=	**add** | **sub** | **mul** | **div** | **min** | **max** | **copysign**	
$testop_{\text{iN}}$	::=	**eqz**	
$relop_{\text{iN}}$	::=	**eq** | **ne** | **lt** sx | **gt** sx | **le** sx | **ge** sx	
$relop_{\text{fN}}$	::=	**eq** | **ne** | **lt** | **gt** | **le** | **ge**	
cvtop	::=	**convert** | **reinterpret**	

Top-level declarations

functions	f	::=	$ex^*\ \textbf{function}\ \chi\ \textbf{local}\ sz^*\ e^*$ | $ex^*\ \textbf{function}\ im$
globals	$glob$	::=	$ex^*\ \textbf{glob mut}^?\ p\ i^*$ | $ex^*\ \textbf{glob}\ im$
table	tab	::=	$ex^*\ \textbf{table}\ i^*$ | $ex^*\ \textbf{table}\ im$
exports	ex	::=	**export** "*name*"
imports	im	::=	**import** "*name*"
modules	m	::=	**module** $f^*\ glob^*\ tab$

Fig. 2. RichWasm abstract syntax

Modules. A top-level RichWasm program is a *module*. A module consists of code in the form of a list of *function*s and data in the form of a list of *global* declarations that can be mutable or immutable. Each module also has a *table* that, as in WebAssembly, stores references to functions which facilitate indirect calls. Functions, globals, and tables can be exported, making them visible to other modules for import.

Functions. Functions are sequences of RichWasm instructions, taking as input a sequence of values and returning a sequence of values. The function type, χ, can be polymorphic over various type-level entities: memory locations, sizes, qualifiers (i.e., linearity annotations), and types. Functions

define their list of local variable using their sizes (sz^*) which will be initialized with an unrestricted unit value. They are not tied to a particular type and their type may change during evaluation.

Value Types. The types of values consist of a *pretype* that is annotated with a qualifier. Concrete qualifiers can be linear (**lin**) or unrestricted (**unr**) and indicate whether a value must be treated linearly. Qualifiers can also be abstract variables bound in function quantification. Qualifiers are ordered with $\textbf{unr} \preceq \textbf{lin}$. This ordering helps us put restrictions on abstract types bound in polymorphic, existential, or recursive types.

Simple pretypes include unit, numeric types (32 or 64 bit signed and unsigned integers and floats), and tuples (τ^*). Next we have references, pointers, and capabilities. Pointers are used to access a memory location and capabilities provide ownership to a memory location. References represent the pair of a capability and a pointer. Capabilities are a type-level reasoning tool and are erased in RichWasm programs compiled to Wasm. The ability to split a reference into a capability and a pointer allows a program to separate its static notion of ownership from its runtime data layout. If pointers to a location are stashed in separate parts of a large data strucutre, ownership (in the form of a capability) can be stored with one of them and then temporarily borrowed by the other location by moving the capability. Since capabilities will be erased upon compilation to Wasm, this transfer of ownership satisifies the type system without incurring any runtime cost. The pretype $\textbf{ref}\ \pi\ \ell\ \psi$ is a reference to a location ℓ that contains a *heap type* ψ and provides read or read-write priviledge (π) to it. A capability carries similar information, while a pointer is annotated only with the location that it points to. The pretype $\textbf{own}\ \ell$ is an *ownership token* that represents write ownership of a location. A read-write capability can be temporarily split into a read-only capability and an ownership token and later recombined.

RichWasm has isorecursive types to represent recursive data structures. The pretype $\textbf{rec}\ q \preceq \alpha.\ \tau$ is recursive over α. The constraint $q \preceq$, which will be discussed further below, asserts that the recursive pretype will be unfolded only into locations with qualifiers greater than or equal to q.

To facilitate location abstraction, which is necessary to statically represent references to memory locations, RichWasm has existential pretypes over locations $\exists \rho.\ \tau$. For instance, $\exists \rho.\ (\textbf{ref}\ \textbf{rw}\ \rho\ \psi)^q$ represents a read-write reference to a statically unknown location with heap type ψ.

Wasm allows "indirect" function calls, using a runtime value to lookup a function in a table. We use the pretype $\textbf{coderef}\ \chi$ to represent a code pointer to some function in that table with type χ.

Lastly, a pretype can be a pretype variable α, referring to a universal, existential, or recursive binding site.

Heap Types and Memory Model. The memory model of RichWasm consists of two global flat memories: the linear memory and the unrestricted memory. The linear memory is manually managed and references to it must be treated linearly. The unrestricted memory is garbage collected and stores ML-like references. Unlike Wasm, where memories are essentially byte arrays, in RichWasm memories store high-level structured data. Heap types can be variants, structs, arrays, and existential packages abstracting over pretypes. A variant heap type ($\textbf{variant}\ \tau^*$) describes a heap value that can contain more than one kind of value, where τ^* is a list containing the type for each case of the variant. An array heap type ($\textbf{array}\ \tau$) is a variable-length array containing values of type τ. A struct heap type ($\textbf{struct}\ (\tau,\ sz)^*$) is a record type with the list $(\tau, sz)^*$ indicating the type τ and *the size sz* of each field. Keeping the size of each field is necessary to support *strong updates*. Strong updates are necessary when consuming linear struct fields, as the old value must be replaced with a new value in order to prevent duplication. An existential heap type ($\exists\ q \preceq \alpha \lesssim sz.\ \tau$) abstracts over a pretype α in a type τ. The qualifier q denotes the minimum qualifier that the pretype should have when it is used inside the type. The size sz denotes an upper bound for the size of the abstracted type, which is useful when we want to do a strong update with this type (either into a field of a struct or a local).

Locations are natural numbers that refer either to the unrestricted or linear memory. Since we have location polymorphism, locations can also be variables.

Function Types and Polymorphism. Function types are arrow types ${\tau_1}^* \to {\tau_2}^*$ that indicate that a function consumes values with types ${\tau_1}^*$ from the stack, and leaves values with types ${\tau_2}^*$ on the stack after its execution. To facilitate polymorphism, function types can quantify over four different kinds: locations, sizes, qualifiers, and pretypes. Quantification over sizes, qualifiers, and pretypes can impose constraints over the abstracted kind.

Location quantification is most straightforward: function declarations should be polymorphic over locations since concrete locations are only available at runtime.

Functions can also be polymorphic over sizes. For instance, projecting a struct field is an operation that should be agnostic to the size of the field, therefore the size must be abstracted. Quantification over sizes can also be subject to constraints. In particular, the size variable σ can be upper-bounded and lower-bounded by some sizes, written $sz^* \leq \sigma \leq {sz'}^*$. Allowing quantification over sizes to be constrained in this way allows for richer reasoning about location usage. For instance, if a function takes two arguments of sizes σ_1 and σ_2 and places a tuple consisting of the two arguments into a local of size σ_3, it must be known that $\sigma_1 + \sigma_2 \leq \sigma_3$.

To write functions that operate on both linear and unrestricted data, functions can also be polymorphic on qualifiers. Qualifier quantification can be subject to the same sort of contraint as size quantification, in this case written $q^* \preceq \delta \preceq {q'}^*$. To see why qualifier constraints are necessary, consider a function that takes two arguments and constructs a tuple. The qualifier of the tuple must be greater than the qualifiers on the values inside the tuple. If this were not the case, we could have a non-linear tuple which can be duplicated, containing a linear value which should not be.

Lastly, we have pretype polymorphism, which is the usual parametric polymorphism we find in ML programs. The type constraints are $q \preceq \alpha^{(c^?)} \lesssim sz$ where q is a lower bound for the qualifier of the pretype, sz an upper bound for its size, and the presence of ^{c} denotes whether this type can contain capabilities. Upper bounds on sizes are straightforwardly useful to demonstrate that types will fit into locations. A qualifier lower bound is, however, a bit less intuitive. Keep in mind that we are quantifying over *pretypes* and not types, so any location where this pretype variable appears will have a qualifier on it. These bounds provide two guarantees. Firstly, this pretype variable will only be substituted into positions with qualifiers greater than or equal to the bound. Secondly, we can only substitute a pretype in for such a pretype variable if it would be valid at that qualifier. To see why this is important, consider an attempt to instantiate the variable in the type $\alpha^{\mathbf{unr}}$ with the tuple pretype $(\mathbf{unit}^{\mathbf{lin}})$. This instantiation would leave us with an unrestricted tuple containing a linear value, a type we just determined would violate the guarantees of our type system. Having quantifier lower bounds will guarantee that such instantiations never take place. Whether or not a type contains capabilities is relevant only for garbage collection and will be discussed below.

Using polymorphism, we can write the type of a function that projects a linear field of a singleton struct and replaces it with a value of a new type, performing a strong update:

$$\forall\, \rho\, \sigma\, \alpha_1\, (\mathbf{lin} \preceq \alpha_2^c \lesssim \sigma).\, \alpha_2^{\mathbf{lin}}\, (\mathbf{ref}\ \mathbf{rw}\ \rho\ (\mathbf{struct}\ (\alpha_1^{\mathbf{lin}}, \sigma)))^{\mathbf{lin}} \to \alpha_1^{\mathbf{lin}}\, (\mathbf{ref}\ \mathbf{rw}\ \rho\ (\mathbf{struct}\ (\alpha_2^{\mathbf{lin}}, \sigma)))^{\mathbf{lin}}$$

The function is polymorphic over the actual location of the struct, ρ, the size of the struct field, σ, the pretype of the struct field α_1, and the pretype of the new value, α_2, that is going to be put in the struct slot. The constraints over α_2 ensure that it will fit into the slot of size σ and can be safely stored in the heap. The qualifier lower bound ensures that this function can be used on types which must appear in at least linear positions.

Heap Values. Variant heap values are written $(\mathbf{variant}\ i\ v)$, where i is tag of the variant. Struct values are written $(\mathbf{struct}\ v^*)$. Array values are written $(\mathbf{array}\ i\ v^*)$, where i is the size of the array.

Lastly, existential packages are writen (**pack** $p\ v\ \psi$), where p is the pretype witness, ψ the heap type of the existential package, and v the value that is being packed.

Values and Instructions. Each RichWasm type has a corresponding value form. For unit, numeric types, and tuples, these values are straightforward. References and pointers are annotated with the memory location they point to. Capabilities and ownership tokens carry no information and are computationally irrelevant. We have the typical constructor for isorecursive types (**fold** v). Existential location packages (**mempack** $\ell\ v$) contain the packed value as well as the location witness of the package, used to instantiate abstract locations in instructions once unpacked. The value form for function references is **coderef** $i\ j\ z^*$, where i is the index of the module instance, j is the index of the function in the function table, and z^* the concrete instantiation of the polymorphic indices.

New RichWasm instructions include instructions for folding and unfolding recursive types, packing and unpacking existential memory locations, grouping and ungrouping tuples, joining and spliting capabilties and ownership tokens, joining and spliting capabilities and pointers to form references, and manipulating heap values (each type of heap value has its own set of instructions).

Instructions that introduce blocks of instructions, like **block** or **if then else**, are annotated not only with their type as in Wasm, but also with their *local effects*, $(i, \tau)^*$. These describe the effect that a list of instructions may have on the type of their local variable slots: the slot at position i gets type τ.

2.2 Simple ML-L^3 FFI and Catching Unsafe Interoperability

We extend the languages in the style of *linking types* [18, 19], which allow languages to maintain their native reasoning principles while interacting with foreign types near language boundaries. ML gets a type, written <τ>, which directs the compiler to make an ML type τ linear in RichWasm. To store linear types, it also has a new construct `ref_to_lin` which creates a reference that can either be empty or contain a linear value of the given type. Normal `Ref` operations can be used on these references, but ML compiles them in such a way that if they are read from or written to twice, they will fail at runtime, as this would violate linearity and is unsound[3]. L^3, which typically only has capabilities and pointers, gets a new ML-like `Ref` type. To convert a capability plus pointer to and from a `Ref` type at the boundary with ML, we give L^3 two new constructs: `join` and `split`.

```
let c = ref_to_lin <Ref Int> in
fun stash (r : <Ref Int>) : <Ref Int> =
  c := r; r
in
fun get_stashed (() : Unit) = !c
```

(a) ML program

```
import (ml.stash :
  !((∃ ℓ. Ref ℓ !Int 1) ⊸ (∃ ℓ. Ref ℓ !Int 1)));
import (ml.get_stashed :
  !(!Unit ⊸ (∃ ℓ. Ref ℓ !Int 1)));
free (split (stash (join (new !42 1))));
free (split (get_stashed !())) (* CRASH! *)
```

(b) L^3 program

Fig. 3. Unsafe interoperability

Fig. 3 shows an example akin to that in Fig. 1, but with syntax more accurate to ML and L^3. The key differences are that the programs must use their new extensions to agree on types at the linking boundary. The problematic function here is still ML's `stash`, and RichWasm will not allow it to typecheck since it duplicates a linear value. If `stash` were to not return the linear value (and correspondingly L^3 were to no longer attempt to free that value, since it's not returned), this program would type check and L^3's previously illegal attempt to free would be safe.

3 RICHWASM DYNAMIC SEMANTICS

Execution in RichWasm terms closely follows Wasm and is defined as a reduction relation. The relation, written $s; v^*; sz^*; e^* \hookrightarrow_j s'; v'^*; e'^*$, represents a reduction step in module j from one

[3]In the example in Fig. 3, the ML code uses the linear reference exactly once, which is valid.

program configuration $s; v^*; sz^*; e^*$ to another, where s is the store, v^* the local values and sz^* the sizes of their slots, and e^* the instructions to be evaluated. Fig. 4 shows the definition of runtime objects as well as important rules of the reduction relation. We elide rules that are identical to Wasm, and we focus on the reduction rules of new constructs.

Store and Module Instances. A store represents the execution state. It holds the list of *module instances*, which is the dynamic representation of static modules and the *global memory*. A module instance consists of a dynamic function table that holds a list of *closures* which, as in Wasm, is used for direct calls, a list of global values, and the dynamic table which is a list of closures that is used for indirect calls. A closure, in the Wasm sense, is a dynamic instance of a function that consists of the code and a pointer to the module that provides the function's environment. The global memory has two components: the *linear* memory and the *unrestricted* memory. Both memories are maps from locations to high-level heap values.

Administrative Instructions. As in Wasm, some reduction rules generate instructions that are not part of the syntax of source programs and represent administrative operations. A **trap** instruction signals an execution trap. We add two administrative instructions: **malloc** sz hv q to allocate memory of size sz holding a value hv in the memory q, and **free** to deallocate parts of the *linear* memory, given a reference. Allocation and deallocation of heap types reduce to these instructions. The administrative instruction **call** cl z^* is similar to Wasm's, but along with the closure cl, it is annotated with the concrete instantiation of the polymorphic quantifiers in the function's type, z^*.

Control Flow. Much like in Wasm, when running RichWasm programs, evaluation occurs within some number of label instructions, each corresponding to a source block of code (introduced by instructions like **block**, **if**, **loop**). The break instructions (**br**, **br_if**, **br_table**) allow programs to jump to any of the surrounding locations by specifying how many labels to jump over. Nested label instructions are represented with *local contexts*. A local context $L^N[e^*]$ represents N nested label instructions. A local context has a hole at its deepest label instruction, where evaluation can occur.

3.1 Reduction Relation

We give an overview of the most important reduction rules of RichWasm that are new or different from those in Wasm. For space reasons, we drop the store and local variables in all the rules that leave them unchanged. We may still use the store s in side conditions where it is relevant, without explicitly mentioning it in the rule.

Garbage Collection. The reduction relation has a rule that can be applied at any point, allowing collection of unrestricted locations that are no longer accessible from the configuration. The roots of collection are the unrestricted locations that appear in **ref** / **cap** values in the instructions, local variables, or the module instances. Any location not reachable from the roots is collected.

If a reference to linear memory is placed into garbage collected memory, we say that the garbage collector now *owns* that memory and is responsible for collecting it if the unrestricted location, and thus the only reference to the linear location, should be collected. Freeing of linear memory is a type-directed operation, and when compiling to Wasm, we can generate finalizer functions that get called when such references are collected. But what would we do if a capability to linear memory were in garbage-collected memory? When compiling to Wasm, capabilities will be erased, which would leave the garbage collector with no way to reference the linear memory location it owns at runtime. To resolve this, we require that capabilities always be paired with a pointer in the form of a reference when placed in memory.

Function Calls. Function calls follow similar principles to those of Wasm. Few differences arise from the polymorphic types of RichWasm functions and the way polymorphic binders are instantiated. Direct function calls are performed with the **call** j z^* instruction, where j is the index of the

$$\begin{array}{llll} \textit{store} & s & ::= & \{\text{inst } \textit{inst}^*, \text{mem } \textit{mem}\} \\ \textit{instances} & \textit{inst} & ::= & \{\text{func } \textit{cl}^*, \text{glob } v^*, \text{tab } \textit{cl}^*\} \end{array} \qquad \begin{array}{llll} \textit{memory} & \textit{mem} & ::= & \{\text{lin } i \mapsto hv, \text{unr } i \mapsto hv\} \\ \textit{closure} & \textit{cl} & ::= & \{\text{inst } i, \text{code } f\} \end{array}$$

$$\begin{array}{llll} \textit{administrative instructions} & e & ::= & \cdots \mid \textbf{trap} \mid \textbf{call}\ \textit{cl}\ z^* \mid \textbf{label}_i\ \textit{tf}\ \{e_1{}^*\}\ e_2{}^*\ \textbf{end} \mid \\ & & & \quad \textbf{local}_i\ \{j;\ (v, sz)^*\}\ e^*\ \textbf{end} \mid \textbf{malloc}\ sz\ hv\ q \mid \textbf{free} \\ \textit{local contexts} & L^0 & ::= & v^*\ [\_]\ e^* \\ & L^{k+1} & ::= & v^*\ \textbf{label}_i\ \textit{tf}\ \{e^*\}\ L^k\ \textbf{end}\ e^* \end{array}$$

Reduction

$$\boxed{s; v^*; sz^*; e^* \hookrightarrow_j s'; v'^*; e'^*}$$

$$\frac{s; v^*; sz^*; e^* \hookrightarrow_j s'; v'^*; e'^*}{s; v^*; sz^*; L^k[e^*] \hookrightarrow_j s'; v'^*; L^k[e'^*]} \qquad \frac{s; v^*; sz^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v_0{}^*; i^*; \textbf{local}_n\ \{i;\ (v, sz)^*\}\ e^*\ \textbf{end} \hookrightarrow_j s'; v_0{}^*; \textbf{local}_n\ \{i;\ (v', sz)^*\}\ e'^*\ \textbf{end}}$$

$$\frac{\textbf{collect}(\textbf{locs}(e^*) \cup \textbf{locs}(v^*) \cup \textbf{locs}(\textit{inst}), \textit{mem}, \textit{mem}'),}{\{\text{inst } \textit{inst}^*, \text{mem } \textit{mem}\}; v^*; sz^*; e^* \hookrightarrow_j \{\text{inst } \textit{inst}^*, \text{mem } \textit{mem}'\}; v^*; e^*}$$

$$\begin{array}{rcll}
s; v^*; sz^*; \textbf{call}\ j\ z^* & \hookrightarrow_i & s; v^*; \textbf{call}\ \textit{cl}\ z^* & \text{where } \textit{cl} = (s_{\text{inst}}(i))_{\text{tab}}(j) \\
s; v^*; sz^*; \textbf{coderef}\ i\ j\ z^*; \textbf{call_indirect} & \hookrightarrow & s; v^*; \textbf{call}\ \textit{cl}\ z^* & \text{where } \textit{cl} = (s_{\text{inst}}(i))_{\text{tab}}(j) \\
v^n\ \textbf{call}\ \{\text{inst } i, \text{code } \textbf{function}\ \forall \kappa^*.\ \tau_1^n \to \tau_2^m\ \textbf{local}\ sz^k\ e^*\}\ z^* & \hookrightarrow & \textbf{local}_m\ \{i;\ \textit{locals}\}\ e^*[z^*/\kappa^*]\ \textbf{end} & \text{where} \\
 & & \quad \textit{locals} = (v, \textbf{size}(\tau_1[z^*/\kappa^*]))^n((), sz[z^*/\kappa^*])^k & \\
v^n\ \textbf{struct.malloc}\ sz^n\ q & \hookrightarrow & \textbf{malloc}\ \textbf{size}(sz^n)\ (\textbf{struct}\ v^n)\ q & \\
\textbf{struct.free} & \hookrightarrow & \textbf{free} & \\
(\textbf{ref}\ l_{mem})\ \textbf{struct.get}\ i & \hookrightarrow & s; v^*; (\textbf{ref}\ l_{mem})\ v_i & \text{where} \\
 & & \quad (s_{\textbf{mem}})_{mem}(l) = (\textbf{struct}\ v_1 \dots v_i \dots) & \\
s; v^*; sz^*; (\textbf{ref}\ l_{mem})\ v\ \textbf{struct.set}\ i & \hookrightarrow & s'; v^*; \textbf{ref}\ l_{mem} & \text{where} \\
 & & \quad (s_{\textbf{mem}})_{mem}(l) = (\textbf{struct}\ v_1 \dots v_i \dots) & \\
 & & \quad s' = s \text{ with } \textbf{mem}_{mem}(l) = (\textbf{struct}\ v_1 \dots v_{i-1}\ v \dots) & \\
s; v^*; sz^*; (\textbf{ref}\ l_{mem})\ v\ \textbf{struct.swap}\ i & \hookrightarrow & s'; v^*; (\textbf{ref}\ l_{mem})\ v_i & \text{where} \\
 & & \quad (s_{\textbf{mem}})_{mem}(l) = (\textbf{struct}\ v_1 \dots v_i \dots) & \\
 & & \quad s' = s \text{ with } \textbf{mem}_{mem}(l) = (\textbf{struct}\ v_1 \dots v_{i-1}\ v \dots) & \\
v\ \textbf{variant.malloc}\ j\ \tau^*\ q & \hookrightarrow & \textbf{malloc}\ (32 + \textbf{size}(v))\ (\textbf{variant}\ j\ v)\ q & \\
\textbf{ref}\ l_{mem}\ v^n\ \textbf{variant.case unr}\ \psi\ \textit{tf}\ (i, \tau)^*\ (e^*)^m\ \textbf{end} & \hookrightarrow & (\textbf{ref}\ l_{mem})\ v^n\ \textbf{block}\ \textit{tf}\ (i, \tau)^*\ v'\ (e^*)^m_{(i)}\ \textbf{end} & \\
 & & \quad \text{where } \psi = (\textbf{variant}\ \tau^m) & \\
 & & \quad \textit{tf} = \tau_1^n \to \tau_2{}^* & \\
 & & \quad (s_{\textbf{mem}})_{mem}(l) = (\textbf{variant}\ i\ v') & \\
s; v^*; sz^*; (\textbf{ref}\ l_{\textbf{lin}})\ v^n\ \textbf{variant.case lin}\ \psi\ \textit{tf}\ (i, \tau)^*\ (e^*)^m\ \textbf{end} & \hookrightarrow & s'; v^*; (\textbf{ref}\ l_{\textbf{lin}})\ \textbf{free}\ v^n\ \textbf{block}\ \textit{tf}\ (i, \tau)^*\ v'\ (e^*)^m_{(i)}\ \textbf{end} & \\
 & & \quad \text{where } \psi = (\textbf{variant}\ \tau^m) & \\
 & & \quad \textit{tf} = \tau_1^n \to \tau_2{}^* & \\
 & & \quad (s_{\textbf{mem}})_{\textbf{lin}}(l) = (\textbf{variant}\ i\ v') & \\
 & & \quad s' = s \text{ with } \textbf{mem}_{\textbf{lin}}(l) = (\textbf{array}\ 0\ \epsilon) & \\
v\ \textbf{ui32.const}\ j\ \textbf{array.malloc}\ q & \hookrightarrow & \textbf{malloc}\ (j \times \textbf{size}(v))\ (\textbf{array}\ j\ v^j)\ q & \\
(\textbf{ref}\ l_{mem})\ (np.\textbf{const}\ j)\ \textbf{array.get} & \hookrightarrow & (\textbf{ref}\ l_{mem})\ v_j & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{mem}(l) = (\textbf{array}\ i\ (v_0 \dots v_j \dots)) & \\
(\textbf{ref}\ l_{mem})\ (np.\textbf{const}\ j)\ \textbf{array.get} & \hookrightarrow & \textbf{trap} & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{mem}(l) = (\textbf{array}\ i\ v^*) & \\
 & & \quad j \geq i \text{ or } j < 0 & \\
s; v^*; sz^*; (\textbf{ref}\ l_{mem})\ (np.\textbf{const}\ j)\ v\ \textbf{array.set} & \hookrightarrow & s'; v^*; \textbf{ref}\ l_{mem} & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{mem}(l) = (\textbf{array}\ i\ v_0 \dots v_{j-1} v_j \dots) & \\
 & & \quad s' = s \text{ with } \textbf{mem}_{mem}(l) = (\textbf{array}\ i\ v_0 \dots v_{j-1} v \dots) & \\
(\textbf{ref}\ l_{mem})\ (np.\textbf{const}\ j)\ \textbf{array.set} & \hookrightarrow & \textbf{trap} & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{mem}(l) = (\textbf{array}\ i\ v^*) & \\
 & & \quad j \geq i \text{ or } j < 0 & \\
v\ \textbf{exist.pack}\ p\ \psi\ q & \hookrightarrow & \textbf{malloc}\ (64 + \textbf{size}(v))\ (\textbf{pack}\ p\ v\ \psi)\ q & \\
(\textbf{ref}\ l_{mem})\ v^n\ \textbf{exist.unpack unr}\ \psi\ \textit{tf}\ (i, \tau)^*\ \alpha\ e^*\ \textbf{end} & \hookrightarrow & (\textbf{ref}\ l_{mem})\ v^n\ \textbf{block}\ \textit{tf}\ (i, \tau)^*\ v'\ e^*[p/\alpha]\ \textbf{end} & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{mem}(l) = (\textbf{pack}\ p\ v'\ \psi) & \\
 & & \quad \textit{tf} = \tau_1^n \to \tau_2{}^* & \\
s; v^*; sz^*; (\textbf{ref}\ l_{\textbf{lin}})\ v^n\ \textbf{exist.unpack lin}\ \psi\ \textit{tf}\ (i, \tau)^*\ \alpha\ e^*\ \textbf{end} & \hookrightarrow & s'; v^*; (\textbf{ref}\ l_{\textbf{lin}})\ \textbf{free}\ v^n\ \textbf{block}\ \textit{tf}\ (i, \tau)^*\ v'\ e^*[p/\alpha]\ \textbf{end} & \\
 & & \quad \text{where } (s_{\textbf{mem}})_{\textbf{lin}}(l) (\textbf{pack}\ p\ v'\ \psi) & \\
 & & \quad s' = s \text{ with } \textbf{mem}_{\textbf{lin}}(l) = (\textbf{array}\ 0\ \epsilon) & \\
 & & \quad \textit{tf} = \tau_1^n \to \tau_2{}^* & \\
s; v^*; sz^*; (\textbf{ref}\ l_{\textbf{lin}})\ \textbf{free} & \hookrightarrow & s'; v^*; \epsilon & \text{where} \\
 & & \quad s' = s \text{ with } l \notin \text{dom}(\textbf{mem}_{\textbf{lin}}) & \\
s; v^*; sz^*; \textbf{malloc}\ sz\ hv\ q & \hookrightarrow & s'; v^*; \textbf{mempack}\ \ell_q\ (\textbf{ref}\ \ell_q) & \\
 & & \quad \text{where } s' = s \text{ with } \textbf{mem}_q(\ell) = hv &
\end{array}$$

Fig. 4. RichWasm dynamic semantics

function in the function table and z^* are concrete instantiations of the polymorphic binders. The instruction **call** j z^* is reduced to the administrative instruction **call** cl z^*, where cl is the corresponding closure in the function table. Indirect function calls are performed with **call_indirect**, whose argument is a code pointer **coderef** i j z^* that points to the jth function of the ith module instance and z^* are the concrete instantiations of polymorphic indices.

The administrative call instruction **call** cl z^*, where cl is a closure containing a module number and function body, takes as stack arguments the arguments of the function and reduces to a local frame that performs the necessary substitutions of the polymorphic indices. For the sizes of local variable slots we use the metafunction **size**$(\cdot)$ that returns the size of a type.

Heap Manipulation. Let's consider the rules for manipulating RichWasm's new heap structures. All heap types have their own allocation instruction that reduces to the **malloc** instruction, which in turn reduces to a reference contained in an existential package that abstracts the location.

Reduction rules for struct's get, set and swap operations are straightforward, taking a reference from the stack and perhaps the value to put at the ith field, performing any necessary memory updates, and leaving the reference back on the stack, together with the read value, if any.

Variants can be used to perform case analysis. The case instruction expects on the stack the reference to the variant and a list of values expected by the branches, that have types τ_1^n. If the allocated value is the ith case of the variant, then we create an instruction block using the ith element of the list of instruction blocks $(e^*)^m$, written $(e^*)^m_{(i)}$. If the case instruction is annotated with an unrestricted qualifier, the reference is returned to the stack for reuse. If the annotation is linear, then a **free** instruction is generated to consume the reference. In this case the contents of the memory are replaced with an empty array, so that the linearity invariants of the type system are preserved. Existential types can be packed and unpacked. Packing an existential type triggers the allocation of a **pack** heap value. Much like variants, the unpacking operation is a block instruction which can optionally free the underlying memory. The unpack operation additionally needs to substitute the witness pretype in the list of instructions to be evaluated.

4 RICHWASM TYPE SYSTEM

We now give a technical account of the type system as well as its safety properties.

Typing Environments. Typing environments in RichWasm (fig. 5) are similar to Wasm, but they also keep track of the new kind variables (types, sizes, locations, qualifiers) and their constraints.

The *local* environment is a list of the types and sizes of the local variables of a program configuration, where the ith element corresponds to the size and the type of local variable i.

The *function environment* stores information about the current evaluation context. It has 7 components. As in Wasm, the label component keeps track of the return type of all the available jump locations (i.e., nested labels). In RichWasm, it additionally tracks the resulting local environment, as all jumps to a location must have the same view of the types of locals. The return component keeps track of the return type of the current block of instructions. qual, size, and type are partial maps from the variables in scope to the constraints placed on them, as explained in §2. loc keeps track of declared location variables. Lastly, the linear environment contains a list of qualifiers representing the greatest lower bound of the qualifiers of the values on the stack between two jump locations. Jumping from the current evaluation context to an outer label will drop the contents of that label, including any potentially linear values. The linear environment can be used to verify that all values dropped when performing a jump to a given label are unrestricted.

To update a component of a typing environment we use similar notation as the WebAssembly paper[9]. For instance, we write $F, \mathsf{linear}\ \mathbf{unr} :: F_{\mathsf{linear}}$ to update the linear environment of F by adding the qualifier **unr** to the top of the list.

$$
\begin{array}{lll}
\textit{Local Env} & L & ::= (\tau, sz)^* \\
\textit{Function Env} & F & ::= \{\ \text{label}\ (\tau^*, L)^*,\ \text{return}\ (\tau^*)^?,\ \text{qual}\ \delta \rightharpoonup (q^*, q^*),\ \text{size}\ \sigma \rightharpoonup (sz^*, sz^*),\ \text{type}\ \alpha \rightharpoonup (sz, q, c^?),\ \text{loc}\ \ell^*,\ \text{linear}\ q^*\ \} \\
\textit{Module Env} & M & ::= \{\ \text{func}\ tf^*,\ \text{global}\ tg^*,\ \text{table}\ tf^*\ \} \\
\textit{Store Typing} & S & ::= \{\ \text{inst}\ M^*,\ \text{unr}\ \ell \rightharpoonup \psi,\ \text{lin}\ \ell \rightharpoonup \psi\ \}
\end{array}
$$

Fig. 5. Typing environments.

The module environment keeps track of the declared functions and globals in the current module. Finally, we have the store typing that keeps track of the list of module instances (inst) and the typing of the linear and unrestricted memories. The memory typing is a partial map from memory locations to the stored heap type (ψ). The linear memory typing is split across the typing of the subexpressions in the premises of rules to ensure that no linear resource is used twice in the program. We write $S = S_1 \uplus S_2$ to denote that the linear memory typing of S_{lin} is the *disjoint* union of $(S_1)_{\mathsf{lin}}$ and $(S_2)_{\mathsf{lin}}$, whereas the other components are exactly the same in all three store typings. When typing a base value or instruction that has no linear memory locations, we require that the linear store typing is empty, ie., no linear resource is ever dropped.

Value Typing. The value typing judgment, written $S; F \vdash v : \tau$, asserts that a value v has type τ in a store typing S and function environment F. Selected value typing rules are shown in Fig. 6. Numeric constants have the corresponding numeric type and can have any qualifier. Tuples have a tuple type consisting of the types of individual values. The top-level qualifier must be an upper bound for each individual qualifier q_i of any type $p_i^{q_i}$ inside the list τ^*. Pointers have type $\mathbf{ptr}\ \ell^q$, which is independent of the heap type of the location, and do not consume a location from the memory typing as they do not represent memory ownership. Typing of capabilities and references is parallel in structure. We only explain references. In the unrestricted case, a reference $\mathbf{ref}\ \ell_{\mathbf{unr}}$ has type $(\mathbf{ref}\ \pi\ \ell_{\mathbf{unr}}\ \psi)^q$, where ψ is the type of $\ell_{\mathbf{unr}}$ in the unrestricted component of the heap. The qualifier q must be provably unrestricted in the constraints of F_{qual}, written $q \preceq_{F_{\mathsf{qual}}} \mathbf{unr}$. The linear component of the memory typing must be empty. The case with a linear location is similar: the qualifier on the reference must be provably linear, and the linear component of the memory typing must be the singleton map from ℓ to ψ. The value $\mathbf{fold}\ v$ has the recursive type $(\mathbf{rec}\ \alpha \preceq q.\ p^q)^{q'}$, if the value v has type $(p[\mathbf{rec}\ \alpha \preceq q.\ p^q/\alpha])^q$ where we have substituted α with $\mathbf{rec}\ \alpha \preceq q.\ p^q$, and the qualifier q is upper bounded by q'. Next we have existential packages $\mathbf{mempack}\ \ell\ v$ that have an existential type $(\exists \rho.\ p^q)^{q'}$ if the type of value v is $(p[\ell/\rho])^{q'}$ and qualifier q is upper bounded by q'. Lastly, we have typing for code references $\mathbf{coderef}\ i\ j\ z^*$: if M is the ith module instance and $\forall \kappa^*.tf$ the type of the jth function in the table of M, then the code reference has type $(tf[z^*/\kappa^*])^q$ where quantifier variables κ have been substituted with the indices z.

$$\boxed{S; F \vdash v : \tau}$$

$$\frac{S = \biguplus_{i \le n} S_i \qquad \forall\ v_i \in v^n\ \tau_i \in \tau^n,\ S_i; F \vdash v_i : \tau_i \qquad \forall\ p_i^{q_i} \in \tau^n,\ q_i \preceq_{F_{\mathsf{qual}}} q}{S; F \vdash v^n : (\tau^n)^q} \qquad \frac{S_{\mathsf{lin}} = \emptyset}{S; F \vdash \mathbf{ptr}\ \ell : \mathbf{ptr}\ \ell^q}$$

$$\frac{S_{\mathsf{lin}} = \emptyset \qquad S_{\mathsf{unr}}(\ell) = \psi \qquad q \preceq_{F_{\mathsf{qual}}} \mathbf{unr}}{S; F \vdash \mathbf{ref}\ \ell_{\mathbf{unr}} : (\mathbf{ref}\ \pi\ \ell_{\mathbf{unr}}\ \psi)^q} \qquad \frac{S_{\mathsf{lin}} = [\ell \mapsto \psi] \qquad \mathbf{lin} \preceq_{F_{\mathsf{qual}}} q}{S; F \vdash \mathbf{ref}\ \ell_{\mathbf{lin}} : (\mathbf{ref}\ \pi\ \ell_{\mathbf{lin}}\ \psi)^q} \qquad \frac{S; F \vdash v : (p[\ell/\rho])^q \qquad q \preceq_{F_{\mathsf{qual}}} q'}{S; F \vdash \mathbf{mempack}\ \ell\ v : (\exists \rho.\ p^q)^{q'}}$$

$$\frac{S_{\mathsf{lin}} = \emptyset \qquad S; F \vdash v : (p[\mathbf{rec}\ \alpha \preceq q.\ p^q/\alpha])^q \qquad q \preceq_{F_{\mathsf{qual}}} q'}{S; F \vdash \mathbf{fold}\ v : (\mathbf{rec}\ \alpha \preceq q.\ p^q)^{q'}} \qquad \frac{S_{\mathsf{lin}} = \emptyset \qquad S_{\mathsf{inst}}(i) = M \qquad M.\mathsf{table}(j) = \forall \kappa^*.tf}{S; F \vdash \mathbf{coderef}\ i\ j\ z^* : (tf[z^*/\kappa^*])^q}$$

Fig. 6. Value typing.

Heap Typing. The heap typing judgement is written $S; F \vdash hv : \psi$ and asserts that the heap value hv has type ψ in the store typing S and function environment F.

Instruction Typing. The core of RichWasm's type system is typing of instructions. The typing judgement, written $S; M; F; L \vdash e^* : {\tau_1}^* \to \tau_2^* \mid L'$ asserts that a list of instructions e^* has type ${\tau_1}^* \to \tau_2^*$. S, M, and F are the store typing, module environment and function environment respectively. L is the local environment that keeps track of the types and sizes of the local variables, and L' is the typing of the local variables after execution of e^*, which might change the types of local variables. Many rules follow Wasm with the addition that we add the necessary premises to ensure linearity (i.e., constraints on qualifiers and linear memory typing). In Fig. 7, we show several important rules of the system.

$$\boxed{S; M; F; L \vdash e^* : {\tau_1}^* \to \tau_2^* \mid L'}$$

$$\frac{L' = (i, \tau)^*[L] \qquad F' = F, \text{label } (\tau_2, L') :: F_{\text{label}}, \text{linear } \mathbf{unr} :: F_{\text{linear}} \qquad S; M; F'; L \vdash e^* : {\tau_1}^* \to {\tau_2}^* \mid L'}{S; M; F; L \vdash \mathbf{block}\ {\tau_1}^* \to {\tau_2}^*\ (i, \tau)^*\ e^*\ \mathbf{end} : {\tau_1}^* \to {\tau_2}^* \mid L'}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad L_{(i)} = (p^q, sz) \qquad (q \preceq_{F_{\text{qual}}} \mathbf{unr} \wedge \tau' = p^q) \vee (\neg q \preceq_{F_{\text{qual}}} \mathbf{unr} \wedge \tau' = \mathbf{unit}^q)}{S; M; F; L \vdash \mathbf{get\_local}\ i\ q : \epsilon \to p^q \mid L[i \mapsto (\tau', sz)]}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad L_{(i)} = (p^q, sz) \qquad q \preceq_{F_{\text{qual}}} \mathbf{unr} \qquad ||\tau||_{F_{\text{type}}} \preceq_{F_{\text{size}}} sz}{S; M; F; L \vdash \mathbf{set\_local}\ i : \tau \to \epsilon \mid L[i \mapsto (\tau, sz)]} \qquad \frac{S_{\text{lin}} = \emptyset}{S; M; F; L \vdash \mathbf{mem.pack}\ \ell : (p[\ell/\rho])^q \to (\exists \rho.\ p^q)^q \mid L}$$

$$\frac{L' = (i, \tau)^*[L] \qquad F' = F, \text{label } ({\tau_2}^*, L') :: F_{\text{label}}, \text{linear } \mathbf{unr} :: F_{\text{linear}}, \text{loc } \rho :: F_{\text{loc}} \qquad S; M; F'; L \vdash e^* : {\tau_1}^*\ \tau_\rho \to {\tau_2}^* \mid L'}{S; M; F; L \vdash \mathbf{mem.unpack}\ {\tau_1}^* \to {\tau_2}^*\ (i, \tau)^*\ \rho.\ e^* : {\tau_1}^*\ (\exists \rho.\ \tau_\rho)^{q_\rho} \to {\tau_2}^* \mid L'}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad no\_caps_{F_{\text{type}}}\ \tau^n \qquad \forall\ (\tau_i, sz_i) \in (\tau, sz)^n,\ ||\tau_i||_{F_{\text{type}}} \preceq_{F_{\text{size}}} sz_i}{S; M; F; L \vdash \mathbf{struct.malloc}\ sz^n\ q : \tau^n \to \exists \rho.\ (\mathbf{ref}\ \mathbf{rw}\ \rho\ (\mathbf{struct}\ (\tau, sz)^n))^q \mid L}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad \tau^*_{(i)} = p_i^{q_i} \qquad q_i \preceq_{F_{\text{qual}}} \mathbf{unr}}{S; M; F; L \vdash \mathbf{struct.get}\ i : (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau, sz)^*))^q \to (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau, sz)^*))^q; p_i^{q_i} \mid L}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad \tau^*_{(i)} = p_i^{q_i} \qquad q_i \preceq_{F_{\text{qual}}} \mathbf{unr} \qquad ||\tau_i'||_{F_{\text{type}}} \preceq_{F_{\text{size}}} sz^*_{(i)} \qquad no\_caps_{F_{\text{type}}}\ \tau_i' \qquad \mathbf{lin} \preceq_{F_{\text{qual}}} q \vee \tau_i' = p_i^{q_i}}{S; M; F; L \vdash \mathbf{struct.set}\ i : (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau, sz)^*))^q; \tau_i' \to (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau[i \mapsto \tau_i'], sz)^*))^q \mid L}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad \tau^*_{(i)} = p_i^{q_i} \qquad ||\tau_i'||_{F_{\text{type}}} \preceq_{F_{\text{size}}} sz^*_{(i)} \qquad no\_caps_{F_{\text{type}}}\ \tau_i' \qquad \mathbf{lin} \preceq_{F_{\text{qual}}} q \vee \tau_i' = p_i^{q_i}}{S; M; F; L \vdash \mathbf{struct.swap}\ i : (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau, sz)^*))^q; \tau_i' \to (\mathbf{ref}\ \pi\ \ell\ (\mathbf{struct}\ (\tau[i \mapsto \tau_i'], sz)^*))^q; p_i^{q_i} \mid L}$$

$$\frac{\begin{array}{c} S_{\text{lin}} = \emptyset \qquad L' = (i, \tau)^*[L] \qquad F' = F, \text{label } ({\tau_2}^*, L') :: F_{\text{label}}, \text{linear } \mathbf{unr} :: F_{\text{linear}} \\ \forall\ i \le n,\ S; M; F'; L \vdash (e^*)_{(i)} : {\tau_1}^*\ (\tau^n)_{(i)} \to {\tau_2}^* \mid L' \qquad \mathbf{lin} \preceq_{F_{\text{qual}}} q \qquad \mathbf{lin} \preceq_{F_{\text{qual}}} q_v \qquad \psi = (\mathbf{variant}\ \tau^n) \end{array}}{S; M; F; L \vdash \mathbf{variant.case}\ q\ \psi\ {\tau_1}^* \to {\tau_2}^*\ (i, \tau)^*\ (e^*)^n\ \mathbf{end} : (\mathbf{ref}\ \pi\ \ell\ \psi)^{q_v}\ {\tau_1}^* \to {\tau_2}^* \mid L'}$$

$$\frac{\begin{array}{c} S_{\text{lin}} = \emptyset \\ q_v \preceq_{F_{\text{qual}}} q' \qquad get\_hd\ F_{\text{linear}} \preceq_{F_{\text{qual}}} q' \qquad L' = (i, \tau)^*[L] \qquad F' = F, \text{label } ({\tau_2}^*, L') :: F_{\text{label}}, \text{linear } \mathbf{unr} :: set\_hd\ q'\ F_{\text{linear}} \\ \forall\ i \le n,\ S; M; F'; L \vdash (e^*)_{(i)} : {\tau_1}^*\ (\tau^n)_{(i)} \to {\tau_2}^* \mid L' \qquad \forall\ p_i{}^{q_i} \in \tau^n,\ q_i \preceq_{F_{\text{qual}}} \mathbf{unr} \qquad q \preceq_{F_{\text{qual}}} \mathbf{unr} \qquad \psi = (\mathbf{variant}\ \tau^n) \end{array}}{S; M; F; L \vdash \mathbf{variant.case}\ q\ \psi\ {\tau_1}^* \to \tau_2^*\ (i, \tau)^*\ (e^*)^n\ \mathbf{end} : (\mathbf{ref}\ \pi\ \ell\ \psi)^{q_v}\ {\tau_1}^* \to (\mathbf{ref}\ \pi\ \ell\ \psi)^{q_v}\ {\tau_2}^* \mid L'}$$

$$\frac{S; F \vdash hv : \psi \qquad F \vdash q\ \text{qual} \qquad no\_caps_{F_{\text{type}}}\ \psi}{S; M; F; L \vdash \mathbf{malloc}\ sz\ hv\ q : \epsilon \to (\exists \rho.\ (\mathbf{ref}\ \mathbf{rw}\ \rho\ \psi)^q)^q \mid L}$$

$$\frac{S_{\text{lin}} = \emptyset \qquad \mathbf{lin} \preceq_{F_{\text{qual}}} q \qquad F \vdash (\mathbf{ref}\ \mathbf{rw}\ \ell\ \psi)^q\ \text{type}}{S; M; F; L \vdash \mathbf{free} : (\mathbf{ref}\ \mathbf{rw}\ \ell\ \psi)^q \to \epsilon \mid L}$$

$$\frac{\begin{array}{c} q_{hd} = get\_hd\ F_{\text{linear}} \qquad q_{hd} \preceq_{F_{\text{qual}}} q_f \\ \forall\ p_i{}^{q_i} \in \tau^*,\ q_i \preceq_{F_{\text{qual}}} q_f \qquad F' = F, \text{linear } set\_hd\ q_f\ F_{\text{linear}} \qquad \forall\ \tau \in \tau^*,\ F \vdash \tau\ \text{type} \qquad S; M; F'; L \vdash e^* : {\tau_1}^* \to {\tau_2}^* \mid L' \end{array}}{S; M; F; L \vdash e^* : \tau^*\ {\tau_1}^* \to \tau^*\ {\tau_2}^* \mid L'}$$

Fig. 7. Instruction typing.

All block-style instructions follow similar principles, so let's examine the block instruction **block** tf $(i, \tau)^*$ e^* **end**. The block instruction is annotated with the type of the inner list of instructions ${\tau_1}^* \rightarrow \tau_2^*$ and the local effects $(i, \tau)^*$, which describe the effect that this block of instructions has on the local environment. The premises of the rule first construct the returned local environment L', by applying the local effects to the initial local environment, which is written $(i, \tau)^*[L]$ and means that the type of the ith slot of L changes to $\tau^*_{(i)}$. Then the premises assert that block of instructions has the expected type inside a new function environment that we obtain by pushing (τ_2, L') to the label component and **unr** to the linear component. The former tracks the return type and local environment of the label we are creating. The latter gives us a new qualifier with which we will track linearity of values on the stack inside this new block and "locks in" the qualifier corresponding to the linearity of the values on the stack between the previous enclosing block and this new block. We choose **unr** because upon entering a block there are no linear values which might be jumped over. The head element of the linear environment can be increased by the frame rule, which as in Wasm, allows typing rules to ignore values lower on the stack. Since break instructions will not see ignored values, the only way they can know whether they will be dropping linear values is by consulting this environment.

Let's look at some instructions for manipulating locals. The **get_local** i q instruction fetches the value of the ith local slot. If the qualifier of the slot, q, is linear, then the contents of the slot must be updated to ensure linearity. We replace it with the unit value and update the typing of the slot accordingly. The **set_local** i instruction updates the ith local slot, allowing to update its type as well. It ensures that the previous value was unrestricted, so it can freely be dropped, and that the upper bound of the size of the new type $||\tau||_{F_{\text{type}}}$, fits into the size of the slot, written $||\tau||_{F_{\text{type}}} \leq_{F_{\text{size}}} sz$. The size function takes the type environment component as parameter to lookup the upper bound for the size of type variables. The $\leq_{F_{\text{size}}}$ operation takes as parameter the size component to take into account the size constraints that are in scope.

Creating an existential package is done with **mem.pack** ℓ. It receives a value from the stack containing a location ℓ and creates an existential package that hides this location. The typing rule for **mem.unpack** tf $(i, \tau)^*$ ρ. e^* is more complicated. It combines the typing of instructions that introduce a new block with unpacking an existential location. The instruction receives from the stack the arguments of the instruction block ${\tau_1}^*$ and a packed value with an existential location type. Then it puts the location variable in the F_{loc} environment component to type the instructions in the block. The handling of the F_{label} and F_{linear} components is the same as in the block instruction.

Next we discuss instructions for manipulation heap values. Each family of heap values has its own malloc instruction. For example, the **struct.malloc** sz^n q instruction allocates space for a struct of n values with sizes sz^n. It receives from the stack n values of types τ^n and it returns a reference whose location is abstracted with an existential type and has the corresponding struct type. We require that the size of the types fit into the requested sizes of the slots. We also put the restriction that there are no capabilities in the types, for the reasons described in §3. Structs have get, set, and swap operations shown in the following rules. **struct.get** i gets the ith element of a struct, that must be an unrestricted value, and **struct.set** i sets the ith element with the a value it finds on the stack. The qualifier of the previous value that is dropped must be unrestricted. If the reference on the stack is linear then the type of the new value can be arbitrary, otherwise it must be the same as the type of the previous value, as only linear structs support strong updates. There is also a check that the new value fits in the size of the slot. The only way to read and write a linear entry from a struct is with a **struct.swap** i operation that combines set and get by simultaneously getting and setting a struct cell and therefore ensuring that neither value is dropped or duplicated.

The instruction **variant.case** q (**variant** τ^n) ${\tau_1}^* \to {\tau_2}^*$ $(i, \tau)^*$ $(e^*)^n$ **end** performs case analysis on a variant with type (**variant** τ^n) and, depending on the result, executes one block of instructions from the branch list $(e^*)^n$. The rule expects ${\tau_1}^*$ on the stack and returns ${\tau_2}^*$. If the qualifier of the instruction is linear, then the underlying memory will be freed after the case analysis and the given (linear) reference will not be returned. Each of the instruction blocks $(e^*)_i$ are typed in the updated (in the usual way) function context and are required to have type ${\tau_1}^*\ \tau_i \to {\tau_2}^*$, where τ_i is the type of the ith variant. The unrestricted case is similar, with the distinction that the reference is returned onto the stack after the case analysis and the underlying memory is left intact. In addition, the second element of the F_{linear} component (corresponding to the values outside this case block, but inside the nearest surrounding block) is switched to an arbitrary q' that is stricter than both the qualifier of the variant reference and the previous qualifier of F_{linear}. This ensures that any jumps beyond this case block will need to consider whether the reference we're leaving on the stack is linear, as such a jump would drop it.

Configuration and Store Typing. At the top level we have the typing judgements for stores and program configurations, show in Fig. 8. The spirit is the same as in Wasm, but we have to split the linear store typing across the typing of different components of the configuration. First we introduce an auxiliary judgement $S; (\tau^*)^? \vdash_i v^n; sz^n; e^* : \tau^*$ that asserts that in the store typing S the configuration $s; v^n; sz^n; e^*$ will result in a stack of type (or potentially return) τ^*. The premises require that the local values are well-typed with some types τ_v^n and that the size of each (closed) value fits in the size of the corresponding slot. The instructions have type $\epsilon \to \tau^*$ under the empty function context containing only an optional return type and under the local context $(\tau_v, sz)^n$. The store typing S is the disjoint union of the store typing used across all typing judgements of the premises. Furthermore, we require that in the final local environment there are no linear values, as all of them must be consumed.

The store typing judgement $\vdash s : S_{heap}; S_{prog}$ asserts that the store s has typing $S_{heap}; S_{prog}$. The two store typings S_{heap} and S_{prog} have identical instances and unrestricted memory typings, but disjoint linear typings. The linear typing of S_{prog} contains the surface locations found syntactically in a configration, i.e., the root pointers, while the linear typing of S_{heap} contains the linear locations needed to type the contents of the memory. The rule asserts that the domain of the linear memory coincides with the locations contained in the linear typing of S_{heap} and S_{prog}, and the same for the unrestricted memory. It also asserts that every pair of location and heap value in the unrestricted store has the corresponding type prescribed by the unrestricted memory typing. For the reasons described previously, it also requires that no capabilities are present on the heap. Additionally, S_{heap} must be the disjoint union of all the individual store typing components used in typing the heap components. Lastly, the rule asserts that the length of the module instances $|s_{\mathsf{inst}}|$ must coincide with the length of the instance typings $|S_{\mathsf{inst}}|$ and that each instance in the list has the corresponding instance typing. We elide the instance typing judgement as it is similar to that of Wasm.

The top level judgement $\vdash_i s; v^n; sz^n; e^* : \tau^*$ asserts that the store is well typed in some store typings S_{heap} and S_{prog}, and that the configuration is well typed in the store typing S_{prog}.

4.1 Type Safety

We prove, in Coq, that our language is type safe by proving soundness via progress and preservation.

Progress. If a configuration is well-typed $\vdash_i s; v, sz^*; e^* : \tau^*$, then either e^* are all values, or it is a single **trap** instruction, or the configuration can take a non-GC step $s; v^*; sz^*; e^* \hookrightarrow_j s'; v'^*; e'^*$.

Preservation. If a well-formed and well-typed configuration, $\vdash_i s; v, sz^*; e^* : \tau^*$, takes any step $s; v^*; sz^*; e^* \hookrightarrow_j s'; v'^*; e'^*$ then the resulting configuration is well-typed with the same type $\vdash_i s'; v', sz^*; e'^* : \tau^*$

$$\boxed{S; (\tau^*)^? \vdash_i v^n; sz^n; e^* : \tau^*}$$

$$\frac{S = S_{stack} \uplus S_1 \uplus \ldots \uplus S_n \qquad \forall v_i \in v^n\ sz_i \in sz^n,\ S_i; F_{empty} \vdash v_i : \tau_{v_i} \wedge \|v_i\|_\epsilon \leq sz_i \qquad F = F_{empty}, \text{return}(\tau^*)^? \qquad S_{stack}; S_{\text{inst}}(i); F; (\tau_v, sz)^n \vdash e^* : \epsilon \to \tau^* \mid L' \qquad \forall (p^q, sz) \in L',\ q \preceq_{F_{\text{qual}}} \textbf{unr}}{S; (\tau^*)^? \vdash_i v^n; sz^n; e^* : \tau^*}$$

$$\boxed{\vdash s : S_{heap}; S_{prog}}$$

$$\frac{S = S_{heap} \uplus S_{prog} \qquad dom\ s_{\text{lin}} = dom\ S_{\text{lin}} \qquad dom\ s_{\text{unr}} = dom\ S_{\text{unr}} \qquad s_{\text{unr}} = \{(l_{\textbf{unr}_1}, hv_{\textbf{unr}_1}), \ldots, (l_{\textbf{unr}_n}, hv_{\textbf{unr}_n})\} \qquad s_{\text{lin}} = \{(l_{\textbf{lin}_1}, hv_{\textbf{lin}_1}), \ldots, (l_{\textbf{lin}_m}, hv_{\textbf{lin}_m})\} \qquad \forall\ i \leq\ |S_{\text{unr}}|,\ S_{\text{unr}_i}; F_{empty} \vdash hv_{\textbf{unr}_i} : S_{\text{unr}}(l_{\textbf{unr}_i}) \qquad \forall\ i \leq\ |S_{\text{lin}}|,\ S_{\text{lin}_i}; F_{empty} \vdash hv_{\textbf{lin}_i} : S_{\text{lin}}(l_{\textbf{lin}_i}) \wedge l_{\textbf{lin}_i} \in codom\ s_{\text{unr}} \Rightarrow no\_caps(hv_{\textbf{lin}_i}) \qquad S_{heap} = \biguplus_{i \leq n} S_{\text{unr}_i} \uplus \biguplus_{i \leq m} S_{\text{lin}_i} \qquad |s_{\text{inst}}| = |S_{\text{inst}}| \qquad \forall\ i \leq\ |S_{\text{inst}}|,\ S_{\text{inst}} \vdash s_{\text{inst}}(i) : S_{\text{inst}}(i)}{\vdash s : S_{heap}; S_{prog}}$$

$$\boxed{\vdash_i s; v, sz^*; e^* : \tau^*}$$

$$\frac{\vdash s : S_{heap}; S_{prog} \qquad S_{prog}; \epsilon \vdash_i v^n; sz^n; e^* : \tau^*}{\vdash_i s; v^n; sz^n; e^* : \tau^*}$$

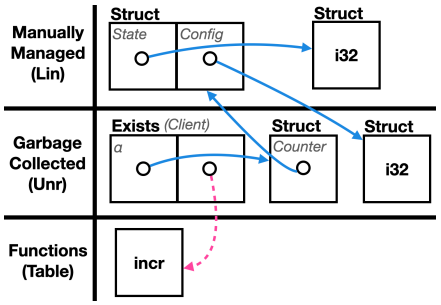
Fig. 8. Configuration typing.

Coq Development. We have formalized the language, its static and dynamic semantics, and the proof of type safety via progress and preservation in Coq. The effort is substantial, consisting of 17k lines of specifications (definitions and theorem statements) and 58k lines of proofs, all directly related to the type system. The proof development is available as an artifact. It contains one remaining admitted lemma related to substitution (among many others that we have fully proved).

4.2 RichWasm Example

We conclude this section with an example of how RichWasm's fine-grained memory sharing might be useful for real programs. Imagine a library for some performance-critical operation implemented in a manually managed language, such as a graphics library. Instances of this graphics data structure might take some mutable configuration state that can change over the course of a program, such as quality settings or dimensions. Next, suppose we want to write the higher-level logic of our program in a GC'd language that uses this graphics library. That would require the GC'd code to reference linear values, which in turn reference some shared mutable state.

For our example, we'll keep this structure, but simplify our library down to a small mutable counter. The shared runtime state will configure how much to increment counters by. The GC'd portion of our program will use this linear library, but hide it behind an interface which allows it to use the library without reasoning about linearity.



$$\text{Client} := (\exists\ \textbf{unr} \preceq \alpha \lesssim 64.\ (\alpha^{\textbf{unr}}\ (\textbf{coderef}\ \forall\ \epsilon.\ \alpha^{\textbf{unr}} \to \textbf{i32}^{\textbf{unr}})^{\textbf{unr}})^{\textbf{unr}})$$

$$\alpha := (\exists \ell.\ (\textbf{ref}\ \ell\ \textbf{rw}\ (\textbf{struct}\ Counter,\ 64))^{\textbf{unr}})^{\textbf{unr}}$$

$$\text{Counter} := (\exists \ell.\ (\textbf{ref}\ \ell\ \textbf{rw}\ (\textbf{struct}\ (State,\ 64)\ (Config,\ 64)))^{\textbf{lin}})^{\textbf{lin}}$$

$$\text{State} := (\exists \ell.\ (\textbf{ref}\ \ell\ \textbf{rw}\ (\textbf{struct}\ \textbf{i32}^{\textbf{unr}},\ 32))^{\textbf{lin}})^{\textbf{lin}}$$

$$\text{Config} := (\exists \ell.\ (\textbf{ref}\ \ell\ \textbf{r}\ (\textbf{struct}\ \textbf{i32}^{\textbf{unr}},\ 32))^{\textbf{unr}})^{\textbf{unr}}$$

Fig. 9. Memory layout. (ref = solid blue line) (coderef = dashed pink line) and type definitions.

Fig. 9 presents a memory layout for such an example. Client is the type of the value that our GC-ed portion will be interacting with. It contains a pair of an abstract value α and a coderef which, given that value, increments the counter and returns the new count. The witness for the hidden type α contains a struct referencing Counter, the main data structure provided by our linear library. Counter points at a struct that contains a pair of references, one granting write access (**rw**) to its internal mutable state (State), and one granting read access (**r**) to the shared configuration (Config). With this heap in place, the GC'd portion of the program can configure and use the counter without reasoning about linearity.

```
function ∀ϵ. α → i32^unr local 32
  get_local 0 unr
  mem.unpack ϵ → ϵ (1, i32^unr) ℓ.
  | (i32.const 0) (struct.malloc 32 unr)
  | mem.unpack ϵ → Config ϵ ℓ.
  |    (ref.demote) (mem.pack ℓ)
  | call 0 ϵ #creates a new Counter
  | struct.swap 0
  | mem.unpack ϵ → Counter (1, i32^unr) ℓ.
  | | struct.get 1
  | | mem.unpack ϵ → ϵ (1, i32^unr) ℓ.
  | | | (struct.get 0) (set_local 1)
  | | | drop
  | | () (struct.swap 0)
  | | mem.unpack ϵ → State ϵ ℓ.
  | | | (struct.get 0) (get_local 1 unr)
  | | | (i32.add) (tee_local 1)
  | | | (struct.set 0) (mem.pack ℓ)
  | | (struct.swap 0) (drop) (mem.pack ℓ)
  | struct.swap 0
  | mem.unpack ϵ → ϵ ϵ ℓ.
  | | () (struct.swap 0)
  | | mem.unpack ϵ → ϵ ϵ ℓ. struct.free
  | | struct.free
  | drop
get_local 1 unr
```

Fig. 10. The `incr` function

Fig. 10 presents the RichWasm code for the `incr` function, which is referenced by the coderef in the Client value and increments the contents of State by the current contents of Config. This function relies on another function of type $\forall\epsilon$. Config → Counter, which creates a fresh Counter. To use the linear data in a Client, `incr` swaps a new Counter for the existing one. Because the GC expects to be pointing at a Counter and linear data cannot be owned by both the stack and the heap, we give the GC a replacement reference temporarily. We could avoid recreating the linear component of the data structure by using a variant type, but we've elected to do something inefficient to keep the example manageably small.

After getting a Counter on the stack, we get the current value of Config and store it in local 1 (see the innermost **mem.unpack** in the left column). Next, we use a strong update to swap a unit value for the State, read the current contents, add the number we stored in local 1, and store the result in both local 1 and in the State reference (see the innermost **mem.unpack** in the right column). The remainder of the function stores the State back in the Counter, swaps the Counter back into its original position, frees the temporary Counter, and returns the int in local 1.

Note that despite working with both manually managed and GC'd memory, it's not immediately apparent from the code where the boundary is because of RichWasm's ability to abstract over management strategies. The full code for this example is available in our artifact.

5 COMPILING ML AND L^3 TO RICHWASM

To demonstrate that RichWasm suffices as a target for very different high-level languages, we implement type-preserving compilers from a garbage-collected language (ML) and a manually managed language (L^3). To exhibit RichWasm's ability to serve as a platform for interop between languages, we extend ML and L^3 with the necessary FFI constructs, discussed in §2.2, to reference each other's types in a limited fashion. These extensions follow a linking-types [18, 19] approach, which aims to allow users of a source language to link with types inexpressible in their own language, without losing native reasoning principles. Programs that do not mention any linking-types extensions are unaffected by the extensions' presence in the language.

ML. Our ML implementation supports the standard types: unit, integers, references, variants, products, recursive types, and functions with parametric polymorphism. Since RichWasm has similar types, the choice of representation is straightforward. We also extend ML with standard constructs for writing multi-module code, such as function imports and exports and the ability to define global state that exported functions can close over. Such extensions provide a good basis for exploring multi-module code in RichWasm and eventually interoperability with other languages.

As discussed in §2.2, we extend ML with the ability to direct the compiler to compile particular types as linear and provide a construct `ref_to_lin` that allows the creation of references containing linear types. The ML type checker explicitly does not check whether types annotated as linear are used linearly, as we can rely on link-time type checking in RichWasm to ensure safety. The goal of interoperability is to allow programmers to use the right language for each task, without burdening them by turning the type system of their source into something as complex as RichWasm. Programmers are still writing ML, only using these linking types at the boundary between languages. In our current implementation, the programmer will see RichWasm errors when violating invariants not checked by their source language, but we think it would be fairly straightforward to tag RichWasm code with some generic source information to improve error messages. Rather than providing an opaque message about qualifiers, we might inform a user that there was a linearity issue with a particular source variable.

L^3. We compile the core L^3 language [13, 2] with a minor adjustment. We require that L^3 capabilities explicitly track the size of the memory they reference. This reflects the fact that here L^3 is a type-safe but C-like low-level source language with precise control over use of memory. While L^3's ability to perform strong updates is powerful, it must be accompanied by reasoning about the size of the location being updated to be useful. As discussed in §2.2, we extend L^3 with an ML-like reference Ref type in order to allow memory interop between the two languages, as well as constructs (join and split) to convert between capability-pointer pairs and references.

Readers familiar with L^3 might note that its source programs often put capabilities on the heap, an operation which is disallowed in the version of RichWasm presented in this paper. We plan to relax this restriction in RichWasm such that capabilities are only disallowed in the parts of the heap owned by the garbage collector. This requires updating our Coq formalism, but our prototype compilers and RichWasm typechecker already support this more relaxed restriction.

Compilation. Compilation to untyped or less strongly typed targets requires compiler writers to expend significant effort on low-level details like data layout, calling conventions, and bit manipulation. In contrast, compilation to RichWasm is largely an exercise in satisfying RichWasm's type system. While this is not entirely straightforward, we have found it to be tractable.

The ML compiler has three phases of note: typed closure conversion, annotation, and code generation. Typed closure conversion is standard in compilers for functional languages. What is notable is that (after the remaining passes) RichWasm is going to check the soundness of these passes, examining the size and qualifier annotations that we use when generating existential packages to hide the environments of closures. Since all type variables in RichWasm have size and qualifier bounds and ML has no such notion at the source, we introduce an annotation phase to change the types of functions appropriately, annotating definitions and call sites with the appropriate information. Code generation is overall quite similar to code generation to any stack-based language. However, there are some type-based complications. For instance, instead of relying on a compiler writer's carefulness when shuffling values around, RichWasm requires explanation of one's use of locals in the form of a local effect annotation on every block of code, allowing it to check that duplication only occurs when allowed.

L^3 is a much lower level language, without the ability to existentially or universally quantify types. Thus, it is much easier to compile to RichWasm, and we can do so in one code generation phase which primarily determines what locals are needed and generates the appropriate type annotations. To keep things simple and since we've already demonstrated how to do so in ML, we don't implement closure conversion in L^3.

5.1 Source-Language Interoperability Examples

```
fun swap_pair : (p : (Int * Int)) =
  let (i1, i2) = p in (i2, i1)
in
import l3.create_cell : (Int → <Ref Int>);
...
```

(a) ML

```
import (ml.swap_pair
  : !(!<!Int * !Int> ⊸ !<!Int * !Int>));
fun create_cell (n : !Int) = join (new n 1) in
let <a, b> = swap_pair !<!1, !2> in
...
```

(b) L^3

Fig. 11. An attempt at interop: without and with linking types

We now present examples to illustrate source-level interoperability between ML and L^3 atop RichWasm. First, to see what goes wrong if programs try to interoperate without the use of linking types, consider Fig. 11, but to start the reader should ignore any text in red. Both `swap_pair`, a function that swaps the positions of two integers in a pair, and `create_cell`, a function that puts a given integer into a fresh manually managed location on the heap, *seem* to superficially match the type at which they're imported. But the L^3 program mistakenly claims that `swap_pair` is a function on linear pairs (if we ignore the code in red), and the ML program mistakenly claims `create_cell` returns a normal ML reference. As a result, neither function import will succeed in RichWasm. A programmer at a language boundary must be aware of these differences and insert the correct annotations (shown in red in Fig. 11). This is where linking types come in. The angle bracket syntax in ML allows the programmer to mark any type (in this case a reference) as linear. For L^3, the programmer can insert !s where necessary to mark types as unrestricted and use the new `join` construct to join capability-pointer pairs into a reference. The versions modified with the red annotations are then accepted by RichWasm.

```
type ravg =
  <Ref <Int * Int * Ref Int>>;
fun get_multiplier (r : Ref Int) =
  !r
in
import l3.create_ravg : (Ref Int → t);
import l3.free_ravg : (ravg → Unit);
import l3.add_sample :
  (ravg Int → ravg);
import l3.average :
  (ravg → <ravg * Int>);
let multiplier = new 2 in
let r = create_ravg multiplier in
let r = add_sample r 4 in
multiplier := 6;
let r = add_sample r 8 in
multiplier := 10;
let r = add_sample r 16 in
let r, average = average r in
free_ravg r;
average (* returns 12 *)
```

(a) ML

```
type ravg = ∃ ℓ.
  Ref ℓ <!Int * !Int * !∃ ℓ. !Ref ℓ !Int 1> 4;
type m = !∃ ℓ. !Ref ℓ !Int 1;
import ml.get_multiplier : !(m ⊸ !Int);
fun create_ravg (m : m) = join (new <!0, !0, m> 4) in
fun free_ravg (r : ravg) = free (split r); !* in
fun add_sample (r : ravg) (sample : !Int) =
  let {ℓ, <cap, ptr>} = split r in
  let <cap, vals> = swap ptr <cap, !*> in
  let <sum, count, m_ref> = vals in
  let <m> = <get_multiplier m_ref> in
  let <cap, *> = swap ptr <cap,
    <sum + (sample * m), (count + m), m_ref>>
  in join {ℓ, <cap, ptr>}
in
fun average (r : ravg) =
  let {ℓ, <cap, ptr>} = split r in
  let <cap, vals> = swap ptr <cap, !*> in
  let <sum, count, m_ref> = vals in
  let <cap, *> = swap ptr <cap, vals> in
  <join {ℓ, <cap, ptr>}, sum / count>
in *
```

(b) L^3

Fig. 12. Computation of rolling averages

Next, let's examine a more substantial example of ML and L^3 interop. Fig. 12 presents two programs: an L^3 program that provides an interface for computing rolling averages and an ML

program that uses that interface to compute a result. The type `ravg` (presented in each of the languages' syntax) is the type of the main data structure, which both languages compile down to the same RichWasm type. It is a reference to a manually managed location containing a total sum, number of samples, and a shared mutable reference representing how much we should believe any new samples. Should the confidence level in new data change, the ML program can freely update its confidence level directly, and all future data points will be adjusted accordingly. If the ML program were to attempt to use `r` after the call to `free_ravg`, that would be caught *statically* by the RichWasm typechecker because it would duplicate a linear value.

Static vs Dynamic Interop Errors. Note that all of the examples we've shown so far have caught all interop errors at compile time. We have, however, also included a construct, namely `ref_to_lin`, in our ML and L^3 FFI that would allow some errors to be caught at runtime. Imagine taking a version of the example in Fig. 3 that type checks as discussed at the end of §2.2, and modifying it so that the function `get_stashed` reads from `c` twice in an attempt to duplicate the linear value inside. Recall from §2.2 that references created with `ref_to_lin`, such as `c`, are compiled as a variant of either a unit value or the linear value we wish to store. In the case where we try to read from an empty reference or write to a reference already containing a value, the ML compiler generates an **unreachable** instruction. In RichWasm and Wasm, **unreachable** typechecks to any valid type, but aborts the program if it is executed, allowing us to defer the error to runtime. Our program in Fig. 12 could easily be written in the `ref_to_lin` style, stashing and retrieving the data structure at every use site, but that seems like a poor choice as it would give us fewer static guarantees. Note that in this paper it was our choice as FFI designers to provide a `ref_to_lin` construct in ML. If we had chosen to omit it, there would be no way to write programs where ML and L^3 interop errors are caught at runtime. The design of more ergonomic source-level FFIs that do not make use of runtime errors deserves thorough investigation as part of future work, but we believe that RichWasm's ability to support more and less principled FFIs (more static error detection vs. more runtime error detection) while guaranteeing safety is a valuable property.

6 COMPILING RICHWASM TO WASM

Compilation from RichWasm to WebAssembly is type directed. It requires some type information that is implicit in RichWasm instructions which is provided by the type checker. The RichWasm to Wasm compiler takes in the type annotated RichWasm code produced by the RichWasm type checker and compiles to WebAssembly 1.0 with the multi-value extension, where functions and instructions can return more than one value.

Lowering RichWasm's Type System. Every RichWasm type will be translated to a series of base Wasm numeric types. Types with no runtime information, such as **unit**, **cap** and **own** , are erased. Numeric types are translated to the corresponding WebAssembly type. **ref** and **ptr** are lowered to a single **i32** pointer. If the size of a pretype α is (transitively) bounded above by a constant, we compile that variable into a series of numeric types. If that size is unbounded, the α is boxed on the heap and it is translated to an **i32** pointer. To translate recursive types, we just compile the inner type, since RichWasm gives us an invariant that the recursive type appears only inside a level of indirection. For existential pretypes on locations, we similarly just lower the inner type.

Operations on Local and Global Variables. Local variables in RichWasm can have arbitrary sizes and can have multiple types over their lifetime. In Wasm, however, local variables can only be one of four numeric types. Therefore, we lower a RichWasm local to a series of Wasm locals. For example, if a RichWasm local has size 160 it will be stored across three Wasm locals of types **i64**, **i64** and **i32**. This sequence of locals might be used to store any type of size up to 160, for example (**i32 i64 i64**). Therefore, the first local will store the first **i32** component of the tuple and the first

half of the second **i64** component. The compilation of **local**.**set** and **local**.**get** needs to perform the correct accesses to fetch the entire value onto the stack, and is informed by the RichWasm type. Operations on global variables in RichWasm are similar to locals.

Memory Model and Heap Types. In Wasm we use only one flat memory to represent both RichWasm's memories. We use a simple free list allocator to allocate and free pointers in Wasm memory. Structs and arrays are encoded in the Wasm memory as a consecutive bytes. Similar to local variables, the representation of a field or an array slot might need to use more than one consecutive memory slot. Variants are represented in memory as a sequence of bytes containing the numeric tag followed by the corresponding type. **variant**.**case** instructions are compiled as a switch case for every case in the variant. Switch cases are represented in Wasm using nested blocks, with blocks for every case, and a **br_table** instruction in the innermost block that jumps to the case being executed. At the start of every case, we provide instructions to read data from the heap according to the type of that case. **exists**.**pack** stores a single RichWasm value on the heap, and **exists**.**unpack** reads it, with the help of an annotation of the type.

Function Calls. Functions in RichWasm can be polymorphic on types with unknown size bounds that are represented in Wasm as **i32** pointers. Say that a caller needs to pass an argument of type (**i64 i32**) to a function that expects a boxed representation of the same argument as it is polymorphic on its type. The caller will put an **i64** and **i32** on the stack. Then, we need to perform a stack coercion, replacing the **i64** and **i32** with a pointer to the same data on the heap. Stack coersions like this will always be required when functions expect or return values of boxed α types.

For indirect calls, **coderef** instructions compile to an **i32** index into the function table. To coerce the stack to the shape that the callee expects, we make a case for each possible shape in the table that could correspond to this call, and at runtime we jump to the correct case depending on the value of the index to the table.

Remaining Instructions. Instructions that have identical counterparts in Wasm are left unaltered. **rec**.**fold** , **rec**.**unfold**, **mem**.**pack** ℓ, **seq**.**group** , **seq**.**ungroup**, **cap**.**split**, **cap**.**join**, **ref**.**demote**, **ref**.**split**, **ref**.**join**, **qualify** , **inst** are all erased since they are type level operations.

7 RELATED WORK

In §1, we've already discussed the three most closely related piece of work: L^3, the Component Model, and Patterson et al.'s semantic framework for sound interoperability. In a general sense, RichWasm is also influenced by work on substructural types, using **unr** and **lin** qualifiers to annotate pretypes as in [1], and by work on type-preserving compilation and typed low-level languages [12, 14, 15, 10].

Wasm does not currently support garbage collection natively, but a proposal to do so is currently working its way through the standardization process [5]. RichWasm's heap types are intentionally designed to be compatible with this proposal, but we believe it lacks one crucial feature: finalizers. Many languages use finalizers, and for RichWasm they are essential to allowing the garbage collector to own, and thus sometimes free, linear memory. At present, with Wasm's current GC proposal, RichWasm's runtime needs to implement its own garbage collector.

MSWasm [11] (Memory-Safe Wasm) is an extension of Wasm designed to enforce memory-safe execution of unsafe code, e.g., code compiled from C or C++. MSWasm extends Wasm with language constructs for CHERI-like [26] fine-grained dynamically checked memory capabilities so code compiled from C will be checked for memory safety at runtime. In contrast, RichWasm has static rather than dynamic capabilities, which have the benefit of zero runtime overhead. But RichWasm is meant to be a target for type-safe source languages; when compiling an unsafe language like C, one would have to do significant static analysis to produce type-annotated RichWasm code that is

well typed since the type information and safety guarantees don't exist in the source. Even then, this analysis could only work on the subset of C programs which are memory safe.

Iris-Wasm [22] is a mechanized higher-order separation logic for modularly verifying Wasm 1.0 programs. Iris-Wasm has been used to build a logical relation for Wasm and prove type safety. By contrast, we have a mechanized type safety proof for a language with a far richer type system, but RichWasm is a typed language not a verification logic.

8 DISCUSSION AND FUTURE WORK

As discussed in §1, this paper describes the RichWasm MVP, which we plan to extend in future work so it can serve as a platform for safe interoperability between a wide range of typed languages. Our first priority in future work is type-preserving compilation from safe Rust to RichWasm. There are two interesting challenges: how to encode lifetime constraints and how to encode immutable borrows in RichWasm. The latter may require fractional capabilities so we can create many borrows, such that when all of these borrows end, we can collect together the fractions and produce a linear capability to return to the owner. A further challenge is to compile Rust with concurrency to a concurrent extension of RichWasm, in turn compiling that to the recent Wasm threads proposal [8].

Next, we will tackle compiling languages with advanced control effects, e.g., algebraic effect handlers. This will require extending RichWasm with linearly typed continuations and compiling RichWasm to Wasm with the recent typed-continuations proposal [3, 21]. The latter dynamically ensures that continuations are used linearly instead of providing static assurance using linear types. The latter are expensive to implement as they preclude in-place updates. In RichWasm, we can statically ensure linear usage of continuations and then have the compiler to Wasm perform the optimization to use in-place updates. We plan to verify correctness of such optimizations using a logical relation.

Wasm provides custom sections and the ability to implement additional type checking by examining annotations in custom sections. We can leverage this to keep RichWasm type information around in Wasm, enabling a rich form of proof-carrying code [17, 16].

Another broad area of future work is designing safe FFIs between practical typed languages that compile to RichWasm. We plan to start with OCaml and Rust and investigate what form of "linking types" or other interoperability mechanisms make sense, but numerous other language pairs would be good candidates for researchers to study. Evaluation of such FFI designs ought to involved user studies to to assess ease of interoperability, as well as measure the annotation and comprehension burden on programmers. There is also a need to assess the ease of building type-preserving compilers for these core-languages-plus-FFIs.

We would like to support safe interop between type-safe languages compiled to RichWasm and unsafe languages such as C and C++ compiled to Wasm. The simplest solution for safe interoperabiltiy is to use Component Model [4] interface types at the boundary between RichWasm (compiled to Wasm) and Wasm modules, ensuring that safe and unsafe code never mix. If we instead want to support memory sharing between RichWasm and Wasm, the interop between more precisely and less precisely typed modules is reminicent of gradual typing [23, 25, 24], but more general since the term languages have differences. This essentially requires tackling gradual typing between a language with linear capability types and one without. We plan to design a RichWasm-Wasm multilanguage, which would have to identify the static guarantees or dynamic checks we need at boundaries; then we will investigate a combination of type inference and static contract verification to eliminate these static and dynamic checks. Interop of RichWasm with MSWasm [11] would be an easier way of achieving the goal since MSWasm already supports dynamic capability checking, but to be performant, MSWasm needs specialized hardware such as CHERI [26].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF awards CCF-1816837 and grant 2030859 to the Computing Research Association for the CIFellows Project) and the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

We would like to thank John Li for his help with our initial formalization of substitution in Coq.

REFERENCES

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. "A Step-Indexed Model of Substructural State". In: *International Conference on Functional Programming (ICFP)*. Sept. 2005, pp. 78–91. https://doi.org/10.1145/1086365.1086376.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. "L3 : A Linear Language with Locations". In: *Fundamenta Informaticae* 77.4 (June 2007), pp. 397–449. https://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06.

[3] WasmFX developers. *Effect handlers for WebAssembly*. 2022. https://wasmfx.dev.

[4] WebAssembly GitHub. *Component Model Design and Specification*. 2022. https://github.com/WebAssembly/component-model/blob/main/design/mvp/Explainer.md.

[5] WebAssembly GitHub. *GC Extension Proposal*. 2021. https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md.

[6] WebAssembly GitHub. *Interface Types Proposal*. 2021. https://github.com/WebAssembly/interface-types/blob/main/proposals/interface-types/Explainer.md.

[7] WebAssembly GitHub. *Multi Memory Proposal for WebAssembly*. 2022. https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md.

[8] WebAssembly GitHub. *Threading Proposal for WebAssembly*. 2022. https://github.com/WebAssembly/threads/blob/main/proposals/threads/Overview.md.

[9] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: PLDI 2017 (June 2017). https://doi.org/10.1145/3062341.3062363.

[10] Chris Hawblitzel et al. "A Garbage-Collecting Typed Assembly Language". In: *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*. Jan. 2007. https://doi.org/10.1145/1190315.1190323.

[11] Alexandra E. Michael et al. "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). https://doi.org/10.1145/3571208.

[12] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. "Typed Closure Conversion". In: *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*. Jan. 1996, pp. 271–283. https://doi.org/10.1145/237721.237791.

[13] Greg Morrisett, Amal Ahmed, and Matthew Fluet. "L3: A Linear Language with Locations". In: (2005), pp. 293–307. https://doi.org/10.1007/11417170_22.

[14] Greg Morrisett et al. "From System F to Typed Assembly Language". In: *ACM Transactions on Programming Languages and Systems* 21.3 (May 1999), pp. 527–568. https://doi.org/10.1145/319301.319345.

[15] Greg Morrisett et al. "Stack-based typed assembly language". In: *Journal of Functional Programming* 12.1 (2002), pp. 43–88. https://doi.org/10.1017/S0956796801004178.

[16] George Necula. "Proof-carrying Code". In: *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*. ACM Press, Jan. 1997, pp. 106–119. https://doi.org/10.1145/263699.263712.

[17] George Necula and Peter Lee. "Safe Kernel Extensions Without Run-Time Checking". In: *Proceedings of Operating System Design and Implementation*. Seattle, Washington, Oct. 1996, pp. 229–243. https://doi.org/10.1145/238721.238781.

[18] Daniel Patterson and Amal Ahmed. "Linking Types for Multi-Language Software: Have Your Cake and Eat It Too". In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. Leibniz International Proceedings in

Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 12:1–12:15. ISBN: 978-3-95977-032-3. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12.

[19] Daniel Patterson, Andrew Wagner, and Amal Ahmed. "Semantic Encapsulation using Linking Types". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2023. , Seattle, WA, USA, Association for Computing Machinery, 2023, pp. 14–28. ISBN: 9798400702990. https://doi.org/10.1145/3609027.3609405.

[20] Daniel Patterson et al. "Semantic Soundness for Language Interoperability". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 609–624. ISBN: 9781450392655. https://doi.org/10.1145/3519939.3523703.

[21] Luna Phipps-Costin et al. "Continuing WebAssembly with Effect Handlers". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 7.OOPSLA (2023). https://doi.org/10.1145/3622814.

[22] Xiaojia Rao et al. "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). https://doi.org/10.1145/3591265.

[23] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *Scheme and Functional Programming Workshop (Scheme)*. Sept. 2006, pp. 81–92.

[24] Asumu Takikawa et al. "Is Sound Gradual Typing Dead?" In: *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*. 2016. https://doi.org/10.1145/2837614.2837630.

[25] Sam Tobin-Hochstadt and Matthias Felleisen. "Interlanguage Migration: From Scripts to Programs". In: *Dynamic Languages Symposium (DLS)*. Oct. 2006, pp. 964–974. https://doi.org/10.1145/1176617.1176755.

[26] Robert N. M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. USA: IEEE Computer Society, 2015, pp. 20–37. ISBN: 9781467369497. https://doi.org/10.1109/SP.2015.9.

Received 2023-11-16; accepted 2024-03-31