Iris-Wasm: Robust and Modular Verification of WebAssembly Programs

XIAOJIA RAO*, Imperial College London, UK
AÏNA LINN GEORGES*, Aarhus University, Denmark
MAXIME LEGOUPIL†, Aarhus University, Denmark
CONRAD WATT, University of Cambridge, UK
JEAN PICHON-PHARABOD, Aarhus University, Denmark
PHILIPPA GARDNER‡, Imperial College London, UK
LARS BIRKEDAL‡, Aarhus University, Denmark

WebAssembly makes it possible to run C/C++ applications on the web with near-native performance. A WebAssembly program is expressed as a collection of higher-order ML-like modules, which are composed together through a system of explicit imports and exports using a host language, enabling a form of higher-order modular programming. We present Iris-Wasm, a mechanized higher-order separation logic building on a specification of Wasm 1.0 mechanized in Coq and the Iris framework. Using Iris-Wasm, we are able to specify and verify individual modules separately, and then compose them modularly in a simple host language featuring the core operations of the WebAssembly JavaScript Interface. Building on Iris-Wasm, we develop a logical relation that enforces robust safety: unknown, adversarial code can only affect other modules through the functions that they explicitly export. Together, the program logic and the logical relation allow us to formally verify functional correctness of WebAssembly programs, even when they invoke and are invoked by unknown code, thereby demonstrating that WebAssembly enforces strong isolation between modules.

CCS Concepts: • **Theory of computation** → **Programming logic**; **Logic and verification**; **Separation logic**; **Higher order logic**; **Formalisms**.

Additional Key Words and Phrases: WebAssembly, separation logic, higher-order logic, formal verification

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

1 INTRODUCTION

WebAssembly (Wasm) is a new bytecode language, supported by all major Web browsers and designed primarily to be an efficient compilation target for low-level languages such as C/C++

*Shared first author.
†This work was carried out while the author was affiliated with École Normale Supérieure, France.
‡Shared senior author.

Authors' addresses: Xiaojia Rao, Imperial College London, UK, xiaojia.rao19@imperial.ac.uk; Aïna Linn Georges, Aarhus University, Denmark, ageorges@cs.au.dk; Maxime Legoupil, Aarhus University, Denmark, maxime@cs.au.dk; Conrad Watt, University of Cambridge, UK, conrad.watt@cl.cam.ac.uk; Jean Pichon-Pharabod, Aarhus University, Denmark, jean.pichon@cs.au.dk; Philippa Gardner, Imperial College London, UK, p.gardner@imperial.ac.uk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

151

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART151
https://doi.org/10.1145/3591265

and Rust. It is officially specified using a formal operational semantics in the W3C Wasm 1.0 standard [Rossberg 2019]. The formal nature of the official Wasm standard and the existence of a well-exercized language mechanization give us a standout opportunity to define a higher-order program logic that covers the full definition of an industrial programming language. We introduce Iris-Wasm, a mechanized higher-order separation logic for Wasm 1.0 which builds on the WasmCert-Coq mechanized specification of the Wasm 1.0 language standard [Watt et al. 2021] and the Iris framework [Jung et al. 2018b, 2015]. In Iris-Wasm, we present an interactive formal verification framework that exactly reflects the Wasm semantics. The result is a semantic and compositional characterization of all Wasm definitions, which can be used to prove separation logic assertions about real Wasm programs, and which lays the foundation for rigorous investigations of the Wasm ecosystem.

A Wasm program is expressed as a collection of higher-order ML-like *modules*, which are composed together through a system of explicit imports and exports. This process of composing Wasm modules into a full program is not performed within Wasm itself. Instead, Wasm is embedded within a *host language*, which provides several important capabilities not available to core Wasm code, including a complex, inherently higher-order, *instantiation* operation in which the declared state of a WebAssembly module is allocated, the module's requested imports are satisfied, and the module's declared exports are registered for use in satisfying further imports requested during subsequent instantiations. The Wasm standard defines instantiation in a host-agnostic way, to be then satisfied by the specific host-language instantiation. For example, a typical Wasm program on the web will involve individual Wasm modules which are instantiated and composed together by a top-level JavaScript host script using the functions of the WebAssembly JavaScript Interface [Ehrenberg 2019].

Iris-Wasm is a higher-order mechanized program logic for the W3C Wasm 1.0 industrial standard using the Iris framework, inspired by a previous Isabelle-mechanized first-order program logic for the language draft [Watt et al. 2019]. Our implementation of the Wasm run-time semantics, with its difficult constructs such as complex control-flow commands, is given directly in Iris, instead of being translated into an existing intermediate Iris language. This choice requires considerable Iris engineering, but provides more trust in our mechanization, as it is line-by-line close to the Wasm semantics, and should lead to the mechanization being comparatively straightforward to extend as the standard expands. We make a minor reformulation of the host function semantics (see §2.2.3), so that our core Wasm semantics and program logic are properly separate from the host.

We provide a host-agnostic axiomatic characterization of Wasm module instantiation by establishing a lemma which lifts the complex W3C Wasm 1.0 instantiation predicate to our Iris-Wasm logic, describing the state before and after instantiation using our logical assertions. We illustrate this instantiation lemma on a simple host language designed to capture the core functionality of the WebAssembly JavaScript Interface [Ehrenberg 2019], and corresponding host program logic, where the soundness of our host instantiation proof rule is established using our instantiation lemma. The Iris-Wasm program logic thus gives a semantic characterization of the host-agnostic instantiation operation.

By capturing the semantics of the full Wasm 1.0 industrial standard directly, Iris-Wasm lays the groundwork for a wide range of future analyses. Iris-Wasm can be used to validate proposed extensions to Wasm such as MSWasm, a memory safe extension of Wasm [Michael et al. 2023]. It can be used to rigorously investigate compilers that either target Wasm or compile Wasm down to some low-level assembly language. Jacobs et al. [2022] demonstrate that Iris can be a useful tool to prove results such as full abstraction. Iris-Wasm sets the groundwork for similar results for realistic compilers involving Wasm.

We demonstrate our compositional higher-order reasoning about Wasm modules in our host language by developing a series of examples. Our main running example is a higher-order stack example comprising a stack module and a client module. The stack module defines and exports stack functions, including a higher-order map function for the stack. The client module imports and uses some of them, including map, in its main function. Using our Wasm program logic and a program logic for the simple host we introduce, we provide specifications for both modules: the stack module's specification contains specifications for all the stack functions, and the client module's specification depends on the stack module's specification. Finally, we verify a host program which instantiates the two modules in sequence, by modularly combining the proofs for the two module specifications. In addition, we demonstrate how to reason about *reentrancy* between the host and Wasm, by having the client module invoke a host function to modify the function table to provide a different input function for subsequent applications of map. The higher-order reasoning of the Iris framework provides an ideal environment to reason about Wasm modules. Nevertheless, it's a substantial task to apply Iris to a true industrial standard. Our implementation precisely follows the design decisions of the W3C Wasm 1.0 standard, and by using a rich logic such as Iris, we have laid the foundations for deep semantic investigations of WebAssembly and its future iterations.

In a case study, we investigate the intuitive coarse-grained encapsulation property of Wasm modules, stated in the standard: 'code from a module can arbitrarily affect its own state, but can only access the state of another module through the module's exports'. Several systems rely on this important property of Wasm to provide a form of sandboxing: for example, Fastly's 'Compute@Edge' [Hickey 2020] platform and the RLBox tool [Narayan et al. 2020]. Both depend on the encapsulation property of a module, regardless of behaviour of other modules, which are validated but not necessarily trusted. Reasoning about such modules necessarily involves the interaction between the known, verified code of one module against unknown, untrusted, and unverified code from other modules, something that cannot be done with a program logic. Building on top of Iris-Wasm, we define a relational interpretation of WebAssembly types through a unary logical relation, which is then used to verify specific *robust safety* properties of a known module, that hold even when composed with unknown modules. We demonstrate this by proving robust safety properties of our stack module composed with arbitrary clients. Our relational interpretation is entirely host agnostic, and can modularly be applied to any host language.

In summary, our contributions are:

(1) Iris-Wasm, a Coq-mechanized higher-order program logic for the Wasm run-time semantics.
(2) A host-agnostic module instantiation lemma, and a program logic for a simple example host language with the specific host instantiation rule proved using our general instantiation lemma.
(3) A semantic interpretation of the Wasm type system, defined via a logical relations interpretation using our Wasm program logic.
(4) Illustrative examples and case studies that demonstrate the expressiveness of Iris-Wasm; we show that an implementation of a higher-order stack module satisfies a very modular abstract specification; we verify a reentrant module that uses host language features to modify function tables dynamically; and we use Iris-Wasm to define and prove the properties of our logical relation, which we use to verify robust safety of higher-order examples.

All results, including soundness of the program logic and logical relations, are formalized in Coq. We hope this will prove useful to other researchers for further investigating the Wasm ecosystem.

Higher-order programming in WebAssembly and reentrancy. Consider the WebAssembly snippet in Figure 1, which contains a module that works as a library implementing a stack of **i32**s (on the left), and a module that works as a client of that library (on the right). The library module, which the host language calls "stack" here, uses a **memory** (with initial size 0; some other function is

```
stack_module ≜
(module ;; "stack"
  (type $t1 (func (param i32) (result i32)))
  (table (export "tab1") 3 funcref)
  (memory 0)
  (func (export "map")
        (param $i i32) (param $stk i32)
   ...
   loop
    ...
    local.get $i
    call_indirect $t1
    ...
   end ...)
```

```
client_module ≜
(module ;; "client"
  (import "stack" "tab1" (table 3 funcref))
  (import "stack" "map"
   (func $map (param i32 i32)))
  (elem (i32.const 0) $f0 $f1 $f2)
  (func $f0 (param $n i32) (result i32)
   ...)
  (func (export "main")
        (param $stk i32) (result i32)
   i32.const 0
   local.get $stk
   call $map
   ... ;; Rest of the code))
```

Fig. 1. A module implementing a stack library, and a client module. Module boundaries enforce isolation. This example uses the Wasm text format; below, we work directly with the AST.

in charge of allocating space for the stack) to implement a stack. The "stack" module exports a "map" function that maps a function over a stack. However, because WebAssembly is a first-order language, "map" does not take the function to map as an argument. Instead, "map" takes as argument an index, $i, into a table of 3 functions, "tab1", that this module creates and exports, and calls the function at that index in the table using **call_indirect**. The client module imports the same shared table of functions, and uses the **elem** directive to populate it (from offset 0) with functions it defines: $f0, $f1, and $f2. It also imports the "map" function from the "stack" module as $map, and its "main" function then calls the $map function with function index 0 as argument, which makes it map $f0 on the stack.

In §2 we describe our program logic and we show in §2.2 how it can be used to give a modular specification of the stack module, and, in particular, in §2.3, the "map" function. A proof of the specification of the *instantiation* of the stack module is given at the end of §3. We emphasize that our logic supports verification of the client module relative to an abstract logical specification of the stack module; in other words, the encapsulation of the internal representation of the stack module is reflected in its specification.

We now consider a simple extension of this example to demonstrate the need for reasoning about reentrancy between WebAssembly and the host. To this end, we will let the "main" function, after the call to $map, dynamically modify the contents of the table to now contain a new function $f3 at index 0. Dynamic modification of the table cannot be performed in WebAssembly 1.0, as WebAssembly only has the **elem** directive available to statically provide an initial value for the elements of the table. WebAssembly code can, however, call functions defined by the host, and those may modify the state of the WebAssembly program. Thus we add an import (**import** "host" "mut" (**func** $mut (**param** **i32** **i32**))) to the preamble of the client module and then complete the code of the "main" function with 6 more instructions: **i32.const** 0; **i32.const** $f3; **call** $mut; **i32.const** 0; **local.get** $stk; **call** $map. The first three of these call the host function $mut that we assume will modify the function table at address 0, replacing the previous value ($f0) by $f3. The last three instructions are a call to $map identical to the one at the beginning of the body of "main" function (see Figure 1), but this time, when mapping the 0th function from the table onto the stack, it maps function $f3 instead of $f0 like it did during the first call to $map. Thus calling "main" on a value that represents stack $[x_0, \ldots, x_n]$ will modify the stack so that the argument value now represents $[f_3(f_0(x_0)), \ldots, f_3(f_0(x_n))]$.

$$
\begin{array}{rll}
\text{(value type) } t & ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} & \text{(function type) } \mathit{ft} ::= \mathit{ts} \rightarrow \mathit{ts} \\
\text{(value) } v & ::= t.\mathbf{const}\ c & \text{(immediate) } i, \mathit{min}, \mathit{max} ::= \mathit{nat}
\end{array}
$$

$$
\begin{array}{rl}
\text{(instructions) } e ::= & v \mid t.\mathbf{add} \mid \mathit{other\ stackops} \mid \mathbf{local}.\{\mathbf{get/set}\}\ i \mid \mathbf{global}.\{\mathbf{get/set}\}\ i \mid t.\mathbf{load}\ \mathit{flags} \mid \\
& t.\mathbf{store}\ \mathit{flags} \mid \mathbf{memory.size} \mid \mathbf{memory.grow} \mid \mathbf{block}\ \mathit{ft}\ \mathit{es} \mid \mathbf{loop}\ \mathit{ft}\ \mathit{es} \mid \mathbf{if}\ \mathit{ft}\ \mathit{es}\ \mathit{es} \mid \\
& \mathbf{br}\ i \mid \mathbf{br_if}\ i \mid \mathbf{br_table}\ \mathit{is} \mid \mathbf{call}\ i \mid \mathbf{call_indirect}\ i \mid \mathbf{return}
\end{array}
$$

$$
\begin{array}{rll}
\text{(functions) } \mathit{func} & ::= \mathbf{func}\ i\ \mathit{ts}\ \mathit{es} & \text{(tables) } \mathit{tab} ::= \mathbf{tab}\ \mathit{min}\ \mathit{max} \\
\text{(memories) } \mathit{mem} & ::= \mathbf{mem}\ \mathit{min}\ \mathit{max} & \text{(globals) } \mathit{glob} ::= \mathbf{glob}\ \text{mutable}\ t\ e_{\text{init}} \\
\text{(elem segments) } \mathit{elem} & ::= \mathbf{elem}\ i\ \mathit{es}_{\text{off}}\ \mathit{is} & \text{(data segments) } \mathit{data} ::= \mathbf{data}\ i\ \mathit{es}_{\text{off}}\ \mathit{bytes}
\end{array}
$$

$$
\begin{array}{rl}
\text{(import descriptions) } \mathit{importdesc} & ::= \mathbf{func_i}\ i \mid \mathbf{tab_i}\ \mathit{min}\ \mathit{max} \mid \mathbf{mem_i}\ \mathit{min}\ \mathit{max} \mid \mathbf{glob_i}\ \text{mutable}^?\ t \\
\text{(imports) } \mathit{import} & ::= \mathbf{import}\ \mathit{string}\ \mathit{string}\ \mathit{importdesc} \\
\text{(export descriptions) } \mathit{exportdesc} & ::= \mathbf{func_e}\ i \mid \mathbf{tab_e}\ i \mid \mathbf{mem_e}\ i \mid \mathbf{glob_e}\ i \\
\text{(exports) } \mathit{export} & ::= \mathbf{export}\ \mathit{string}\ \mathit{exportdesc} \\
\text{(start) } \mathit{start} & ::= \mathbf{Some}\ i \mid \mathbf{None}
\end{array}
$$

$$
\begin{array}{rl}
\text{(function instances) } \mathit{finst} & ::= \{(\mathit{inst}; \mathit{ts}); \mathit{es}\}_{\mathit{tf}}^{\text{NativeCl}} \mid \{\mathit{hidx}\}_{\mathit{tf}}^{\text{HostCl}} \\
\text{(table instances) } \mathit{tinst} & ::= \{\text{elem} : \mathit{is},\ \ \text{max} : \mathit{max}^?\} \\
\text{(memory instance) } \mathit{minst} & ::= \{\text{data} : \mathit{bytes},\ \ \text{max} : \mathit{max}^?\} \\
\text{(global instance) } \mathit{ginst} & ::= \{\text{mut} : \mathit{mutable}^?,\ \ \text{value} : v\} \\
\text{(store) } S & ::= \{\text{funcs} : \mathit{finsts},\ \ \text{globs} : \mathit{ginsts},\ \ \text{mems} : \mathit{minsts},\ \ \text{tabs} : \mathit{tinsts}\} \\
\text{(frame) } F & ::= \{\text{locs} : \mathit{vs},\ \ \text{inst} : \mathit{inst}\} \\
\text{(module instance) } \mathit{inst} & ::= \{\text{types} : \mathit{fts},\ \ \text{funcs} : \mathit{is},\ \ \text{globs} : \mathit{is},\ \ \text{mems} : \mathit{is},\ \ \text{tabs} : \mathit{is}\}
\end{array}
$$

$$
\text{(modules) } m ::= \left\{ \begin{array}{l} \text{types} : \mathit{fts},\ \text{funcs} : \mathit{funcs},\ \text{globs} : \mathit{globs},\ \text{mems} : \mathit{mems},\ \text{tabs} : \mathit{tabs}, \\ \text{data} : \mathit{datas},\ \text{elem} : \mathit{elems},\ \text{imports} : \mathit{imports},\ \text{exports} : \mathit{exports},\ \text{start} : \mathit{start} \end{array} \right\}
$$

Fig. 2. WebAssembly 1.0 Abstract Syntax

This example illustrates how programs may take advantage of the stronger expressive power of the host. In §2.2, we show how we deal with calls to host functions in Iris-Wasm, and in §3, we introduce a simple host language and a program logic for it and show how it can be used in combination with our WebAssembly program logic to reason about complex interaction between WebAssembly code and the host language code that embeds it, including this example.

2 MODULAR REASONING FOR WEBASSEMBLY MODULES

In this section, we introduce Iris-Wasm. We present our proof rules for WebAssembly language features, and outline how they are used to prove a specification for the stack module from the Introduction. For reasons of space, we only discuss selected proof rules; we stress that we have proved program logic rules for *all* of WebAssembly and used them to give full formal proofs of examples, including the stack module; see the accompanying Coq formalization for details. Then, in §3, we present the operational semantics and proof rules for our host language, and show how they are used to verify the interaction of a client module with the stack module; we focus on instantiation and reentrancy. Finally, in §4 we discuss how our program logic is defined within the Iris program logic framework, we overview some of the generic features and proof rules we inherit from Iris, and we state the soundness and adequacy of Iris-Wasm.

2.1 Proof Rules for Basic WebAssembly Stack Operations

WebAssembly is a stack language with structured control. Its dynamics is specified by a small-step operational semantics on *configuration tuples* of the form $(S; F; es)$, where es is a hybrid stack of values and instructions,[1] S is the global *store*, and F is the current function *frame*. The store S

[1]The standard uses '*' to stand for 'a list of', but we prefer using s as a suffix to avoid confusion with the symbol for separating conjuction, so 'es' is a list of 'e's, 'vs' is a list of 'v's, etc.

contains information about the global variables, the tables, the memories and the functions declared in all modules instantiated thus far, and the frame F contains the values of all local variables, as well as an *instance* that handles indirection, as will be explained progressively below. We recall the abstract syntax in Figure 2.

Reductions are structural: for any program fragment[2] es that reduces to es', the same reduction can occur under a context; for example, for any list vs of constants and es_2 of expressions, $vs +\!\!+\ es +\!\!+\ es_2$ reduces to $vs +\!\!+\ es' +\!\!+\ es_2$. We give the general meaning of contexts in §2.2.

The overall structure of the operational semantics is as expected for a stack language; for example, the stack $[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ binop]$ reduces to $[t.\mathbf{const}\ c]$, where c is the result of applying $binop$ to c_1 and c_2. Let us introduce the corresponding proof rule in our program logic.

Weakest preconditions. Our proof rules are phrased using Iris' *weakest precondition*. Intuitively, $\mathsf{wp}\ es\ \{w, \Phi(w)\}$ states that the program fragment es computes safely, and, if it terminates with result w, predicate Φ holds of w (we discuss the formal meta-theory in §4). This construct is close to Hoare triples, as we have the following equality in Iris[3]:

$$\{P\}\ es\ \{w, \Phi(w)\} = \Box(P -\!\!* \ \mathsf{wp}\ es\ \{w, \Phi(w)\})$$

Logical values. Because we reason about *fragments* of WebAssembly programs, execution does not always terminate with a stack of WebAssembly values, but more generally with a *logical value*:

$$\mathit{LogVal} \ni w ::= \mathbf{immV}\ vs \mid \mathbf{trapV} \mid \mathbf{brV}\ i\ vh_i \mid \mathbf{retV}\ lh_k \mid \mathbf{call_hostV}\ tf\ hidx\ vs\ llh$$

which is one of the following:

- **immV** vs, the 'normal' result: a stack of WebAssembly values;
- a trap **trapV**, which represents that the program has encountered an error in its execution;
- a break (or branching) value **brV**, a return value **retV**, or a host call value **call_hostV**, which correspond to program fragments that are stuck as such, but can get unstuck when placed in an appropriate context; we explain their meaning, and the meaning of their arguments, in §2.2.

Accordingly, in our proof rules, the postcondition Φ takes a logical value w as an argument.

Proof rule. We prove the following Iris-Wasm proof rule for binary operators:

wp_binop

$$\frac{[\![t.binop]\!](c_1, c_2) = c \ * \ \triangleright\Phi(\mathbf{immV}\ [t.\mathbf{const}\ c]) \ * \ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ binop]\ \left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

which states that, with two constants $t.\mathbf{const}\ c_1$ and $t.\mathbf{const}\ c_2$ on the value stack, and any function frame F, if an arbitrary predicate Φ holds *later* of the result c of the binop of type t on c_1 and c_2, then this program fragment executes safely, and if it terminates (which it does in this case), Φ holds of the execution result w, because it will be the value stack **immV** $[t.\mathbf{const}\ c]$. The frame resource is a special resource which needs to be included in every proof rule where we 'take a reduction step'.

We merely require that Φ holds after one step of execution, as expressed by the later $\triangleright$ modality of Iris [Jung et al. 2018b]. One may choose to ignore this, but it is necessary in the presence of Iris' higher-order features, to avoid cyclicity.

[2]For simplicity, in this paper, we conflate what WebAssembly calls 'basic instructions' and 'administrative instructions'; see beginning of §2.2.

[3]The persistent modality $\Box$ indicates that the Hoare triple is a proposition that can be duplicated as many times as needed.

2.2 Control and Function Calls

Control and function calls in WebAssembly are intricate, but still feature locality, as expected; for example, blocks can be reasoned about in isolation, and function scope is still respected. We present an approach that allows us to reason about code fragments without needing knowledge of their environment; it improves over the approach taken in the earlier Wasm program logic [Watt et al. 2019] which does not scale to higher-order programs. In this section, we show how our rules capture this locality to make reasoning tractable.

2.2.1 Administrative Instructions. To define reduction of blocks and functions calls, WebAssembly adds an extra layer on top of the surface language, to represent intermediate states by *administrative instructions*, which are defined by the following grammar:

$$AI \ ::= \ \mathbf{basic}\ e \mid \mathbf{trap} \mid \mathbf{invoke}\ i \mid \mathbf{label}_i\{es\}\ es\ \mathbf{end} \mid \mathbf{local}_i\{F\}\ es\ \mathbf{end} \mid \mathbf{call_host}\ tf\ hidx\ vs$$

- A **basic** instruction is a plain WebAssembly expression, as described in Figure 2. When clear from the context, we conflate **basic** e and e, for example in weakest preconditions.
- A **trap** represents a program that has encountered an error in its dynamic execution.
- An **invoke** represents an intermediate step when reducing a **call** or **call_indirect** .
- A **label** represents a block or a loop that is being executed.
- A **local** represents a function call that is being executed.
- A **call_host** represents a program that performs a call to a function defined the host language.

We discuss the last four kinds of administrative instructions below, as we describe control flow and function calls in WebAssembly.

2.2.2 Blocks, Labels, and Breaks. WebAssembly is somewhat unusual as an assembly-like language in that it features only structured control, including labeled breaks. We show how we use the higher-order nature of Iris to ease reasoning about the control structure of WebAssembly.

WebAssembly has (aside from function calls) two core constructs for control flow: **block**, and **loop** (and the conditional **if**, which reduces immediately to a **block**). These take as arguments a function type, and a list of expressions constituting the body of the **block** or **loop**. This body will reduce until either it becomes a list of constants and the **block** or **loop** is exited, or a **br** instruction is its first non-constant instruction. In a **block**, the body is then exited, and execution continues with whatever follows the block; and in a **loop**, the full original body of the **loop** is repeated from the beginning. The function type $ts_1 \to ts_2$ describes the $|ts_1|$ values[4] needed to enter the **block** or **loop**, and the $|ts_2|$ values that need to be on the stack if a **br** is encountered.

Because of the similarity between these two constructs, the WebAssembly semantics has them both reduce to a **label** administrative instruction. $\mathbf{label}_n\{es_{\text{cont}}\}\ es_{\text{body}}\ \mathbf{end}$ is a label with body es_{body} that will execute continuation expression es_{cont} if it encounters a **br** instruction preceeded by n values. We come back later to the exact semantics of **br**. When preceeded with $|ts_1|$ values vs of the right type, $\mathbf{block}\ (ts_1 \to ts_2)\ es$ reduces to $\mathbf{label}_{|ts_2|}\{[]\}\ vs +\!\!+\ es\ \mathbf{end}$ and $\mathbf{loop}\ (ts_1 \to ts_2)\ es$ reduces to $\mathbf{label}_{|ts_1|}\{[\mathbf{loop}\ (ts_1 \to ts_2)\ es]\}\ vs +\!\!+\ es\ \mathbf{end}$.

Once the **block** or **loop** instruction has been reduced to a **label**, reduction steps can be taken in the body of the **label**. As this may happen under many nested labels, WebAssembly defines evaluation contexts lh_k, which describe stack environments consisting of k nested *labels* surrounding a *hole* [_] where the next step of execution takes place:

$$lh_0 ::= vs +\!\!+\ [_] +\!\!+\ es \qquad lh_{k+1} ::= vs +\!\!+\ \mathbf{label}_n\{es_{cont}\}\ lh_k\ \mathbf{end} +\!\!+\ es$$

[4] In WebAssembly 1.0, ts_1 is always empty.

Note how only (constant) values *vs* can be on the left of the hole and label instructions: this enforces that we can only 'zoom in' on the next expression to reduce.

As expected, steps can be taken under an evaluation context: if *es* reduces to *es'*, then $lh_k[es]$ reduces to $lh_k[es']$. Taking $k = 0$ yields the expected sequencing rule mentioned at the start of §2.1.

Correspondingly, we prove the following Iris-Wasm rule, which reduces reasoning about a program fragment that can be decomposed as $lh_i[es]$ to reasoning about $lh_i[vs]$, that is, the result *vs* of evaluating the expression to a list of constants, placed in the evaluation context.[5]

`wp_ctx_bind`

$$\frac{\mathsf{wp}\ es\ \{w, \mathsf{wp}\ lh_i[w]\ \{w', \Phi(w')\}\}}{\mathsf{wp}\ lh_i[es]\ \{w', \Phi(w')\}}$$

This rule leverages the fact that in Iris, weakest preconditions are propositions themselves, and can therefore be nested. Notice how we have implicitly cast w, a logical value, into an expression when plugging it into lh_i. This is done in the intuitive way: **immV** *vs* is cast into *vs*, **trapV** is cast into the single administrative instruction [**trap**], etc.

While control flow in WebAssembly is structured, the presence of labelled breaks makes it slightly involved. A break targets a particular level of the evaluation context, and skips the rest. As a result, the default evaluation context rules provided by Iris are inadequate, and we have to build our own reasoning principles for contexts.

The **br** i instruction targets the ith label from the context. Crucially, breaking relies on the instruction **br** i being in an evaluation context lh_k with $i = k$: the break index indicates what context depth is targeted. If $i > k$, the expression $lh_k[\mathbf{br}\ i]$ is stuck and can only reduce if placed in a deeper context. Correspondingly, we introduce a new type of logical values: **brV** i vh_i, representing the program fragment $vh_i[\mathbf{br}\ i]$. The *breaking context* vh_i is similar to an evaluation context lh_i, except that the meaning of the subscript i is that the context has depth *at most* i, instead of exactly i. If $i < k$, a **br** i nested in context lh_k will only break out of the i first **label**s, and the result will be in the form $lh_{k-i}[vs \mathbin{+\!\!+} es]$. The break value **brV** allows to bind into any number of **label**s without needing to worry about getting stuck at a **br** i statement: when encountering such a statement, we simply bind back $i + 1$ times to get a wp in a form where our rule for **br** can be applied.

2.2.3 Functions. There are two ways to call a function in WebAssembly: statically with **call**, or by dynamically fetching a function from a table, with **call_indirect**. We focus on the simpler direct **call** here, and explain **call_indirect** in §2.3.

The instruction **call** n calls the nth function declared in the current module. Indexing starts at 0 with the imported functions, followed by the functions defined in the module itself. The store S keeps a list of the *function closures* (which we describe below) of *all* the instantiated modules. This means the nth function in the current module will not always be the nth function in the store: the *instance* in the function frame F is in charge of remembering that indirection. The instance also contains this indirection information for global variables, memories, and tables.

A **call** i retrieves the address *addri* of the relevant closure in the store from the frame's instance, and reduces to **invoke** *addri*. We prove the corresponding Iris-Wasm rule:

`wp_call`

$$\frac{(F.\mathsf{inst.funcs}[i] = addri) * \xhookrightarrow{\text{FR}} F * \triangleright\left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\ [\mathbf{invoke}\ addri]\ \{w, \Phi(w)\}\right)}{\mathsf{wp}\ [\mathbf{call}\ i]\ \{w, \Phi(w)\}}$$

[5]The version we show here is meant for evaluation contexts with at least one label constructor; in our Coq formalization, we prove more intricate variations of this rule, to be applied for sequencing, with for instance $lh_i[es]$ replaced with $lh_i[es_1 \mathbin{+\!\!+} es_2]$.

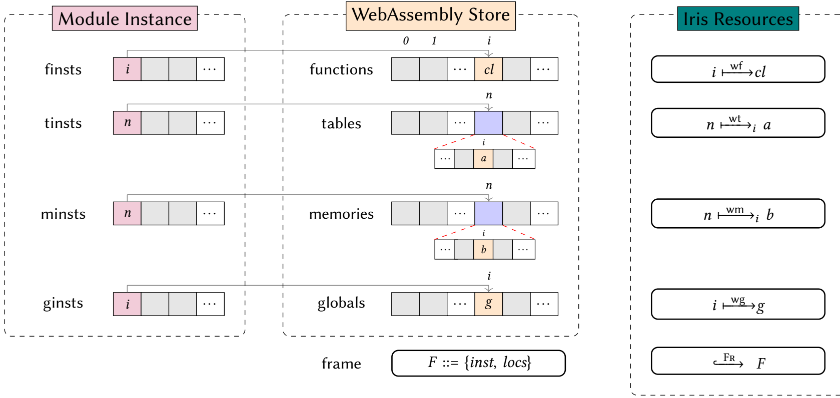

Fig. 3. Points-to predicates for the store and the frame

which requires ownership of the frame, not only because we are taking a reduction step, but also to know where to look up index *addri*.

The function closures *cl* (also called function instances *finst* in Figure 2) stored in the store S are of two kinds: native and host. Let us focus first on native closures, and come back to host closures at the end of this section. The closure $\{(inst; ts); es\}^{\text{NativeCl}}_{ts_1 \to ts_2}$ describes a *native* function that was defined in a WebAssembly module with instance *inst* (this is the environment for the closure), which expects arguments of type ts_1, defines additional local variables of type *ts* for the computation of its body, yields results of type ts_2, and has body *es*. When reducing **invoke**, we look up the closure in the store, and check that the stack contains the appropriate number of values to be passed as parameters to the function. If the closure is native, **invoke** is replaced with the body of the function. In order to properly encapsulate the function call, WebAssembly places the function body inside a **local** administrative instruction, and inside a **block**, as captured by the following Iris-Wasm proof rule (we say more about **local** and the meaning of F' further down):

wp_invoke_native

$$\frac{\begin{array}{c}|vs| = |ts_1| * cl = \{(inst; ts); es\}^{\text{NativeCl}}_{(ts_1 \to ts_2)} * F' = \{\text{locs} := vs +\!\!+ \textbf{zeros}(ts); \text{inst} := inst\} * \\ i \xmapsto{\text{wf}} cl * \xhookrightarrow{\text{Fr}} F * \triangleright \left[\begin{array}{c}(i \xmapsto{\text{wf}} cl * \xhookrightarrow{\text{Fr}} F) -\!\!* \\ \text{wp } \left[\textbf{local}_{|ts_2|}\{F'\}\ (\textbf{block}\ ([] \to ts_2)\ es)\ \textbf{end}\right]\ \{w, \Phi(w)\}\end{array}\right]\end{array}}{\text{wp } (vs +\!\!+ \textbf{invoke}\ i)\ \{w, \Phi(w)\}}$$

Unlike for the function frame F, we do not assert ownership of the whole store S. Instead, we rely on points-to predicates to assert ownership of specific components: for instance, the predicate $i \xmapsto{\text{wf}} cl$ asserts ownership of S.funcs$[i]$ in the store.

In general, we define points-to predicates for each component of the Wasm store. Fig. 3 illustrates all the points-to predicates used in this paper, and how they relate to the physical Wasm store. Functions and globals are referred to directly via their indices, while function tables and linear memories can be viewed as two dimensional structures, where an index is used to refer to a particular table or memory, and another index is used to refer to a particular cell within that table or memory. For example, $n \xmapsto{\text{wm}}_i b$ asserts that the ith byte of memory n is b. The WebAssembly frame F tracks the scope of the currently executing function, namely its enclosing instance and local variables. The enclosing instance collects indices of all the entities of the Wasm store that the module may access, and is crucial for enforcing the encapsulation properties of Wasm modules.

Encapsulation. Let us return to why the function body is placed inside a **local** and inside a **block**. The first of these is to provide proper encapsulation, as reduction of an expression nested in a **local** takes place with respect to the nested frame of the **local**: when reducing $[\mathbf{local}_n\{F_1\}\ es\ \mathbf{end}]$, one reduces es with respect to frame F_1 rather than the current function frame F.

For our native invocation, the frame used will be F'. Note that the `inst` field of F' is the instance that was declared in the closure (to enforce static scoping), and that the local variables in F' are the function parameters from the stack, followed by a list of zeros corresponding to the types of local variables required by the function. We prove the corresponding proof rule for `local`:

`wp_local_bind`

$$\frac{\xhookrightarrow{\text{FR}} F * \left(\xhookrightarrow{\text{FR}} F_1 \twoheadrightarrow * \ \text{wp}\ es \left\{w, \exists F_1', \xhookrightarrow{\text{FR}} F_1' * \left(\xhookrightarrow{\text{FR}} F \twoheadrightarrow * \ \text{wp}\ [\mathbf{local}_n\{F_1'\}\ w\ \mathbf{end}]\ \{w', \Phi(w')\}\right)\right\}\right)}{\text{wp}\ [\mathbf{local}_n\{F_1\}\ es\ \mathbf{end}]\ \{w', \Phi(w')\}}$$

which is reminiscent of `wp_ctx_bind`; the only reason this rule looks like more of a mouthful, is that the frame changes. As discussed above, this frame change is necessary for proper encapsulation.

Finally, the reason WebAssembly puts the function body in a **block** is to allow the function body to contain a **br** (with the right index) to exit the function-body's execution. Alternatively, a **return** instruction will work like a **br**, but target the closest **local** instruction. The **return** instruction also has an associated logical value **retV** lh_k, representing the expression $lh_k[\mathbf{return}]$.

Example. Consider the increment function with body $es_{incr} = [\mathbf{i32.local.get}\ 0; \mathbf{i32.const}\ 1; \mathbf{i32.add}]$ of type $[\mathbf{i32}] \to [\mathbf{i32}]$. We show that calling it on input 3 returns 4.

Define es as $[\mathbf{i32.const}\ 3; \mathbf{call}\ \$incr]$, and let $F.\text{inst.funcs}[\$incr] = i$. We prove that

$$i \overset{\text{wf}}{\longmapsto} \{(inst; []); es_{incr}\}^{\text{NativeCl}}_{[\text{i32}]\to[\text{i32}]} \twoheadrightarrow * \xhookrightarrow{\text{FR}} F \twoheadrightarrow * \ \text{wp}\ es\ \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\}$$

Here, the first precondition asserts that we know that function number i in the store is the increment function (we denote by $inst$ the instance of the module where the increment function was defined), and the second precondition is ownership of the frame F.

We introduce the two preconditions by moving them to a proof environment Γ. For the first step of derivation, we apply the `wp_call` rule[6]. To fulfill the premises of the `wp_call` rule, the resource $\xhookrightarrow{\text{FR}} F$ from Γ is consumed, and it remains to prove

$$\triangleright(\xhookrightarrow{\text{FR}} F \twoheadrightarrow * \ \text{wp}\ [\mathbf{i32.const}\ 3; \mathbf{invoke}\ i]\ \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\})$$

Now we introduce the $\triangleright$, move the frame resource back to our proof environment Γ, and are left with a new weakest precondition to prove. This first proof step corresponds to the bottom-most rule of the following simplified proof-tree:

$$\Gamma \vdash \mathbf{immV}\ [\text{i32.const}\ 4] = \mathbf{immV}\ [\text{i32.const}\ 4]$$
`wp_local_value`
$$\Gamma \vdash \text{wp}\ [\mathbf{local}_1\{F_1'\}\ [\text{i32.const}\ 4]\ \mathbf{end}]\ \{w, w = \mathbf{immV}\ [\text{i32.const}\ 4]\}$$
`wp_label_value`
$$\Gamma' \vdash \text{wp}\ [\mathbf{label}_1\{[]\}\ [\text{i32.const}\ 4]\ \mathbf{end}]\ \{w, \Phi(w)\}$$
`wp_binop`
$$\Gamma' \vdash \text{wp}\ [\text{i32.const}\ 3; \text{i32.const}\ 1; \text{i32.add}]\ \{w, \text{wp}\ [\mathbf{label}_1\{[]\}\ w\ \mathbf{end}]\ \{w', \Phi(w')\}\}$$
`wp_ctx_bind`
$$\Gamma' \vdash \text{wp}\ [\mathbf{label}_1\{[]\}\ [\text{i32.const}\ 3; \text{i32.const}\ 1; \text{i32.add}]\ \mathbf{end}]\ \{w', \Phi(w')\}$$
`wp_local_get`
$$\Gamma' \vdash \text{wp}\ [\text{local.get}\ 0]\ \{w, \text{wp}\ [\mathbf{label}_1\{[]\}\ w +\!\!+ [\text{i32.const}\ 1; \text{i32.add}]\ \mathbf{end}]\ \{w', \Phi(w')\}\}$$
`wp_ctx_bind`
$$\Gamma' \vdash \text{wp}\ [\mathbf{label}_1\{[]\}\ es_{incr}\ \mathbf{end}]\ \{w, \Phi(w)\}$$
`wp_block`
$$\Gamma' \vdash \text{wp}\ [\mathbf{block}([] \to [\text{i32}])es_{incr}]\ \{w, \Phi(w)\}$$
`wp_local_bind`
$$\Gamma \vdash \text{wp}\ [\mathbf{local}_1\{F'\}\ \mathbf{block}([] \to [\text{i32}])es_{incr}\ \mathbf{end}]\ \{w, w = \mathbf{immV}\ [\text{i32.const}\ 4]\}$$
`wp_invoke_native`
$$\Gamma \vdash \text{wp}\ [\text{i32.const}\ 3; \mathbf{invoke}\ i]\ \{w, w = \mathbf{immV}\ [\text{i32.const}\ 4]\}$$
`wp_call`
$$\Gamma \vdash \text{wp}\ es\ \{w, w = \mathbf{immV}\ [\text{i32.const}\ 4]\}$$

As illustrated, we proceed by applying rule `wp_invoke_native`, leaving us with a new weakest precondition to prove with the same environment Γ. In the figure, F' is defined as

[6] Some structural rules, which we have omitted here, allow it to be applied despite the constant preceding the **call** instruction.

$\{\text{locs} := [\textbf{i32.const}\ 3]; \text{inst} := inst\}$, which is the frame where the call to the increment function needs to be executed in. Next we apply the rule `wp_local_bind` to bind the contents of the **local**. We give up the $\overset{\text{FR}}{\hookrightarrow} F$ resource to fulfill one premise. In its last premise, the new frame resource $\overset{\text{FR}}{\hookrightarrow} F'$ is introduced back to the context, and will be the frame we use to reason within the call to the increment function. We denote by Γ' this new proof environment where we own frame F' instead of F, and let $\Phi(w) = \exists F_1', \overset{\text{FR}}{\hookrightarrow} F_1' * \left(\overset{\text{FR}}{\hookrightarrow} F \mathrel{-\!\!*} \text{wp}\ [\textbf{local}_1\{F_1'\}\ w\ \textbf{end}]\ \{w', w' = \textbf{immV}\ [\textbf{i32.const}\ 4]\}\right)$, which corresponds to the postcondition in the premise of the rule `wp_local_bind`.

The next few steps are mechanical, and we omit the details of some rules for brevity. We apply `wp_block` followed by `wp_ctx_bind` to focus on the first instruction of es_{incr}, **local.get**. We resolve it by applying rule `wp_local_get`, which inspects the locs field of the frame, and leaves us to prove the post-condition for 3. We apply `wp_ctx_bind` again to bind the binary operation **i32.add**, resolve it by applying `wp_binop`[7], and then `wp_label_value` to exit the label. It now remains to show $\Phi(\textbf{immV}\ \ [\textbf{i32.const}\ 4])$, which expands to

$$\exists F_1', \overset{\text{FR}}{\hookrightarrow} F_1' * \left(\overset{\text{FR}}{\hookrightarrow} F \mathrel{-\!\!*} \text{wp}\ [\textbf{local}_1\{F_1'\}\ [\textbf{i32.const}\ 4]\ \textbf{end}]\ \{w, w = \textbf{immV}\ [\textbf{i32.const}\ 4]\}\right)$$

We satisfy the existential with F', give up the resource $\overset{\text{FR}}{\hookrightarrow} F'$ from the context Γ' to satisfy the first part of the separating conjunction, and obtain $\overset{\text{FR}}{\hookrightarrow} F$ back, making our proof environment Γ again. We exit the **local** instruction (which is the function call context) by applying `wp_local_value`, and are left with our original postcondition to prove, which is now trivial when substituted with the value we obtained inside **local**. This completes the detailed proof.

Example. Coming back to the stack module from §1, we now outline what specifications for functions look like and, how they can be used by client modules. Take any function f. We write its specification in the general form:

$$\square \exists cl\ P, \forall i\ vs\ xs, \quad \Psi(P, vs, xs) \mathrel{-\!\!*} (i \overset{\text{wf}}{\longmapsto} cl) \mathrel{-\!\!*} \text{wp}\ vs \mathbin{+\!\!+} [\textbf{invoke}\ i]\ \{w, \Phi(P, w, xs)\}$$

with Φ and Ψ some predicates specific to the function f. The persistence modality $\square$ simply indicates this specification can be duplicated as many times as needed;[8] we omit this modality in every specification that follows, for simplicity. Note the existential quantifiers. The first one, cl, abstracts over the actual closure of function f; because it is hidden behind an existential, it is hidden from clients. The second one, P, allows the specification to reference some abstract representation predicate. In the case of the functions from the "stack" module, we will have an existentially quantified predicate **isStack**, which hides the data representation from clients. We put all specifications under one large existential $\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots\ \textbf{isStack}$, so that all specifications can share the predicate **isStack**.

The specification is thus a weakest precondition[9] on an **invoke**, with some precondition Ψ on the arguments vs given and some postcondition Φ. Both Ψ and Φ can mention the existentially quantified predicate P, as well as some universally quantified variables xs. The invocation address i is linked to the function f by the condition $i \overset{\text{wf}}{\longmapsto} cl$, that asserts that the function body is stored at address i. Let us give the concrete Φ and Ψ used for function "push":

$$\begin{aligned}\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots\ \textbf{isStack}, \Big(&\forall i\ v\ x\ s, \textbf{isStack}(v, s) \mathrel{-\!\!*} (i \overset{\text{wf}}{\longmapsto} cl_{\text{push}}) \mathrel{-\!\!*} \\ &\text{wp}\ [\textbf{i32.const}\ x; v; \textbf{invoke}\ i]\ \{w, w = \textbf{immV}\ [\,] * \textbf{isStack}(v, x :: s)\}\Big) * \ldots \text{(other specs)}\end{aligned}$$

---

[7] Formally, to use the rule as it was presented earlier, one must first frame in the resource $\overset{\text{FR}}{\hookrightarrow} F'$ in order to have the postcondition be of the right form. This means that, just like for every rule we have applied so far, even though we give up ownership of $\overset{\text{FR}}{\hookrightarrow} F'$ to fulfill one premise, we still get to use it to prove the other premise.

[8] As a counterpart, proving this specification cannot rely on usage of any non-duplicable resource.

[9] In practice, we use the host weakest precondition $\text{wp}_{\textbf{HOST}} - \{-\}$ that we introduce in §3, as to allow functions to interact with the host via host calls. For functions that do not interact with the host, this makes no difference.

To present the corresponding Φ and Ψ predicates for the "map" function, we need first to introduce some aspects about higher-order code in WebAssembly, which we do in §2.3.

Given a specification written in this form, and given the resource $i \xmapsto{\text{wf}} cl_{\text{map}}$,[10] a client can verify its code in the presence of a call to the imported map function: when arriving at the instruction **call** $map, `wp_call` reduces **call** to **invoke**, and now the specification shown above can be applied.

Host functions. WebAssembly is meant to be defined independently of the host language in which it is embedded. However, the way the WebAssembly standard is phrased assumes that it is given some operational semantics of the host language as input, and embeds it in the operational semantics of WebAssembly. This phrasing suffices for defining the semantics of WebAssembly alone, which is what the WebAssembly standard does. However, when providing the first formal integration of WebAssembly with a separately-defined host language, we identified that this phrasing is limiting, because it prevents formally giving the semantics of the combined host and embedded language as the integration of two concrete, separately defined language.

To account for this, we modify the presentation of the WebAssembly semantics (this is our only point of departure from the Coq formalization of Watt et al. [2021]) so that the **invoke** of a host function reduces to a new **call_host** administrative instruction:

`invoke_host`

$$\frac{\left(S.\text{funcs}[i] = \{hidx\}^{\text{HostCl}}_{ts_1 \to ts_2}\right) * (|ts_1| = |vs|)}{(S; F; vs \mathbin{+\!\!+} [\textbf{invoke } i]) \hookrightarrow (S; F; [\textbf{call_host } (ts_1 \to ts_2)\ hidx\ vs])}$$

The closure $\{hidx\}^{\text{HostCl}}_{ts_1 \to ts_2}$ represents a *host* function imported from the host language that expects arguments of type ts_1 and yields results of type ts_2. The argument $hidx$ is an identifier that the host will use to determine what the desired function is. The **call_host** instruction remembers the function type tf, the 'host identifier' $hidx$ that allows the host language to identify which function is being called, and the function arguments vs. A **call_host** is stuck, and can only be unstuck by the host language, which typically replaces it by the return value of the call, possibly changing the frame or the store in doing so. We say more about the host interaction in §3.

We prove the following Iris-Wasm proof rule:

`wp_invoke_host`

$$\frac{|vs| = |ts_1| * cl = \{hidx\}^{\text{HostCl}}_{(ts_1 \to ts_2)} * i \xmapsto{\text{wf}} cl * \overset{\text{Fr}}{\hookrightarrow} F * \triangleright \begin{bmatrix} (i \xmapsto{\text{wf}} cl * \overset{\text{Fr}}{\hookrightarrow} F) \twoheadrightarrow \\ \text{wp } (\textbf{call_host } (ts_1 \to ts_2)\ hidx\ vs)\ \{w, \Phi(w)\} \end{bmatrix}}{\text{wp } (vs \mathbin{+\!\!+} [\textbf{invoke } i])\ \{w, \Phi(w)\}}$$

We introduce the **call_hostV** tf $hidx$ vs llh logical value, representing the stuck value llh[**call_host** tf $hidx$ vs]. This allows for seamless binding rules when we introduce the host language's logical rules in §3. Since a **call_host** instruction is also stuck if it is under a **local** or under a **label**, we remember the context llh around the **call_host** as the fourth argument of **call_hostV**. This context llh is a generalized version of lh_k, that has a hole in nested **local**s and **label**s. In the rule above, wp (**call_host** $(ts_1 \to ts_2)$ $hidx$ vs) $\{\Phi\}$ is thus a weakest precondition on a value, and it thus suffices to show that $\Phi($**call_hostV** $(ts_1 \to ts_2)$ $hidx$ vs [_]).

For example, when specifying the "main" function of the extended client module from §1, one intermediate goal, when verifying the part of the code corresponding to the call to the host function $mut, would have the form wp $vs \mathbin{+\!\!+}$ **call** $mut $\{\Phi\}$, where vs represents the constant arguments we have pushed onto the stack prior to making the call. To prove this, one can simply apply rule `wp_call` to reduce **call** to **invoke**, and then rule `wp_invoke_host` to reduce the **invoke** to a **call_hostV** value. The computation is now reduced to a logical value, thus we now must prove that

[10] The name of the index i and ownership of this resource are provided by instantiation when the client does the import.

the postcondition Φ holds of the host call value. We cannot carry on to the rest of the code of the reentrant example if we stick at the WebAssembly level; this is in line with the nature of this call: it is a host call and needs interaction with the host to be unstuck. We will see in §3 how to reason about interaction with the host to prove the full specification of the reentrant example.

2.3 Higher-Order Code with Indirect Calls

As explained in §1, one can use **call_indirect** to implement higher-order functions with the help of the host language. The instruction **call_indirect** i, where i is an index into the types field of the module instance in the function frame, takes one argument k from the stack, and uses it as an index to look up the function to call in the table. The table itself is located in the store. Like for function invocation, the instance in the frame F finds the store-index ta of the correct table (i.e. the one at the head of the tables field). Now the kth element a of the table indexed ta can be looked up, and used as the index in the function closures component of the store, to find the closure cl to execute. As a side condition, the type of the closure must match the one declared by index i (that **call_indirect** takes as an immediate). Finally, [**call_indirect** i] reduces to [**invoke** a], setting cl to be invoked in the next reduction step.

We prove the following program logic rule:

wp_call_indirect_success
$$\frac{\overset{\text{FR}}{\hookrightarrow} F * (F.\text{inst.tabs}[0] = ta) * (ta \overset{\text{wt}}{\longmapsto}_k a) * (a \overset{\text{wf}}{\longmapsto} cl) * (F.\text{inst.types}[i] = \text{typeof } cl) * \triangleright\Big((ta \overset{\text{wt}}{\longmapsto}_k a) -\!\!* (ta \overset{\text{wf}}{\longmapsto} cl) -\!\!* (\overset{\text{FR}}{\hookrightarrow} F) -\!\!* \text{wp } [\textbf{invoke } a]\ \{w, \Phi(w)\}\Big)}{\text{wp } [\textbf{i32.const } k; \textbf{call_indirect } i]\ \{w, \Phi(w)\}}$$

Here, we use the points-to predicate for elements of the table: only ownership of the relevant kth element of the table is required. Notice how the rule passes the ownership of all three points-to predicates (frame ownership, table element ownership and function closure ownership) to the continuing weakest precondition.

Example. The higher-order "map" function of our stack module in §1 calls its argument function on each element in the stack by using **call_indirect**. We have now introduced enough logical machinery to present our modular specification of "map":

$$\begin{aligned}
&\exists cl_{\text{map}}\ \textbf{isStack}, \forall \Phi\ \Psi\ a\ v\ s\ F\ j\ k\ i, && (1)\\
&\Box(\forall u.\ \Phi\ u -\!\!* \ldots -\!\!* \text{wp } (\textbf{i32.const } u; \textbf{invoke } a)\ \{v, \Psi\ u\ v * \ldots\}) -\!\!* && (2)\\
&\quad \textbf{isStack } v\ s -\!\!* \text{stack_all } s\ \Phi -\!\!* && (3)\\
&\quad\quad (\overset{\text{FR}}{\hookrightarrow} F) -\!\!* (F.\text{inst.tabs}[0] = j) -\!\!* (j \overset{\text{wt}}{\longmapsto}_k a) -\!\!* \ldots -\!\!* (i \overset{\text{wf}}{\longmapsto} cl_{\text{map}}) -\!\!* && (4)\\
&\quad\quad\quad \text{wp } [\textbf{i32.const } k; v; \textbf{invoke } i]\ \{w, \exists s'.\ \textbf{isStack } v\ s' * \text{stack_all2 } s\ s'\ \Psi * \ldots\} && (5)
\end{aligned}$$

Let us describe the specification line by line: (1) As explained in §2.2, we existentially quantify over a closure cl_{map} and a predicate **isStack**, to hide our implementation of the stack and the body of the "map" function. We then universally quantify over many variables, including notably Φ and Ψ used in the specification of the mapped function, stressing this specification can be as general as needed. (2) The first precondition is a specification for the mapped function; it uses two predicates Φ and Ψ to express that for any **i32** input u that satisfies Φ, the mapped function returns an **i32** result v such that Ψ relates u with v. We have used '…' to elide some predicates, which are simply a copy of some of the resources from line 4, so as to allow usage of those resources (like frame ownership) in the proof of the specification of the mapped function. (3) Next, we describe the argument value v: it must represent a mathematical stack s, all elements of which satisfy Φ. This is captured by the **isStack** v s predicate. (4) A points-to predicate for table j links the argument value k to the function index a (from the **invoke** in line 2). For brevity, we elide other side-conditions pertaining to typechecking the mapped function. At the end of the line, we have the function closure points-to

(import variable) vi ::= nat (module variable) vm ::= nat (host action id) $hidx$ ::= nat

(declaration)	δ ::= **inst_decl** vis vm vis | **get_global** i
(host action)	a ::= **nop** | **print** | **instantiate** δ | **call_wasm** | **table.set**
(import variable store)	I ::= $vi \hookrightarrow export$
(host state)	H ::= {store : S, frame : F, imports : I, modules : ms, actions : as}
(host expression)	he ::= $(es; \delta s)$ (host value) hw ::= $(vs; [\,])$ | $(\textbf{trap}; [\,])$

Fig. 4. Host Syntax (definitions reference the grammar in Fig. 2)

predicate that links the index i of the invocation on line 5 to the "map" function closure. (5) After running "map", we have a stack with logical state s' at location v, whose elements are related one-to-one to that of the previous logical state s by Ψ. For readability, we omit the second part of the postcondition, which simply gives back all of the resources from line 4.

To prove the above specification, the $stack module, who has access to the actual code of the "map" function, simply fills in the existential quantifiers with the actual closure of "map" and the definition of **isStack** reflecting the actual implementation. Then all that remains is a weakest precondition to prove, which is done by applying the rules in §2.2: `wp_invoke_native` using hypothesis $i \overset{\text{wf}}{\longmapsto} cl_{\text{map}}$, then `wp_local_bind`, to enter the local etc.

Note that we rely on the fact that our ambient logic, Iris, is a higher-order separation logic, in which weakest preconditions are just usual propositions. We stress again that the user of "map" does not need to know how **isStack** is defined (and in fact, we hide it with an existential quantifier surrounding the specification of the stack module, again exploiting the higher-order logic of Iris) or the physical state of the stack representation in memory: they only need to reason about the mathematical state, s; for example, stack_all only refers to s.

This example demonstrates that Iris-Wasm can be used to prove specifications for modules that cleanly hide the heavy indirection and low-level details of WebAssembly.[11] The use of **call_indirect** for higher-order programming, to call an arbitrary client function, goes beyond the 'encapsulated' fragment of WebAssembly of Watt et al. [2019], and yet is captured modularly in the first line of our specification. Our accompanying Coq formalization contains a formal proof that a simple implementation of the stack module meets the specification. We can then apply the specification to different clients. In this paper, we focus on the reentrant client introduced in §1, see §3, and a client that applies "map" to an unknown and potentially malicious imported function (see §5). The code for these examples, and a few more, can be found in our Coq development.

3 HOST LANGUAGE AND PROOF RULES

In this section, we define a minimal host language featuring the core operations of the WebAssembly JavaScript Interface. The host fulfils two important roles; first, it embeds WebAssembly and defines the interoperability between WebAssembly and the host; and, second, it implements *module instantiation*, in which the host language handles the allocation of WebAssembly states. Our minimal host language also has the ability to mutate WebAssembly function tables.

We begin by introducing the syntax of the host language and selected proof rules, with a focus on the interoperability with WebAssembly. We then detail the rules for module instantiation.

The syntax of the host language is shown in Fig. 4. Host expressions are pairs of WebAssembly expressions and host-specific declarations; host values are pairs of WebAssembly values, and an empty list of declarations. Finally, the host state is a record of the WebAssembly store and frame,

[11]Indeed, the specification shown here is akin to the specification for a stack module implemented in an ML-like programming language in standard Iris [Birkedal and Bizjak 2017].

as well as host-specific state. Host specific state has three components. First, it includes a store of export objects, to store the exports of an instantiated module, and to feed the imports of future instantiations. Note that while we call them import variables, they are used both for imports and exports. Subsequently, an *export* object refers to any object passed from one module to another, either as import or export. Second, it keeps track of a list of WebAssembly modules. Finally, to maintain the generality of host calls, host actions are indirectly referenced by indices into a list of available host actions.

To illustrate the expressive power of a host, our minimal host language includes five different host actions. **nop**, **print** and **instantiate** δ are pure operations that do not depend on host or WebAssembly store. More noteworthy are the **call_wasm** and **table.set** operations: **call_wasm** reduces to a WebAssembly call instruction, which opens up the possibility of reentrancy between the host and WebAssembly; **table.set** displays the expressive power of the host over the WebAssembly store, by mutating a given function table with a function from the WebAssembly store.

Declarations are either (1) instantiations **inst_decl** *vis vm vis*, which consist of a list of import/export variables to feed into the imports of a module (referenced indirectly by its index into the module store), whose exports are stored in the subsequent list of import/export variables, or (2) load declarations for WebAssembly globals, to load the final output of a Wasm module's main function. The host operational semantics prioritises the reduction of WebAssembly expressions over that of instantiation declarations. We refer to the Coq formalization for a full account of the host operational semantics.

In the remainder of this section, we will discuss the proof rules of our new program logic for the host. We define our host logic using a weakest precondition predicate $\mathsf{wp}_{\mathbf{HOST}}\,(es;\delta s)\,\{hw, \Phi(hw)\}$, which intuitively means that the host expression $(es;\delta s)$ does not get stuck and, if it terminates with the host value hw, then the predicate Φ holds for hw.

While the host weakest precondition is not to be confused with the Wasm weakest precondition, it shares some similarities in its memory model. The memory model of the host program logic extends the memory model of the Wasm program logic, as it includes the Wasm store. We reason about the host-specific part of the host state using three new predicates: (1) $vi \xmapsto{\mathsf{vis}} export$: a points-to predicate for the export object store; (2) $vm \xhookrightarrow{\mathsf{mod}} m$: a points-to predicate for the module store; (3) $hidx \xhookrightarrow{\mathsf{ha}} a$: a points-to predicate for the host action store. We present the host program logic in two parts: first we discuss the rules that implement interoperability between WebAssembly and the host, and second we discuss module instantiation.

Interoperability. The first key to WebAssembly and host interoperability is the WebAssembly lifting step. Any reduction in the WebAssembly part of a host expression corresponds to a step in the host expression, as captured by the following bind rule:

$$\texttt{wp_lift_wasm}\quad \frac{\mathsf{wp}\ es\,\left\{w, \mathsf{wp}_{\mathbf{HOST}}\,(w;\delta s)\,\{hw, \Phi(hw)\}\right\}}{\mathsf{wp}_{\mathbf{HOST}}\,(es;\delta s)\,\{hw, \Phi(hw)\}}$$

Note that w may be a logical value, in particular a suspended host call from Wasm to the host, which can now be resolved via the host proof rules for **call_host**. Recall the definition of a stuck host call: the **call_host** *tf hidx vs* administrative instruction is considered stuck in any nested WebAssembly context *llh*, and is interpreted as the logical value **call_hostV** *tf hidx vs llh*, in which *hidx* refers to the host action identifier which is storing the executing host action, *tf* refers to its type, and *vs* refers to the parameters of the invocation. Each host action is resolved via a different proof rule.

In particular, one such host action is a call in the other direction, from the host to Wasm. In that case, the inner **call_wasm** action, performed by the host function *hidx*, reduces to the WebAssembly

instruction **call** as follows.

`wp_host_action_call_wasm`

$$\frac{hidx \xhookrightarrow{\text{ha}} \textbf{call_wasm} \ast \triangleright (hidx \xhookrightarrow{\text{ha}} \textbf{call_wasm} \mathrel{-\!\!\ast} \mathsf{wp}_{\textbf{HOST}}\ (llh[\textbf{call}\ i]; \delta s)\ \{hw, \Phi(hw)\})}{\mathsf{wp}_{\textbf{HOST}}\ (llh[\textbf{call_host}\ tf\ hidx\ [\textbf{i32.const}\ i]]; \delta s)\ \{hw, \Phi(hw)\}}$$

Reentrant example. We now have all we need to prove a specification for the extended (reentrant) client introduced in §1. This specification will be parametrized with specifications for all the functions from the stack module (and thus with all the existentials of those specifications, most importantly the **isStack** predicate), and can be *modularly* combined with a specification for the stack module.

Our specification could look like this:

$$\begin{gathered}\exists cl_{\text{main}}, \forall v\ x_1\ \ldots\ x_n\ i\ hidx,\quad \textbf{isStack}\ v\ [x_1, \ldots, x_n] \mathrel{-\!\!\ast} i \xmapsto{\text{wf}} cl_{\text{main}} \mathrel{-\!\!\ast} \\ \textbf{OwnClosures}([\$f0; \$f3; \$map]) \mathrel{-\!\!\ast} \$mut \xmapsto{\text{wf}} \{hidx\}^{\text{HostCl}}_{[\text{i32};\text{i32}]\to[]} \mathrel{-\!\!\ast} hidx \xhookrightarrow{\text{ha}} \textbf{table.set} \mathrel{-\!\!\ast} \\ \mathsf{wp}_{\textbf{HOST}}\ ([\textbf{i32.const}\ v; \textbf{invoke}\ i], [])\ \{hw, \textbf{isStack}\ hw\ [f_3(f_0(x_1)), \ldots, f_3(f_0(x_n))] \ast \ldots\}\end{gathered}$$

The elided postconditions give back all the preconditions; $\textbf{OwnClosures}(fs)$ asserts ownership, for all functions $f \in fs$, of a closure cl_f. For the function $map imported from the stack module, the closure is the one referenced in the specification of the stack module. In order to carry out our proof, we assume we are given specifications for functions $f0 and $f3 that reference cl_{f_0} and cl_{f_3}.

To prove this specification, we fill in the existential quantifier for cl_{main} with the actual code of the "main" function. Now we apply `wp_lift_wasm` to bring ourselves to proving a WebAssembly weakest preconditon: the postcondition now becomes $w, \mathsf{wp}_{\textbf{HOST}}\ w\ \{hw, \Phi(hw)\}$ where Φ is the postcondition in the weakest precondition shown above. We can now begin the proof just like we proved all the specifications for the functions in the stack module: we apply `wp_invoke_native`, then `wp_local_bind`, etc.

As showcased in §2.2, the WebAssembly weakest precondition gets stuck on a value when it arrives at the host call: we now need to show that the postcondition holds of the **call_hostV** value, i.e. that

$$\mathsf{wp}_{\textbf{HOST}}\ llh[\textbf{call_host}\ tf\ hidx\ vs]\ \{hw, \Phi(hw)\}$$

where llh is the context in which the host call was, containing for instance all the code that follows the host call. To prove this, we have a rule `wp_host_action_table_set` similar to rule `wp_host_action_call_wasm` shown above, that, given our knowledge of $n \xmapsto{\text{wt}}_0$ $f0, gives back $n \xmapsto{\text{wt}}_0$ $f3, and brings us to prove a (host) weakest precondition statement on the code that follows the host call, with this new function at the 0th place in the table. We can prove this by lifting to WebAssembly and carrying out the proof in the WebAssembly program logic until the end.

Module instantiation. While WebAssembly 1.0 does not depend on any particular host language, it does define a *specification* for module instantiation. Any host language is tasked with implementing instantiation according to that specification. We thus conceptually distinguish between the parts of module instantiation pertaining to the official WebAssembly specification, and the parts that deal with the host language. $\mathsf{Instantiate}(S, m, exportdescs, ((S', inst, exports), start))$ defines the specification for module instantiation. The full definition is quite elaborate; we refer to the Coq mechanization for all details, and provide an intuitive overview here. In essence, it states that *inst* is the result of instantiating module m while importing *exportdescs*, *exports* are the resulting exports, and S' is the resulting WebAssembly store, in which all the relevant state has been allocated.

The specification enforces various side conditions. First, the module must be well typed according to a list of relevant import and export types. Next, it asserts the necessary operational conditions on the allocated state and created instance; that all the fields of the instance are properly initialized (e.g. any function table is initialized with the proper elements as defined by the module), that all the

initialized values are within the bounds of the initialized object, and finally that the start function is either empty, or refers to a function of the module of type $[] \rightarrow []$.

The instantiation specification specifies the outcome of module instantiation on the WebAssembly store. Note that the specification is host language agnostic. The semantic outcome of instantiation on the WebAssembly store ought likewise to be independent of the host language that implements it. The following lemma captures the effects of instantiation on the interpretation of the WebAssembly store as Iris resources, according to the host agnostic instantiation specification. The lemma is thus independent of any host language definition.

Lemma 3.1 (Module Instantiation Resource Allocation).

$$\begin{array}{ll} \textit{If} & \vdash m : \textit{timps} \longrightarrow \textit{texps} \;\wedge\; \mathsf{constInits}(m) \\ \textit{and} & \mathsf{Instantiate}(S, m, \textit{imports}, ((S', \textit{inst}, \textit{exports}), \textit{start})) \\ \textit{then} & \mathsf{resourcesImports}(m, \textit{imports}, \textit{timps}, \textit{wfs}, \textit{wts}, \textit{wms}, \textit{wgs}) * \mathsf{stateInterp}(S) \\ & \Rrightarrow \mathsf{resources}(m, \textit{imports}, \textit{timps}, \textit{wfs}, \textit{wts}, \textit{wms}, \textit{wgs}, \textit{start}, \textit{inst}) * \mathsf{stateInterp}(S') \end{array}$$

For readability, we omit the technical details behind some of the above predicates. It suffices to know the following: constInits limits the global initializers and offsets to be constants, resourcesImports defines the points-to predicate associated with each import in imports, and resources defines all the points-to predicates associated to the created instance *inst*, including those that were previously imported. The variables *wfs*, *wts*, *wms* and *wgs* are maps that summarise the values of functions, tables, memories and globals of the created instance. (The $\Rrightarrow$ modality is used in Iris to update ghost resources [Jung et al. 2018b].) Using Lemma 3.1, we can then prove a host weakest precondition rule for host instantiation, that we will refer to as `wp_host_instantiate`.

Example. The complete stack module is an instantiation declaration, which exports closures for push, pop, new_stack, is_empty, is_full, stack_length and map, as well as the function table invoked by map. We recall that exports are passed via indices into the import variable store.

$$vm \triangleq \{0 \mapsto \mathsf{stack_module}\} \qquad \textit{host_program} \triangleq ([], [\mathbf{inst_decl}\ []\ 0\ [0, 1, 2, 3, 4, 5, 6, 7]])$$

The Iris-Wasm specification of the complete stack module from §1 is as follows (we elide the exporting of the table, for simplicity):

$$\begin{array}{l} \exists \mathsf{stack_module}, \forall i\ js, (i \xhookrightarrow{\mathsf{mod}} \mathsf{stack_module}) \mathrel{-\!\!*} \mathop{\huge *}_{j \in js} j \xmapsto{\mathsf{vis}} - \mathrel{-\!\!*} \\ \quad \mathsf{wp}_{\mathbf{HOST}}\ ([]; [\mathbf{inst_decl}\ []\ \mathsf{i}\ js]) \left\{ \begin{array}{l} \exists cl_{\mathsf{push}}\ cl_{\mathsf{pop}} \ldots cl_{\mathsf{map}}, \mathbf{isStack}, \mathsf{spec_push} * \mathsf{spec_pop} \\ \quad * \ldots * \mathsf{spec_map} * \mathop{\huge *}_{j \in js} j \xmapsto{\mathsf{vis}} \mathsf{function_export}\ cl_j \end{array} \right\} \end{array}$$

spec_push is the specification of the "push" method shown earlier. Likewise for the other specifications mentioned in the postcondition. Both the contents of the $stack module and the implementations of the stack operations are hidden from clients because of the existential quantifiers.

This stack module specification is proven by applying rule `wp_host_instantiate`, which populates the value import stores and gives ownership of all the resources necessary for the stack module operations, and then we apply the specifications for the stack operations shown in §2.3.

With this specification for the stack module and a similar one for the client module (parametrized by the specification of the stack), we verify the complete stack program (a sequence of instantiations) in our Coq formalization.

4 MECHANIZATION IN THE IRIS FRAMEWORK

We implement and prove the Iris-Wasm proof rules in this paper in the Iris framework in the Coq proof assistant. Iris was originally developed to reason about programs with complex concurrency; however, the same mechanisms have proven useful to reason about complex sequential programs

Fig. 5. Lines of code of the Iris development, as given by cloc

helpers	language	rules	instantiation	host	examples	logrel	stack	**total**
11836	3685	7123	6828	2339	2754	8145	8787	51497

such as the awkward example, as demonstrated for example by Georges et al. [2021a]. In this paper, we focus our presentation on the novel, language-specific proof rules we introduce and prove, but our program logic also inherits many other logical constructs and proof rules from Iris which we make use of in our development. We have already mentioned the 'later' modality, $\triangleright$, which avoids circularities in the presence of the higher-order features of Iris, and which can be used to define guarded recursive predicates in Iris, as well as the 'persistence' $\Box$ and 'update' $\Rrightarrow$ modalities. Other features we use include the frame rule, non-atomic invariants, ghost state, and other proof rules like Löb induction; for a thorough introduction to those, see Jung et al. [2018b].

We prove all our proof rules in Iris, with respect to the default definition of the weakest precondition predicate (with an extra requirement that the frame resource holds for every step of reduction) instantiated to refer to the Coq formalization of the official WebAssembly 1.0 operational semantics by Watt et al. [2021].

The adequacy theorem of Iris [Jung et al. 2018b, §6.4] then yields the final desired soundness theorem, which intuitively says that if a weakest precondition for a WebAssembly or host program has been proved in Iris-Wasm, then it does indeed mean that the program runs safely, according to the official WebAssembly 1.0 operational semantics, or the host language that embeds it. An example of the latter can be found in the Coq mechanization.

The size of the full Iris development is summarized in Fig. 5. The logrel folder contains a case study presented in the next section, and stack contains the full stack module and associated clients.

The stack module, with a binary size of 637 bytes, is defined in around 200 lines code in Coq, with the module type checking done in 300 lines of code using the type checker from Watt et al. [2021]. The module specification is fully verified using the Iris-Wasm logic in around 3800 lines of code in Coq, where 2100 lines are used to verify each of the module function specifications, and the remaining code is used to prove the top-level instantiation specification and auxiliary lemmas. Such a ratio between program and proof size may hint at a substantial verification effort. However, it's important to note that it reflects a version of Iris-Wasm without a bespoke proof mode; an interesting line of future work is to extend Iris-Wasm with various automation techniques, such as the proof search strategy of Mulder et al. [2022], and use it to prove specifications of large real-world programs.

5 CASE STUDY

We showcase the utility of our program logic through a case study[12]. The goal is to leverage the coarse-grained encapsulation guarantees of WebAssembly modules to prove robust safety of two scenarios involving some interaction between a known module and an unknown, potentially malicious, module. While the coarse-grained encapsulation properties granted by modules are relatively shallow (one module cannot interact with the internals of another), the reasoning principles are not: not only are we reasoning about unknown code, the desired robust safety property can be subtle, and highly specific to the particular implementation of a robustly safe module. We emphasize that we do not seek to either define or prove encapsulation as a meta-property, rather, we define and apply a methodology to prove robust safety of specific modules.

[12]Our Coq mechanization also includes another case study of a program that uses recursion through the store, by applying a host call to mutate the function table, known as Landin's Knot.

WebAssembly's modules are designed to allow trusted code to encapsulate its local state (e.g. variables and memory), by limiting what is shared with untrusted modules via imports and exports. This encapsulation is meant to hold no matter what other modules do, either by accident or by malice, and thus does not rely on compliance. Modules can take advantage of this encapsulation to guarantee various safety properties. To prove those properties formally, we may need to reason about the interaction between known, trusted code and unknown, untrusted code. We have thus far presented a program logic to reason about known code only. In this case study, we use the program logic to build a method to reason about the instantiation of unknown code, and use it to prove the *robust* safety of known code, that is, safety even when composed with adversarial code.

```
m_client ≜ (module ;; Another Stack Client
(import "adv" "f" (func $f (param i32) (result i32)))
(import "stack" "map" (func $map (param i32 i32)))
  ... ;; import global g and the remaining stack module
(elem (i32.const 0) $f) ;; populate table with imported
      function
(func $main (local $i i32)
  call $new_stack; ... ; const 4; call $push;
  local.get $i; const 2; call $push;
  local.get $i; const 0; call $map;
  local.get $i; call $stack_length; global.set $g))

   stack_client ≜
   inst_decl [] "stack" ["tab";...;"pop"]
   inst_decl [] "adv" ["f"]
   inst_decl ["f";"g";"tab";...;"pop"] "client" []
```

Fig. 6. Robust safety example: applying map on an imported function

The methodology is based on a relational interpretation of WebAssembly types, built on top of our Iris-Wasm program logic, by defining logical relations for each WebAssembly type. The key idea is to interpret the types of primitives, functions, etc., all the way to module types, as propositions in Iris-Wasm. The methodology of defining logical relations in Iris is well known [Georges et al. 2021a; Jung et al. 2018a; Krebbers et al. 2017; Swasey et al. 2017], but here it is for the first time applied to the type system of a full industrial standard, namely the WebAssembly type system. We define semantic interpretations for all WebAssembly types. That includes all the internals of a module, and in particular it includes the types of exports and imports. We say that an import object is safe to share, or valid, if it is in the appropriate relation. All the results in this section have been formally proved in Coq. We give an overview here, and refer the reader to the accompanying Coq code for the full definition of the relational interpretation of WebAssembly types.

The interpretation of module types via the instance relation, denoted $\mathcal{I}[\![C]\!]$, is the keystone to derive specifications for unknown functions. The following key theorem states that the result of instantiating a well-typed module $\vdash m : timps \longrightarrow texps$ produces a valid instance, given that all imports are valid according to *timps*.

Theorem 5.1 (Valid Instance Allocation). *If* $\vdash m : timps \longrightarrow texps$*, and inst is the result of instantiating module m with imports imps, then*

$$\mathsf{resources}(m, imps, timps, \cdots, inst) \twoheadrightarrow \mathsf{valid}[\![timps]\!](imps) \twoheadrightarrow \mathcal{I}[\![C]\!](inst)$$

where C is the module type, determined syntactically, $\mathsf{resources}(\cdots, inst)$ *corresponds to the ghost resources allocated by module instantiation as depicted by Lemma 3.1, and* $\mathsf{valid}[\![timps]\!](imps)$ *unfolds the list of imports, and applies the relevant relation on each import object.*

Proof. By unfolding the definition of module typing, inferring properties about the result of instantiating m, and component-wise proving the instance relation. Validity of imported types is established by the $\mathsf{valid}[\![timps]\!](imps)$ assumption, while the rest are established using the fundamental theorem of logical relations (FTLR). The FTLR, which roughly states that all well-typed programs are semantically well-typed, is a key non-trivial language property, and is proved by induction over the full type system. □

Applications of the Logical Relation. Next we describe two scenarios, each involving our stack module interacting with some unknown function. In each case, the two modules interact via imported closures. We will therefore employ the closure relation *Clos* as the principal logical relation in our reasoning.

The two applications highlight a conceptual distinction between two kinds of scenarios in which known code interacts with unknown code. In the first example, known code imports functions from an unknown module, and has a certain amount of control over how these are applied. The second example exports known code to an unknown module, and in that case, exported closures must carefully guard against misuse.

Fig. 6 depicts a client of the stack module, which imports a closure "f" of type $[\mathbf{i32}] \rightarrow [\mathbf{i32}]$ from an unknown module. The client creates a new stack, pushes two values, then applies map using the imported unknown function, and finally computes the length of the stack by calling a function from the stack module. The stack module hides its internal representation from the context. Likewise, the host makes sure to hide the stack module operations from the unknown module. WebAssembly's coarse grained encapsulation thus guarantees that the integrity of the allocated stack is maintained, no matter what the unknown imported function does: as long as it does not trap, the final length operation succeeds and returns the original size of the stack, namely 2. We refer to imports and modules via names rather than indices, for the sake of readability. The following theorem expresses robust safety formally:

Theorem 5.2 (Top-level Host Specification). *If* $\vdash m_{\mathrm{adv}} : [] \; [\mathbf{func_e} \; ([\mathbf{i32}] \rightarrow [\mathbf{i32}])]$ *and the syntactic restrictions on* m_{adv} *hold, then*

$$\left\{\begin{array}{l} \text{"stack"} \xhookrightarrow{\text{mod}} m_{\text{stack}} * \text{"adv"} \xhookrightarrow{\text{mod}} m_{\text{adv}} * \\ \text{"client"} \xhookrightarrow{\text{mod}} m_{\text{client}} * \text{"g"} \xmapsto{\text{vis}} \$g * \$g \xmapsto{\text{wg}} - * \\ [\mathit{NaInv} : \top] * \text{"f"}, \text{"tab1"}, \text{"map"}, \ldots, \text{"pop"} \xmapsto{\text{vis}} - \end{array}\right\} \text{stack_client} \left\{ hw, \begin{array}{l} (hw = ([]; []) \land \$g \xmapsto{\text{wg}} 2) \lor \\ hw = (\mathbf{trap}; []) \end{array} \right\}$$

Proof. Once the host has allocated the unknown module, we apply Theorem 5.1 to conclude that its instance is valid, which guarantees that each of its components, including the exported closure of type $[\mathbf{i32}] \rightarrow [\mathbf{i32}]$, is valid. As a result, we know that the unknown import of our client is in the closure relation *Clos*, which by definition of the relational interpretation includes a specification for the unknown function. Crucially, this specification does not depend on the stack internals, and thus we are able to prove that the stack size is maintained. □

Next we consider a scenario in which an unknown module imports operations from the stack module, namely new_stack, push and pop. The encapsulation of the stack module's internal state, alongside careful checks at the boundaries of each operation, which we will elaborate on below, should guarantee that the stack module memory indeed stores and maintains stacks, as defined by the **isStack** predicate, irrespectively of what the unknown module does. Henceforth we will refer to this as the representation invariant, denoted by stackInvariant(m), where m is the index of the encapsulated memory. Roughly, the representation invariant is an Iris (non-atomic) invariant containing a big separation of **isStack** predicates, one for each allocated stack.

The basic type system of WebAssembly guarantees that the adversary code does not get stuck. However, our goal is to reason about integrity of the data representation enforced by the module system. While the type system defines the typing of an individual module, it does not consider interweaving of module instantiations, since instantiation is handled by a host, typically written in untyped JavaScript. Therefore, the type system is too weak to capture the data abstraction enforced by the module system, which we are relying on here. As such, our interpretation of the type system does not capture the refined interpretation (with the representation invariant) of the stack module.

We use the standard type interpretation of the adversary module to reason about its execution. However, we want this interpretation to depend on the *refined representation invariant* of the stack module internals, rather than the *default interpretation granted by the logical relation*. Since each import must be valid when applying Theorem 5.1, we *manually* prove that, given the representation invariant, each exported function (new_stack, push, and pop) is in the closure relation.

As a result, we must now consider the case where a stack operation is applied on an arbitrary input value. Consider, for instance, push – it takes two arguments, one of which is a stack value, which is interpreted as a memory address. A malicious adversary could apply push to a masked stack value (a bogus memory address), thus breaking the expected internal behavior of the stack module. push must thus guard against such a situation by dynamically checking the validity of all safety-critical parameters. These dynamic checks ensure that no stack gets corrupted. Relying on those dynamic checks, we can then prove specifications that maintain the representation invariant:

Theorem 5.3 (Validity of Select Stack Module Operations). *If* $inst.\mathsf{mems} = [m]$ *then,*

$$\begin{aligned}\mathsf{stackInvariant}(m) \rightarrow\ & \mathit{Clos}[\![[\mathbf{i32};\mathbf{i32}] \rightarrow []]\!](\{(inst, [\mathbf{i32}]); \mathsf{push}\}^{\mathsf{NativeCl}}) \\ & * \mathit{Clos}[\![[\mathbf{i32}] \rightarrow [\mathbf{i32}]]\!](\{(inst, [\mathbf{i32}]); \mathsf{pop}\}^{\mathsf{NativeCl}}) \\ & * \mathit{Clos}[\![[] \rightarrow [\mathbf{i32}]]\!](\{(inst, [\mathbf{i32}]); \mathsf{new_stack}\}^{\mathsf{NativeCl}})\end{aligned}$$

The representation invariant is allocated upon instantiation of the stack module, at which point there are no allocated stacks. Theorem 5.3 is then applied on each of the relevant stack module exports, such that we can apply Theorem 5.1, and conclude with the standard type interpretation of the adversary module, while maintaining the now allocated representation invariant.

6 RELATED WORK

Watt et al. [2019] develop a mechanized first-order separation logic for what they call "encapsulated" WebAssembly, that is, code limited to a single module, with no exports or imports, and no uses of the **call_indirect** instruction or the host, and they do not handle instantiation. For their subset of the language, our proof rules are similar up to presentational details, except for the handling of breaks, where, as mentioned in §2.2, we use a novel approach with a bind rule which scales to higher-order programs, unlike the approach taken by Watt et al. [2019].

WebAssembly provides coarse-grained memory safety, at the boundary of memory objects, and coarse-grained isolation, at the boundary of modules. Lehmann et al. [2020] show that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against Wasm programs that not specifically written to take advantage of module isolation. We show in our examples that, when Wasm programs are written with module isolation in mind, the language specification does indeed enforce expected isolation guarantees.

MSWasm [Disselkoen et al. 2019; Michael et al. 2023] (Memory-Safe Wasm) is a proposed extension of WebAssembly that adds first-class support for CHERI-like [Watson et al. 2015] fine-grained runtime-checked memory capabilities. The logical relation of Cerise [Georges et al. 2021a, 2022a, 2021b, 2022b], mechanized in Iris, captures encapsulation for hardware capabilities in an idealized assembly model and may be used as a starting point to formalize the guarantees of MSWasm on top of Iris-Wasm.

CapableWasm [Fitzgibbons 2022] is a (work-in-progress) extension of the type system of WebAssembly to support compositional compilation from different languages. They rely on their type system to enforce finer-grained encapsulation than at the module boundary.

Kolosick et al. [2022] use a logical relation to show that WebAssembly programs naturally compile to unsafe platform assembly in such a way that the compiled code obeys a safe calling convention and certain isolation properties with respect to the rest of the system. Narayan et al.

[2020] rely on this result to implement a sandboxing technique whereby C code is first compiled to WebAssembly which is then ultimately compiled to native assembly for linking. They use this technique to sandbox a number of Firefox libraries.

Many related works deal with the mechanized formalization of low-level languages. RockSalt [Morrisett et al. 2012] is a verified checker that validates code binaries against a sandbox policy, similar to that of Google's Native Client (NaCl). RockSalt is mechanically verified using a formalization of a subset of x86 in Coq. Kennedy et al. [2013] use Coq to build a macro assembler for x86, while relating machine code to separation logic formulas suitable for program verification.

The Certified Assembly Programming (CAP) family of frameworks [Feng and Shao 2005; Ni and Shao 2006; Yu et al. 2003; Yu and Shao 2004] support the definition of second-order Hoare logics for verifying modular specifications of low-level assembly programs, using expressive features such as embedded code pointers, concurrency, and dynamic thread creation. As such, CAP focuses on features that are abstracted away by Wasm. Gu et al. [2016] presents CertiKOS, an extensible architecture for certifying concurrent OS kernels. Using CertiKOS, Gu et al. [2016, 2018] develop and verify a concurrent OS kernel consisting of both C and x86 assembly code. By leveraging CompCertX [Gu et al. 2015], CertiKOS is able to reason about interactions between C and x86 assembly. As is the case with Iris-Wasm, the setup assumes that the two languages share the same memory model. The recent DimSum [Sammler et al. 2023] framework supports reasoning about multilingual programs between languages with different memory models. However, while Iris-Wasm focuses on mechanizing the full language of a real industrial standard, the DimSum approach has only been applied to a simple high-level imperative language and an idealized assembly language so far.

The W3C have announced a Public Working Draft for WebAssembly 2.0. It includes several features orthogonal to our focus on security, such as extra numeric operations. The two relevant features are: the lifting of the artificial restriction to one table per module (we have done this too), which corresponds to a simple update to the relation on instances; and the addition of opaque reference types to objects of the host language, which adds new WebAssembly values, but no actual complexity because of their opacity (this is trivial to do).

7 CONCLUSION

We have presented Iris-Wasm, a practical higher-order, mechanized program logic for the W3C WebAssembly 1.0 official language standard [Rossberg 2019], building on the mechanized WasmCert-Coq specification [Watt et al. 2021]. We show how the reasoning of Iris-Wasm can handle the intricacies of WebAssembly, including interaction with its host language and the higher-order programs and reentrancy that it enables, going far beyond the 'encapsulated' fragment of WebAssembly in previous work [Watt et al. 2019]. We then leverage our program logic to build a logical relation which enforces robust safety, demonstrating that we can prove properties of encapsulation at module boundaries. This example illustrates the potential of what can be done with formal methods. We hope other researchers will use our formalization to further investigate the WebAssembly ecosystem, and that industrial language communities will thereby be further enticed to embrace the formalization of language specifications.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. Conrad Watt is supported by a Research Fellowship from Peterhouse, University of Cambridge. Rao is supported by a Doctoral Scholarship Award from Department of Computing, Imperial College London. Gardner is supported by the EPSRC fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1).

ARTIFACT AVAILABILITY

The artifact [Rao et al. 2023] containing the full Coq development is available on Zenodo.

REFERENCES

Lars Birkedal and Aleš Bizjak. 2017. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic.* Technical Report. Aarhus University.

Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3337167.3337171

Daniel Ehrenberg. 2019. *WebAssembly JavaScript Interface W3C Recommendation.* Technical Report. W3C. https://www.w3.org/TR/wasm-js-api-1/

Xinyu Feng and Zhong Shao. 2005. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 254–267. https://doi.org/10.1145/1086365.1086399

Michael Fitzgibbons. 2022. CapableWasm: Bringing Better Interop Down to WebAssembly. https://www.youtube.com/watch?v=E44lTaa2qHk POPL'22 student research competition presentation.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021a. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2022a. *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code.* Technical Report. Aarhus University. https://cs.au.dk/~birke/papers/cerise.pdf

Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu, Alix Devriese, and Lars Birkedal. 2021b. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021.* https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022b. *Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities.* Technical Report. Aarhus University. https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 646–661. https://doi.org/10.1145/3192366.3192381

Pat Hickey. 2020. How Fastly and the developer community are investing in the WebAssembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem

Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527326

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 637–650. https://doi.org/10.1145/2676726.2676980

Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, Ricardo Peña and Tom Schrijvers (Eds.). ACM, 13–24. https://doi.org/10.1145/2505879.2505897

Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael Lemay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022. Isolation Without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/3498688

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL (2023), 425–454. https://doi.org/10.1145/3571208

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 395–404. https://doi.org/10.1145/2254064.2254111

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 809–824. https://doi.org/10.1145/3519939.3523432

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, USA, Article 40, 18 pages.

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. *SIGPLAN Not.* 41, 1 (Jan. 2006), 320–333. https://doi.org/10.1145/1111320.1111066

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs (Artefact). https://doi.org/10.5281/zenodo.7808708

Andreas Rossberg. 2019. *WebAssembly Core Specification W3C Recommendation.* Technical Report. W3C. https://www.w3.org/TR/wasm-core-1/

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 775–805. https://doi.org/10.1145/3571220

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. 2019. A Program Logic for First-Order Encapsulated WebAssembly. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30. https://doi.org/10.4230/LIPIcs.ECOOP.2019.9

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 61–79. https://doi.org/10.1007/978-3-030-90870-6_4

Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. 2003. Building Certified Libraries for PCC: Dynamic Storage Allocation. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2618)*, Pierpaolo Degano (Ed.). Springer, 363–379. https://doi.org/10.1007/3-540-36575-3_25

Dachuan Yu and Zhong Shao. 2004. Verification of safety properties for concurrent assembly code. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 175–188. https://doi.org/10.1145/1016850.1016875

Received 2022-11-10; accepted 2023-03-31