A Program Logic for First-Order Encapsulated WebAssembly

Conrad Watt
University of Cambridge, UK
conrad.watt@cl.cam.ac.uk

Petar Maksimović
Imperial College London, UK
Mathematical Institute SASA, Serbia
p.maksimovic@imperial.ac.uk

Neelakantan R. Krishnaswami
University of Cambridge, UK
nk480@cl.cam.ac.uk

Philippa Gardner
Imperial College London, UK
p.gardner@imperial.ac.uk

Abstract

We introduce Wasm Logic, a sound program logic for first-order, encapsulated WebAssembly. We design a novel assertion syntax, tailored to WebAssembly's stack-based semantics and the strong guarantees given by WebAssembly's type system, and show how to adapt the standard separation logic triple and proof rules in a principled way to capture WebAssembly's uncommon structured control flow. Using Wasm Logic, we specify and verify a simple WebAssembly B-tree library, giving abstract specifications independent of the underlying implementation. We mechanise Wasm Logic and its soundness proof in full in Isabelle/HOL. As part of the soundness proof, we formalise and fully mechanise a novel, big-step semantics of WebAssembly, which we prove equivalent, up to transitive closure, to the original WebAssembly small-step semantics. Wasm Logic is the first program logic for WebAssembly, and represents a first step towards the creation of static analysis tools for WebAssembly.

2012 ACM Subject Classification Theory of computation → Separation logic

Keywords and phrases WebAssembly, program logic, separation logic, soundness, mechanisation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.9

Acknowledgements We would like to thank the reviewers, whose comments were valuable in improving the paper. All authors were supported by the EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems" (EP/K008528/1). In addition: Watt was supported by an EPSRC DTP award (EP/N509620/1); Maksimović was supported by the Serbian Ministry of Education and Science through the Mathematical Institute SASA, projects ON174026 and III44006; Krishnaswami was supported by the EPSRC Programme Grant "Semantic Foundations for Interactive Programs" (EP/N02706X/2); and Gardner was supported by the EPSRC Fellowship "VeTSpec: Verified Trustworthy Software Specification" (EP/R034567/1).

1 Introduction

WebAssembly [16] is a stack-based, statically typed bytecode language. It is the first new language to be natively supported on the Web in nearly 25 years, following JavaScript (JS). It was created to act as the safe, fast, portable low-level code of the Web, in answer to the growing sophisticated, computationally intensive demands of the Internet of today, such as 3D visualisation, audio/video processing, and games. For years, developers wishing to execute calculation-heavy programs written in C/C++ on the Web had to compile them to asm.js [17], a subset of JS. In time, such code has become widespread [53, 25, 11], but the fundamental limitations of JS as a compilation target have become too detrimental to ignore. WebAssembly is designed from the ground up to be an efficient, Web-compatible compilation target, obsoleting asm.js and other similar endeavours, such as Native Client [52]. All major browser vendors, including Google, Microsoft, Apple, and Mozilla, have pledged to support WebAssembly, and the past two years have seen a flurry of implementation activity [49].

These facts alone would be enough to motivate that WebAssembly will be an important technology, and a worthy target for formal methods. The designers of WebAssembly have anticipated this, and have specified WebAssembly using a precise formal small-step semantics, combined with a sound type system. Moreover, WebAssembly's semantics, type system, and soundness have already been fully mechanised [46], and the WebAssembly Working Group requires any further additions to WebAssembly to be formally specified.

The main use case for WebAssembly is to inter-operate with JS in creating content for the Web. More precisely, WebAssembly functions can be grouped into *modules*, which provide interfaces through which users can call WebAssembly code, and self-contained (encapsulated) modules can be used as drop-in replacements for their existing JS counterparts and already constitute a major design pattern in WebAssembly. We believe that having a formalism for describing and reasoning about WebAssembly modules and their interfaces is essential, in line with WebAssembly's emphasis on formal methods. Thus far, very little work has been done on static analysis for WebAssembly (cf. §6).

We present Wasm Logic, a sound program logic for reasoning about first-order, encapsulated WebAssembly modules, such as data structure libraries. Enabled by the strong guarantees of WebAssembly's type system, we design a novel assertion syntax, tailored to WebAssembly's stack-based semantics. We further adapt the standard separation logic triple and proof rules in a principled way to capture WebAssembly's uncommon structured control flow.

Having a program logic for WebAssembly is valuable for several reasons. First, as WebAssembly programs are distributed without their originating source code, any client-side verification would have to rely on a WebAssembly-level logic. Similarly, verification techniques such as proof-transforming compilation [31, 1, 26, 37] rely on the existence of a program logic for the target language. Finally, some fundamental data structure libraries are expected to be implemented directly in WebAssembly for efficiency reasons. For example, the structure of B-trees strongly aligns with the way in which WebAssembly memory is managed (cf. §4.2).

To demonstrate the usability of Wasm Logic, we implement, specify, and verify a simple WebAssembly B-tree library. In doing so, we discuss how the new and adapted Wasm Logic proof rules can be used in practice. The specifications that we obtain are abstract, in that they do not reveal any details about the underlying implementation.

We mechanise Wasm Logic and its soundness proof in full in Isabelle/HOL, building on a previous WebAssembly mechanisation of Watt [46]. We prove Wasm Logic sound against a novel, big-step semantics of WebAssembly, and also mechanise a proof of equivalence between the transitive closure of the original small-step semantics and our big-step semantics. Our mechanisation totals ~10,400 lines of non-comment, non-whitespace Isabelle code, not counting code inherited from the existing mechanisation.

2 A Brief Overview of WebAssembly

We give the syntax and an informal description of the semantics of WebAssembly. A precise account of its semantics is given through our program logic in §3 and also through our big-step semantics, introduced in §5 and presented in full in [47].

2.1 WebAssembly Syntax

WebAssembly has a human-readable *text format* based on s-expressions, which we use throughout. The abstract syntax of WebAssembly programs [16], is given in full in Figure 1. As we consider first-order, encapsulated modules, we grey out the remaining, non-relevant syntax. We describe the semantics of the instructions informally in §2.3, and additional syntax as it arises in the paper. A full description of WebAssembly can be found in [16].

$$
\begin{array}{llcl}
\text{(constants)} & k & ::= & \dots \\
\text{(immediates)} & im & ::= & i, a, o \in nat \\
\text{(packed types)} & pt & ::= & \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32} \\
\text{(value types)} & t & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(function types)} & ft & ::= & t^* \to t^* \\
\text{(global types)} & gt & ::= & \mathsf{mut}^? \; t
\end{array}
$$

$$
\begin{array}{rcl}
unop_{iN} & ::= & \mathbf{clz} \mid \mathbf{ctz} \mid \mathbf{popcnt} \\
unop_{fN} & ::= & \mathbf{neg} \mid \mathbf{abs} \mid \mathbf{ceil} \mid \mathbf{floor} \mid \mathbf{trunc} \mid \mathbf{nearest} \mid \mathbf{sqrt} \\
binop_{iN} & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div\_}sx \mid \mathbf{rem\_}sx \mid \mathbf{and} \mid \\
 & & \mathbf{or} \mid \mathbf{xor} \mid \mathbf{shl} \mid \mathbf{shr\_}sx \mid \mathbf{rotl} \mid \mathbf{rotr} \\
binop_{fN} & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \\
 & & \mathbf{min} \mid \mathbf{max} \mid \mathbf{copysign} \\
testop_{iN} & ::= & \mathbf{eqz} \\
relop_{iN} & ::= & \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt\_}sx \mid \mathbf{gt\_}sx \mid \\
 & & \mathbf{le\_}sx \mid \mathbf{ge\_}sx \\
relop_{fN} & ::= & \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt} \mid \mathbf{gt} \mid \mathbf{le} \mid \mathbf{ge} \\
cvtop & ::= & \mathbf{convert} \mid \mathbf{reinterpret} \\
sx & ::= & \mathbf{s} \mid \mathbf{u}
\end{array}
$$

$$
\begin{array}{llcl}
\text{(instructions)} & e & ::= & t.\mathbf{const}\; k \mid \mathbf{drop} \mid \mathbf{nop} \mid \mathbf{select} \mid \mathbf{unreachable} \mid \\
 & & & t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid \\
 & & & t.cvtop_{t\_sx^?} \mid \mathbf{get\_local}\; im \mid \mathbf{set\_local}\; im \mid \\
 & & & \mathbf{tee\_local}\; im \mid \mathbf{get\_global}\; im \mid \\
 & & & \mathbf{set\_global}\; im \mid t.\mathbf{store}\; pt^?\; a\; o \mid \\
 & & & t.\mathbf{load}\; (pt\_sx)^?\; a\; o \mid \mathbf{mem.size} \mid \mathbf{mem.grow} \mid \\
 & & & \mathbf{block}\; ft\; e^*\; \mathbf{end} \mid \mathbf{loop}\; ft\; e^*\; \mathbf{end} \mid \\
 & & & \mathbf{if}\; ft\; e^*\; \mathbf{else}\; e^*\; \mathbf{end} \mid \\
 & & & \mathbf{br}\; im \mid \mathbf{br\_if}\; im \mid \mathbf{br\_table}\; im^+ \mid \\
 & & & \mathbf{return} \mid \mathbf{call}\; im \mid \mathbf{call\_indirect}\; ft \\
\text{(functions)} & func & ::= & ex^*\; \mathbf{func}\; ft\; \mathbf{local}\; t^*\; e^* \mid ex^*\; \mathbf{func}\; ft\; imp \\
\text{(globals)} & glob & ::= & ex^*\; \mathbf{global}\; gt\; e^* \mid ex^*\; \mathbf{global}\; gt\; imp \\
\text{(tables)} & tab & ::= & ex^*\; \mathbf{table}\; n\; im^* \mid ex^*\; \mathbf{table}\; n\; imp \\
\text{(memories)} & mem & ::= & ex^*\; \mathbf{memory}\; n \mid ex^*\; \mathbf{memory}\; n\; imp \\
\text{(imports)} & imp & ::= & \mathbf{import}\; \text{“}name\text{”}\; \text{“}name\text{”} \\
\text{(exports)} & ex & ::= & \mathbf{export}\; \text{“}name\text{”} \\
\text{(modules)} & mod & ::= & \mathbf{module}\; func^*\; glob^*\; tab^?\; mem^?
\end{array}
$$

Note: we denote lists with a $*$ superscript: for example, t^* denotes a list of types.

Figure 1 WebAssembly Abstract Syntax of [16], with aspects not relevant to this work greyed out.

2.2 The WebAssembly Memory Model

Values. WebAssembly values, v, may have one of four *value types*, representing 32- and 64-bit IEEE-754 integers and floating-point numbers: i32, i64, f32, or f64. We denote values using their type: for example, a 32-bit representation of the integer 42 is denoted 42_{i32}. If the type of a value is not given, it is assumed to be i32 by default.

Local and Global Variables. WebAssembly programs have access to statically declared variables, which may be *local* or *global*. Local variables are declared per-function. They live in local variable stores, which exist only in the body of their declaring function. They include function arguments, followed by a number of "scratch" local variables initialised to zero when the function is called. Global variables are declared by the enclosing module. They live in a global variable store, are initialised to zero at the beginning of the execution, and are accessible by all of the functions of the module.

In contrast to most standard programming languages, WebAssembly variables cannot be referenced by name. Instead, both the global and local variable stores are designed as mappings from natural numbers to WebAssembly values, and variables are referenced by their index in the corresponding variable store, as shown in §2.3.

Stack. WebAssembly computation is based on a stack machine: all instructions pop their arguments from and push their results onto a stack of WebAssembly values. By convention, stack concatenation is implicit and the top of the stack is written on the right-hand side: for example, a stack with a 32-bit 0 at its top followed by m WebAssembly values would be denoted as $v^m\ 0$. Note that the type system of WebAssembly allows us to statically know both the number of elements on the stack and their types at every point of program execution.

Memory. WebAssembly has a linear memory model. A WebAssembly memory is an array of bytes, indexed by i32 values, which are interpreted as offsets. Memory is allocated in units of *pages*, and each page is exactly 64k bytes in size.

2.3 WebAssembly Instructions

WebAssembly has a wide array of instructions, which we divide into: basic instructions, variable management instructions, memory management instructions, function-related instructions, and control flow instructions, all of which we discuss below. Every instruction consumes its arguments from the stack, carries out its operation, and pushes any resulting value back onto the stack. Moreover, every instruction is typed, with its type describing the types of its arguments and result. We illustrate how this works in Figure 2, which describes WebAssembly addition of two 32-bit integers starting from an empty stack. In particular, the **i32.const** command, whose type is $[] \to [\mathsf{i32}]$, does not require any arguments and puts the given value on the stack, whereas the **i32.add** instruction, whose type is $[\mathsf{i32}, \mathsf{i32}] \to [\mathsf{i32}]$, takes two arguments from the stack and returns their sum.

WebAssembly gives two official, equivalent, semantics: a semi-formal prose semantics and an entirely formal small-step semantics [50]. In this paper, we introduce an additional, equivalent, big-step semantics as part of the soundness proof of our logic. Most of our diagrams and explanatory text throughout the paper follow the style of the prose semantics, as its treatment of the value stack is most useful in explaining the behaviour of the logic. We denote prose-style execution steps using $\rightsquigarrow$, and introduce the other semantics as necessary.

Basic Instructions. WebAssembly values can be declared using the t.**const** command, typed $[] \to [t]$, in the style of (**i32.const** 2) of Figure 2. The (**drop**) command, typed $[t] \to []$, pops and discards the top stack item, while (**nop**), typed $[] \to []$ has no effect. The (**select**) instruction, typed $[t, t, \mathsf{i32}] \to []$, takes three values from the stack, v_1, v_2, and c. If c is non-zero, v_1 is pushed back onto the stack, and v_2 otherwise. The (**unreachable**) instruction, typed $[] \to t^*$, causes the program to halt with a runtime error, which is represented in WebAssembly by a special **Trap** execution result (cf. §2.4).

WebAssembly also provides a variety of (type-annotated) unary and binary arithmetic operations (Figure 1, *unop* and *binop*, respectively), unary and binary logical operations (Figure 1, *testop* and *relop*, respectively), and casting operations (Figure 1, *cvtop*). Some of these operations can cause a **Trap**: for example, if we attempt division by zero or try to convert a floating-point number to an integer when the result is not representable. Their meaning is detailed in [16], and we address them in this paper by need.

Variable Management Instructions. Local and global variables can be read from and written to using the appropriate **get** and **set** instructions, and all variable accesses are performed using static indexes. For example, (**get_local** i), typed $[] \to [t]$ (where t is the statically known type of the i-th local variable), will push the value of the i-th declared local variable of the current function onto the stack, and (**set_global** i), typed $[t] \to []$, will set

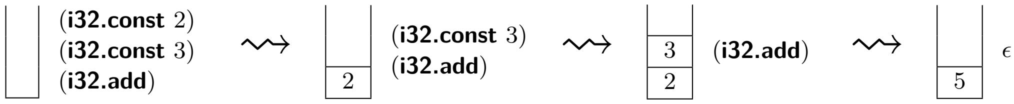


Figure 2 Addition in WebAssembly.

the value of the i-th declared global variable to the value at the top of the stack, which is consumed in the process. It is also possible to set a local variable without consuming this value from the stack by using the **tee_local** instruction, typed $[t] \to [t]$.

Memory Management Instructions. Stack values may be serialised and copied into the appropriate number of bytes in memory through the type-annotated **store** instruction. The (t.**store**) instruction, typed $[\mathsf{i32}, t] \to []$, interprets its i32 argument as an index into the memory, while the second is serialised into the appropriate number of bytes to be stored sequentially, starting from the indexed memory location.

Conversely, the type-annotated **load** instruction reads bytes from the memory and produces the appropriate stack value. (t.**load**), typed $[\mathsf{i32}] \to [t]$, will consume a single i32 value (the address), and then read the appropriate number of bytes starting from that address, leaving the corresponding value of type t on the top of the stack. WebAssembly specifies that every value can be serialised, and every byte sequence of the appropriate length can be interpreted as a value; there are no trap representations for values.

The size of the memory can be inspected by executing the (**mem.size**) instruction, typed $[] \to [\mathsf{i32}]$, which returns an 32-bit integer denoting the current memory size in pages. The WebAssembly memory may also be grown by executing the (**mem.grow**) instruction, typed $[\mathsf{i32}] \to [\mathsf{i32}]$, which takes a single i32 value from the top of the stack and attempts to grow the memory by that many pages, returning the previous size of the memory, in pages, as a 32-bit integer if successful. (**mem.grow**) is always allowed to fail non-deterministically, to represent some memory limitation of the host environment. In this case, the memory is not altered, and the value -1_{i32} is returned.

Control Flow Instructions. Most WebAssembly features have many similarities to other bytecodes, such as that of the Java Virtual Machine [24]. WebAssembly's approach to control flow, however, is uncommon. WebAssembly does not allow unstructured control flow in the style of a **goto** instruction. Instead, it has three control constructs that implement structured control flow: (**block** ft e^* **end**), (**loop** ft e^* **end**), and (**if** ft e^* **else** e^* **end**). Each of these control constructs is annotated with a function type ft of the form $t^m \to t^n$, meaning that its body, e^*, requires m elements from the stack and places back n elements onto the stack on exit. The semantics guarantees that this type precisely describes the effect the construct will have on the stack after it/its body terminates. For example, a (**loop** $(t^m \to t^n)$ e^* **end**), no matter the behaviour of its body, will always leave precisely n additional values on the stack upon termination. Control constructs may be nested within each other in the intuitive way. The execution of a control construct consists of executing its body to termination.

Within the body of a control construct, a break instruction, (**br** i), may be executed. As control constructs can be nested, **br** is parameterised by a static index i, indicating the control construct that it targets (indexing inner to outer). The behaviour of **br** depends on the type of its target. When targeting a **block** or an **if**, **br** acts as a "break" statement of a high-level language, which transfers control to the matching **end** opcode, jumping out of all intervening constructs. When targeting a **loop**, the break instruction acts like a "continue"

(**loop** *tf*
    (**if** *tf′* (**br** 0) **else** (**br** 1) **end**)
end)

Figure 3 Example of WebAssembly control flow. Executing (**br** 0) jumps to the end of the **if**, while (**br** 1) jumps to the *start* of the **loop**.

statement, transferring control back to the beginning of the loop. If the body of a **loop** terminates without executing a **br**, the loop terminates with the result of the body. The **br** instruction is, therefore, required for loop iteration. We illustrate this in Figure 3. The first break, (**br** 0), targets its enclosing **if** instruction, meaning that control should be transferred to the end of that **if** instruction. The second break, (**br** 1), targets the outer **loop** instruction, meaning that control should be transferred to the beginning of that loop.

WebAssembly also has two instructions for conditional breaking: **br_if** and **br_table**. The (**br_if** i) instruction takes one i32 value off the stack and, if this value is not equal to zero, behaves as (**br** i), and as (**nop**) otherwise. On the other hand, the (**br_table** $i_0 \ldots i_n$ i) instruction acts like a switch statement. It takes one i32 value v off the stack and then: if $0 \leqslant v \leqslant n$, it behaves as (**br** i_v); otherwise, it behaves as (**br** i).

Function-related Instructions. WebAssembly supports two types of functions. First, the host environment wil supply *import* functions for use by the WebAssembly module. These functions may be JavaScript *host* functions or may come from other WebAssembly modules. Second, the module itself will define its own native WebAssembly functions.

Functions are called using the (**call** i) instruction, which executes the i-th function, indexing imports first, followed by module-native functions in order of declaration. As WebAssembly functions are declared with a precise type annotation, (**call** i) also takes the type of the i-th function. WebAssembly also provides a mechanism for dynamic dispatch through the **call_indirect** instruction.

Our core logic does not support imported functions, as well as the **call_indirect** dynamic dispatch, as all of these features require JavaScript intervention for non-trivial use. Without **call_indirect**, WebAssembly provides no mechanism for higher-order code – this is why we characterise our logic as supporting "first-order, encapsulated WebAssembly". We view these features as part of further work on JavaScript/WebAssembly interoperability and discuss the ramifications of providing support for them in §7.

Finally, the (**return**) instruction is analogous to **br**, except that it breaks out of *all* enclosing constructs, concluding the execution of the function.

Modules. A WebAssembly program is represented as a module, which consists of: a list of functions; a list of global variables; the (optional) **call_indirect** table; and the (optional) linear memory. Formally, this is written as **module** func^* glob^* $\mathit{tab}^?$ $\mathit{mem}^?$. Functions are made of a function type ft, a series of typed local variable declarations t^*, and a function body e^*. Globals are made up of a type declaration gt (including an optional immutable flag for declaring constants) and an initializer expression e^*. Tables collect a list of function indexes for use by the **call_indirect** instruction. Memories declare their initial size measured in pages. Functions, globals, tables, and memories may be shared between modules through a system of imports and exports, but we do not support this in our current logic, in large part because WebAssembly modules cannot satisfy each other's imports natively, but must currently rely on JavaScript "glue code" to compose together.

2.4 WebAssembly Semantics

WebAssembly's official specification [16] provides a formal small-step semantics, mechanised in Isabelle/HOL by Watt [46]. As part of the soundness proof of our program logic, we define and mechanise in Isabelle/HOL a WebAssembly big-step semantics that we formally prove equivalent, up to transitive closure, to the mechanised small-step semantics of [46]. We introduce a fine-grained semantics of the **br** and **return** instructions, which is independent of the style of semantics chosen and streamlines formal reasoning.

Execution Results. WebAssembly executions terminate with one of the following results:

- **Normal** $v*$, representing standard termination with a list of values $v*$ (in future, we often elide the **Normal** constructor and consider it to be the default result type);
- **Trap**, representing a runtime error (cf. §2.3 for examples of instructions that can trap);
- **Break** n $v*$, describing an in-progress **br** instruction;
- **Return** $v*$, describing an in-progress **return** instruction.

Whereas the first two types of results are introduced by Haas et al. in [16], the last two are introduced by us in this paper. The reason for this is that the WebAssembly formal semantics of [16] gives a very coarse-grained semantics to the **br** and **return** instructions. A **br** instruction targetting a control construct is defined as breaking to it immediately in a single step, discarding everything in between, including all other nested control constructs.

This complicates inductive proofs over the semantics, impairing formal reasoning [46]. In fact, this semantics is too coarse-grained for our proof system and we need to introduce the notions of "in-progress" **br** and **return** instructions as explicit execution results.

We illustrate the difference between the approach of Haas et al. [16] and our approach in Figure 4. The top reduction follows the official semantics of [16]. There, (**br** 1) breaks out of two blocks in a single step, transferring exactly one value out of the **block**, in order to satisfy the targeted block's type signature. We make this semantics more granular by introducing an auxiliary **Break** result type. Concretely, **Break** n $v*$ denotes an in-progress **br** instruction, with n remaining contexts to break out of, in the process of transferring $v*$ values to the

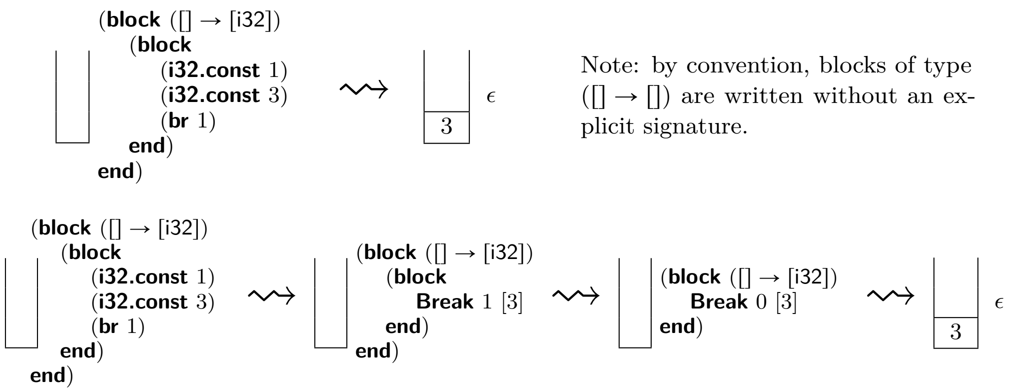


Figure 4 Granularity of **br** executions: Haas et al. [16] (top); our approach (bottom).

target context, as shown in the bottom reduction of Figure 4. Similarly, **Return** v^* represents an in-progress **return** instruction, with the only difference being that **Return** does not require a remaining context count, as it breaks out of all enclosing constructs.

Big-Step Semantic Judgement. The judgement of our big-step semantics is of the form

$$(s, loc^*, v_e^* \, e^*) \Downarrow_{inst}^{labs,ret} (s', loc'^*, res).$$

On the left-hand side of the judgement, we have *configurations* of the form $(s, loc^*, v_e^* \, e^*)$, where s is a *store* containing whole-program runtime information (e.g. global variables and the memory), loc^* is the list of current local variables, and v_e^* is a value stack v^* lifted to **const** instructions, which is then directly concatenated with e^*, the list of instructions to execute.[1] Configuration execution yields an updated store s', updated local variables loc'^*, and a result res, which has one of the four above-mentioned result types.

Additionally, execution is defined with respect to a subscript *inst*. This is the *run-time instance*, a record which keeps track of which elements of s have been allocated by the current program. In the case of the encapsulated modules that we consider, its role in the formalism is trivial, but its full role is described in the official specification [16], and we give a full definition in [47], along with our big-step semantics.

Finally, execution is also defined with respect to a list of break label arities *labs* (a *nat list*), and a return label arity *ret* (a single *nat*). As depicted in Fig. 4, **Break** and **Return** results must transfer precisely the correct number of values to satisfy the type of the context it is targeting. The *labs* and *ret* parameters keep track of the number of values required, so that, for example, if *res* is of the form **Break** $k\ v^n$, then $labs!k = n$. Similarly, if *res* is of the form **Return** v^n, then $ret = n$.

Equivalence Result. We recall the original formal small-step semantic judgement of [16], which is of the form $(s, loc^*, v_e^* \, e^*) \hookrightarrow_{inst} (s', loc'^*, v_e'^* e'^*)$. This judgement does not include our break or return labels.

We state our equivalence result in Theorem 1 and mechanise its proof in Isabelle/HOL. We denote the transitive closure of the small-step semantics by $\hookrightarrow^*$. Both $\hookrightarrow$ and $\Downarrow$ are subscripted by the instance *inst*, the big-step derivation starts with empty *labs* ([]) and empty *ret* (ϵ) components, and v'^* denotes the list of values obtained from $v_e'^*$ by removing their leading **const**s.

▶ **Theorem 1** (reduce_trans_equiv_reduce_to)**.**

$$(s, loc^*, v_e^* \, e^*) \hookrightarrow_{inst}^* (s', loc'^*, v_e'^*) \Longleftrightarrow (s, loc^*, v_e^* \, e^*) \Downarrow_{inst}^{[],\epsilon} (s', loc'^*, \textbf{Normal}\ v'^*) \ \wedge$$
$$(s, loc^*, v_e^* \, e^*) \hookrightarrow_{inst}^* (s', loc'^*, [\textbf{trap}]) \Longleftrightarrow (s, loc^*, v_e^* \, e^*) \Downarrow_{inst}^{[],\epsilon} (s', loc'^*, \textbf{Trap})$$

Theorem 1 relates terminal states (values e^* or a trap result [**trap**]) in the small step semantics with execution results in the big-step semantics. In particular, it shows that the small- and big-step semantics give equivalent results for all terminating programs. The proof also requires auxiliary lemmas about how the big-step **Break** and **Return** execution results correspond to behaviours in the small-step semantics. These lemmas are not included here for space, but can be found in the mechanisation.

[1] This treatment of the value stack is a key difference between the official prose and formal semantics. In the prose semantics, the stack is represented as a list of values v^*, together with an executing list of instructions e^*, which modifies the stack. In the formal semantics, the value stack is represented as a list of **const** instructions, and directly concatenated with the executing list of instructions to form a single list. Reduction rules are defined between configurations, pattern-matching between **const** instructions and other instructions, such as **add**, without ever explicitly manipulating a separate value stack.

3 Wasm Logic

We present Wasm Logic, a program logic for first-order, encapsulated WebAssembly modules. We define a novel assertion syntax, with a highly structured stack assertion which takes advantage of WebAssembly's strict type system. Our proof rules for the WebAssembly **br** and **return** instructions are inspired by a foundational proof rule for "structured **goto**" by Clint and Hoare [7], and extend their work to the world of separation logic [35]. We fully mechanise and prove soundness of Wasm Logic in Isabelle/HOL, as detailed in §5.

3.1 Assertion Language

Wasm Logic assertions encode information about WebAssembly runtime states. Their semantic interpretation is formally described in §5, in the context of our soundness result.

In many programming languages, program state is made up of the values stored in *variables* and the values stored in the *heap*. In this case, it is natural for assertions to be expressed using a *separation logic*, which extends predicate logic with connectives for reasoning about resource separation, and is useful for modular client reasoning [35].

WebAssembly, however, also allows values to be stored in the stack. Given how the WebAssembly's type system provides static knowledge of the stack size and of the types of each of its elements at every program point, we believe that reasoning about the WebAssembly stack *should* be simple: that is, it should not result in proofs more complicated than those of traditional separation logic. We manage to achieve this thanks to our structured stack assertion and the associated proof rules. While one's first instinct could be to treat assertions about stack values like assertions about local variables, such a system would require substantial bookkeeping, since the stack changes shape during execution. Benton [3] uses this approach for a language with a similar typed-value stack, but ends up describing the resulting proofs as "fussily baroque" and "extremely tedious to construct by hand".

constants	$c \in \mathit{Const}$	$::=$	$\mathrm{c_{i32}} \mid \mathrm{c_{i64}} \mid \mathrm{c_{f32}} \mid \mathrm{c_{f64}}$
variables (logical/local/global)	$\nu \in \mathcal{V}\mathit{ar}$	$::=$	$x \mid l_i \mid g_i$, where $i \in \mathbb{N}$
terms	$\tau \in \mathcal{T}\mathit{erm}$	$::=$	$c \mid \nu \mid f(\tau_1 \ldots \tau_n)$
heap assertions	$H, H' \in \mathcal{A}_{ph}$	$::=$	$\bot \mid \neg H \mid H \wedge H' \mid$ $\exists x.\ H \mid p(\tau_1 \ldots \tau_n) \mid$ $\mathbf{emp} \mid H * H' \mid \circledast_{\tau_1 < x < \tau_2} H \mid$ $\tau_1 \mapsto \tau_2 \mid \mathbf{size}(\tau)$
stack assertions	$S \in \mathcal{A}_s$	$::=$	$[] \mid S :: \tau$
assertions	$P, Q \in \mathcal{A}$	$::=$	$(S \mid H) \mid \exists x.\ P$

$$[\exists \vec{x}.\ (S \mid H)] \otimes H_f \ \triangleq\ \exists \vec{x}.\ (S \mid H * H_f) \quad \text{iff } \mathit{fv}(H_f) \cap \vec{x} = \varnothing$$

Figure 5 Syntax of Wasm Logic assertions.

The syntax of Wasm Logic assertions is defined in Fig. 5. Constants, c, can have one of the four WebAssembly value types. Next we have logical, local, and global variables, with local/global variables having dedicated variable names, l_i/g_i, where $i \in \mathbb{N}$. Terms can either be constants, or variables, or functions (for example, unary and binary operators).

Heap assertions are mostly familiar from traditional separation logic [35]. First, we have the pure assertions of predicate logic, including predicates $p(\tau_1 \dots \tau_n)$ over terms (for example, term equality). We also have the standard spatial assertions: emp describes an empty heap, $H * H'$ is the separating conjunction (star), and the iterated star operator, $\circledast$, aggregates assertions composed by $*$ in the same way that $\sum$ aggregates arithmetic expressions composed by $+$. Finally, we have two WebAssembly-specific spatial assertions: the cell assertion $\tau_1 \mapsto \tau_2$ describes a single heap cell at address denoted by τ_1 with contents denoted by τ_2, and the **size**(τ) assertion states that the number of pages currently allocated is denoted by τ.

A stack assertion, denoted by S, is a list of terms, each of which represents the value of the corresponding stack position in the value stack. This is possible due to the size of the WebAssembly stack always being precisely known statically. Were this not true, the stack assertion would need to be able to represent that the stack may have multiple sizes, and could not be represented purely as a single list of terms. The list appends on the right, to match the conventions of the WebAssembly type system.

Finally, a Wasm Logic assertion is a two-part, possibly existentially quantified assertion consisting of a stack assertion S, and a pure/heap assertion H. We define an operator, $\otimes$, for distributing heap frames through Wasm Logic assertions, which will be used later in §3.3 to define our frame rule. The notation $\exists\overrightarrow{x}$ is a shorthand for some set of outer existentially quantified variables, while $fv(H_f)$ returns the set of *free variables* in the heap assertion H_f.

Notation. For clarity of presentation, we introduce the following notational conventions:

- (Stack Length) We denote by P_n an assertion whose stack part is of length n.
- (Type Annotations in Cell Assertions) The cell assertion $\tau_1 \mapsto \tau_2$ encodes the value of a single byte in memory. As WebAssembly values normally take up either four or eight bytes, it is convenient for us to define the corresponding shorthand, which we do by annotating the arrow with the appropriate type: $\tau_1 \mapsto_t \tau_2$. For example, we have that $\tau_1 \mapsto_{i32} \tau_2 \triangleq \tau_1 \mapsto b_0 * (\tau_1 + 1) \mapsto b_1 * (\tau_1 + 2) \mapsto b_2 * (\tau_1 + 3) \mapsto b_3$, where b_k denotes the k^{th} least significant byte of the 32-bit representation of τ_2.
- (Operator Domain) To avoid clutter, we overload all mathematical operators (e.g., $+$, $\cdot$, $\leqslant$, $\dots$) instead of explicitly stating their domain (i32, i64, f32, f64, $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$) on each use. When required, we state the domain either of a single operator (e.g., $+_{\mathsf{i32}}$, $+_{\mathsf{i64}}$, $\dots$) or of a parenthesised expression (e.g., $(3.14 - 2.71 \cdot x)_{\mathsf{f64}})$), in which case the domain applies to all operators and operands of the expression. The default domain is i32.

3.2 Wasm Logic Triple

We define a program logic for first-order, encapsulated WebAssembly modules. We base our encoding of program behaviour on *Hoare triples* [18]. Wasm Logic triples are of the form

$$\Gamma \vdash \{P\}\ e^*\ \{Q\}$$

where e^* is the WebAssembly program to be executed, P is its *pre-condition*, Q is its *post-condition*, and Γ represents the context in which the program is executed.

Before giving the interpretation of the Wasm Logic triple, we have to explain the context Γ in detail. A context contains four fields: **(1)** the *functions* field, F, containing a list of all function definitions of the module; **(2)** the *assumptions* field, A, containing a set of assertions of the form $\{P\}$ **call** i $\{Q\}$, used by the [call] rule to correctly capture mutually recursive functions; **(3)** the *labels* field, L, containing a list of assertions used to describe the behaviour of the **br** instruction; and **(4)** the *return* field, R, containing an optional return assertion, used to describe the behaviour of the **return** instruction. A context may be alternatively presented as (F, A, L, R), and any of its fields may be referenced directly: for example, $\Gamma.F$ refers to the functions field of the context. We use $P; \Gamma$ as syntactic shorthand for Γ with P appended to the head of its labels field, since this pattern occurs commonly.

Interpretation of Wasm Logic Triples. The meaning of the triple $\Gamma \vdash \{P\}\ e^*\ \{Q\}$ is, informally, as follows. Let e^* be executed from a state satisfying P. Then: if e^* terminates normally, it will terminate in a state satisfying Q; if it terminates with a **Return** v^* result, the resulting state must satisfy $\Gamma.R$; and if it terminates with a **Break** $i\ v^*$ result, the resulting state must satisfy the i-th assertion of $\Gamma.L$. A formal definition is given in §5.

3.3 Proof Rules

Basic Instructions. The proof rules for basic instructions are given in Figure 6. These rules manipulate only the stack and pure logical assertions, and can be intuitively motivated by their effects on the stack. In particular, the effect of the [select] rule is conditional on the value of τ_3: we know that it has placed exactly one value on the stack, but whether it is τ_1 or τ_2 depends on whether or not $\tau_3 \neq 0$. These rules, despite manipulating the WebAssembly stack, appear very standard: this is precisely due to our structured stack assertions.

Variable Management Instructions. We give the proof rules for variable management instructions in Figure 7 (left). Just like the rules for basic instructions, these also require an empty heap. By observing these rules, we can understand how the dedicated local/global variable names are manipulated. For example, (**get_local** i) simply puts the variable l_i on

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ t.\mathsf{const}\ c\ \{[c] \mid \mathbf{emp}\}}\,[\text{const}] \qquad \frac{}{\Gamma \vdash \{[] \mid \bot\}\ \mathsf{unreachable}\ \{Q\}}\,[\text{unreachable}]$$

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \mathsf{nop}\ \{[] \mid \mathbf{emp}\}}\,[\text{nop}] \qquad \frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ \mathsf{drop}\ \{[] \mid \mathbf{emp}\}}\,[\text{drop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2, \tau_3] \mid \mathbf{emp}\}\ \mathsf{select}\ \{\exists x.\ [x] \mid \mathbf{emp} \wedge (\tau_3 \neq 0 \to x = \tau_1) \wedge (\tau_3 = 0 \to x = \tau_2)\}}\,[\text{select}]$$

$$\frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ t.unop\ \{[unop(\tau)] \mid \mathbf{emp}\}}\,[\text{unop}] \qquad \frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ t.testop\ \{[testop(\tau)] \mid \mathbf{emp}\}}\,[\text{testop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid \text{defined}(binop, \tau_1, \tau_2) \wedge \mathbf{emp}\}\ t.binop\ \{[binop(\tau_1, \tau_2)] \mid \mathbf{emp}\}}\,[\text{binop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid \mathbf{emp}\}\ t.relop\ \{[relop(\tau_1, \tau_2)] \mid \mathbf{emp}\}}\,[\text{relop}]$$

$$\frac{}{\Gamma \vdash \{[\tau] \mid \text{defined}(cvtop, \tau) \wedge \mathbf{emp}\}\ t.cvtop\ \{[cvtop(\tau)] \mid \mathbf{emp}\}}\,[\text{cvtop}]$$

Note: The defined$(binop, \tau_1, \tau_2)$ and defined$(cvtop, \tau)$ predicates describe conditions sufficient for binary and conversion operators to be non-trapping.

Figure 6 Proof Rules: Basic Instructions.

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \textsf{get\_local}\ i\ \{[l_i] \mid \mathbf{emp}\}}\ [\text{get\_local}]$$

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textsf{set\_local}\ i\ \{[] \mid \mathbf{emp} \wedge l_i = x\}}\ [\text{set\_local}]$$

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textsf{tee\_local}\ i\ \{[x] \mid \mathbf{emp} \wedge l_i = x\}}\ [\text{tee\_local}]$$

$$\frac{\text{isDeclaredGlobal } i}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \textsf{get\_global}\ i\ \{[g_i] \mid \mathbf{emp}\}}\ [\text{get\_global}]$$

$$\frac{\text{isDeclaredGlobal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textsf{set\_global}\ i\ \{[] \mid \mathbf{emp} \wedge g_i = x\}}\ [\text{set\_global}]$$

$\{[] \mid l_1 = 2 \wedge \mathsf{emp}\}$
(**get_local** 1)
$\{[l_1] \mid l_1 = 2 \wedge \mathsf{emp}\}$
$\{[2] \mid l_1 = 2 \wedge \mathsf{emp}\}$
(**i32.const** 3)
$\{[2, 3] \mid l_1 = 2 \wedge \mathsf{emp}\}$
(**i32.add**)
$\{[5] \mid l_1 = 2 \wedge \mathsf{emp}\}$

Note: The (isDeclaredLocal i) and (isDeclaredGlobal i) predicates are an internal detail of the meta-theory ensuring that l_i and g_i do not refer to local/global variables that are not declared by the module. They always hold for any well-typed WebAssembly program.

Figure 7 Proof Rules: Variable Management Instructions (left); Simple Proof Sketch (right).

the top of the stack. On the other hand, (**set_global** i) requires one value from the value stack in the pre-condition, and in the post-condition has consumed it, and guarantees that g_i, the i-th global variable, holds this value.

In Figure 7 (right), we give a proof sketch of a simple WebAssembly program that uses basic and variable management instructions, illustrating how stack assertions behave. We start from the pre-condition $\{[] \mid l_1 = 2 \wedge \mathsf{emp}\}$, which tells us that the stack and the heap are empty and that the first local variable, l_1, equals 2. Executing (**get_local** 1) adds l_1 to the stack, which we can immediately replace with 2 due to our pure knowledge that $l_1 = 2$. The second line of the program pushes the constant 3 onto the stack (the top of the stack is on the *right-hand side* of the assertion). Finally, the two values are added together, and the resulting stack holds a single value, 5.

Memory Management Instructions. Proof rules for instructions that interact with the WebAssembly memory are given in Figure 8. The (t.**load**) and (t.**store**) proof rules are similar to standard separation heap rules, except that they are annotated with the type of the value in the heap, which determines the number of bytes that this value occupies, and also a static offset, which is added to the given address.

As discussed, the (**mem.size**) and (**mem.grow**) instructions allow WebAssembly to alter the memory size. The "permission" to observe the memory size is encoded using the **size**(τ) assertion, which states that the memory is currently τ pages long. This permission, however, does not imply permission to access in-bounds locations; the logic still requires $x \mapsto_t n$ to be held in order to access the location x, even if x is known to be in-bounds because **size** is held. Growing the memory using the (**mem.grow**) instruction confers ownership of all newly created locations, and leaves the index of the first newly allocated location on the stack.

Control Flow Instructions. The proof rules for WebAssembly control constructs are given in Figure 9. These rules illustrate how the labels (L) and return (R) fields of the context are used in practice. In particular, L contains a list of assertions, and the i-th assertion describes the state that has to hold if we break out of i enclosing contexts. Similarly, the R assertion describes the state that has to hold if we execute a function return.

In line with this, the precondition of (**br** i) in the [br] rule equals the i-th assertions of L. On the other hand, its post-condition is arbitrary, which is justified by the fact that any code following a **br** instruction in the same block of code cannot be reached due to the structured control flow of WebAssembly. Analogously, the precondition of a (**return**) statement in the [return] rule equals the return field of the context, and its post-condition is arbitrary. Observe the clear analogy between the role of *labs* and *ret* in the semantics and the role of L and R in the proof rules for **br** and **return**, respectively.

The main aspect of the [block] and [loop] rules is how they interact with the context. Concretely, in the [block] rule, the labels field is extended with the post-condition of the block, whereas in the [loop] rule, it is extended with its pre-condition. Bearing in mind the [br] rule, this precisely captures the WebAssembly control flow: when we break to a block, we exit the block, and when we break to a loop, we continue with the next iteration and the pre-condition of the loop acts as its invariant.

This approach is inspired by the proof rule for "structured" **goto** statements of Clint and Hoare [7], as WebAssembly's **block** and **br** opcodes replicate the structural conditions imposed by [7] on the use of **goto**. Note also that the explicit type annotations of [block] and [loop], combined with the guarantees of the WebAssembly type system, allow the rules to precisely fix the size of the stack in both the pre- and post-condition.

Next, the [if] rule branches depending on the value that is on the top of the stack. If this value is non-zero, the **then** branch is taken, and the **else** branch otherwise. As is commonplace, the post-conditions of the two **if** branches have to match.

The [br_if] rule is a conditional break. If the break is taken, the value on the top of the stack is popped, and known to be non-zero, and the instruction functions identically to **br**. The post-condition represents the case where the break is not taken: the value on the top of the stack is popped, and known to be 0.

Finally, the **br_table** instruction acts like the switch statement of modern languages, breaking to the appropriate label depending on the value on the top of the stack.

Structural Proof Rules. Structural proof rules, shown in Figure 10 and demonstrated in practice throughout §4, are needed to compose proofs together. The [seq] rule for program concatenation is inherited from standard separation logic, whereas the others are either new or require adjustment for Wasm Logic.

$$\frac{}{\Gamma \vdash \{[\tau_1] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}\ t.\mathbf{load}\ \mathit{off}\ \{[\tau_2] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}}\ [\text{load}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid (\tau_1 + \mathit{off}) \mapsto_t -\}\ t.\mathbf{store}\ \mathit{off}\ \{[] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}}\ [\text{store}]$$

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{size}(\tau)\}\ \mathbf{mem.size}\ \{[\tau] \mid \mathbf{size}(\tau)\}}\ [\text{mem.size}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1] \mid \mathbf{size}(\tau_2)\}\ \mathbf{mem.grow}\ \left\{ \exists v.\ [v] \;\middle|\; \begin{array}{l} \left(\begin{array}{l} \circledast_{\tau_2 \leqslant i/64k < (\tau_2+\tau_1)} i \mapsto 0 * \mathbf{size}(\tau_2 + \tau_1) \\ \wedge\ v = \tau_2 \wedge ((\tau_2 + \tau_1) \leqslant 2^{16})_{\mathbb{N}} \end{array} \right) \\ \vee\ (\mathbf{size}(\tau_2) \wedge v = -1) \end{array} \right\}}\ [\text{mem.grow}]$$

Figure 8 Proof Rules: Memory Management Instructions.

$$\frac{L!i = P}{F, A, L, R \vdash \{P\}\ \textbf{br}\ i\ \{Q\}}\text{[br]} \qquad \frac{}{F, A, L, R \vdash \{R\}\ \textbf{return}\ \{Q\}}\text{[return]}$$

$$\frac{Q_m\ ; \Gamma \vdash \{P_n\}\ e^*\ \{Q_m\}}{\Gamma \vdash \{P_n\}\ \textbf{block}\ t^n\ \to t^m\ e^*\ \textbf{end}\ \{Q_m\}}\text{[block]} \qquad \frac{P_n\ ; \Gamma \vdash \{P_n\}\ e^*\ \{Q_m\}}{\Gamma \vdash \{P_n\}\ \textbf{loop}\ t^n\ \to t^m\ e^*\ \textbf{end}\ \{Q_m\}}\text{[loop]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{S \mid H \wedge \tau \neq 0_{i32}\}\ \textbf{block}\ \mathit{tf}\ e_1^*\ \textbf{end}\ \{Q\} \\ \Gamma \vdash \{S \mid H \wedge \tau = 0_{i32}\}\ \textbf{block}\ \mathit{tf}\ e_2^*\ \textbf{end}\ \{Q\}\end{array}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{if}\ \mathit{tf}\ e_1^*\ \textbf{else}\ e_2^*\ \ \textbf{end}\ \{Q\}}\text{[if]} \qquad \frac{\Gamma \vdash \{S \mid H \wedge \tau \neq 0_{i32}\}\ \textbf{br}\ i\ \{Q\}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{br\_if}\ i\ \{S \mid H \wedge \tau = 0_{i32}\}}\text{[br\_if]}$$

$$\frac{\begin{array}{c}\forall k.\, 0 \leqslant k < \mathsf{llen}(i^*) \to \Gamma \vdash \{S \mid H \wedge \tau = k_{\mathrm{i32}}\}\ \textbf{br}\ (i^*!k)\ \{Q\} \\ \Gamma \vdash \{S \mid H \wedge \neg(0 \leqslant \tau < \mathsf{llen}(i^*))_{\mathrm{i32}}\}\ \textbf{br}\ i\ \{Q\}\end{array}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{br\_table}\ i^*\ i\ \{Q\}}\text{[br\_table]}$$

Figure 9 Proof Rules: Control Flow Instructions.

$$\frac{\Gamma \vdash \{P\}\ e_1^*\ \{Q\} \quad \Gamma \vdash \{Q\}\ e_2^*\ \{R\}}{\Gamma \vdash \{P\}\ e_1^*\ e_2^*\ \{R\}}\text{[seq]} \qquad \frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\}}{F, A, (\mathit{map}\ (\exists x.)\ L), (\exists x.\ R)^? \vdash \{\exists x.\ P\}\ e^*\ \{\exists x.\ Q\}}\text{[exists]}$$

$$\frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\} \quad \mathit{fv}(H) \cap \mathit{mv}(e^*) = \varnothing}{F, A, (\mathit{map}\ (\otimes\ H)\ L), (R \otimes H)^? \vdash \{P \otimes H\}\ e^*\ \{Q \otimes H\}}\text{[frame]}$$

$$\frac{\begin{array}{c}F, A, L', R'_{n'} \vdash \{P'\}\ e^*\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \mathsf{llen}(L) = \mathsf{llen}(L') \\ \forall i < \mathsf{llen}(L).\exists L_n\ L'_{n'}.\ L!i = L_n \wedge L'!i = L'_{n'} \wedge n' \leqslant n \wedge L'_{n'} \Rightarrow L_n \quad n' \leqslant n \wedge R'_{n'} \Rightarrow R_n\end{array}}{F, A, L, R_n \vdash \{P\}\ e^*\ \{Q\}}\text{[consequence]}$$

$$\frac{\Gamma \vdash \{\exists \vec{x}.\ (S_p \mid H)\}\ e^*\ \{\exists \vec{y}.\ (S_q \mid H')\} \qquad fv(S_k) \cap (mv(e^*) \cup \vec{x} \cup \vec{y}) = \varnothing}{\Gamma \vdash \{\exists \vec{x}.\ (S_k; S_p \mid H)\}\ e^*\ \{\exists \vec{y}.\ (S_k; S_q \mid H')\}}\text{[extension]}$$

$$\frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\}}{F, A, (L; L_f), R \vdash \{P\}\ e^*\ \{Q\}}\text{[context]}$$

Note: $mv(e^*)$ denotes the set of local and global variables modified by the execution of e^*.

Figure 10 Proof Rules: Structural.

The existential elimination rule, [exists], has to eliminate the existential from all assertions in L and also the R. If we were only to eliminate the existential from the pre- and post-condition, as is standard, the rule would be unsound, as we could derive the following:

$$\frac{-, -, [([] \mid l_0 = k)], - \vdash \{[] \mid l_0 = k\}\ (\textbf{br}\ 0)\ \{Q\}}{-, -, [([] \mid l_0 = k)], - \vdash \{\exists k'.\ [] \mid l_0 = k'\}\ (\textbf{br}\ 0)\ \{\exists x.\ Q\}}\text{[unsound exists]}$$

which does not correspond to the intended meaning of the context, as the pre-condition of the break no longer implies its matching assertion in L. For similar reasons, the [frame] rule must frame off from all assertions in L and also the R. As shown in §4, we can derive simpler proof rules for straight-line code that do not require irrelevant manipulation of L and R.

$$\frac{\begin{array}{c} \mathit{func} = \textsf{func}\ t^n \to t^m\ \textsf{local}\ t^k\ e^* \quad S_n = [x_0..x_{n-1}] \quad \forall i.\ l_i \notin fv(S_n) \cup fv(H) \cup fv(Q_m) \\ (F, A, [Q_m], Q_m) \vdash \{[] \mid H \wedge \bigwedge_{0 \leqslant i < n}(l_i = x_i) \wedge \bigwedge_{n \leqslant i < n+k}(l_i = 0)\}\ e^*\ \{Q_m\} \end{array}}{F, A, L, R \vdash \{S_n \mid H\}\ \textsf{callcl}\ \mathit{func}\ \{Q_m\}}\ [\text{function}]$$

$$\frac{\{P\}\ \textsf{call}\ i\ \{Q\} \in A(\Gamma) \quad i < \textsf{llen}(F(\Gamma))}{\Gamma \vdash \{P\}\ \textsf{call}\ i\ \{Q\}}\ [\text{call}]$$

$$\frac{\forall(\{P\}\ e^*\ \{Q\}) \in \mathit{specs}.\ \Gamma \vdash \{P\}\ e^*\ \{Q\}}{\Gamma \Vdash \mathit{specs}}\ [\text{specsI}] \qquad \frac{\Gamma \Vdash \mathit{specs} \quad (\{P\}\ e^*\ \{Q\}) \in \mathit{specs}}{\Gamma \vdash \{P\}\ e^*\ \{Q\}}\ [\text{specsE}]$$

$$\frac{\begin{array}{l} \forall \mathit{spec} \in \mathit{specs}.\ \mathit{spec} = \{\_\}\ \textsf{call}\ \_\ \{\_\} \\ F, \mathit{specs}, [], [] \Vdash \{\ \{P\}\ \textsf{callcl}\ (F!j)\ \{Q\} \mid \{P\}\ \textsf{call}\ j\ \{Q\} \in \mathit{specs}\ \} \end{array}}{F, [], [], [] \Vdash \mathit{specs}}\ [\text{module}]$$

Figure 11 Proof Rules: Function-Related Instructions, Modules.

In addition to the standard strengthening of the pre-condition and weakening of the post-condition, the [consequence] rule allows us to weaken the assertions in L and also the R. This weakening comes with a side condition that we are not allowed to increase the number of elements on the corresponding stack, which comes from the intuition that breaking out carrying n values does not necessarily imply that we can break out with $n + 1$ values. The [consequence] rule uses the entailment relation of Wasm Logic, denoted by $P \Rightarrow Q$ and defined in the standard way in §5, Figure 16.

The two new rules introduced for Wasm Logic are [extension] and [context]. The [extension] rule is the analog of [frame] for stacks, and it allows us to arbitrarily extend the "bottom" of the stack. This, in turn, enables the proof rules of Figures 6, 7, and 8 to be generalised to arbitrary stacks, with the rules modifying only the head. The [context] rule allows us to remove unneeded assertions from L and also, potentially, R. This rule is sound because the triple encodes that e^*, when executed, will only jump to targets in L, so it is trivially correct for L to be further enlarged.

Function-Related Instructions, Modules. The proof rules for function-related instructions and modules are given in Fig. 11. We give a unified semantics to function calls in WebAssembly through the auxiliary **callcl** instruction and the corresponding [function] rule, which we now explain in detail. First, when inside a function body, if we execute (**br** 0) at top-level or (**return**) anywhere, the function terminates. For this reason, the context from which we start proving a function body has the labels and the return field set to the post-condition of the function Q_m. Next, as previously described, the function arguments are taken from the stack. Therefore, we require the length of the stack to match the number of function parameters, n, given in the function definition. Next, the n arguments themselves are transferred into the first n local variables (l_0 through l_{n-1}), whereas the remaining declared local variables (l_n through l_{n+k}) are set to 0. Finally, as local variables are declared per-function, we forbid function pre- and post-conditions from talking about local variables altogether in order to avoid name clashes. Note that, as with [block] and [loop], the function's explicit type annotation allows us to precisely fix the stack size of both the pre- and post-condition.

At the top level, we have rules for proving specifications for sets of mutually recursive functions. We follow the strategy described by Oheimb [33] and Nipkow [30]. There, each individual function body is initially proven while assuming the specifications of all other functions (the [function] rule), recursive calls and calls to other functions only use the assumptions (the [call] rule), and from this, it can be concluded that all function specifications are correct without any assumptions (the [module] rule).

4 Using Wasm Logic: A Verified B-Tree Library

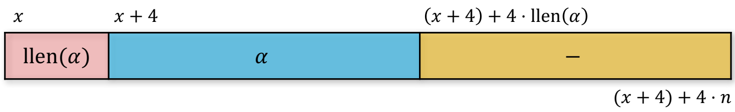
We demonstrate the applicability of Wasm Logic by specifying and verifying a simple WebAssembly B-tree library. B-trees are one of the data structures that we expect to be implemented directly in WebAssembly for efficiency reasons. In particular, a B-tree node commonly occupies an entire page of secondary storage (for example, a hard drive) and WebAssembly memory is allocated in pages. Our B-tree implementation is underpinned by the ordered, bounded array data structure, which we use to demonstrate in detail how Wasm Logic rules can be used in practice (§4.1). We focus on the two non-standard aspects of the logic: stack manipulation and the interplay between structural rules (framing, existential variable elimination, and consequence) and WebAssembly's control flow. We further describe the structure of the B-trees that we implement and present abstract specifications for some of the main B-tree operations (§4.2). The full details of our B-tree implementation are available in the accompanying technical report [47].

Additional Notation (Lists/Sets). We denote: the empty list by [ ]; the list resulting from prepending an element a to a list α by $a{:}\alpha$; concatenation of two lists α and β by $\alpha \cdot \beta$; the length of a list α by $\mathsf{llen}(\alpha)$; the n-th element of a list α by $\alpha!n$; the sublist of a list α starting from index k and containing n elements by $\mathsf{SubList}(\alpha, k, n)$; and the set corresponding to a list α by $\mathsf{ToSet}(\alpha)$. We also denote the number of elements of a set X by $\mathsf{card}(X)$.

4.1 Ordered, Bounded Arrays in WebAssembly

An ordered, bounded array (OBA) is an array whose elements are ordered and which has a fixed upper bound on the number of elements it can contain. We have found OBAs to be an appropriate data structure for representing B-tree nodes, as discussed in detail in [47].

In separation logic, it is commonplace to describe data structures using *abstract predicates* in order to abstract their implementation and simplify the textual representation of the associated proofs.[2] We define the abstract predicate for a 32-bit OBA at address x, with maximum size n and contents α, written $\mathsf{OBA}(x, n, \alpha)$. Informally, the layout of OBAs in memory, illustrated below, is as follows: the first 32-bit cell holds the length of the list α; the next $\mathsf{llen}(\alpha)$ 32-bit cells hold the contents of the list α; and the remaining $(n - \mathsf{llen}(\alpha))$ 32-bit cells constitute over-allocated space.



[2] In some separation logics, abstract predicates are distinct formal entities, but in Wasm Logic they are simply a syntactic shorthand for some particular assertion.

$\{[x, k] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$
(**func** OBAGet [i32, i32] → [i32]
  $\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \wedge l_0 = x \wedge l_1 = k\}$

frame
    $\{[] \mid \mathsf{emp}\}$
    (**get_local** 0)
    $\{[l_0] \mid \mathsf{emp}\}$

extension
      $\{[] \mid \mathsf{emp}\}$
      (**get_local** 1)
      $\{[l_1] \mid \mathsf{emp}\}$

    $\{[l_0, l_1] \mid \mathsf{emp}\}$
    (**i32.const** 4)
    $\{[l_0, l_1, 4] \mid \mathsf{emp}\}$
    (**i32.mul**)(**i32.add**)
    $\{[l_0 + 4 \cdot l_1] \mid \mathsf{emp}\}$
  $\{[l_0 + 4 \cdot l_1] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \wedge l_0 = x \wedge l_1 = k\}$
  $\{[x + 4 \cdot k] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$ (by consequence)
  [[Unfold $\mathsf{OBA}(x, n, \alpha)$]]

$$\left\{ \begin{array}{l} [x + 4 \cdot k] \mid (x \mapsto_{i32} \mathsf{llen}(\alpha) * \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 + 4 \cdot i \mapsto_{i32} \alpha!i) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge \\ \qquad (\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \wedge (x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \end{array} \right\}$$

frame
    $\{[x + 4 \cdot k] \mid x + 4 + 4 \cdot k \mapsto_{i32} \alpha!k\}$
    (**i32.load** offset=4)
    $\{[\alpha!k] \mid x + 4 + 4 \cdot k \mapsto_{i32} \alpha!k\}$

$$\left\{ \begin{array}{l} [\alpha!k] \mid (x \mapsto_{i32} \mathsf{llen}(\alpha) * \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 + 4 \cdot i \mapsto_{i32} \alpha!i) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge \\ \qquad (\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \wedge (x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \end{array} \right\}$$

  [[Fold $\mathsf{OBA}(x, n, \alpha)$]]
  $\{[\alpha!k] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$
end)
$\{[\alpha!k] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$

Figure 12 OBAGet: Specification and Verification.

Formally, the definition of the $\mathsf{OBA}(x, n, \alpha)$ predicate is:

$$\mathsf{OBA}(x, n, \alpha) := (x \mapsto_{i32} \mathsf{llen}(\alpha) * \mathsf{Aseg}(x + 4, \alpha) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge$$
$$(\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \wedge (x + 4 + 4 \cdot n) \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}),$$

where: the predicate $\mathsf{Aseg}(x, \alpha)$ describes the contents as an array segment:

$$\mathsf{Aseg}(x, \alpha) := \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 \cdot i \mapsto_{i32} \alpha!i);$$

the predicate $\mathsf{Ordered}(\alpha)$ denotes that α is ordered in ascending order:

$$\mathsf{Ordered}(\alpha) := \forall i.\, 0 < i < \mathsf{llen}(\alpha) \Rightarrow \alpha!(i - 1) \leqslant \alpha!i;$$

and INT32_MAX denotes the maximal positive integer of i32. Additionally, we require that the length of the list be bounded ($\mathsf{llen}(\alpha) \leqslant n$). Finally, since we are working in i32, we have to explicitly prevent overflow by stating that $(x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}$.

Straight-Line Code: OBAGet. We demonstrate the basics of proof sketches in Wasm Logic using the example of the $\mathsf{OBAGet}(x, k)$ function, specified and verified in Figure 12. OBAGet takes two parameters: x, denoting the memory address at which the OBA starts; and k, denoting the (non-negative) index of the OBA element to be retrieved. Assuming that k does not exceed the current OBA length, the function returns the k-th element of the OBA.

This example illustrates the following aspects of Wasm Logic: the interaction between function parameters, the stack, and the local variables; basic stack and heap manipulation; basic use of the frame, extension, and consequence rules; and predicate unfolding and folding.

In Wasm, function inputs are taken from and function outputs are put onto the stack, as specified in the pre- and post-conditions. When verifying the function body, the values of the function parameters are introduced as local variables (here, l_0 and l_1), which are propagated throughout the proof and are forgotten in the post-condition (cf. the [function] rule).

When the code being verified is straight-line, i.e. when the labels and the return fields of the context are empty, the [frame] and [consequence] rules can be used as in standard separation logic. On the other hand, the [extension] rule, which manipulates the stack analogously to [frame] manipulating the heap, can always be applied independently of the context (to limit clutter, in Figure 12, we show only one use of the [extension] rule and do not show the context Γ, since it is not relevant for this particular proof).

Predicate unfolding and folding in Wasm Logic is standard. For example, in Figure 12, we have to unfold the OBA predicate and frame off the excess resource in order to isolate the k-th element of the OBA in the heap, perform the lookup according to the [load] rule, and then frame the resource back on and fold the predicate.

Conditionals and Loops: OBAFind. We demonstrate how to reason about WebAssembly conditionals and loops in Wasm Logic using the example of the $\mathsf{OBAFind}(x, e)$ function, specified and verified in Figure 13. OBAFind takes two parameters: x, denoting the memory address at which the OBA starts; and e, a 32-bit integer. The function returns the index i of the first element of the OBA that is not smaller than e, or $\mathsf{llen}(\alpha)$ if such an element does not exist. The index i effectively tells us the position in the OBA at which either e appears for the first time or would be inserted.

This example addresses, among other things, the following features of Wasm Logic: interaction between conditionals, loops, and the break statement; advanced use of the frame, existential elimination, and consequence rules; and function calls. To focus on these features, we elide previously discussed details, such as predicate management, from the proof sketch.

First, observe how local variables are initialised. The function itself expects two parameters, as given by the type of the function (cf. the [function] rule). These form the first two local variables. The explicitly declared local variables, starting from index 2, are initialised to zero.

The body of the function is a loop that uses the local variable l_2 to iterate over the OBA and find its first element that is not smaller than e. First, the loop checks if l_2 is smaller than the length of the OBA. If it is, the loop terminates (by reaching the loop end), and we know that all of the elements of the OBA are smaller than e. Otherwise, it checks if the l_2-nd element of the OBA is smaller than e. If it is, the loop terminates, and we know that we have found an element not smaller than e in the OBA. Otherwise, l_2 is incremented and the loop restarts (by executing the break instruction).

For the loop construct, we establish the appropriate invariant, $([] \mid P_{inv})$, using the [consequence] rule in the standard way. This invariant essentially states that all of the previously examined elements are smaller than e. Then, following the [loop] rule, we verify the body of the loop while extending the labels field of the context with the invariant. We explicitly state modifications to the context at the point at which they first occur.

As soon as the labels or the return field of the context is not empty, the use of the frame and existential elimination becomes more involved. For example, when framing off, we have to frame off not only from the current state, but also from all of the labels, as well as from the return assertion. We illustrate this in Figure 13, using the first instruction of the loop body, (**get_local** 2), where we have to frame off P_{inv} both from the state and the labels of the context in order to apply the [get_local] rule.

$\{[x, e] \mid \mathsf{OBA}(x, n, \alpha)\}$
(**func** OBAFind [i32, i32] → [i32]
(**locals** i32)
$\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2 = 0\}$
$P_{inv} : \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge 0 \leqslant l_2 \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e)$
$\{[] \mid P_{inv}\}$ (by consequence)
(**loop**
$([] \mid P_{inv}) \vdash$
$\{[] \mid P_{inv}\}$
frame:
$([] \mid \mathsf{emp}) \vdash \{[] \mid \mathsf{emp}\}$
(**get_local** 2)
$([] \mid \mathsf{emp}) \vdash \{[l_2] \mid \mathsf{emp}\}$
$\{[l_2] \mid P_{inv}\}$
(**get_local** 0) (**i32.load**)
$\{[l_2, \mathsf{llen}(\alpha)] \mid P_{inv}\}$
(**i32.lt**)
$C_1 : (v = 0 \Rightarrow l_2 = \mathsf{llen}(\alpha)) \wedge (v \neq 0 \Rightarrow l_2 < \mathsf{llen}(\alpha))$
$\{\exists v.\, [v] \mid P_{inv} \wedge C_1\}$
$(\exists v.\, [] \mid P_{inv}) \vdash \{\exists v.\, [v] \mid P_{inv} \wedge C_1\}$ (by consequence)
exists:
$([] \mid P_{inv}) \vdash \{[v] \mid P_{inv} \wedge C_1\}$
(**if**
$([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash$
$\{[] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
(**get_local** 0) (**get_local** 2)
$\{[x, l_2] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
(S2):
$\vdash \{[x, l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant l_2 < \mathsf{llen}(\alpha)\}$
(**call**OBAGet)
$\vdash \{[\alpha!l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant l_2 < \mathsf{llen}(\alpha)\}$
$([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash \{[\alpha!l_2] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
(**get_local** 1) (**i32.lt**)
$\{\exists v.\, [v] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge (v = 0 \Rightarrow \alpha!l_2 \geqslant e) \wedge (v \neq 0 \Rightarrow \alpha!l_2 < e)\}$
(**if**
$C_2 : ((l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 \geqslant e) \vee l_2 = \mathsf{llen}(\alpha))$
$([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash$
$\{[] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 < e\}$
(**get_local** 2) (**i32.const** 1) (**i32.add**)
$\{[l_2 + 1] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 < e\}$
(**set_local** 2)
$\left\{\begin{array}{l} [] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2{-}1 < \mathsf{llen}(\alpha) \wedge \\ \quad (\forall j.\, 0 \leqslant j < l_2{-}1 \Rightarrow \alpha!j < e) \wedge \alpha!(l_2{-}1) < e \end{array}\right\}$
$\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e) \wedge l_2 \leqslant \mathsf{llen}(\alpha)\}$
$\{[] \mid P_{inv}\}$
(**br** 2)
$\{[] \mid P_{inv} \wedge C_2\}$
end)
$\{[] \mid P_{inv} \wedge C_2\}$
end)
$\{[] \mid P_{inv} \wedge C_2\}$
$(\exists v.\, [] \mid P_{inv}) \vdash \{\exists v.\, [] \mid P_{inv} \wedge C_2\}$
$([] \mid P_{inv}) \vdash \{[] \mid P_{inv} \wedge C_2\}$ (by consequence)
end)
$\{[] \mid P_{inv} \wedge C_2\}$
(**get_local** 2)
$\{[l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge 0 \leqslant l_2 \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e) \wedge C_2\}$
$\left\{\begin{array}{l} \exists i.\, [i] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2 = i \wedge 0 \leqslant i \leqslant \mathsf{llen}(\alpha) \wedge \\ \quad (\forall j.\, 0 \leqslant j < i \Rightarrow \alpha!j < e) \wedge (\forall j.\, i \leqslant j < \mathsf{llen}(\alpha) \Rightarrow e \leqslant \alpha!j) \end{array}\right\}$
end)
$\{\exists i.\, [i] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant i \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < i \Rightarrow \alpha!j < e) \wedge (\forall j.\, i \leqslant j < \mathsf{llen}(\alpha) \Rightarrow e \leqslant \alpha!j)\}$

Figure 13 OBAFind: Specification and Verification.

In the general case, however, the label assertions, the return assertion, and the state need not match in resource, meaning that the [frame] rule may be unable to manipulate the label/return context. In practice, we have identified two strategies for handling this issue: **(S1)** specialising "falsey" labels/return via the [consequence] rule; or **(S2)** adjusting the context via the [context] rule.

We illustrate the first strategy using the following derivation tree:

$$\dfrac{\dfrac{-,-,[(S_1 \mid \bot),(S_2,\bot)],(S_R,\bot) \vdash \{\ S_P \mid P\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q\ \}}{\begin{array}{c}-,-,[(S_1 \mid \bot * F),(S_2 \mid \bot * F)],(S_R \mid \bot * F) \vdash \\ \{\ S_P \mid P * F\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q * F\ \}\end{array}}\text{[frame]} \quad \begin{array}{l}(S_1 \mid \bot * F) \Rightarrow (S_1 \mid H_1)\\(S_2 \mid \bot * F) \Rightarrow (S_2 \mid H_2)\\(S_R \mid \bot * F) \Rightarrow (S_R \mid H_R)\end{array}}{-,-,[(S_1 \mid H_1),(S_2 \mid H_2)],(S_R \mid H_R) \vdash \{\ S_P \mid P * F\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q * F\ \}}\text{[cons]}$$

This strategy takes advantage of the fact that if e^* never actually executes (for example) (**br** n), then $L!n$ can have a $\bot$ component, allowing the manufacturing of any frame through application of the [consequence] rule.

An example of the second strategy works as follows:

$$\dfrac{\dfrac{-,-,[],\texttt{None} \vdash \{\ S_P \mid P\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q\ \}}{-,-,[],\texttt{None} \vdash \{\ S_P \mid P * F\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q * F\ \}}\text{[frame]}}{-,-,[L_1,L_2],R \vdash \{\ S_P \mid P * F\ \}\ \mathrm{e}^*\ \{\ S_Q \mid Q * F\ \}}\text{[context]}$$

Here, we use the [context] rule to temporarily remove all of the labels and the return, allowing us to frame off only from the state. This strategy be seen in action immediately before the function call to OBAGet in Figure 13.

Both strategies can normally be applied before any non-break, non-return instruction, although the second strategy is preferred. However, there are occasions where the first strategy must be used. For example, if e^* executes (**br** 1), then $L!0$ can no longer be removed by [context]. However, it can still be falsified, allowing the first approach.

Existential elimination is another fundamental separation logic rule that needs to consider the context in Wasm Logic and can only be applied if all of the labels, the return, and the state have the same leading existential variable(s). This requirement can normally be established via the [consequence] rule and can be used regardless of the context and the position in the code. For example, consider the following part of the proof derivation for the first **if** statement of OBAFind (cf. Figure 13 for more details):

$$\dfrac{\dfrac{-,-,[([] \mid P_{inv})],\texttt{None} \vdash \{\ [v] \mid P_{inv} \wedge C_1\ \}\ (\textbf{if}\ldots\textbf{end})\ \{\ [] \mid P_{inv} \wedge C_2\ \}}{-,-,[(\exists v.\,[] \mid P_{inv})],\texttt{None} \vdash \{\ \exists v.\,[v] \mid P_{inv} \wedge C_1\ \}\ (\textbf{if}\ldots\textbf{end})\ \{\ \exists v.\,[] \mid P_{inv} \wedge C_2\ \}}\text{[exists]}}{-,-,[([] \mid P_{inv})],\texttt{None} \vdash \{\ \exists v.\,[v] \mid P_{inv} \wedge C_1\ \}\ (\textbf{if}\ldots\textbf{end})\ \{\ [] \mid P_{inv} \wedge C_2\ \}}\text{[cons]}$$

Here, we use [consequence] to add the existential v directly to the label (possible because v is not featured in P_{inv}) and remove it from the obtained post-condition (possible because v is not featured in R_2). In cases where this direct approach would lead to variable capture, we would have an additional first step of renaming the existentials appropriately.

In the first **if** statement of OBAFind, we also encounter a call to the OBAGet function. In Wasm Logic, function calls are handled in the standard way, meaning that frame and consequence are used first to isolate the appropriate pre-condition from the current state and then to massage the obtained post-condition into a desired form. For simplicity, in the code we call the functions by name, rather than by index.

Finally, we comment on the treatment of break statements, using the example of the (**br** 2) statement seen in OBAFind. Given the [br] rule, the pre-condition of that break statement must match the loop invariant ($[] \mid P_{inv}$), which we establish. The post-condition,

$\{[x, e] \mid \mathsf{OBA_{nd}}(x, n, \alpha) \wedge \mathsf{llen}(\alpha) < n\}$
(**func** OBAInsert [i32, i32] → [] ... **end**)
$\left\{\begin{array}{l}\exists \alpha'. \, [] \mid \mathsf{OBA_{nd}}(x, n, \alpha') \wedge \\ \qquad \mathsf{ToSet}(\alpha') = \mathsf{ToSet}(\alpha) \cup \{e\}\end{array}\right\}$

$\{[x, e] \mid \mathsf{OBA_{nd}}(x, n, \alpha)\}$
(**func** OBADelete [i32, i32] → [] ... **end**)
$\left\{\begin{array}{l}\exists \alpha'. \, [] \mid \mathsf{OBA_{nd}}(x, n, \alpha') \wedge \\ \qquad \mathsf{ToSet}(\alpha') = \mathsf{ToSet}(\alpha) \backslash \{e\}\end{array}\right\}$

$\{[t] \mid \mathbf{size}(0) \wedge 2 \leqslant t \leqslant 4095\}$
(**func** BTreeCreate [i32] → [] ... **end**)
$\{[] \mid \mathsf{BTree}(t, \varnothing) \wedge 2 \leqslant t \leqslant 4095\}$

$\{[k] \mid \mathsf{BTree}(t, \kappa)\}$
(**func** BTreeSearch [i32] → [i32] ... **end**)
$\left\{\begin{array}{l}\exists b. \, [b] \mid \mathsf{BTree}(t, \kappa) \wedge \\ \qquad (k \in \kappa \Rightarrow b = 1) \wedge (k \notin \kappa \Rightarrow b = 0)\end{array}\right\}$

$\{[k] \mid \mathsf{BTree}(t, \kappa)\}$
(**func** BTreeInsert [i32] → [i32] ... **end**)
$\{[] \mid \mathsf{BTree}(t, \kappa \cup \{k\})\}$

Figure 14 Specifications of: OBAInsert/OBADelete (left); B-Tree operations (right).

however, is left free in the [br] rule, and has to be chosen correctly so that the subsequent derivation makes sense. Observe that, due to the design of WebAssembly, any code found between a break statement and the end of the block of code in which it is found is dead code. In our case, this means that we never reach the exit of that **if** branch – instead, we unconditionally jump to the head of the main loop. The only way to reach the end of that **if** statement is if the test of that **if** yields zero, in which case our state would be $([] \mid P_{inv} \wedge C_2)$. Now, since the [if] rule requires the final states from both branches to be the same, we can choose precisely $([] \mid P_{inv} \wedge C_2)$ to be the post-condition of the break statement. More generally, a safe option is to always choose the post-condition of a break statement to be $([] \mid \bot)$, and from there derive any required assertion using the [consequence] rule.

Additional OBA Functions. In order to support basic B-tree operations, we also need to be able to insert/delete elements into/from an OBA. Moreover, as B-tree keys are unique (cf. §4.2), we strengthen the OBA predicate to enforce non-duplication of elements:

$$\mathsf{OBA_{nd}}(x, n, \alpha) := \mathsf{OBA}(x, n, \alpha) \wedge \mathsf{llen}(\alpha) = \mathsf{card}(\mathsf{ToSet}(\alpha)).$$

Note that the previously presented OBA functions, OBAGet and OBAFind, can also be used with an $\mathsf{OBA_{nd}}$. We give the specifications of OBAInsert and OBADelete in Figure 14 (left). Their corresponding proof sketches are available in [47].

4.2 B-Trees in WebAssembly

B-trees are self-balancing tree data structures that allow search, sequential access, insertion, and deletion in logarithmic time. They generalise binary search trees in that a node of a B-tree can have more than two children. B-trees are particularly well-suited for storage systems that manipulate large blocks of data, such as hard drives, and are commonly used in databases and file systems [8].

Every node x of a B-tree contains: an indicator denoting whether or not it is a leaf, λ; the number of keys that it holds, n; and the n keys themselves, $\kappa_1, \ldots \kappa_n$. Additionally, each non-leaf node contains $n + 1$ pointers to its children, $\pi_1, \ldots, \pi_{n+1}$.

The number of keys that a B-tree node may have is bounded. These bounds are expressed in terms of a fixed integer $t \geqslant 2$, called *the branching factor* of the B-tree. In particular, every node except the root must have at least $t - 1$ keys, and every node must have *at most* $2t - 1$ keys. Moreover, if a B-tree is non-empty, the root must have at least one key. Finally, all of the leaves of the B-tree have the same depth.

The keys of a B-tree are ordered, in the sense that the keys of every node are ordered (for us, in ascending order), and that every key of a non-leaf node is greater than all of the keys of its left child and smaller than all of the keys of its right child.

As an illustrative example, in Figure 15 we show a B-tree with branching factor $t = 2$ that contains all prime numbers between 1 and 100. It has 25 keys distributed over 12 nodes, with every node having at least $t - 1 = 1$ and at most $2t - 1 = 3$ keys.

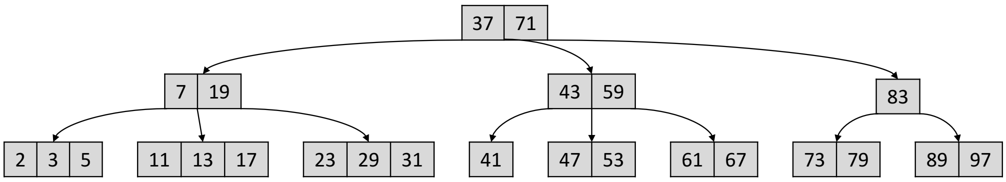


Figure 15 Prime numbers from 1 to 100 in a B-tree of branching factor two, with the λ and n parameters of the nodes elided.

Onward, we describe the layout of a B-tree in WebAssembly memory, define the associated predicates, and show the specifications for B-tree creation, search, and insertion, implemented based on the algorithms and auxiliary functions in [8]. The implementations are available, together with their accompanying proof sketches, in full in [47].

B-Tree Metadata Page. The first page of memory is reserved for keeping track of information about the state of the module. For example, one aspect of module state are the addresses of "free" pages where nodes can be allocated, and another is the root node address.

We first define what it means to be a page in memory with (non-negative integer) index n:

$$\mathsf{Page}(n) := \mathop{\circledast}_{n \cdot 64k \leqslant i < (n+1) \cdot 64k} (i \mapsto_{i32} -) \land 0 \leqslant n \land ((n+1) \cdot 64k \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}$$

Next, we define the predicate capturing the free pages, $\mathsf{Free}(\varphi)$, which stores the list of free pages, φ, in an $\mathsf{OBA_{nd}}$, and confers ownership of all of the pages in φ. The $\mathsf{OBA_{nd}}$ length $(64k/4 - 3 = 16381)$ is chosen to ensure that it can never overflow over the bounds of the metadata page, taking into account the two first elements of the page as well as the length of the array itself that is stored in the $\mathsf{OBA_{nd}}$.

$$\mathsf{Free}(\varphi) := \mathsf{OBA_{nd}}(8, 16381, \varphi) \mathop{\circledast}_{0 \leqslant i < \mathsf{llen}(\varphi)} (\mathsf{Page}(\varphi!i));$$

The full metadata predicate, $\mathsf{Meta}(t, r, l, \varphi)$, describes the metadata page layout: t denotes the branching factor of the B-tree; r denotes the address of its root; μ denotes the current memory size in pages; and φ denotes the list of free pages.

$$\mathsf{Meta}(t, r, \mu, \varphi) := 0 \mapsto_{i32} t * 4 \mapsto_{i32} r * \mathbf{size}(\mu) * \mathsf{Free}(\varphi).$$

B-Tree Nodes. We next show the definition of the abstract predicate $\mathsf{Node}(x, \lambda, \kappa, \pi)$, which captures a B-tree node at page x, with leaf indicator λ, keys κ, and pointers π. A B-tree node takes up an entire WebAssembly page in memory, which can hold 16384 32-bit integers. The first 32-bit integer of the page is the leaf indicator (non-zero means non-leaf); the next 8191 32-bit integers hold information about the node keys; and the last 8192 32-bit integers hold information about the node pointers. The associated predicates are defined as follows:

$$\begin{aligned}\mathsf{Keys}(x,\kappa) &:= \mathsf{OBA}_{\mathsf{nd}}(x \cdot 64k + 4, 8090, \kappa);\\ \mathsf{Ptrs}(x,\pi) &:= \mathsf{BA}(x \cdot 64k + 32k, 8091, \pi);\\ \mathsf{Node}(x,\lambda,\kappa,\pi) &:= x \cdot 64k \mapsto_{i32} \lambda * \mathsf{Keys}(x,\kappa) * \mathsf{Ptrs}(x,\pi).\end{aligned}$$

Note that, since the pointers need not be ordered, we describe them using use a simpler bounded array predicate, $\mathsf{BA}(x, n, \alpha)$, whose definition is the same as that of the OBA predicate given in §4.1, but without the ordering requirement. Recall also that the OBAs and BAs come with a leading 32-bit integer capturing their length, meaning that the maximum number of keys/pointers our B-tree node can hold is 8090/8091 and that the maximal branching factor of our B-trees is 4095.

B-Tree Definition and Operations. Finally, we define an abstract predicate, $\mathsf{BTree}(t, \kappa)$, capturing a WebAssembly B-Tree with branching factor t and set of keys κ:

$$\mathsf{BTree}(t,\kappa) \triangleq \exists r, \mu, \varphi, \lambda, \phi.\, \mathsf{Meta}(t, r, \mu, \varphi) * \mathsf{BTreeRec}^{t,r,\mu}(r, \kappa, \lambda, \phi).$$

Due to lack of space, the full definition of the $\mathsf{BTreeRec}$ predicate is shown and explained in detail in [47]. Informally, $\mathsf{BTreeRec}^{t,r,\mu}(r', \kappa, \lambda, \phi)$ captures a subtree of a B-tree with branching factor t, root r, in a memory of size μ. This subtree has root r' and set of keys κ. Additionally, the B-tree node at r' is a leaf iff $\lambda \neq 0$ and is full iff $\phi \neq 0$.

In Figure 14 (right), we give the specifications of WebAssembly functions for basic B-tree operations: creation; search; and insertion. The specifications are abstract, in that they do not reveal any detail of the underlying implementations.

5 Soundness

The semantic interpretation of our triple and the accompanying soundness proof are informed by the approaches of de Bruin [9] and Oheimb [33]. The former gives us a semantics for **goto** which we use as the foundation for WebAssembly's **br** and **return** instructions. The latter gives us a strategy for handling mutual recursion.

Interpretation is defined against an *abstract variable store*, $\rho \in \mathcal{S}to$. Abstract variable stores are finite partial mappings from variables to constants: $\mathcal{S}to \equiv \mathcal{V}ar \rightharpoonup \mathcal{C}onst$.

Defining interpretation for terms and stack assertions is straightforward. On the other hand, interpretation of heap assertions is more involved. In traditional separation logic [35], ownership and existence of memory locations are conflated to simplify the soundness proof. This, however, cannot be done for WebAssembly: in the concrete WebAssembly linear memory, the existence of the addressable location $x + 1$ implies that the addressable location x also exists. However, asserting ownership of location $x + 1$ should not imply ownership of x.

To address this, we define a two-stage interpretation of heap assertions. We first define their interpretation into a set of abstract heaps, $\mathcal{AH}eap$. An abstract heap, $h \in \mathcal{AH}eap$, is a map from locations to bytes that additionally keeps track of the memory size, which may be fixed by ownership of the **size** resource. The **size** resource can be thought of as tracking the state of memory allocation, with ownership of **size** implying permission to perform allocations through **mem.grow**, similarly to the "free set" resource of [34]. Each abstract heap that is a member of the assertion interpretation represents a possible set of owned locations. Our separation algebra is defined over abstract heaps, as shown in Figure 16.

Interpretation of terms

$$[\![\cdot]\!] :: \mathcal{T}erm \Rightarrow \mathcal{S}to \Rightarrow \mathcal{C}onst$$

$$\begin{aligned} [\![c]\!](\rho) &\triangleq c \\ [\![\nu]\!](\rho) &\triangleq \rho(\nu) \\ [\![f(\tau_1,\dots,\tau_n)]\!](\rho) &\triangleq f([\![\tau_1]\!](\rho),\dots,[\![\tau_n]\!](\rho)) \end{aligned}$$

Interpretation of stack assertions

$$[\![\cdot]\!] :: \mathcal{T}erm \text{ list} \Rightarrow \mathcal{S}to \Rightarrow \mathcal{C}onst \text{ list}$$

$$\begin{aligned} [\![\ [\,]\ ]\!](\rho) &\triangleq [\,] \\ [\![\ \mathcal{S} :: \tau\ ]\!](\rho) &\triangleq [\![\ \mathcal{S}\ ]\!](\rho) :: [\![\tau]\!](\rho) \end{aligned}$$

Abstract heaps

$$size ::= \bullet \mid i32$$

$$\begin{aligned} \mathcal{AH}eap &::= (i32 \rightharpoonup byte) \times size \\ (h_m, \bullet) \uplus (h'_m, \bullet) &\triangleq (h_m \uplus h'_m, \bullet) \\ (h_m, \bullet) \uplus (h'_m, n) &\triangleq (h_m \uplus h'_m, n) \\ (h_m, n) \uplus (h'_m, \bullet) &\triangleq (h_m \uplus h'_m, n) \end{aligned}$$

Note: the two last cases require that $\forall i \in dom(h_m) \uplus dom(h'_m).\ i < n * 64k$

Interpretation of pure/heap assertions

$$[\![\cdot]\!] :: \mathcal{A}_{ph} \Rightarrow \mathcal{S}to \Rightarrow \mathcal{AH}eap \text{ set}$$

$$\begin{aligned} [\![\bot]\!](\rho) &\triangleq \varnothing \\ [\![\tau_1 = \tau_2]\!](\rho) &\triangleq \{\ h \mid [\![\tau_1]\!](\rho) = [\![\tau_2]\!](\rho)\ \} \\ [\![\tau_1 \mapsto \tau_2]\!](\rho) &\triangleq \{\ ([\![\tau_1]\!](\rho) \mapsto [\![\tau_2]\!](\rho), \bullet)\ \} \\ [\![\tau_1 \wedge \tau_2]\!](\rho) &\triangleq [\![\tau_1]\!](\rho) \cap [\![\tau_2]\!](\rho) \\ [\![\neg H]\!](\rho) &\triangleq ([\![H]\!](\rho))^c \\ [\![\exists x.\ H]\!](\rho) &\triangleq \{\ h \mid \exists c.\ h \in [\![H]\!](\rho[x \mapsto c])\ \} \\ [\![p(\tau_1,\dots,\tau_n)]\!](\rho) &\triangleq \{\ h \mid p(\ [\![\tau_1]\!](\rho),\dots,[\![\tau_n]\!](\rho)\ )\ \} \\ [\![H * H']\!](\rho) &\triangleq \{\ h_1 \uplus h_2 \mid h_1 \in [\![H]\!](\rho),\ h_2 \in [\![H']\!](\rho)\ \} \\ [\![\mathbf{size}(\tau)]\!](\rho) &\triangleq \{\ (\varnothing, [\![\tau]\!](\rho))\ \} \end{aligned}$$

Interpretation of assertions

$$\begin{aligned} [\![\cdot]\!] &:: \mathcal{A} \Rightarrow \mathcal{S}to \Rightarrow (\mathcal{C}onst \text{ list} \times \mathcal{AH}eap) \text{ set} \\ [\![\ \mathcal{S} \mid H\ ]\!](\rho) &\triangleq \{(v^*, h) \mid v^* = [\![\mathcal{S}]\!](\rho), h \in [\![H]\!](\rho)\} \\ [\![\ \exists x.\ P\ ]\!](\rho) &\triangleq \{(v^*, h) \mid \exists x.\ (v^*, h) \in [\![P]\!](\rho[x \mapsto c])\} \end{aligned}$$

Entailment

$$P \Rightarrow Q \triangleq \forall \rho.\ [\![P]\!](\rho) \subseteq [\![Q]\!](\rho)$$

Figure 16 Interpretations of Terms and Assertions.

Before describing the second, *reification* stage, we recall the definition of *instances* and *WebAssembly stores* as defined in the official WebAssembly specification [16] (the table fields are elided as they are only used by **call_indirect**):

$$s ::= \{\ \text{funcs: } func\ list,\ \text{mems: } mem\ list,\ \text{globs: } glob\ list\ \} \qquad inst ::= \{\ \text{faddrs: } nat\ list,\ \text{maddr: } nat\ option,\ \text{gaddrs: } nat\ list\ \}$$

$$locs ::= Const\ list \qquad labs ::= nat\ list \qquad ret ::= nat\ option$$

The reification stage further relates abstract heaps to WebAssembly stores, giving the concrete WebAssembly memories that are consistent with the **size** resource, such that all owned locations exist. Store reification is defined between a WebAssembly store, instance, abstract heap, abstract variable store, and function list, as follows:

$$\frac{\begin{array}{c} \forall i.\ F!i = \text{funcs}(s)!((\text{faddrs}(inst))!i) \\ \forall (i, c) \in \text{fst}(h).\ c = (\text{mems}(s)!(\text{maddr } inst))!i \\ \text{snd}(h) \neq \bullet \Longrightarrow pages((\text{mems}(s)!(\text{maddr } inst))) = \text{snd}(h) \\ \forall (g_i, c) \in \rho.\ c = \text{globs}(s)!((\text{gaddrs}(inst))!i) \end{array}}{reifies_{sto}(s,\ inst,\ h,\ \rho,\ F)}\ \text{rei}_{sto}$$

We also define reification for local variables, labels, and returns:

$$\frac{\forall (l_i, v) \in \rho.\ v = locs!i}{reifies_{loc}(locs,\ \rho)}\ \text{rei}_{loc} \qquad \frac{\forall i.\ (L!i = P_n) \Longleftrightarrow (labs!i = n)}{reifies_{lab}(labs,\ L)}\ \text{rei}_{lab} \qquad \frac{(R = R_n) \Longleftrightarrow (ret = n)}{reifies_{ret}(ret,\ R)}\ \text{rei}_{ret}$$

Semantic Interpretation. We define the semantic interpretation of Wasm Logic triples in Figure 17. We say that a triple $(s, locs, v^*)$ satisfies an assertion P if its members can be reified from a member of the interpretation of P. The judgement $F, L, R \vDash \{P\}\ e^*\ \{Q\}$ means,

$$
\begin{array}{l}
F, L, R \vDash \{P\}\ e^*\ \{Q\} \triangleq \forall s, locs, v^*, labs, labs^f, v^{f*}, h, h^f, \rho, ret, s', locs', res.\ (v^*, h) \in [\![P]\!](\rho)\ \wedge \\
\quad reifies_s(s, inst, h \uplus h^f, \rho, F) \wedge reifies_{loc}(locs, \rho) \wedge reifies_{lab}(labs, L) \wedge reifies_{ret}(ret, R)\ \wedge \\
\quad\quad (s, locs, v_e^{f*} v_e^* e^*) \Downarrow_{inst}^{(labs;labs^f), ret}\ (s', locs', res) \Longrightarrow \\
\quad\quad\quad res \neq \textsf{Trap}\ \wedge \\
\quad\quad\quad \exists h', \rho'.\ reifies_s(s',\ inst,\ h' \uplus h^f,\ \rho',\ F) \wedge reifies_{loc}(locs',\ \rho')\ \wedge \\
\quad\quad\quad\quad (res = \textsf{Normal}\ v^* \Rightarrow \exists v'^*.\ v^* = v^{f*} v'^* \wedge (v'^*, h') \in [\![Q]\!](\rho'))\ \wedge \\
\quad\quad\quad\quad (res = \textsf{Break}\ i\ v^* \Rightarrow (v^*, h') \in [\![L!i]\!](\rho'))\ \wedge \\
\quad\quad\quad\quad (res = \textsf{Return}\ v^* \Rightarrow (v^*, h') \in [\![R]\!](\rho'))
\end{array}
$$

$$F, L, R \Vdash\!\!\vDash specs \triangleq (\forall (\{P\}\ e^*\ \{Q\}) \in specs.\ F, L, R \vDash \{P\}\ e^*\ \{Q\})$$

$$F, A, L, R \Vdash\!\!\vDash specs \triangleq (F, [], \epsilon \Vdash\!\!\vDash A \Rightarrow F, L, R \Vdash\!\!\vDash specs)$$

Figure 17 Semantic interpretation of the specification triple.

intuitively, that for all triples $(s, locs, v_e^*)$ that satisfy P, executing $(s, locs, (v_e^{f*})(v_e^*)e^*)$ to completion will result in a triple $(s', locs', res)$ with the following properties: if res is of the form **Normal** v^*, then $(s', locs', v^*)$ satisfies Q; if res is of the form **Break** $i\ v^*$, then $(s', locs', v^*)$ satisfies $L!i$; if res is of the form **Return** v^*, then $(s', locs', v^*)$ satisfies R.

Note that framing is featured in three places in the definition: in the heap (h^f); in the stack (v^{f*}); and in the labels $(labs^f)$. The heap frame is treated in the standard way. The stack frame remains in the case of a **Normal** result, but is discarded in case of the **Break** and **Return** results automatically, by WebAssembly's semantics. Finally, the labels frame encodes that the full label context during reduction may be arbitrarily large, but that only the initial labels *labs* will be targeted by the **br** instructions present in e^*.

Soundness. We now state our soundness result, fully mechanised in Isabelle/HOL.

▶ **Theorem 2** (inference_rules_sound).

$$\Gamma \Vdash specs \Longrightarrow \Gamma \Vdash\!\!\vDash specs$$

6 Related Work

WebAssembly's official specification is given as a pen-and-paper formal semantics [16, 36], a large core of which has been mechanised in Isabelle [46]. Our mechanised soundness results build on this existing mechanisation. CT-Wasm [48] is a proposed cryptographic extension to WebAssembly's type system that protects against side-channel and information flow leaks. Aside from this, research on WebAssembly has focussed mainly on dynamic analysis. Wasabi [23] is a general purpose framework for dynamic analysis. Other work has focussed on taint tracking and binary instrumentation [14, 41]; and the detection of unauthorised WebAssembly-based cryptocurrency miners [45, 27].

Control Flow. Our proof rules for Wasm Logic's break/continue-to-block-style semi-structured control flow take inspiration from the program logic for "structured **goto**" proposed by Clint and Hoare [7] and first proven sound by de Bruin [9]. These works use a traditional Hoare Logic based on first-order logic; we have adapted their approach to our Wasm Logic. In doing so, we have observed that the existential elimination and consequence rules of Hoare logic, and the frame rule of separation logic, require modification, as detailed in §3.2.

Huisman and Jacobs [19] describe an early Hoare logic for Java, and their treatment of Java's **break** and **continue** statements in their operational semantics is similar to our use of the **Break** and **Return** execution results. However, their specifications must explicitly track in the post-condition that a statement terminates via **break** or **continue**, leading to unwieldy proof rules for loops, since separate specifications must be proven for each possible kind of termination of the loop body.

It is common for program logics which handle unstructured control flow, such as **goto** or continuations, to include a context of target assumptions in the semantics of the triple [3, 9, 42, 38]. Separation logics for such languages require a "higher-order frame rule", which distributes the frame across all such assumptions [20, 5, 51, 32, 22]. Similarly, our adaptions to the "structured **goto**" approach result in rules akin to a higher-order frame rule, despite the first-order nature of our logic.

Stack-Based Logics. Two existing program logics are defined over languages which are close to WebAssembly in their typed treatments of the stack: Benton [3], and Bannwart and Müller [1]. However, unlike Wasm Logic, these works does not propose a structured assertion syntax for the stack, instead using unstructured assertions about the values of individual stack positions. This means that assertions must be re-written with a *shift* operation whenever the shape of the stack changes due to the execution of an instruction, and irrelevant portions of the assertions cannot be framed off during local proofs without keeping track of the necessary resulting shift. Saabas and Uustalu [38] give a program logic for a low-level stack-based language with no heap. Their stack assertion is related to ours in that it has a list structure, but their proof rules rely on a global style of term substitution, and their discussion of compositionality does not appear to extend to generalising existing specifications to larger stacks. This means that one cannot conduct local proofs over just the portion of the stack that is changing in the program fragment, which we permit thanks to our [extension] rule. There has been other previous work on program logics for low-level, assembly-like languages, often incorporating a stack [29, 4, 10, 28, 2, 20]. These languages do not have type system restrictions on the stack that are as strong as WebAssembly's, and must therefore find other, less structured ways to represent the stack formally.

7 Conclusions and Future Work

We have presented Wasm Logic, a sound program logic for first-order, encapsulated WebAssembly, and proven the soundness result in Isabelle/HOL. Using Wasm Logic, we have specified and verified a simple WebAssembly B-tree library, giving abstract specifications independent of the underlying implementation.

In designing Wasm Logic, we have found the properties of WebAssembly's type system helpful for streamlining the assertions of Wasm Logic. The restrictions placed on the runtime behaviour of the WebAssembly stack by the type system are mirrored in the structured nature of our logic's stack assertions. To account for WebAssembly's uncommon control flow, we have adapted the standard separation logic triple and proof rules, inspired by the early approach of Clint and Hoare [7] for "structured **goto**".

We plan to extend Wasm Logic to handle programs made up of multiple WebAssembly modules composed together. To do this, we must extend Wasm Logic with the ability to reason about multiple, disjoint memories. Moreover, we would need to account for the JavaScript "glue code", mandatory for module interoperability. This is part of our broader goal of integrating JavaScript and WebAssembly reasoning. To achieve this, however, we will need to support some higher-order reasoning, as WebAssembly modules and functions are first-class entities in JavaScript. We also plan to extend Wasm Logic to be able to reason about higher-order pure WebAssembly code and the **call_indirect** instruction. For both of these goals, we will refer to existing work on higher-order separation logics [44, 21]. Although WebAssembly's higher-order constructs are not entirely standard, we believe that it is possible to map WebAssembly's use of the *table* as a higher-order store to the more traditional program states of other higher-order logics, and hence take direct inspiration from their proof rules and soundness approaches. Again, we would also need to account for the JavaScript component required to mutate the table.

Our long-term goal is to be able to reason, in a single formalism, about integrated JavaScript/WebAssembly programs as they will appear on the Web. We ultimately hope to integrate our work on Wasm Logic with existing work on program analysis for JavaScript [15, 12, 13] to provide a combined proof system, as well as a verification tool.

We expect WebAssembly to be extended with threads and concurrency primitives in the near future [40]. Because there is no sharing of stacks in the WebAssembly threads proposal, we believe that many of our proof rules will be fully transferrable to a hypothetical concurrent separation logic for WebAssembly with threads, although proof rules for the (now shared) heap will need revising, as will the semantic interpretation. For this, we will take inspiration from various modern concurrent separation logics [6, 43, 39].

References

1 Fabian Bannwart and Peter Müller. A Program Logic for Bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):255–273, December 2005. doi:10.1016/j.entcs.2005.02.026.

2 Björn Bartels and Nils Jähnig. Mechanized, Compositional Verification of Low-Level Code. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 98–112, Cham, 2014. Springer International Publishing.

3 Nick Benton. A Typed, Compositional Logic for a Stack-based Abstract Machine. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 364–380, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11575467_24.

4 Lennart Beringer and Martin Hofmann. A Bytecode Logic for JML and Types. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems*, APLAS'06, pages 389–405, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11924661_24.

5 Lars Birkedal and Hongseok Yang. Relational Parametricity and Separation Logic. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'07, pages 93–107, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1760037.1760047.

6 Stephen Brookes and Peter W. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3):47–65, August 2016. doi:10.1145/2984450.2984457.

7 M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224, September 1972. doi:10.1007/BF00288686.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

9 Arie de Bruin. Goto statements: semantics and deduction systems. *Acta Informatica*, 15(4):385–424, August 1981. doi:10.1007/BF00264536.

10 Y. Dong, S. Wang, L. Zhang, and P. Yang. Modular Certification of Low-Level Intermediate Representation Programs. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 563–570, July 2009. doi:10.1109/COMPSAC.2009.81.

11 Jonas Echterhoff. On the future of Web publishing in Unity, 2014. URL: https://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/.

12 José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.*, 2(POPL):50:1–50:33, December 2017. doi:10.1145/3158138.

13 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proc. ACM Program. Lang.*, 3(POPL):66:1–66:31, January 2019. doi:10.1145/3290379.

14 William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly, 2018. arXiv:arXiv:1802.01050.

15 Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103663.

16 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062363.

17 David Herman, Luke Wagner, and Alon Zakai. asm.js, 2014. URL: http://asmjs.org/spec/latest.

18 C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

19 Marieke Huisman and Bart Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, pages 284–303, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

20 Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level Separation Logic for Low-level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 301–314, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429105.

21 Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, pages 696–723, New York, NY, USA, 2017. Springer-Verlag New York, Inc. doi:10.1007/978-3-662-54434-1_26.

22 Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA., July 2011.

23 Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. ACM. doi:10.1145/3297858.3304068.

24 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

25 Mozilla. Mozilla and Epic Preview Unreal Engine 4 Running in Firefox, 2014. URL: https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/.

26 Peter Müller and Martin Nordio. Proof-transforming Compilation of Programs with Abrupt Termination. In *Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 39–46, New York, NY, USA, 2007. ACM. doi:10.1145/1292316.1292321.

27 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Web-based Cryptojacking in the Wild, 2018. arXiv:arXiv:1808.09474.

28 Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare Logic for ARM Machine Code. In *Proceedings of the 2007 International Conference on Fundamentals of Software Engineering*, FSEN'07, pages 272–286, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1775223.1775241`.

29 Magnus O. Myreen and Michael J. C. Gordon. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 568–582, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1763507.1763565`.

30 Tobias Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In Julian Bradfield, editor, *Computer Science Logic*, pages 103–119, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

31 Martin Nordio, Peter Müller, and Bertrand Meyer. Proof-Transforming Compilation of Eiffel Programs. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, pages 316–335, 2008. `doi:10.1007/978-3-540-69824-1_18`.

32 Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and Information Hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM. `doi:10.1145/964001.964024`.

33 David von Oheimb. Hoare Logic for Mutual Recursion and Local Variables. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 168–180, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646837.708364`.

34 Mohammad Raza and Philippa Gardner. Footprints in Local Reasoning. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 201–215, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

35 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=645683.664578`.

36 Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. Bringing the Web Up to Speed with WebAssembly. *Commun. ACM*, 61(12):107–115, November 2018. `doi:10.1145/3282510`.

37 Ando Saabas and Tarmo Uustalu. A Compositional Natural Semantics and Hoare Logic for Low-Level Languages. *Electron. Notes Theor. Comput. Sci.*, 156(1):151–168, May 2006. `doi:10.1016/j.entcs.2005.09.031`.

38 Ando Saabas and Tarmo Uustalu. Compositional Type Systems for Stack-based Low-level Languages. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium - Volume 51*, CATS '06, pages 27–39, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. URL: `http://dl.acm.org/citation.cfm?id=2523791.2523798`.

39 Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A Separation Logic for Fictional Sequential Consistency. In *Programming Languages and Systems*, ESOP '15, pages 736–761, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

40 Ben Smith. Threading proposal for WebAssembly, 2018. URL: `https://github.com/WebAssembly/threads`.

41 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly, 2018. `arXiv:arXiv:1807.08349`.

42 Gang Tan and Andrew W. Appel. A Compositional Logic for Control Flow. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 80–94, Berlin, Heidelberg, 2006. Springer-Verlag. `doi:10.1007/11609773_6`.

43 Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object*

Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM. doi:10.1145/2509136.2509532.

44 Carsten Varming and Lars Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electron. Notes Theor. Comput. Sci.*, 218:371–389, October 2008. doi:10.1016/j.entcs.2008.10.022.

45 Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 122–142, Cham, 2018. Springer International Publishing.

46 Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. ACM. doi:10.1145/3167082.

47 Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A Program Logic for First-Order Encapsulated WebAssembly, 2018. arXiv:1811.03479.

48 Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, January 2019. doi:10.1145/3290390.

49 WebAssembly Community Group. Roadmap, 2018. URL: https://webassembly.org/roadmap/.

50 WebAssembly Community Group. WebAssembly Specifications, 2018. URL: https://webassembly.github.io/spec/.

51 Hongseok Yang. Semantics of Separation-Logic Typing and Higher-Order Frame Rules. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, pages 260–269, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/LICS.2005.47.

52 Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.

53 Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM. doi:10.1145/2048147.2048224.