A Program Logic for First-Order Encapsulated WebAssembly

Anonymised Authors

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.

1 Introduction

WebAssembly [15] 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. 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 modern Web, 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 have been required to compile them to asm.js [16], a subset of JavaScript. In time, such code has become widespread on the Web [42, 23, 10], but the fundamental limitations of JavaScript 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 [41]. 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 [39].

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 [37], and the WebAssembly Working Group requires any further additions to WebAssembly to be formally specified.

WebAssembly functions are grouped into modules. Modules provide interfaces through which users may call WebAssembly code. The main use case for WebAssembly modules is for them to inter-operate with JavaScript code in creating content for the Web. In particular, self-contained (encapsulated) WebAssembly modules are meant to be used as drop-in replacements for their existing JavaScript counterparts, and already constitute a major design pattern in WebAssembly. Therefore, we believe that having a formalism for describing and reasoning about WebAssembly modules and their interfaces is an important task, in line with WebAssembly’s emphasis on formal methods. So far, very little work has been done on program 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. We design a novel assertion syntax, tailored to WebAssembly’s stack-based semantics and the strong guarantees given
by WebAssembly’s type system. Moreover, we show how to adapt the standard separation logic triple and proof rules in a principled way to capture WebAssembly’s uncommon structured control flow. In doing so, we bring the foundational work of Clint and Hoare [6] on reasoning about “structured goto” to the world of modern separation logic.

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 [37]. 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.

Wasm Logic is the first program logic for WebAssembly. It demonstrates that modern separation logic can be used for reasoning about WebAssembly programs, and represents the first step towards the creation of program analysis tools for WebAssembly.

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 Appendix A.

2.1 WebAssembly Syntax

WebAssembly has a human-readable text format based on s-expressions, which we adopt and use throughout. The abstract syntax of WebAssembly programs, taken from [15], is given in full in Figure 1. As in this work 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 [15].

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 42i32, while its 64-bit floating-point representation is denoted 42f64. 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 pre-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 that are 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.
example, a stack with a 32-bit 0 at its top followed by arguments from and push their results onto a stack of WebAssembly values. By convention, Stack.

Anonymised Authors 1:3

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

Figure 1 WebAssembly Abstract Syntax of [15], with aspects not relevant to this work greyed out.

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 thirty-bit integers starting from an empty stack. In particular, the i32.const command, whose type is [[]] → [i32], does not require any arguments and puts the given value on the stack, whereas the i32.add instruction, whose type is [i32,i32] → [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 [40]. 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 $\rightarrow$, and introduce the other semantics as necessary.

**Basic Instructions.** WebAssembly values can be declared using the $t$.const command, typed $[] \rightarrow [t]$, in the style of $(i32$.const 2) of Figure 2. The (drop) command, typed $[t] \rightarrow []$, pops and discards the top stack item, while (nop), typed $[] \rightarrow []$ has no effect.

The (select) operation, typed $[t, t, i32] \rightarrow []$, 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 $[] \rightarrow t^*$, causes the program to halt with a runtime error, which is represented in WebAssembly by a special Trap execution result (cf. 2.4).

![Figure 2 Example of addition in WebAssembly.](image)

WebAssembly also provides a variety of (type-annotated) arithmetic and logical unary operations (Figure 1, unop and testop, respectively), arithmetic and logical binary operations (Figure 1, binop 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 [15], 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 $[] \rightarrow [t]$ (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 $[t] \rightarrow []$, will set 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 $[t] \rightarrow [t]$.

**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 $[i32, t] \rightarrow []$, 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 $[i32] \rightarrow [t]$, 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 WebAssembly memory may also be grown by executing the (mem.grow) instruction, typed $[i32] \rightarrow [i32]$, 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.

Finally, we can inspect the size of the memory by executing the (mem.size) instruction, typed $[] \rightarrow [i32]$, which returns an 32-bit integer denoting the current memory size in pages.
Control Flow Instructions. Most WebAssembly features have many similarities to other bytecodes, such as that of the Java Virtual Machine [22]. 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 f t 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 \( t^n \rightarrow t^n \), 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 will be respected after the body of the construct terminates. 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, \((\text{br} \; i)\), may be executed. As control constructs can be nested, \(\text{br} \) is parameterised by a static index \( i \), indicating the control construct that it targets (indexing inner to outer). The behaviour of \(\text{br} \) depends on the type of its target. When targeting a block or an if, \(\text{br} \) functions 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 functions like a “continue” statement, transferring control back to the beginning of the loop. If the body of a loop terminates without executing a \(\text{br} \), the loop terminates with the result of the body. The \(\text{br} \) instruction is, therefore, required for loop iteration. We illustrate this can in Figure 3.

The first break, \((\text{br} \; 0)\), targets its enclosing if instruction, meaning that control should be transferred to the end of that if instruction. The second break, \((\text{br} \; 1)\), 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: \((\text{br}_{\text{if}} \; i)\) and \((\text{br}_{\text{table}} \; i)\).

The \((\text{br}_{\text{if}} \; i)\) instruction takes one i32 value off the stack and, if this value is not equal to zero, behaves as \((\text{br} \; i)\), and as \((\text{nop})\) otherwise. On the other hand, the \((\text{br}_{\text{table}} \; i_0 \ldots i_n \; i)\) instruction acts like a switch statement. It takes one i32 value \(v\) off the stack and then: if \(0 \leq v \leq n\), it behaves as \((\text{br} \; i_v)\); otherwise, it behaves as \((\text{br} \; i)\).

Function-related Instructions. WebAssembly supports two types of functions. First, native WebAssembly functions are declared as part of the module definition or imported from other modules. Second, the JavaScript environment can supply JavaScript functions to be called from WebAssembly, known as host functions.

Functions are called using the \((\text{call} \; i)\) 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, \((\text{call} \; i)\) also takes the type of the \(i\)-th function. WebAssembly also provides a mechanism for dynamic dispatch through the \(\text{call}_{\text{indirect}}\) instruction.

Our core logic does not support imported or host functions, as well as the \((\text{call}_{\text{indirect}}\) dynamic dispatch, as all of these features require JavaScript intervention for non-trivial use. Without \((\text{call}_{\text{indirect}}\), 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 briefly discuss the ramifications of providing support for them in §7.

Finally, the \((\text{return})\) instruction is analogous to \((\text{br} \), except that it breaks out of all enclosing constructs, concluding the execution of the function.
A Program Logic for First-Order Encapsulated WebAssembly

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* tab? 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 [15] provides a formal small-step semantics, mechanised in Isabelle/HOL by Watt [37]. 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 [37]. We introduce a novel, 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 [15], the last two are introduced by us in this paper. The reason for this is that the WebAssembly formal semantics of [15] 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.

It has been previously observed that this complicates inductive proofs over the semantics, impairing formal reasoning [37]. In fact, this semantics is too course-grained for our proof system to function. For this reason, we need to introduce a notion of an “in-progress” br and return instruction, as an explicit execution result.
We illustrate the difference between the approach of Haas et al. [15] and our approach in Figure 4. The top reduction follows the official semantics of [15]. 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 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

\[(s, \text{loc}^*, v^*_e e^*) \xrightarrow{\text{labs, ret}}_{\text{inst}} (s', \text{loc}'^*, \text{res})\]

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

Additionally, execution is defined with respect to a subscript \(\text{inst}\). 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 function in the formalism is trivial, but its full function is described in the official specification [15], and we give a full definition in the appendix along with our big-step semantics.

Finally, execution is also defined with respect to a list of break label arities \(\text{labs}\) (a natlist), and a return label arity \(\text{ret}\) (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 \(\text{labs}\) and \(\text{ret}\) parameters keep track of the number of values required, so that, for example, if res is of the form Break \(k v^n\), then \(\text{labs}!k = n\). Similarly, if res is of the form Return \(v^n\), then \(\text{ret} = n\).

**Equivalence Result.** We recall the original formal small-step semantic judgement of [15], which is of the form \((s, \text{loc}^*, v^*_e e^*) \red_{\text{inst}} (s', \text{loc}'^*, v'^*_e e'^*)\). Note that 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 \(\rightarrow^*\). Both \(\rightarrow^*\) and \(\rightarrow\) are subscribed by the instance \(\text{inst}\), the big-step derivation starts with empty \(\text{labs} ([])\) and \(\text{ret (e)}\) components, and \(v'^*\) denotes the list of values obtained from \(v^*_e\) by removing their leading consts.

**Theorem 1 (reduce_trans_equiv_reduce_to)**

\[(s, \text{loc}^*, v^*_e e^*) \red_{\text{inst}} (s', \text{loc}'^*, v'^*_e e'^*) \iff (s, \text{loc}^*, v^*_e e^*) \reduce_{\text{inst}} (s', \text{loc}'^*, \text{Normal } v'^*) \land (s, \text{loc}^*, v^*_e e^*) \reduce_{\text{inst}} (s', \text{loc}'^*, \text{Trap})\]

---

1 This treatment of the value stack is a key difference between the official prose and formal semantics. In the prose semantics, which we have adopted so far, 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 permanently 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 operations are inspired by a foundational proof rule for “structured goto” by Clint and Hoare [6], ad extends their work to the world of separation logic [29]. We fully mechanise and prove soundness of Wasm Logic in Isabelle/HOL, as detailed in §5.

3.1 Assertion Language

Wasm Logic assertions encode knowledge about a single WebAssembly runtime state. They include a novel structure for representing the state of the value stack. A formal description of their semantic interpretation is given in the context of our soundness result in §5.

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 the predicate calculus with connectives for reasoning about resource separation, and is necessary for modular client reasoning [29].

WebAssembly, however, in addition to variables and the heap, also allows values to be stored in the stack. The type system of WebAssembly gives us static knowledge of the size of the stack and the types of each of its elements at every program point. This allows us to define a more structured assertion syntax representing the state of the value stack, as opposed to the unstructured composition of individual heap cell assertions used in separation logic.

```
| constants        | c :::= c_i32 | c_i64 | c_f32 | c_f64 |
| variables        | ν :::= x | l_n | g_n |
| terms            | τ :::= c | ν | f(τ_1 ... τ_n) |
| pure/heap assertions | H, H' :::= p(τ_1 ... τ_n) | ⊥ | emp | τ_1 ⊸ τ_2 | size(τ) | ¬H | H ∧ H' | H * H' | ⊕ | H | ∃x, H |
| stack assertions  | S :::= [] | S :: τ |
| assertions       | P, Q :::= (S | H) | ∃x, P |
```

Figure 5 Syntax of Wasm Logic assertions.

The syntax of Wasm Logic assertions is defined in Fig. 5. We have constants, c, which can have one of the four WebAssembly value types. We have logical, local, and global variables, with local and global variables having dedicated variable names, l_n and g_n, respectively, where n ∈ N. Terms are either constants, or variables, or functions (such as, for example, unary and binary operators). Pure assertions are described using meta-level predicates, p(τ_1 ... τ_n), one example of which is term equality. The emp assertion describes an empty heap, the cell assertion τ_1 ⊸ τ_2 describes a single heap cell at address denoted by τ_1 with contents denoted by τ_2, and the size(τ) assertion is a new, WebAssembly-related assertion that states that the number of pages in current memory is denoted by τ. Next follow the standard separation logic negation, conjunction, separating conjunction, iterated separating conjunction, and existential quantification. We highlight the iterated separating conjunction operator, ⊕, which aggregates assertions composed by * in the same way that \( \sum \) aggregates arithmetic expressions composed by +.

A stack assertion, denoted by S, has a novel structure: it 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 $H$.

**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 $\tau_1 \rightarrow \tau_2$ 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: $\tau_1 \rightarrow_t \tau_2$. For example, we have that $\tau_1 \rightarrow_{\text{f32}} \tau_2 \equiv \tau_1 \rightarrow b_0 \ast (\tau_1 + 1) \rightarrow b_1 \ast (\tau_1 + 2) \rightarrow b_2 \ast (\tau_1 + 3) \rightarrow b_3$, where $b_k$ denotes the $k$\textsuperscript{th} least significant byte of the 32-bit representation of $\tau_2$.
- (Operator Domain) To avoid clutter when writing assertions, we overload all mathematical operators (e.g., $+,-,\leq,\ldots$) instead of explicitly stating their domain ($\text{i32}, \text{i64}, \text{f32}, \text{f64}, \text{N}, \text{Z}, \text{or} \text{R}$) on each use. When required, we do state the domain either of a single operator (e.g., $+_{\text{i32}}, +_{\text{i64}}, \ldots$) or of a parenthesised expression (e.g., $(3.14 - 2.71 \cdot x)_{\text{f64}}$), in which case the domain applies to all of the operators and operands featured in the expression. The default domain is i32.

### 3.2 Proof System

We have defined and mechanised a program logic capable of verifying specifications of first-order, encapsulated WebAssembly modules. We base our encoding of program behaviour on Hoare triples [17]. 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 $\Gamma$ represents the context in which the program is executed.

In order to give the interpretation of the Wasm Logic triple, we have to explain the context $\Gamma$ in detail. A context consists of the following four fields: (1) the functions field, $F$, which contains a list of all function definitions of the module; (2) the assumptions field, $A$, which contains a set of assertions of the form $\{P\} \ \text{call} \ i \ \{Q\}$, and is used by the [call] rule to correctly capture mutually recursive functions; (3) the labels field, $L$, which contains a list of assertions used to describe the behaviour of the br instruction; and (4) the return field, $R$, which contains an optional return assertion, used to describe the behaviour of the return instruction. A context may be split into its components and presented as $(F, A, L, R)$. Also, any single field of the context may be referenced: for example, $F(\Gamma)$ refers to the functions field of the context. Since this pattern occurs commonly, we use $P; \Gamma$ as syntactic shorthand for $\Gamma$ with $P$ appended to the head of its labels field.

**Interpretation of Wasm Logic Triples.** The meaning of the triple $\Gamma \vdash \{P\} \ e^* \ \{Q\}$ can be informally understood 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 $e^*$ result, the resulting state must satisfy $R(\Gamma)$; and if it terminates with a Break $i \ v^*$ result, the resulting state must satisfy the $i$-th assertion of $L(\Gamma)$.
3.2.1 Proof Rules

**Basic Instructions.** The proof rules for basic instructions may be found in Figure 6. These rules manipulate only stack and pure logical assertions, requiring an empty heap. The proof rules for each instruction can be intuitively motivated by their effects on the value stack. The select rule is complicated by the fact that its effect is conditional on the value of \( t_3 \), and both possibilities must be encoded. Once it has terminated, we know that it has placed exactly one value on the stack, but whether it is \( t_1 \) or \( t_2 \) depends on whether \( c \) was non-zero.

\[
\begin{align*}
\Gamma \vdash [[] | \text{emp}] \text{t}.\text{const} \ {}c \ {}|[c] | \text{emp} &\quad \text{[const]} \\
\Gamma \vdash [[] | \bot] \text{unreachable} \ {}Q &\quad \text{[unreachable]} \\
\Gamma \vdash [[] | \text{emp}] \text{nop} \ {}|[[]] | \text{emp} &\quad \text{[nop]} \\
\Gamma \vdash [[] | \text{emp}] \text{drop} \ {}|[[]] | \text{emp} &\quad \text{[drop]} \\
\Gamma \vdash [[t_1, t_2, t_3] | \text{emp}] \text{select} \ {}(\exists x. [x] | \text{emp} \land (t_3 \neq 0 \rightarrow x = t_1) \land (t_3 = 0 \rightarrow x = t_2)) &\quad \text{[select]} \\
\Gamma \vdash [[t] | \text{emp}] \text{t}.\text{unop} \ {}|[\text{unop}(t)] | \text{emp} &\quad \text{[unop]} \\
\Gamma \vdash [[t] | \text{emp}] \text{t}.\text{testop} \ {}|[\text{testop}(t)] | \text{emp} &\quad \text{[testop]} \\
\Gamma \vdash [[t_1, t_2] | \text{emp}] \text{t}.\text{binop} \ {}|[\text{binop}(t_1, t_2)] | \text{emp} &\quad \text{[binop]} \\
\Gamma \vdash [[t_1, t_2] | \text{emp}] \text{t}.\text{relop} \ {}|[\text{relop}(t_1, t_2)] | \text{emp} &\quad \text{[relop]} \\
\Gamma \vdash [[t] | \text{emp}] \text{t}.\text{cvtop} \ {}|[\text{cvtop}(t)] | \text{emp} &\quad \text{[cvtop]}
\end{align*}
\]

**Note:** The defined(\( \text{binop}, t_1, t_2 \)) and defined(\( \text{cvtop}, t \)) predicates describe conditions sufficient for binary and conversion operators to be non-trapping.

- **Figure 6** Proof Rules: Basic Instructions.

**Variable Management Instructions.** We give the proof rules for variable management instructions in Figure 6. Just as 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, \( \text{(get\_local} \ i) \) simply puts the variable \( l_i \) on the top of the stack. On the other hand, \( \text{(set\_global} \ i) \) 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 8, we give a proof sketch of a simple WebAssembly program that uses basic and variable management instructions, essentially illustrating how stack assertions function. We start with the pre-condition \( \{[] | l_1 = 2 \land \text{emp} \} \), which encodes that the stack and the heap are empty, as well as the knowledge that the first local variable has the value 2. Executing \( \text{(get\_local} \ 1) \) adds \( l_1 \) to the stack, which we can immediately replace it with 2 due to our pure knowledge that \( l_1 = 2 \). The second line of the program pushes the constant value 1 onto the stack. Note again that 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 operations which interact with the WebAssembly memory are given in Figure 9. The \( \text{(t.load)} \) and \( \text{(t.store)} \) proof rules are...
similar to standard separation logic cell assertions, 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 \texttt{(mem.grow)} and \texttt{(mem.size)} operations allow WebAssembly to alter

the size of memory. The “permission” to observe the size of memory is encoded using the

\texttt{size(x)} assertion, which encodes that the memory is currently \( x \) pages long. Note that

the permission to observe the memory size does not imply permission to access in-bounds

locations; the logic still requires \( x \mapsto n \) to be held in order to access the location \( x \), even if \( x \)
is known to be in-bounds because \texttt{size} is held. Growing the memory using the \texttt{(mem.grow)}
instruction confers ownership of all newly-created locations.

\[
\Gamma \vdash \{ [\tau_1 | (\tau_1 + \text{off}) \mapsto_i \tau_2] \quad \text{[load]} \\
\Gamma \vdash \{ [\tau_1, \tau_2] \quad \text{(\tau_1 + \text{off}) \mapsto_i -} \quad \text{[store]} \\
\Gamma \vdash \{ [\text{size}(\tau)] \quad \text{mem.size} ([\tau] \quad \text{size}(\tau)) \text{[mem.size]} \\
\Gamma \vdash \{ [\text{size}(\tau_2)] \quad \text{mem.grow} \quad \exists v. [v] \left\{ \begin{array}{l}
\tau_2 \times 64k \mapsto 0 \ast \ldots \ast (l + m) \times 64k - 1 \mapsto 0 \ast \\
\text{size}(\tau_2 + \tau_1) \land v = \tau_2 \\
\text{size}(\tau_2) \land v = -1 \\
\end{array} \right. \} \quad \text{[mem.grow]} \\
\]

\textbf{Control Flow Instructions.} The proof rules for WebAssembly control constructs are
given in Figure 10. 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 \texttt{(br \( i \))} in the \texttt{[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 \texttt{br} instruction in the same block of code cannot be reached due to
the structured control flow of WebAssembly. Analogously, the precondition of a \texttt{(return)}
statement in the \texttt{[return]} rule equals the return field of the context, and its post-condition is
arbitrary. Observe the clear analogy between the role of \texttt{labs} and \texttt{ret} in the semantics and
the role of \( L \) and \( R \) in the proof rules for \texttt{br} and \texttt{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 like its invariant.

This approach is inspired by the proof rule for “structured” goto statements of Clint and Hoare [6], as WebAssembly’s block and br opcodes replicate the structural conditions imposed by [6] on the use of goto. We extend this foundational work to a separation logic setting, and additionally generalise it in the [loop] rule, by allowing control to be transferred to the beginning of the nesting structure instead of to the end.

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.

Next, the [br_if] rule simulates an if-break: if the value on the top of the stack is non-zero, we break, and proceed otherwise. Note that for this rule, the post-condition is not arbitrary, but instead describes the condition under which we do not break.

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 11 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.

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

\[ \vdash [[[] | l_0 = k]], \vdash [[] | l_0 = k'] \{ \text{br 0} \} \{ Q \} \]

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 has to frame off from all assertions in L and also the R. In addition, it requires that the assertions in L and also R are not existentially quantified on the outside. This is not an issue in practice, as it can normally be established by use of [exists] and [consequence], as shown in §4.
Anonymised Authors

1:13

Figure 11 Structural Proof Rules.

\[
\begin{align*}
\Gamma \vdash \{P\} e^*_{a} \{Q\} & \quad \Gamma \vdash \{Q\} e^*_{a} \{R\} \quad [\text{seq}] \\
\Gamma \vdash \{P\} e^*_{a} e^*_{b} \{R\} & \quad [\text{exists}] \\
F.A.L, R' \vdash \{P\} e^* \{Q\} & \quad F.A, \{\text{map} (\exists x.) L\}, (\exists x. R)' \vdash (\exists x. P) e^* (\exists x. Q) \\
F.A, \{\text{map} (\ast F) L\}, (R * F)' \vdash \{P \ast F\} e^* \{Q \ast F\} & \quad [\text{frame}] \\
F.A, L', R'_{n} \vdash \{P'\} e^* \{Q'\} & \quad P \Rightarrow P' \quad Q' \Rightarrow Q \quad \text{llen}(L') = \text{llen}(L) \\
\forall i < \text{llen}(L). L'_{i} = L_{n} \land L'_{i+1} = L'_{n} \land n' \leq n' \land L'_{n'} \Rightarrow R_{n} & \quad [\text{consequence}] \\
\Gamma \vdash \{S_{p}\} \{P\} e^* \{S_{q}\} \land \text{map}(e^*) = \emptyset & \quad [\text{extension}] \\
\Gamma \vdash \{S_{k} ; S_{p} ; P\} e^* \{S_{k} ; S_{q} ; Q\} & \quad [\text{context}] \\
\end{align*}
\]

Note: \(\text{map}(e^*)\) denotes the set of local and global variables modified by the execution of \(e^*\). \(\langle \text{map} (\ast F) L\rangle\) is only defined when the assertions in \(L\) contain no outer existential quantifiers.

Figure 12 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 19.

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 9 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. 12. We give a unified semantics to function calls in WebAssembly through the auxiliary \(\text{call}_{\text{cl}}\) instruction and the corresponding [function] rule, which we now explain in detail. First, when inside a function body, if we execute \(\text{br} 0\) at top-level or \(\text{return}\) 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 de-
clared 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-condition from talking about local variables
altogether in order to avoid name clashes.

At the top level, we have rules for proving specifications for sets of mutually recursive
functions. We follow the strategy described by Oheimb [28] and Nipkow [27]. 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 as-
sumptions (the [call] rule), and from this, it can be concluded that all function specifications
are correct without any assumptions (the [module] rule).

Using Wasm Logic: Verified B-Tree Library

We demonstrate how Wasm Logic can be used for specification and verification of Web-
Assembly programs by specifying and verifying a simple WebAssembly B-tree library. We
show in detail how Wasm Logic rules can be used in practice on examples relevant to the
ordered, bounded array data structure, which underpins our B-tree implementation. We
focus on the non-standard aspects of the logic: stack manipulation; WebAssembly control
flow; and the interplay between structural separation logic rules (framing, existential variable
elimination, and consequence) and the control flow constructs of WebAssembly.

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

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 §4.2.

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. We define the abstract predicate representing what it means to be
a 32-bit OBA at address $x$, with maximum size $n$ and contents $\alpha$, written $\text{OBA}(x, n, \alpha)$.
Informally, the layout of OBAs in memory, illustrated below, is as follows: the first 32-bit
cell holds the length of the list $\alpha$; the next $\text{llen}(\alpha)$ 32-bit cells hold the contents of the list
$\alpha$; and the remaining $(n - \text{llen}(\alpha))$ 32-bit cells constitute over-allocated space.

\[
\begin{array}{c}
x \quad x + 4 \\
\text{llen}(\alpha) \quad (x + 4) + 4 \cdot \text{llen}(\alpha) \\
\alpha \quad (x + 4) + 4 \cdot n
\end{array}
\]

---

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.
Formally, the definition of the \( OBA(x, n, \alpha) \) predicate is:

\[
OBA(x, n, \alpha) := (x \mapsto_{i32} \text{llen}(\alpha) \land \text{Aseg}(x + 4, \alpha) \land \bigoplus_{\text{llen}(\alpha) < i \leq n} (x + 4 \cdot i \mapsto_{i32} \alpha!)) \land
\]

\( (\text{Ordered}(\alpha) \land \text{llen}(\alpha) \leq n \land (x + 4 \cdot (n + 1) \leq \text{INT32\_MAX})_Z) \),

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

\[
\text{Aseg}(x, \alpha) := \bigoplus_{0 \leq i < \text{llen}(\alpha)} (x + 4 \cdot i \mapsto_{i32} \alpha!);
\]

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

\[
\text{Ordered}(\alpha) := \forall i, 0 < i < \text{llen}(\alpha) \rightarrow \alpha!(i - 1) \leq \alpha i;
\]

and \( \text{INT32\_MAX} \) denotes the maximal positive integer of \( i32 \). Additionally, we require that the length of the list be bounded (\( \text{llen}(\alpha) \leq n \)). Finally, since we are working in \( i32 \), we have to explicitly prevent overflow by stating that \( (x + 4 \cdot (n + 1) \leq \text{INT32\_MAX})_Z \).

**Straight-Line Code: OBAGet.** We demonstrate the basics of proof sketches in Wasm Logic using the example of the OBAGet function, specified and verified in Figure 13. 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.

\[
\begin{align*}
\{ & (x, k) \mid \text{OBA}(x, n, \alpha) \land 0 \leq k \leq \text{llen}(\alpha) \} \\
\text{func} \ & \text{OBAGet} \ [i32, i32] \rightarrow [i32] \\
\{ & \{ \} \mid \text{emp} \\
\{ & \text{get\_local} \ 0 \\
\{ & \text{local} \ \ell_0 \mid \text{emp} \\
\{ & \{ \} \mid \text{emp} \\
\{ & \text{get\_local} \ 1 \\
\{ & \{ \ell_1 \} \mid \text{emp} \\
\{ & \text{i32.const} \ 4 \\
\{ & \{ \ell_0, \ell_1, 4 \} \mid \text{emp} \\
\{ & \text{i32.mul} \ i32.add \\
\{ & \{ \ell_0 + 4 \cdot \ell_1 \} \mid \text{emp} \\
\{ & \{ \text{OBA}(x, n, \alpha) \land 0 \leq k \leq \text{llen}(\alpha) \} \times (x \mapsto_{i32} \text{llen}(\alpha) \land \text{llen}(\alpha) = x \land \ell_1 = k) \\
\{ & \{ \text{OBA}(x, n, \alpha) \land 0 \leq k \leq \text{llen}(\alpha) \} \times (x \mapsto_{i32} \text{llen}(\alpha) \land \text{llen}(\alpha) = x \land \ell_1 = k) \} \} \} \} \} \} \} \} \} \} \}
\end{align*}
\]

\[
\begin{align*}
\{ & (x + 4 \cdot k) \mid \text{OBA}(x, n, \alpha) \land 0 \leq k \leq \text{llen}(\alpha) \land \ell_0 = x \land \ell_1 = k \\
\{ & (x + 4 \cdot k) \mid \text{OBA}(x, n, \alpha) \land 0 \leq k \leq \text{llen}(\alpha) \land \ell_0 = x \land \ell_1 = k) \} \} \} \} \} \} \} \}
\end{align*}
\]

**Figure 13** Specification and Verification: OBAGet
In Wasm, function inputs are taken from and function outputs are put onto the stack, as
stated in the pre- and post-condition. 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 be applied independently of the context
(to limit clutter, in Figure 13, we show only one use of the [extension] rule and do not show
the context \(\Gamma\), since it is not relevant for this particular proof).

Predicate unfolding and folding in Wasm Logic is standard. For example, in Figure 13,
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 Web-
Assembly conditionals and loops in Wasm Logic using the example of the OBAFind function,
specified and verified in Figure 14. 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 \(\ell\text{len}(\alpha)\) if such an element does
not exist. The index \(i\) effectively tells us the position in the OFA at which either \(e\) appears
for the first time or would be inserted.

This example addresses, among others, 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, we can further observe how local variables are initialised: we have that the local
variables declared in the function are initialised to zero and start from the index 2, as the
function itself expects two parameters (cf. the [function] rule).

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 14, 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.

In the general case, however, the resource of each of the labels, the return assertion, and
the state need not match, meaning that the [frame] rule is unable to manipulate the label
Anonymised Authors

Figure 14 OBAFind: Specification and Verification

context. In practice, there are proof strategies which make this issue unimportant. We have identified two strategies for handling this issue: (S1) specialising “falsy” 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:

\[
\begin{array}{c}
\vdash \neg \neg \left( (S_1 \mid \perp) , (S_2 \mid \top) ,(S_R \mid \top) \vdash \left\{ S_P \mid P \right\} e^* \left\{ S_Q \mid Q \right\} \right) \\
\vdash \neg \neg \left( (S_1 \mid \perp \neq F) , (S_2 \mid \perp \neq F) , (S_R \mid \perp \neq F) \vdash \left\{ S_P \mid P \neq F \right\} e^* \left\{ S_Q \mid Q \neq F \right\} \right) \\
\vdash \neg \neg \left( (S_1 \mid H_1) , (S_2 \mid H_2) ,(S_R \mid H_R) \vdash \left\{ S_P \mid P \neq F \right\} e^* \left\{ S_Q \mid Q \neq F \right\} \right)
\end{array}
\]

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

An example of the second strategy is as follows:

\[
\begin{array}{c}
\vdash \neg \neg \left[. \: \text{None} \vdash \left\{ S_P \mid P \right\} e^* \left\{ S_Q \mid Q \right\} \right) \left[\text{frame}\right] \\
\vdash \neg \neg \left[. \: \text{None} \vdash \left\{ S_P \mid P \neq F \right\} e^* \left\{ S_Q \mid Q \neq F \right\} \right) \left[\text{context}\right]
\end{array}
\]

Here, we use the \([\text{context}]\) rule to temporarily remove all of the labels and the return, allowing us to frame off only from the state.

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 \((\text{br} \: 1)\), then \( L!0 \) can no longer be removed by \([\text{context}]\). 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 \([\text{consequence}]\) 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 \textit{if} statement of OBAFind (cf. Figure 14 for more details):

\[
\begin{array}{c}
\vdash \neg \neg \left[. \: \left\{ \left[. \: \text{None} \left( [\top \mid P_{inv}] \right) \right] \vdash \left[. \: \left[ [0] \right] \mid P_{inv} \wedge C_1 \right] \left( \text{if} \ldots \text{end} \right) \left[ [\top \mid P_{inv}] \right) \right) \left[\text{exists}\right] \\
\vdash \neg \neg \left[. \: \left\{ \left[. \: \text{None} \left( [\top \mid P_{inv}] \right) \right] \vdash \left[. \: \left[ \exists [v] \left[ [0] \right] \mid P_{inv} \wedge C_1 \right] \left( \text{if} \ldots \text{end} \right) \left[ [\top \mid P_{inv}] \right) \right) \left[\text{cons}\right]
\end{array}
\]

Here, we use \([\text{consequence}]\) 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 \( R_2 \)). If 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 \textit{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 \((\text{br} \: 2)\) statement seen in OBAFind. Given the \([\text{br}]\) rule, the pre-condition of that break statement must match the loop invariant \([\top \mid P_{inv}]\), which we establish. The post-condition, however, is left free in the \([\text{br}]\) 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 \textit{if} branch—instead, we unconditionally jump to the head of the main loop. The only way to reach the end of that \textit{if} statement is if the test of that \textit{if} yields zero, in which case our state would be \([\top \mid P_{inv} \wedge C_3]\).

Now, since the \([\text{if}]\) rule requires the final states from both branches to be the same, we can
choose precisely \( ([] \land P_{\text{inv}} \land C_3) \) 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 \( ([] \land \bot) \), and from there derive any required assertion using the \([\text{consequence}]\) 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:

\[
\forall x, e. \text{OBA}_\text{nd}(x, n, \alpha) \land \text{llen}(\alpha) < n \Rightarrow \forall x, e. \text{OBA}_\text{nd}(x, n, \alpha) \land \text{ToSet}(\alpha') = \text{ToSet}(\alpha) \cup \{e\}
\]

\[
\forall x, e. \text{OBA}_\text{nd}(x, n, \alpha) \land \text{llen}(\alpha) = \text{card}(\text{ToSet}(\alpha))
\]

Note that the previously presented OBA functions, OBAGet and OBAFind, can also be used with an OBA\text{nd}. We give the specifications of OBAInsert and OBADelete in Figure 15, and their corresponding proof sketches are available in Appendix B.2.

### 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 [7].

Every node \( x \) of a B-tree contains: an indicator denoting whether or not it is a leaf, \( x.\lambda \); the number of keys that it holds, \( x.n \); and the \( x.n \) keys themselves, \( x.K_1, \ldots, x.K_{x.n} \). Additionally, each non-leaf node contains \( x.n + 1 \) pointers to its children, \( x.C_1, \ldots, x.C_{x.n+1} \).

The number of keys that a node in a given B-tree may have is bounded. These bounds are expressed in terms of a fixed integer \( t \geq 2 \), called the minimum degree 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 16 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.

In the remainder of this section, we first describe the layout of a B-tree in WebAssembly memory and define the associated predicates. Next, we give the specifications of basic B-tree node operations, such as allocation, deletion, and key retrieval. Finally, we conclude with the specifications for B-tree creation, search, and insertion, implemented in a standard way, as presented in [7]. All of the proof sketches are given in full in Appendices B.4 and B.5.
A Program Logic for First-Order Encapsulated WebAssembly

**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 $a$:

$$\text{Page}(a) := \begin{cases} 0 & \text{if } 64k \leq i < (a+1) \cdot 64k \\
\end{cases}$$

Next, we define the list of free pages, which is an OBA_{\text{meta}} allocated at a fixed location. Its length $(64k/4 - 3 = 16381)$ is chosen to ensure that it can never overflow the bounds of the first page, taking into account the two first elements of the metadata page as well as the stored length of the array itself. It also confers ownership of the free pages listed.

$$\text{Free}(\alpha) := \text{OBA}_{\text{meta}}(8, 16381, \alpha) \bigoplus_{0 \leq i < \text{len}(\alpha)} \text{Page}(\alpha_i);$$

The full metadata predicate describes the layout of the metadata page: $t$ denotes the branching factor of the B-tree; $r$ denotes the address of its root; $l$ denotes the current memory size resource (expressed in pages); and $\alpha$ denotes the indexes of the free pages.

$$\text{Meta}(t, r, l, \alpha) := 0 \mapsto_{\leq 32} t \mapsto 4 \mapsto_{\leq 32} r \mapsto \text{size}(l) \mapsto \text{Free}(\alpha).$$

**B-Tree Nodes.** A B-tree node takes up an entire WebAssembly page: the first thirty-two-bit integer indicates whether or not the node is a leaf (non-zero indicates non-leaf); the next 4095 thirty-two-bit integers are allocated to hold information about the node keys; and the last 4096 thirty-two-bit integers are allocated to hold information about the node pointers.

We define the following related predicates:

$$\text{Keys}(x, \kappa) := \text{OBA}_{\text{meta}}(x \cdot 64k + 4, 4095, \kappa);$$

$$\text{Ptrs}(x, \pi) := \text{BA}(x \cdot 64k + 32k, 4096, \pi);$$

$$\text{Node}(x, \lambda, \kappa, \pi) := x \cdot 64k \mapsto_{\leq 32} \lambda \ast \text{Keys}(x, \kappa) \ast \text{Ptrs}(x, \pi).$$

---

**Figure 17 Specifications of Basic B-Tree Node Operations**
Note that, since the pointers need not be ordered, we describe them using use a simpler bounded array \( \text{BA}(x, n, \alpha) \) predicate, whose definition is the same as that of the \( \text{OBA} \) predicate, except that there is no ordering requirement.

**B-Tree Node Operations.** In Figure 17, we specify WebAssembly functions for basic B-tree node operations: initialisation; allocation; de-allocation; and leaf information, key, and pointer management.

When a node is allocated, its page is either taken from the list of free pages (if that list is non-empty), or a new memory page is allocated. When a node is deallocated, its page is added to the list of free pages. Also, observe that we can get, but not set node keys using their index, as key sortedness has to be preserved. On the other hand, we can both get and set node pointers by their index. Finally, we can only insert new keys and pointers if there is enough space in memory, and can manipulate pointers only for non-leaf nodes.

**B-Tree Definition.** We define an abstract predicate describing what it means to be a B-Tree

\[
\text{BTree}(t, \kappa) \equiv \exists r, l, \alpha, \lambda. \text{Meta}(t, r, l, \alpha) \land \text{BTreeRec}(r, \kappa, \lambda, \phi),
\]

where:

\[
\text{BTreeRec}(r, \kappa, \lambda, \phi) \equiv \exists x < l \land (\exists \kappa_x, \pi_x, (\text{Node}(x, \lambda, \kappa_x, \pi_x) \land \text{llen}(\kappa_x) < 2t - 1 \land (x \neq r \land \pi_x[i] \land \text{llen}(\kappa_x) < 2t - 1 \land \pi_x[i] = \phi) \land (\text{llen}(\kappa_x) = 2t - 1 \rightarrow \pi_x[i] = \phi) \land \lambda \land \text{ToSet}(\kappa_x) = \kappa \land \text{emp}) \land \\
(\lambda = 0 \rightarrow \pi_x[i] = 0) \land (\text{llen}(\kappa_x) = 1 \land 3\lambda, \pi, \phi, (\text{llen}(\pi_x) = \text{llen}(\phi) = \text{llen}(\pi_x) \land \\
\kappa = (\bigcup_{0 \leq i \leq \text{llen}(\pi_x)} \pi_x[i] \cup \kappa_x \land \\
\bigotimes_{0 \leq i \leq \text{llen}(\pi_x)} \text{BTreeRec}(\pi_x[i], \lambda, \phi[i], \phi[i])) \land \\
(\bigwedge_{0 \leq i \leq \text{llen}(\pi_x) - 1} \forall k, k'. k \in \pi_x[i] \rightarrow k' \in \pi_x[i + 1] \rightarrow k < \kappa_x i < k') \land \\
(\bigwedge_{0 \leq i \leq \text{llen}(\pi_x)} \pi_x[i] \land \text{llen}(\pi_x) \land l)\)
\]

In particular, the \( \text{BTreeRec}(r, \kappa, \lambda, \phi) \) captures a subtree that holds a set of keys \( \kappa \), and whose root; is at address \( x \); has leaf information \( \lambda \) (\( \lambda \) is non-zero iff \( x \) is not a leaf); and fullness information \( \phi \) (\( \phi \) is non-zero iff \( x \) is full). This is all in the context of a B-tree with root at address \( r \), branching factor \( t \), and all in WebAssembly memory with at most \( l \) allocated pages. This predicate expresses all of the B-tree features and constraints that we mentioned earlier. Note that we handle a variable number of existential variables via existentially quantified lists (concretely, \( \pi \) and \( \bar{\phi} \)), and also that we have to explicitly require that the addresses of all the nodes in the B-tree be strictly smaller than the page count \( l \).

**B-Tree Operations.** In Figure 18, we specify WebAssembly functions for basic B-tree operations: creation; search; and insertion. Our specifications are abstract, in that they do not reveal any detail of the underlying implementations. We implement these three functions based on the algorithms and using additional auxiliary functions shown in [7]. Note that we require the branching factor \( t \) as a parameter of the B-tree creation function, and are limited by the page size of WebAssembly to a maximum branching factor of 2048.
Assertion interpretation is defined against an abstract variable store, $\sigma$.

$$\sigma : (\nu \times c) \text{ set}$$

An abstract heap, $heap$, is a pair consisting of address mappings and size. The size component is required for the purpose of the mem.size instruction.

$$heap ::= ((i32 \times byte) \text{ set}) \times size \quad size ::= \cdot | i32$$

$$(h_1, \bullet) \wedge (h_2, \bullet) \models (h_1 \wedge h_2, \bullet)$$

$$(h_1, \bullet) \wedge (h_2, n) \models (h_1 \wedge h_2, n)$$

if $\forall i \in \text{dom}(h_1), i < n \times 64k$

$$(h_1, n) \wedge (h_2, \bullet) \models (h_1 \wedge h_2, n)$$

if $\forall i \in \text{dom}(h_2), i < n \times 64k$

Term interpretation

$$[\cdot] : \tau \Rightarrow \sigma \Rightarrow c$$

Stack assertion interpretation

$$[\cdot] : \tau \text{ list} \Rightarrow \sigma \Rightarrow c \text{ list}$$

Heap assertion interpretation

$$[\cdot] : H \Rightarrow \sigma \Rightarrow heap \text{ set}$$

Assertion interpretation

$$[\cdot] : P \Rightarrow \sigma \Rightarrow (c \text{ list } \times heap) \text{ set}$$

Entailment

$$\exists x. H \Rightarrow \sigma \models h \quad \exists c. h \in [H](\sigma[x \mapsto c])$$

$$P \Rightarrow Q \equiv \forall \sigma, [P](\sigma) \subseteq [Q](\sigma)$$

The definition of \texttt{inst} and \texttt{s} are reproduced here from the official WebAssembly specification [15]. The table fields are elided as they are only used by \texttt{call\_indirect}.

\begin{verbatim}
inst ::= \{ faddr: nat list s ::= \{ funs: func list maddr: nat option labs ::= nat list globs: glob list ret ::= nat option

\forall i. Fli = funs(i)!((faddr(inst))\![i])
\forall(i, c) \in \text{hm}. c = (\text{maddr}(s)!(maddr inst))!c
hs \neq \bullet \Rightarrow (\text{mems}(s)!(maddr inst)) = hs
\forall(g, c) \in \sigma. c = globs(s)!((gaddr(inst))!c)
\end{verbatim}

Note: Store reification is defined between a WebAssembly store, instance, abstract heap, abstract variable store, and function list.

$$\forall(l, v) \in \sigma. v = \text{loc}(L)$$

$$\text{reifies}_{\text{loc}}(\text{locs}, \sigma)$$

$$\forall i. (L_i = P_n) \Leftrightarrow (\text{labs}(i) = n)$$

$$\text{reifies}_{\text{lab}}(\text{labs}, L)$$

$$\text{reifies}_{\text{ret}}(R, R_n) \Leftrightarrow (\text{ret} = n)$$

Figure 19 Interpretation of Assertions.

5 Soundness

The semantic interpretation of our triple and the accompanying soundness proof are informed by the approaches of de Bruin [8] and Oheimb [28]. 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.

Unlike the concrete model used by simple separation logic [29], the concrete heap of WebAssembly is a linear buffer. This means that the existence of the addressable location $x + 1$ also means that the location $x$ is addressable. Care must be taken to ensure that the
\[ F, L, R \models \{ P \} e^* \{ Q \} \triangleq \forall s, locs, v^*, labs, labs', v'^*, h, h', \sigma, \text{ret}. (v^*, h) \in [P](\sigma) \land \] 
\[ \text{reifies}(s, \text{inst}, h \cup h', \sigma, F) \land \text{reifies}_{\text{loc}}(\text{locs}, \sigma) \land \text{reifies}_{\text{lab}}(\text{ labs}, L) \land \text{reifies}_{\text{ret}}(\text{ret}, R) \land \] 
\[ \forall s', \text{locs}', \text{res}. (s, \text{locs}, v^*, v^*', e^*) \models \text{inst} \left( s', \text{locs}', \text{res} \right) \implies \text{res} \neq \text{Trap} \land \] 
\[ \exists h', \sigma'. \text{reifies}(s', \text{inst}, h' \cup h', \sigma', F) \land \text{reifies}_{\text{loc}}(\text{locs}', \sigma') \land \] 
\[ (\text{res} = \text{Normal} v^* \implies \exists v'^*. v'^* = v'^* v'^* \land (v'^*, h') \in [Q](\sigma')) \land \] 
\[ (\text{res} = \text{Break} i v^* \implies (\text{v}^*, h') \in [L!](\sigma')) \land \] 
\[ (\text{res} = \text{Return} v^* \implies v^* \in [R](\sigma')) \] 

**Figure 20** Semantic interpretation of the specification triple.

Semantic Interpretation. We define the semantic interpretation of Wasm Logic triples in Figure 20. We say that a triple \((s, \text{locs}^*, v^*)\) satisfies an assertion \(P\) if its members can be reified from a member of the interpretation of \(P\). The judgement \(F, L, R \models \{ P \} e^* \{ Q \}\) means, intuitively, that for all triples \((s, \text{locs}^*, v^*)\) that satisfy \(P\), executing \((s, \text{locs}^*, v^*) e^*\) to completion will result in a triple \((s', \text{locs}'^*, \text{res})\) with the following properties: if \(\text{res}\) is of the form \(\text{Normal} v^*\), then \((s', \text{locs}'^*, v^*)\) satisfies \(Q\); if \(\text{res}\) is of the form \(\text{Break} i v^*\), then \((s', \text{locs}'^*, v^*)\) satisfies \(L!\); if \(\text{res}\) is of the form \(\text{Return} v^*\), then \((s', \text{locs}'^*, v^*)\) satisfies \(R\).

Observe that frame is featured in three places in this definition: in the heap \((h')\), in the stack \((v'^*)\), and in the labels \((\text{labs}')\). The heap frame is treated in the standard way.

An interesting aspect of the stack frame is that it remains in the case of a \text{Normal} result, but is discarded in case of the \text{Break} and \text{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 \(\text{labs}\) will be targeted by the \text{break} instructions that are present in \(e^*\).

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

**Theorem 2** (\text{inference\_rules\_sound})

\[ \Gamma \vdash \text{specs} \implies \Gamma \models \text{specs} \]

6 Related Work

So far, there has been little work on analysing WebAssembly programs. In particular, there is: unpublished work on run-time analysis of WebAssembly programs, focussing on taint tracking and binary instrumentation [21, 13, 33]; unpublished and published work aimed at the detection of unauthorised WebAssembly-based cryptocurrency miners [36, 24]; and published work on cryptographic extensions to WebAssembly’s type system [38]. Wasm Logic is the first program logic for WebAssembly, mechanised or otherwise. Our mechanised
soundness result builds on the existing work of Watt [37], which mechanises the original pen-and-paper formal semantics [15, 30].

Control Flow. It is common for program logics featuring unstructured control flow such as goto to include a context of target assumptions in the semantics of the triple [3, 8, 34, 31]. This context is usually global across the whole proof, or at the very least exposed at the top level of the specification. Because of WebAssembly’s structured control flow (e.g. block), we are able to adopt a more local and syntax-directed approach. Our approach is inspired by the program logic for “structured goto”, proposed by Clint and Hoare [6], and first proven sound by de Bruin [8]. 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 fundamental frame, existential elimination, and consequence rules of separation logic require modification, as detailed in §3.2. Interestingly, the structured form of the proof rule by Clint and Hoare [6] appears to be essentially unused in other existing work on any program or separation logic. Finally, Huisman and Jacobs [18] give a program logic for Java, and their treatment of break and continue in their formal semantics bears some similarity to our use of the Break and Return execution results.

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 [31] 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 [26, 4, 9, 25, 2, 19]. 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, bringing the early approach of Clint and Hoare [6] for “structured goto” to modern separation logic.

We plan to extend Wasm Logic to handle programs made up of multiple WebAssembly modules composed together. This task has two important aspects. First, we must extend
Wasm Logic with the ability to reason about multiple, disjoint memories. This will involve extending the cell assertion $x \rightarrow n$ with an additional annotation describing which memory it refers to. Moreover, the semantics of the assertion triple must be extended in order to encode inside which module scope we are currently reasoning. Second, 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.

We also plan to extend Wasm Logic to be able to reason about higher-order WebAssembly code and the `call_indirect` instruction. For this, we will refer to existing work on higher-order separation logics [35, 20]. 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 [14, 11, 12] 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 [32]. 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 [5].

---

**References**


A Program Logic for First-Order Encapsulated WebAssembly


Anonymised Authors


Full Big-Step Semantics

The definition of the Wasm AST remains identical to that of [15] Figs. 1 and 2. Our big-step judgement, \( \Downarrow_{\text{inst}}^{\text{labs,ret}} \), is parameterised by a list of labels (nat list), a return (nat option), and the current instance.

Our judgement has a structure almost identical to that of the original small-step judgement. The only difference is that the right-hand side results in a \( \text{res} \) object with the following structure, rather than an intermediate stack.

Note that for a given list of values \( v^* \), by convention we write \( v^*_e \) to represent the same values wrapped by the appropriate \texttt{const} operation.

\[
\text{res} ::= v^* \mid \text{Break} n v^* \mid \text{Return} v^* \mid \text{Trap}
\]

\[
\text{inst} ::= \{ \text{faddrs: nat list} \mid \text{taddr: nat option} \mid \text{maddr: nat option} \mid \text{gaddrs: nat list} \}
\]

**Arithmetic operations**

\[
s, \text{locs}, (t\text{.const } c)(t\text{.unop } op) \Downarrow_{\text{inst}}^{\text{labs,ret}} s, \text{locs}, (op(c))
\]

\[
s, \text{locs}, (t\text{.const } c_1)(t\text{.const } c_2)(t\text{.binop } op) \Downarrow_{\text{inst}}^{\text{labs,ret}} s, \text{locs}, (op(c_1, c_2))
\]

**Control operations**

\[
s, \text{locs}, (\text{label}_n [[]]) (v^n_e es) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res}
\]

\[
s, \text{locs}, v^n_e (\text{block } t^n \rightarrow t^m es) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res}
\]

\[
s, \text{locs}, (\text{label}_n [[(\text{loop } t^n \rightarrow t^m es)]])(v^n_e es) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res}
\]

\[
s, \text{locs}, v^n_e (\text{block } \text{if } es) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \quad c \neq \text{i32}
\]

\[
s, \text{locs}, v^n_e (\text{if } es') \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \quad c = \text{i32}
\]

\[
\text{labs}!n = k
\]

\[
s, \text{locs}, v^k_e (\text{br } n) \Downarrow_{\text{inst}}^{\text{labs,ret}} s, \text{locs}, \text{Break } n v^k
\]

\[
s, \text{locs}, v^k_e (\text{br } n) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \quad c \neq \text{i32}
\]

\[
s, \text{locs}, v^k_e (\text{i32.const } c)(\text{br_if } n) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \quad c = \text{i32}
\]

\[
s, \text{locs}, (\text{i32.const } c)(\text{br_if } n) \Downarrow_{\text{inst}}^{\text{labs,ret}} s, \text{locs}, \epsilon
\]
1:30 A Program Logic for First-Order Encapsulated WebAssembly

\[
s, \text{locs}, v^k_e (\text{br } k') \vdash^\text{inst} \text{lab}_e \text{ret } s', \text{locs}', \text{res}
\]
\[
s, \text{locs}, v^k_e (\text{i32.const } c)(\text{br_table } ns n') \vdash^\text{inst} \text{lab}_e \text{ret } s', \text{locs}', \text{res}
\]
\[
s, \text{locs}, v^k_e (\text{br } n') \vdash^\text{inst} \text{lab}_e \text{ret } s', \text{locs}', \text{res}
\]
\[c \geq \text{length}(ns)\]
\[
s, \text{locs}, v^k_e (\text{i32.const } c)(\text{br_table } ns n') \vdash^\text{inst} \text{lab}_e \text{ret } s', \text{locs}', \text{res}
\]
\[
k = \text{ret}
\]
\[
s, \text{locs}, v^k_e \text{ return } \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, \text{Return } v^k
\]

945 stack operations

\[
s, \text{locs}, v_e (\text{drop}) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, \epsilon
\]
\[
s, \text{locs}, v_1 v_2 (\text{i32.const } c) (\text{select}) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, v_1 \text{ if } c \neq 0_{\text{i32}}
\]
\[
s, \text{locs}, v_1 v_2 (\text{i32.const } c) (\text{select}) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, v_2 \text{ if } c = 0_{\text{i32}}
\]

946 local operations

\[
s, \text{locs}, (\text{get\_local } n) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, v \text{ if } \text{locs! } n = v
\]
\[
s, \text{locs}, v (\text{set\_local } n) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}[n := v], \epsilon
\]
\[
s, \text{locs}, v (\text{tee\_local } n) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}[n := v], v
\]

947 global operations

\[
s, \text{locs}, (\text{get\_global } n) \vdash^\text{inst} \text{lab}_e \text{ret } s, \text{locs}, v \text{ if }\text{gaddrs}(\text{inst})[n] = k
\]
\[
\text{globs}(s)[k] = v
\]
\[
s, \text{locs}, v (\text{set\_global } n) \vdash^\text{inst} s[n := v], \text{locs}, \epsilon \text{ if }\text{gaddrs}(\text{inst})[n] = k
\]
memory operations

\[ s, \text{locs}, (\text{i32.const n})(\text{t.load off}) \psi_{\text{inst}}^{\text{labs,ret}} s, \text{locs}, (\text{t.const c}) \]
if
\[
\text{maddr}(\text{inst}) = k
\]
\[
\text{mems}(s)!k = m
\]
\[
\text{m}[(n + \text{off} + \text{off} + |t|)] = \text{bytes}
\]
\[
\text{from_bytes}(t, \text{bytes}) = c
\]

\[ s, \text{locs}, (\text{i32.const n})(\text{i32.const c})(\text{t.store off}) \psi_{\text{inst}}^{\text{labs,ret}} \text{s}[\text{mems}!k := m'], \text{locs}, \epsilon \]
if
\[
\text{maddr}(\text{inst}) = k
\]
\[
\text{mems}(s)!k = m
\]
\[
\text{to_bytes}(t, c) = \text{bytes}
\]
\[
\text{m}[(n + \text{off}) + (n + \text{off} + |t|)] = \text{bytes} = m'
\]

\[ s, \text{locs}, (\text{i32.const c})(\text{mem.grow}) \psi_{\text{inst}}^{\text{labs,ret}} \text{s}[\text{mems}!k := m'], \text{locs}, (\text{i32.const n}) \]
if
\[
\text{maddr}(\text{inst}) = k
\]
\[
\text{mems}(s)!k = m
\]
\[
\text{pages}(m) = n
\]
\[
\text{add_pages}(m, c) = m'
\]

\[ s, \text{locs}, (\text{i32.const c})(\text{mem.grow}) \psi_{\text{inst}}^{\text{labs,ret}} \text{s}, \text{locs}, (\text{i32.const -1}) \]
if
\[
\text{maddr}(\text{inst}) = k
\]
\[
\text{mems}(s)!k = m
\]

\[ s, \text{locs}, (\text{mem.size}) \psi_{\text{inst}}^{\text{labs,ret}} \text{s}, \text{locs}, (\text{i32.const n}) \]
if
\[
\text{maddr}(\text{inst}) = k
\]
\[
\text{mems}(s)!k = m
\]
\[
\text{pages}(m) = n
\]

call operations

\[ \text{faddrs}(\text{inst})\!n = k \quad \text{funcs}(s)!k = \text{cl} \]
\[ s, \text{locs}, v^* (\text{call cl}) \psi_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \]
\[ s, \text{locs}, v^* (\text{call n}) \psi_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \]

\[ (\text{tables}(s)!((\text{taddr}(\text{inst}))\!n = \text{cl}) \]
\[ s, \text{locs}, v^* (\text{callcl cl}) \psi_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \]
\[ s, \text{locs}, v^* (\text{i32.const n}) (\text{call_indirect tf}) \psi_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res} \]
\[
\text{cl} = \text{f} \; t^n \to t^m \; \text{locals} \; l_n, \ldots, l_{n+n'} \; \text{es} \; \quad z'' = \text{zerowals}(l_n, \ldots, l_{n+n'})
\]

\[
\text{callcl operation}
\]

\[
s, \text{locs}, (\text{callcl } \text{cl}) \Downarrow_{\text{inst}}^{\text{labs,ret}} s', \text{locs}', \text{res}
\]
return congruence

\[
\begin{align*}
\text{s, locs, es} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Return vs} \\
\text{s, locs, } v'_e (\text{es es'}) & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Return vs} \\
\text{s, locs, } \text{label}_w\{e^*\} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Return vs} \\
\text{s, llocs, es} & \vdash_{\text{inst}}^{[r]} s', \text{ llocs}', \text{ Return vs} \\
\text{s, locs, local}_{\text{locos}} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs, vs}
\end{align*}
\]

trap congruence

\[
\begin{align*}
\text{s, locs, es} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Trap} \\
\text{s, locs, } v'_e (\text{es es'}) & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Trap} \\
\text{s, locs, } \text{label}_w\{e^*\} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs}', \text{ Trap} \\
\text{s, llocs, es} & \vdash_{\text{inst}}^{[r]} s', \text{ llocs}', \text{ Trap} \\
\text{s, locs, local}_{\text{locos}} & \vdash_{\text{inst}}^{\text{labs,ret}} s', \text{ locs, Trap}
\end{align*}
\]
B Verification: B-Trees

B.1 Ordered, Bounded Arrays

\[ Aseg(x, \alpha) := (x + 4 \cdot i \mapsto_{i \leq \text{llen}(\alpha)} \alpha) \]

\[ BA(x, n, \alpha) := (x \mapsto_{i \leq \text{llen}(\alpha)} Aseg(x + 4, \alpha) \pi_{\text{llen}(\alpha) < i \leq n} (x + 4 \cdot i \mapsto_{i \leq \text{llen}(\alpha)} -)) \land (\text{llen}(\alpha) \leq n \land (x + 4 \cdot (n + 1) \leq \text{INT32\_MAX})) \]

\[ \text{Ordered}(\alpha) := \forall i, i'. 0 \leq i < \text{llen}(\alpha) \rightarrow 0 \leq i' < i \rightarrow \alpha(i) \leq \alpha(i') \]

\[ OBA(x, n, \alpha) := BA(x, n, \alpha) \land \text{Ordered}(\alpha) \]

**OBAGet**. The OBAGet\((x, k)\) function, specified and verified below, retrieves the \(k\)-th element of the OBA which starts from memory location \(x\).

\[
\{[x, k] \mid OBA(x, n, \alpha) \land 0 \leq k < \text{llen}(\alpha)\}
\]

\[
\begin{align*}
\text{func \ OBAGet} \ [\text{int32}, \text{int32}] & \rightarrow \text{int32} \\
\{[] \mid OBA(x, n, \alpha) \land 0 \leq k < \text{llen}(\alpha) \land l_0 = x \land l_1 = k\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ get\_local} 0 \\
\{[l_0] \mid \text{emp}\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ get\_local} 1 \\
\{[l_1] \mid \text{emp}\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ i32\_const} 4 \\
\{[l_0, l_1] \mid \text{emp}\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ i32\_add} \pi_{\text{i32\_load} \text{offset}=4} \\
\{[\alpha k] \mid x + 4 + 4 \cdot k \mapsto_{i \leq \text{llen}(\alpha)} \alpha k\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ i32\_load} \text{offset}=4 \\
\{[\alpha k] \mid x + 4 + 4 \cdot k \mapsto_{i \leq \text{llen}(\alpha)} \alpha k\} \\
\end{align*}
\]

\[
\begin{align*}
\text{frame} \text{ i32\_load} \text{offset}=4 \\
\{[\alpha k] \mid x + 4 + 4 \cdot k \mapsto_{i \leq \text{llen}(\alpha)} \alpha k\} \\
\end{align*}
\]

\[
\begin{align*}
\{[\alpha k] \mid OBA(x, n, \alpha) \land 0 \leq k < \text{llen}(\alpha) \land l_0 = x \land l_1 = k\} \\
\end{align*}
\]

\[
\begin{align*}
\{[\alpha k] \mid OBA(x, n, \alpha) \land 0 \leq k < \text{llen}(\alpha)\} \end{align*}
\]

\[
\begin{align*}
\{[\alpha k] \mid OBA(x, n, \alpha) \land 0 \leq k < \text{llen}(\alpha)\} \end{align*}
\]
Anonymised Authors

OBAFind. The OBAFind(x, e) function, specified and verified below, finds the appropriate index for an element in an OBA list. It takes two parameters: x denotes the memory address at which the OBA is allocated, while e denotes the element that is being searched for. The function returns an integer i with the following properties: if e is in the OBA, i equals the index of its first occurrence; if e is not in the OBA, i is equal to the index of the first element of the OBA larger than e, if such an element exists, and to the OBA length otherwise.

\[
\{ (x, e) | \text{OBA}(x, n, \alpha) \} \\
(\text{func} \text{ OBAFind} \ [32, 32] \rightarrow [32])
\]

\[
(\text{locals} \ [32])
\]

\[
\{ (x, e) | \text{OBA}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \}
\]

\[P_{inv}: \text{OBA}(x, n, \alpha) \land l_0 = x \land l_1 = e \land 0 \leq l_2 \leq \text{llen}(\alpha) \land (\forall j. 0 < j < l_2 \rightarrow \alpha j < e) \]

\[
(\{ | P_{inv} \}) \quad \text{(by consequence)}
\]

\[
(\text{get} \_\text{local} \ 2)
\]

\[
(\{ | \text{emp} \}) \leftarrow (\{ | \text{emp} \})
\]

\[
(\{ | l_2 \} | P_{inv})
\]

\[
(\text{get} \_\text{local} \ 0) \ (\text{i32.load})
\]

\[
(\{l_2, \text{llen}(\alpha) \} | P_{inv})
\]

\[
(\text{i32.load})
\]

\[
(\exists v. [v] | P_{inv} \land C_1)
\]

\[
(\exists v. [v] | P_{inv} \land C_1) \quad \text{(by consequence)}
\]

\[
(\{ | P_{inv} \} + \{ [v] | P_{inv} \land C_1 \}
\]

\[
(\text{if})
\]

\[
(\{ | P_{inv} \land C_3 \}, (\{ | P_{inv} \}) \leftarrow
\]

\[
(\{ | P_{inv} \land l_2 \land \text{llen}(\alpha) \}
\]

\[
(\text{get} \_\text{local} \ 0) \ (\text{get} \_\text{local} \ 2) \ (\text{call} \ \text{OBAGet})
\]

\[
(\{l_0 \land l_2 \land \text{llen}(\alpha) \}
\]

\[
(\text{get} \_\text{local} \ 1) \ (\text{i32.load})
\]

\[
(\{3n, [v] | P_{inv} \land C_1 \}
\]

\[
(\text{exists})
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\text{get} \_\text{local} \ 2)
\]

\[
(\{ | P_{inv} \land l_2 \land \text{llen}(\alpha) \land \alpha l_2 < e \}
\]

\[
(\text{get} \_\text{local} \ 2) \ (\text{i32.const} \ 1) \ (\text{i32.add})
\]

\[
(\{l_2 + 1 \land P_{inv} \land l_2 \land \text{llen}(\alpha) \land \alpha l_2 < e \}
\]

\[
(\text{exists})
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\text{get} \_\text{local} \ 2)
\]

\[
(\{ | P_{inv} \} + \{ | P_{inv} \land C_3 \}
\]

\[
(\exists v. [v] | P_{inv} \land C_3)
\]

\[
(\exists v. [v] | P_{inv} \land C_3)
\]

\[
(\{ | P_{inv} \} + \{ | P_{inv} \land C_3 \}
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\text{get} \_\text{local} \ 2)
\]

\[
(\{ | P_{inv} \land l_2 \land \text{llen}(\alpha) \land (\forall j. 0 < j < l_2 \rightarrow \alpha j < e) \land \alpha l_2 < e \}
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\text{exists})
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\{ | P_{inv} \land C_3 \}
\]

\[
(\forall j. 0 < j < i \rightarrow \alpha j < e) \land (\forall j. i < j < \text{llen}(\alpha) \rightarrow e \leq \alpha j)
\]

\[
(\exists i | \text{OBA}(x, n, \alpha) \land 0 \leq i < s - \text{llen}(\alpha) \land (\forall j. 0 < j < i \rightarrow \alpha j < e))
\]

\[
(\forall j. i < j < \text{llen}(\alpha) \rightarrow e \leq \alpha j)
\]

\[
(\exists i | \text{OBA}(x, n, \alpha) \land 0 \leq i < s - \text{llen}(\alpha) \land (\forall j. 0 < j < i \rightarrow \alpha j < e))
\]

\[
(\forall j. i < j < \text{llen}(\alpha) \rightarrow e \leq \alpha j)
\]
where:
\[ C_1 \equiv (v = 0 \to l_2 = \text{llen}(\alpha)) \land (v > 0 \to l_2 < \text{llen}(\alpha)); \]
\[ C_2 \equiv (v = 0 \to \alpha l_2 \geq e) \land (v \neq 0 \to \alpha l_2 < e); \]
\[ C_3 \equiv ((l_2 < \text{llen}(\alpha) \land \alpha l_2 \geq e) \lor l_2 = \text{llen}(\alpha)). \]

AsegShr. The AsegShr\((x, n)\) function, specified and verified below, shifts an array segment to the right. It takes two parameters: \(x\) denotes the address in memory at which the array segment is allocated, while \(n\) denotes the length of the segment that is to be shifted to the left. The resulting OBA segment contains all of its previous elements except the one at the front, and the last element of the original array segment is forgotten.

\[
\{\{x, n\} \mid \text{Aseg}(x, a; \alpha) \land n = \text{llen}(\alpha)\}
\]

\text{func AsegShr \:[i32, i32] \rightarrow \{\}}
\text{\{locals i32, i32\}}
\text{\{get_local 0\} (tee_local 2) (i32.add) (set_local 3) (i32.const 0) (set_local 0)}
\text{\{local 0\} (tee_local 1) (i32.add) (set_local 1)}
\text{\{local 0\} (i32.load) (i32.store)}
\text{\{local 0\} (i32.const 1) (i32.add) (set_local 0)}
\text{\{local 0\} (i32.add) (tee_local 2) (i32.const 4) (i32.add) (set_local 3)}
\text{\{local 0\} (i32.const 1) (i32.add) (i32.store)}
\text{\{br 1\}}

AsegShr. The AsegShr\((x, n)\) function, specified below and verified analogously to AsegShl, shifts an array segment to the right. It takes two parameters: \(x\) denotes the memory address at which the (non-empty) array segment is allocated, while \(n\) denotes the length of the segment to be shifted to the right. We also require ownership of one element past the length of the list. The resulting array segment contains an additional, duplicated element at the front.

\[
\{\{x, n\} \mid \text{Aseg}(x, a; \alpha) \land x + 4 \cdot \text{llen}(\alpha) \rightarrow \text{i32} \land n = \text{llen}(\alpha)\}
\]

\text{func \:[i32, i32] \rightarrow \{\} AsegShr \ldots \text{end}}
\text{\{local 0\} (i32.const 0) (set_local 0)}
\text{\{local 0\} (i32.add) (i32.store)}
\text{\{local 0\} (i32.add) (i32.load) (i32.store)}
\text{\{local 0\} (i32.add) (i32.load) (i32.store)}
\text{\{br 1\}}

The AsegShl\((x, n)\) function, specified below and verified analogously to AsegShl, shifts an array segment to the left. It takes two parameters: \(x\) denotes the address in memory at which the array segment is allocated, while \(n\) denotes the length of the segment that is to be shifted to the left. The resulting OBA segment contains all of its previous elements except the one at the front, and the last element of the original array segment is forgotten.

\[
\{\{x, n\} \mid \text{Aseg}(x, a; \alpha) \land n = \text{llen}(\alpha)\}
\]

\text{func AsegShl \:[i32, i32] \rightarrow \{\}}
\text{\{locals i32, i32\}}
\text{\{get_local 0\} (tee_local 2) (i32.add) (set_local 3) (i32.const 0) (set_local 0)}
\text{\{local 0\} (i32.load) (i32.store)}
\text{\{local 0\} (i32.add) (i32.load) (i32.store)}
\text{\{local 0\} (i32.add) (i32.load) (i32.store)}
\text{\{br 1\}}

The AsegShr\((x, n)\) function, specified and verified below, shifts an array segment to the right. It takes two parameters: \(x\) denotes the address in memory at which the array segment is allocated, while \(n\) denotes the length of the segment that is to be shifted to the right. The resulting OBA segment contains all of its previous elements except the one at the front, and the last element of the original array segment is forgotten.

\[
\{\{x, n\} \mid \text{Aseg}(x, a; \alpha) \land n = \text{llen}(\alpha)\}
\]

\text{func AsegShr \:[i32, i32] \rightarrow \{\}}
B.2 Ordered, Bounded Arrays without Duplication

\[ \text{OBA}_{nd}(x, n, \alpha) := \text{OBA}(x, n, \alpha) \land \text{llen}(\alpha) = \text{card}(\text{ToSet}(\alpha)) \]

**OBAInsert.** The OBAInsert\((x, e)\) function, specified and verified below, inserts an element into a given \(\text{OBA}_{nd}\). If the element already is in the OBA, the OBA is not modified.

\[ \{[x, e] \mid \text{OBA}_{nd}(x, n, \alpha) \land \text{llen}(\alpha) < n \} \]

**func** OBAInsert \([i32, i32] \rightarrow []\)

**locals** \(i32\)

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \} \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land \]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land \]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \land
\]

\[ (v = 0 \rightarrow l_2 = \text{llen}(\alpha)) \land (v \neq 0 \rightarrow l_2 < \text{llen}(\alpha)) \]

**if**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \land
\]

\[ (v = 0 \rightarrow l_2 = \text{llen}(\alpha)) \land (v \neq 0 \rightarrow l_2 < \text{llen}(\alpha)) \]

**else**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \land
\]

**end**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha') \land \text{ToSet}(\alpha') = \text{ToSet}(\alpha) \cup \{e\} \]

**end**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha') \land \text{ToSet}(\alpha') = \text{ToSet}(\alpha) \cup \{e\} \]

**end**

\[ \{ [x, e] \mid \text{OBA}_{nd}(x, n, \alpha) \} \]

**func** OBADelete \([i32, i32] \rightarrow []\)

**locals** \(i32\)

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (v \neq 0 \rightarrow \alpha^i = e) \land (v \neq 0 \rightarrow \alpha^{i+1} = e) \]

**if**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (v \neq 0 \rightarrow \alpha^i = e) \land (v \neq 0 \rightarrow \alpha^{i+1} = e) \]

**end**

**end**

\[ \{ [x, e] \mid \text{OBA}_{nd}(x, n, \alpha) \} \]

**func** OBAInsert \([i32, i32] \rightarrow []\)

**locals** \(i32\)

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \}

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \]

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \land
\]

\[ (v = 0 \rightarrow l_2 = \text{llen}(\alpha)) \land (v \neq 0 \rightarrow l_2 < \text{llen}(\alpha)) \]

**if**

\[ \{ \} \mid \text{OBA}_{nd}(x, n, \alpha) \land l_0 = x \land l_1 = e \land l_2 = 0 \land \text{llen}(\alpha) < n \land e \notin \text{ToSet}(\alpha) \land 0 \leq i < \text{llen}(\alpha) \land
\]

\[ (\forall j. 0 \leq j < i \rightarrow \alpha^j < e) \land (\forall j. i \leq j < \text{llen}(\alpha) \rightarrow e < \alpha^j) \land
\]

**end**

**end**
A Program Logic for First-Order Encapsulated WebAssembly

B.3 B-Tree Metadata Page

\[
\text{Page}(a) := \begin{cases} 
0 & (a \cdot 64k) \leq i < ((a + 1) \cdot 64k) \\
\text{Page}(a!i) & \text{otherwise}
\end{cases} \quad (i \mapsto_{a \cdot 32} -) \wedge 0 \leq a \wedge ((a + 1) \cdot 64k) \leq \text{INT32\_MAX}
\]

\[
\text{Free}(\alpha) := \text{OBA}_\text{rd}(8, 16381, \alpha) \begin{cases} 
\text{Page}(\alpha!1) & \text{otherwise}
\end{cases}
\]

\[
\text{Meta}(t, r, l, \alpha) := 0 \mapsto_{a \cdot 32} t \ast 4 \mapsto_{a \cdot 32} r \ast \text{size}(l) \ast \text{Free}(\alpha).
\]

B.4 B-Tree Nodes

\[
\text{Keys}(x, \kappa) := \text{OBA}_\text{rd}(x \cdot 64k + 4, 4095, \kappa);
\]

\[
\text{Ptrs}(x, \pi) := \text{BA}(x \cdot 64k + 32k, 4096, \pi);
\]

\[
\text{Node}(x, \lambda, \kappa, \pi) := x \cdot 64k \mapsto_{a \cdot 32} \lambda \ast \text{Keys}(x, \kappa) \ast \text{Ptrs}(x, \pi).
\]

InitNode. The InitNode function, specified and verified below, initialises a given Web-Assembly memory page to represent a B-tree leaf node.

\[
\begin{align*}
\{[x] | \text{Page}(x)\} \\
\{[x] | \text{Page}(x) \land l_0 = x\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Page}(x) \land l_0 = x \cdot 64k\} \\
\end{align*}
\]

FreeNode. The FreeNode function, specified and verified below, frees the memory page belonging to a given B-tree node. If the free page set is not full, the page is added to the free page set. Otherwise, the function does not terminate.

\[
\begin{align*}
\{[x] | \text{Free}(\alpha) \land \text{lken}(\alpha) < 16381 \ast \text{Node}(x, -, -)\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\{[x] | \text{Free}(\alpha) \ast \text{Node}(x, -, -) \land l_0 = x \cdot 64k\} \\
\end{align*}
\]
AllocNode. The AllocNode function, specified and verified below, allocates a B-tree leaf node. The node’s address is chosen from the list of free pages, if that list is non-empty, otherwise a new page is allocated, if possible. If not, the function does not terminate.

```c
func AllocNode () -> []
{[l] | size(l) = Free(a)}
```

```c
(i32.const 8) (i32.load) (i32.const 0) (i32.gt)
{\exists \alpha \in \varepsilon | \text{size}(l) = \text{Free}(a) \land (v = 0 \rightarrow \alpha = [\ ] \land (v \neq 0 \rightarrow 3l, \alpha^' = l^' \alpha^'))}
```

```
if
{[l] | size(l) = \text{Free}(a^')}
```

```
(i32.const 12) (i32.load) (call InitNode) (We do, initialise node on first free page)
```

```
(i32.const 8) (i32.const 12) (i32.load)
```

```
(call OBADelete) (Remove the re-allocated page from the list of free pages)
```

```
POST
```

```
else
loop
```

```
(i32.const 1) (mem.grow) (We do not, attempt to allocate another page)
```

```
(tee local 0)
```

```
(i32.const -1) (i32.eq) (br_if 0) (Loop forever if we cannot allocate)
```

```
[ ] | \text{Page}(l) = \text{size}(l + 1) \land l_0 = l
```

```
end
```

```
(get_local 0) (Get the address of the newly allocated page)
```

```
([l] | \text{size}(l) = \text{size}(l + 1) \land \text{Free}(a) \land \text{l}_0 = l)
```

```
(call InitNode) (Initialise node on the newly allocated page)
```

```
POST
```

```
end
```

```
POST
```

```
where
\{POST\} = \{3l', [l'] | \text{Node}(l', 1, [\ ], [\ ]) \land ((\alpha = [\ ] \land l' = l \land \text{size}(l + 1) = \text{Free}(a) \lor ))
\}
```

```
(\exists \alpha', \alpha^' = l^' \alpha^' = \text{size}(l) = \text{Free}(a'))
```

Onward, when calling AllocNode, we will assume the set of free pages to be empty, for simplicity, and use the following specification:

```
{[l] | size(l) = \text{Free}(l')}
```

```
(func AllocNode [] -> [32] ... end)
```

```
{[l] | \text{Node}(l, 1, [\ ], [\ ]) \land \text{size}(l + 1) = \text{Free}(l')}
```

GetNodeLeaf. The GetNodeLeaf(x) function, specified and verified below, returns the leaf information of a given B-tree node at address x.

```
{[\ ] | \text{Node}(x, \lambda, \kappa, \pi)}
```

```
(func GetNodeLeaf [32] -> [32]
{[\ ] | \text{Node}(x, \lambda, \kappa, \pi) \land l_0 = x}
```

```
(get_local 0) (i32.const 64k) (i32.mul) (i32.load)
```

```
{[\lambda] | \text{Node}(x, \lambda, \kappa, \pi) \land l_0 = x}
```

```
end
```

```
(\lambda) | \text{Node}(x, \lambda, \kappa, \pi)
```

SetNodeLeaf. The SetNodeLeaf function, specified and verified below, stores the provided leaf information \lambda in a given B-tree node at address x.
GetNodeKey. The GetNodeKey function, specified and verified below, retrieves the $i$-th key of a B-tree node at address $x$.

```
{[x, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ)}
```

```
func GetNodeKey [i32, i32] → i32
{[i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
```

```
get_local 0 (i32.const 64k) (i32.mul) (get_local 1) (i32.store)
```

```
{[i] | Node(x, λ, κ, π) ∧ l0 = x ∧ l1 = λ}
end
```

```
{[i] | Node(x, λ, κ, π)}
```

GetNodePtr. The GetNodePtr function, specified and verified below, retrieves the $i$-th pointer of a non-leaf B-tree node at address $x$.

```
{[x, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(π)}
```

```
func GetNodePtr [i32, i32] → [i32]
{[i] | Node(x, 0, κ, π) ∧ 0 ≤ i < llen(π) ∧ l0 = x ∧ l1 = i}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 4) (i32.add) (get_local 1)
```

```
{[x + 64k + 4, i] | Node(x, 0, κ, π) ∧ 0 ≤ i < llen(π) ∧ l0 = x ∧ l1 = i}
```

```
call OBAGet
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
end
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ)}
```

InsertNodeKey. The InsertNodeKey function, specified and verified below, inserts the key $k$ into the keys of a B-tree node at address $x$, potentially extending the keys.

```
{[x, k] | Node(x, λ, κ, π) ∧ llen(κ) < 4095}
```

```
func InsertNodeKey [i32, i32] → [i32]
{[i] | Node(x, λ, κ, π) ∧ llen(κ) < 4095 ∧ l0 = x ∧ l1 = k}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 4) (i32.add) (get_local 1)
```

```
{x + 64k + 4, i} | Node(x, λ, κ, π) ∧ llen(κ) < 4095 ∧ l0 = x ∧ l1 = k}
```

```
call OBInsert
```

```
{κ, i'} | Node(x, λ, κ', π) ∧ ToSet(κ') = ToSet(κ) ∪ {k} ∧ l0 = x ∧ l1 = k}
end
```

```
{κ, i'} | Node(x, λ, κ', π) ∧ ToSet(κ') = ToSet(κ) ∪ {k}
```

SetNodePtr. The SetNodePtr function, specified and verified below, sets the $i$-th pointer of a given B-tree non-leaf node at address $x$ to value $p$, potentially extending the pointers.

```
{[x, i, p] | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096))}
```

```
func SetNodePtr [i32, i32, i32] → []
{[i] | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096)) ∧ l0 = x ∧ l1 = i ∧ l2 = p}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 32k) (i32.add) (get_local 1) (get_local 2)
```

```
{x + 64k + 32k + i, p} | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096))
```

```
call BASet
```

```
{κ, i'} | Node(x, 0, κ', π) ∧ 0 ≤ i ≤ llen(π) ∧ κ' = SubList(π, 0, i) · e · SubList(π, i + 1, llen(π) − i − 1)
```

GetNodeKey. The GetNodeKey function, specified and verified below, retrieves the $i$-th key of a B-tree node at address $x$.

```
{[x, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ)}
```

```
func GetNodeKey [i32, i32] → i32
{[i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 4) (i32.add) (get_local 1)
```

```
{[x + 64k + 4, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
```

```
call OBAGet
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
end
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ)}
```

GetNodePtr. The GetNodePtr function, specified and verified below, retrieves the $i$-th pointer of a non-leaf B-tree node at address $x$.

```
{[x, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(π)}
```

```
func GetNodePtr [i32, i32] → [i32]
{[i] | Node(x, 0, κ, π) ∧ 0 ≤ i < llen(π) ∧ l0 = x ∧ l1 = i}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 32k) (i32.add) (get_local 1)
```

```
{x + 64k + 32k + i, i} | Node(x, 0, κ, π) ∧ 0 ≤ i < llen(π) ∧ l0 = x ∧ l1 = i}
```

```
call OBAGet
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ) ∧ l0 = x ∧ l1 = i}
end
```

```
{[κ, i] | Node(x, λ, κ, π) ∧ 0 ≤ i < llen(κ)}
```

InsertNodeKey. The InsertNodeKey function, specified and verified below, inserts the key $k$ into the keys of a B-tree node at address $x$, potentially extending the keys.

```
{[x, k] | Node(x, λ, κ, π) ∧ llen(κ) < 4095}
```

```
func InsertNodeKey [i32, i32] → [i32]
{[i] | Node(x, λ, κ, π) ∧ llen(κ) < 4095 ∧ l0 = x ∧ l1 = k}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 4) (i32.add) (get_local 1)
```

```
{x + 64k + 4, i} | Node(x, λ, κ, π) ∧ llen(κ) < 4095 ∧ l0 = x ∧ l1 = k}
```

```
call OBInsert
```

```
{κ, i'} | Node(x, λ, κ', π) ∧ ToSet(κ') = ToSet(κ) ∪ {k} ∧ l0 = x ∧ l1 = k}
end
```

```
{κ, i'} | Node(x, λ, κ', π) ∧ ToSet(κ') = ToSet(κ) ∪ {k}
```

SetNodePtr. The SetNodePtr function, specified and verified below, sets the $i$-th pointer of a given B-tree non-leaf node at address $x$ to value $p$, potentially extending the pointers.

```
{[x, i, p] | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096))}
```

```
func SetNodePtr [i32, i32, i32] → []
{[i] | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096)) ∧ l0 = x ∧ l1 = i ∧ l2 = p}
```

```
get_local 0 (i32.const 64k) (i32.mul) (i32.const 32k) (i32.add) (get_local 1) (get_local 2)
```

```
{x + 64k + 32k + i, p} | Node(x, 0, κ, π) ∧ ((0 ≤ i < llen(π)) ∨ (i = llen(π) < 4096))
```

```
call BASet
```

```
{κ, i'} | Node(x, 0, κ', π) ∧ 0 ≤ i ≤ llen(π) ∧ κ' = SubList(π, 0, i) · e · SubList(π, i + 1, llen(π) − i − 1)
```
B-Tree with branching factor

We define an abstract predicate describing what it means to be a WebAssembly
bounded array to a given value, is:

\{[x, i, e] | \text{BA}(x, n, \alpha) \land (0 \leq i < \text{llen}(\alpha)) \lor (i = \text{llen}(\alpha) < n)\}

\text{func \text{BASet}[32, 132, i32] \to [] ...

\{3a'. [] | \text{BA}(x, n, \alpha') \land 0 \leq i < \text{llen}(\alpha) \land \alpha' = \text{SubList}(\alpha, 0, i) \cdot [e] \cdot \text{SubList}(\alpha, i + 1, \text{llen}(\alpha) - i - 1)\}

B.5 B-Trees

Definition. We define an abstract predicate describing what it means to be a WebAssembly
B-Tree with branching factor t and keys \(\kappa\) as follows:

\(\text{BTree}(t, \kappa) \triangleq \exists r, l, \alpha, \lambda, \phi. \text{Meta}(t, r, l, \alpha) \cdot \text{BTreeRec}^{t,r,l}(r, \kappa, \lambda, \phi),\)

where:

\(\text{BTreeRec}^{t,r,l}(x, \kappa, \lambda, \phi) \triangleq x < l \land\)

\((\exists \pi_x, \pi_x. (\text{Node}(x, \lambda_x, \kappa_x, \pi_x) \land \text{llen}(\kappa_x) < 2t - 1 \land (x \neq r \to t - 1 \leq \text{llen}(\kappa_x)) \land\)

\((\text{llen}(\kappa_x) < 2t - 1 \to \phi = 0) \land (\text{llen}(\kappa_x) = 2t - 1 \to \phi \neq 0)) \cdot\)

\((\lambda \neq 0 \to \pi_x = [] \land \text{ToSet}(\kappa_x) = \kappa \land \text{emp}) \cdot\)

\((\lambda = 0 \to 0 < \text{llen}(\kappa_x) = \text{llen}(\pi_x) - 1 \land \exists \lambda', \pi, \phi. (\text{llen}(\pi) = \text{llen}(\phi) = \text{llen}(\pi_x) \land\)

\(\kappa = \bigcup_{0 \leq i < \text{llen}(\pi_x)} \kappa_x \land \bigotimes_{0 \leq i < \text{llen}(\pi_x)} \text{BTreeRec}^{t,r,l}(\pi_x[i], \pi[i], \lambda', \phi) \land\)

\((\bigwedge_{0 \leq i < \text{llen}(\pi_x)} \forall k, k'. k \in \pi[i] \to k' \in \pi[i] \to k < k_x[i] \land k' < k_x[i]) \land\)

\((\bigwedge_{0 \leq i < \text{llen}(\pi_x)} \pi_x[i] < t)))\).

B.5.1 B-Tree Creation

A B-Tree is always created from an empty memory. We first allocate the metadata page,
then allocate the first, root node, which is set to be a leaf and left empty. The branching
factor of the B-Tree, \(t\) is given as the only parameter.

\{[t] | \text{size}(0) \land 2 \leq t \leq 2048\}

\text{func \text{BTreeCreate [i32] \to []\}

\{[t] | \text{size}(0) \land 2 \leq t \leq 2048 \land l_0 = t\}

\text{loop}

\(\{[t] | \text{size}(0) \land 2 \leq t \leq 2048 \land l_0 = t\}

(i32.const 1) (\text{mem.grow}) (i32.const -1) (i32.eq 0) (\text{Allocate metadata page})\)

\{[t] | \text{Page}(0) \neq \text{size}(1) \land 2 \leq t \leq 2048 \land l_0 = t\}

(i32.const 0) (\text{get.local 0}) (i32.store) (\text{Set the branching factor})\)

(i32.const 4) (i32.const 0) (i32.store) (\text{Set the root node})\)

(i32.const 8) (i32.const 0) (i32.store) (\text{Set the free page list to empty})\)

]] (\text{Fold Free}([]): \text{Fold Meta}(t, 0, 1, [ ]))\]

\text{call \text{AllocateNode}(\text{drop})\}

\{[t] | \text{Meta}(t, 0, 2, [ ] \land \text{Node}(t, 1, [], [ ]) \land 2 \leq t \leq 2048\}

]] (\text{Fold BTreeRec}^{0,0,2}(0, 1, [ ]))\]

\{[t] | \text{Meta}(t, 0, 2, [ ] \land \text{BTreeRec}^{0,0,2}(0, 1, [ ]) \land 2 \leq t \leq 2048\}

]] (\text{Fold BTree}(t, [ ]))\]

\{[t] | \text{BTree}(t, [ ]) \land 2 \leq t \leq 2048\}

]] (\text{Fold Free}([]): \text{Fold Meta}(t, 0, 1, [ ]))\]

\text{call \text{AllocateNode}(\text{drop})\}

\{[t] | \text{Meta}(t, 0, 2, [ ] \land \text{Node}(t, 1, [], [ ]) \land 2 \leq t \leq 2048\}

]] (\text{Fold BTreeRec}^{0,0,2}(0, 1, [ ]))\]

\{[t] | \text{Meta}(t, 0, 2, [ ] \land \text{BTreeRec}^{0,0,2}(0, 1, [ ]) \land 2 \leq t \leq 2048\}

]] (\text{Fold BTree}(t, [ ]))\]

\{[t] | \text{BTree}(t, [ ]) \land 2 \leq t \leq 2048\}
1.42  A Program Logic for First-Order Encapsulated WebAssembly

B.5.2  B-Tree Search

The B-tree search function, BTreeSearch, takes a key \( k \) that is being searched for and returns a non-zero result if the B-tree contains the tree \( k \), and zero if it does not. It uses an auxiliary function, BTreeSearchRec, which traverses the B-tree recursively.

```
{[x, k] | BTreeRec^{r,l}(x, κ, λ, φ)}

(func BTreeSearchRec [i32, i32] → [i32]

(locals [i32])

{[] | BTreeRec^{r,l}(x, κ, λ, φ) ∧ l0 = x ∧ l1 = k ∧ l2 = 0}

(get_local 0) (i32.const 64k) (i32.mul) (i32.const 4) (i32.add) (get_local 1)

{[x · 64k + 4, k] | BTreeRec^{r,l}(x, κ, λ, φ) ∧ l0 = x ∧ l1 = k ∧ l2 = 0}

{[x · 64k + 4, k] | Node(x, κ, λ, πx) * ...

(call OBAFind)

3.1. [1] | Node(x, κ, λ, πx) ∧ 0 < i ≤ len(κx) ∧ (∀ j, 0 ≤ j < i → κx{l}j < k) ∧

(∀ j, i < j < len(κx) → k < κx{l}j) * ...

(tree-local 2) (get_local 0) (i32.load)

{[l1, len(κx)] | Node(x, κ, λ, πx) ∧ 0 < l2 ≤ len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 ≤ j < len(κx) → k < κx{l}j) * ...

(i32.it)

(if

{[] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ * ...

(get_local 0) (get_local 2) (call GetNodeKey) (get_local 1)

{[κx{l}2, l2] | Node(x, κ, λ, πx) ∧ 0 < l2 < len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ * ...

(i32.eq)

(if

{[] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ * ...

(i32.const 1)

{[1] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ κx{l}2 = k ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) * ...

{[1] | BTreeRec^{r,l}(r, κ, λ, φ) ∧ k ∈ κ

(POST)

else

{[] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) * ...

(get_local 0) (call GetNodeLeaf) (i32.const 0)

{λ, 0] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) * ...

(i32.ne)

(if

{[] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ λ ≠ 0 ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ ToSet(κx) = k * ...

(i32.const 0)

{0] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ λ ≠ 0 ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ ToSet(κx) = k * ...

{[0] | BTreeRec^{r,l}(r, κ, λ, φ) ∧ k ≠ κ

(POST)

else

{[] | Node(x, κ, λ, πx) ∧ 0 ≤ l2 < len(κx) ∧ λ = 0 ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ λ = (\bigcup_{0 \leq j < len(κx)} πl_j) \cup κx ∧ k \notin \bigcup_{0 \leq j < len(κx)} πl_j * ...

(i32.const 0) (get_local 3) (call GetNodePtr) (get_local 1)

{[πx{l}2, l2] | BTreeRec^{r,l}(r, κ, λ, φ) ∧ 0 ≤ l2 < len(κx) ∧ λ = 0 ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ λ = (\bigcup_{0 \leq j < len(κx)} πl_j) \cup κx ∧ k \notin \bigcup_{0 \leq j < len(κx)} πl_j * ...

(call BTreeRec)

3b. [b] | BTreeRec^{r,l}(πx{l}2, πl{l}2, λ, κx{l}2) ∧ 0 ≤ l2 < len(κx) ∧ λ = 0 ∧ (∀ j, 0 ≤ j < l2 → κx{l}j < k) ∧

(∀ j, l2 < j < len(κx) → k < κx{l}j) ∧ λ = (\bigcup_{0 \leq j < len(κx)} πl_j) \cup κx ∧ k \notin \bigcup_{0 \leq j < len(κx)} πl_j * ...

(POST)

end)

end)

end)
```
else
  (\| \ Node (x, λ, κx, πx) \land l2 = llen (κx) \land (\forall j < |l2| \rightarrow κx \land j < k) \land * ...)
\end{itemize}

\begin{array}{l}
\mathbf{get\_local\_0}\ (\text{call\ GetNodeLeaf}) (i32.\text{const}\ 0)
\end{array}

\begin{itemize}
\item \{\| \ Node (x, λ, κx, πx) \land l2 = llen (κx) \land (\forall j < |l2| \rightarrow κx \land j < k) \land * ...
\end{itemize}

\subsection{B.5.3 B-Tree Insertion}

\textbf{B-Tree Child Splitting.} The auxiliary function \texttt{BTreeSplitChild} splits the (full) $i$-th child of a non-full, non-leaf node at address $x$ into two nodes with $t - 1$ keys each and moves its median key into the node at address $x$. The set of keys is effectively left unchanged. We note that the \texttt{SubList} predicate returns an empty list instead if its arguments do not make sense in the context of the given list.
A Program Logic for First-Order Encapsulated WebAssembly

```wasm
[locals i32, i32]

(allocnode) (set local 3)

(i32.const 0) (i32.load) (set local 4)

(i32.const 0) (i32.load) (i32.const 2) (i32.mul) (i32.const 2) (i32.sub) (i32.lt)

(if)

(get local 5) (i32.const 1) (i32.add) (br 1)

end)

end)

(get local 0) (get local 2) (i32.const 0) (i32.load) (i32.const 1) (i32.sub)

(call GetNodeKey) (call InsertNodeKey)

(i32.load) (i32.const 0) (i32.load) (i32.const 32) (i32.add)

(i32.const 1) (i32.add) (call AssegShr)

(i32.load) (i32.const 32) (i32.add)

(i32.add) (i32.add) (i32.add) (i32.add)

(i32.load) (i32.const 0) (i32.load)

(i32.const 0) (i32.const 0) (set local 1) (i32.load)

(i32.load) (i32.const 0) (i32.load) (i32.const 32) (i32.add)

(i32.add) (i32.add) (i32.add) (i32.add)
as described in its post-condition.

B-Tree Insertion into Non-Full Nodes. The auxiliary function BTreeInsertNonFull recursively traverses a subtree with a non-full root to insert a key k, which is not already in the subtree, into it. This function may extend the allocated memory by adding new nodes, as described in its post-condition.
Anonymised Authors 1:47

B-Tree Insertion.

The function BTreeInsert inserts a key \( k \) into the B-tree. If the key already exists, the B-tree is not modified.

```
BTreeInsert([x, y], \{ k \})
```

```bash
{[x, y] | BTree(x, \{ k \})}
```
else
  \{[] | \text{Meta}(t, r, l, \alpha) \land \text{BTreeRec}^{t,r,l}(r, \kappa, \lambda, \phi) \land l_0 = k \land l_1 = r \land l_2 = 2t - 1 \land k \neq \kappa \land \phi = 0 \}\}

\text{(get\_local 1)} \ (\text{get\_local 0})

\{[r, k] | \text{Meta}(t, r, l, \alpha) \land \text{BTreeRec}^{t,r,l}(r, \kappa, \lambda, \phi) \land l_0 = k \land l_1 = r \land l_2 = 2t - 1 \land k \neq \kappa \land \phi = 0 \}\}

\text{(call BTreeInsertNonFull)}

\{[] | \text{Meta}(t, r, l', \alpha) \land \text{BTreeRec}^{t,r,l'}(r, \kappa, \lambda, \phi) \land l_0 = k \land l_1 = r \land l_2 = 2t - 1 \land k \neq \kappa \land \phi \neq 0 \}\}

\{[[] | \text{BTree}(x, \kappa \cup \{k\})\}

\text{end)}

\{[] | \text{BTree}(x, \kappa \cup \{k\})\}

\text{end)}

\{[] | \text{BTree}(x, \kappa \cup \{k\})\}