Bridging the Gap: Precise Static Analysis of WebAssembly in a JavaScript World

ANONYMOUS AUTHOR(S)

WebAssembly is a low-level bytecode that runs alongside JavaScript to enable near-native performance on the web. Static analysis of WebAssembly is challenging, particularly in the case of call graph analysis because the dispatch of function calls is done using non-constant values with complex data-flow. Existing state-of-the-art analysis tools do not perform a pointer and value analysis which is essential for precise call graph analysis. Additionally, they analyze the WebAssembly binary in isolation and make worst-case assumptions about a binary's JavaScript client, which leads to loss of precision in a call graph analysis.

We address this gap with WasmBridge, the first multi-language static analysis tool that integrates interoperation between WebAssembly and JavaScript. WasmBridge introduces a refinement type system for Wasm that distinguishes numeric values from pointers and types WebAssembly memory to recover types for marshalled JavaScript data. We prove the type system sound and show that incorporating pointer information substantially improves the precision of call graph construction. Across a corpus of real-world Wasm applications, WasmBridge reduces the number of estimated call targets at 35% of indirect call sites—by up to 17% per site—yielding an overall call-graph edge reduction of 5% compared to state-of-the-art Wasm analyses. We further demonstrate the utility of our approach through dead-code elimination, identifying up to 20% more dead functions than an industry tool. WasmBridge provides an extensible foundation for whole-program, multi-language analysis of JavaScript-WebAssembly applications, enabling future integration with advanced JavaScript value and call graph analyses.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: WebAssembly, JavaScript, Interoperation, Multi-Language Static Analysis, Static Analysis, Constraints, Debloating, Dead Code Elimination

1 INTRODUCTION

Program analysis is essential to understanding the functionality and behavior of a binary program when its source code is unavailable. It has been used extensively to detect malware and vulnerabilities in binary programs [23, 35], to reverse engineer binaries [24] and to optimize binaries outside the compiler toolchain [2]. However, program analysis of binaries is also very challenging since a substantial amount of information is lost during compilation. Compilers erase types, variable and function names, and all high-level structure present in the source to create the most performant binary possible. This loss of information, coupled with the under-specified semantics of low-level languages and large instruction sets, makes writing sound and precise analyses for binaries difficult [3].

The problem of binary analysis is now also on the Web! With the introduction of WebAssembly (Wasm) in 2017 [20], compilers have been able to compile languages like C, C++, Rust, etc. to WebAssembly, a portable, low-level bytecode format that has been designed for computationally intensive tasks in the browser. These WebAssembly binaries interoperate with a JavaScript client on browsers and in Node.js applications. These applications don't include the source code of the binary and so, analysis writers for these applications must reckon with the hardships of binary analysis. Some of these hardships are mitigated by WebAssembly's formally specified semantics [40], structured control flow and low-level value types. Prior work [28] succinctly captures the specific challenges of WebAssembly static analysis with the example of call graph analysis. In WebAssembly, indirect calls are mediated through a table containing a list of functions, where an integer index is used to call a specific function from the table. Standard static analysis techniques dictate that a value analysis be used to determine the index used in indirect calls. However, Lehmann et al. [28] identify several unique challenges to WebAssembly call graph analysis, which they empirically validate over a dataset of 8,461 WebAssembly binaries [21]. They find that the dispatch of function calls is done using non-constant values with complex data-flow in nearly all cases, complicating the task at hand. Furthermore, since WebAssembly commonly interoperates with JavaScript, aspects of this interoperation pose further challenge to call graph analysis. For example, JavaScript can mutate exported WebAssembly function tables. Thus, to precisely determine the value of an index used for indirect calls, a WebAssembly static analysis would have to have both a sophisticated value and pointer analysis, and an analysis of the JavaScript client.

However, few WebAssembly static analysis tools have risen to the task. Tools like Wassail[36] either limit themselves to a constant index analysis, which is not found to occur in practice [28], while tools like Sturdy[22] use abstract interpretation to track the data flow of abstract integers inter-procedurally, but do not perform pointer analysis beyond that of constant memory addresses. Additionally, to the best of our knowledge, no Wasm static analysis tool supports passing in values from JavaScript. Current state-of-the-art tools either analyze WebAssembly in isolation or make worst-case assumptions about JavaScript client behavior, such as that a client always mutates the WebAssembly function table. Adding support for interoperation is non-trivial. While JavaScript Numbers and BigInts are passed to WebAssembly functions directly, other datatypes (strings, arrays, objects) are marshalled into WebAssembly memory and passed to Wasm functions as pointers to memory. No static analysis for WebAssembly performs pointer analysis at all.

In this paper, we address the central problem: the lack of support for interoperation with JavaScript in the WebAssembly static analysis literature. We perform a multi-language analysis to support passing in JavaScript values into a WebAssembly static analysis. Current value analyses cannot perform a multi-language analysis because of a lack of representation of pointers in the WebAssembly type system. WebAssembly has a low-level type system of 32- and 64-bit integer types (i32, i64) and 32- and 64-bit floating point types (f32, f64) and WebAssembly memory is untyped. We design a refinement type system that refines i32 types to either a singleton pointer type $\mathsf{ptr}(l, n)$ or a singleton number $\mathsf{num}(n)$ — where $\mathsf{ptr}(l, n)$ refers to a pointer to location l at offset n. We also type the WebAssembly memory, which lets us recover JavaScript values marshalled into the memory. We prove this refinement type system to be type-safe and show that the addition of a pointer analysis improves the precision of call graph analyses over standalone WebAssembly binaries. We improve precision at an average of 35% of indirect call sites, reduce the number of edges in the call graph by up to 5% compared to other Wasm static analysis tools, and discover a small number of call sites to be monomorphic, which enables additional compiler optimizations.

Along with our refinement type system, we also implement an analysis of the JavaScript code responsible for the marshalling and unmarshalling of JavaScript values to WebAssembly. While this does not improve precision of the WebAssembly analysis, it makes our tool extensible to other JavaScript client analyses. Specifically, to do a precise *open-world analysis* of a JavaScript-WebAssembly application, a tool that performs a value and call graph analysis of the JavaScript client can supply WasmBridge with JavaScript values passed to the relevant subset of WebAssembly-exported functions. We demonstrate the precision of such a multi-language analysis by performing dead-code elimination to specialize a WebAssembly binary to a JavaScript client. Here we are able to find up to 20% more functions to be dead than an industry tool, Wasm-Metadce[41] does. To the best of our knowledge, there is no off-the-shelf sophisticated value and call graph analysis for entire JavaScript applications. Thalakottur et al. [38] find that WebAssembly is often depended upon in JavaScript applications through complex indirect dependencies, which futher complicates the analysis. Therefore, we leave such analyses to future work.

We summarize the contributions of this work as follows:

(1) We present the first multi-language static analysis of WebAssembly and JavaScript, WasmBridge, that analyzes JavaScript wrappers and performs an intra-procedural pointer and value analysis over WebAssembly binaries.

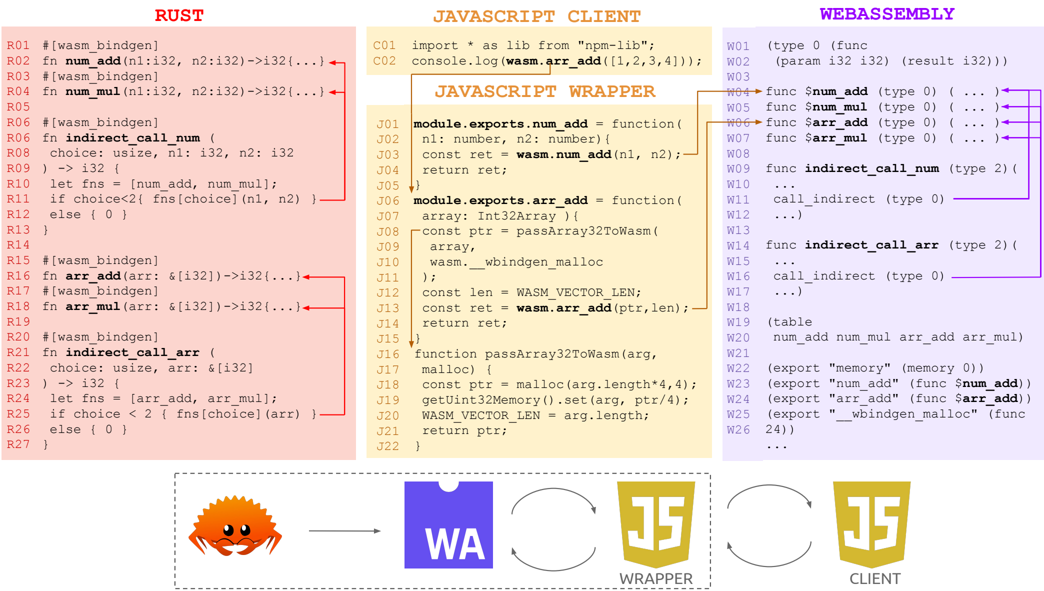

Fig. 1. A Rust Library compiled to WebAssembly along with its associated JavaScript wrapper.

(2) We develop a refinement type system for WebAssembly that refines the WebAssembly value type i32 to distinguish between numbers and pointers to WebAssembly memory.
(3) We show that our approach, WasmBridge, leads to a reduction in the number of estimated targets at an average of 35.63% of indirect call sites, with an average reduction of upto 17% at those call sites.
(4) We evaluate WasmBridge against current state-of-the-art static analyses for WebAssembly and show an edge reduction of 5% when compared against those tools.
(5) We estimate the impact of the increased precision of our analysis on dead-code elimination and find up to 20% more dead functions than a state-of-the-art industry tool, Wasm-Metadce.

The remainder of this paper is organized as follows. Section 2 provides background on the interoperation between WebAssembly and JavaScript and introduces the intuition behind our approach. Section 3, 4 and 5 delve into the specifics of our approach. We describe our evaluation against real-world subjects in Section 6. Section 7 and 8 discuss related and future work respectively. WasmBridge is anonymously available at https://anonymous.4open.science/r/wasmbridge-eval-2D7B/.

2 BACKGROUND AND MAIN IDEAS

2.1 Interoperation Through a Wrapper

We explain the interoperation between WebAssembly and JavaScript using the example shown in Figure 1, where we show a Rust library being compiled to WebAssembly and its associated compiler-generated JavaScript wrapper file. The library functions in the wrapper file are then called from a JavaScript client. We use different colors for each language for clarity.

2.1.1 The Library Developer Perspective. Let us consider the Rust library shown in Figure 1. The library consists of operations over numbers and arrays. num_add adds two numbers, num_mul multiplies two numbers, arr_add returns the sum of the elements of an array and arr_mul returns the result of multiplying all the elements of an array. The library also has two functions, indirect_call_num and indirect_call_arr that are used to indirectly call one of the number or array operations based on some user-specified choice. A Rust library developer who wants to compile this library to WebAssembly annotates the functions in the library that they want to expose to a JavaScript client with a special tag, #[wasm_bindgen][1]. We refer to functions that are exposed to the JavaScript client as *exported functions*. The reader can see in Figure 1 that all the library functions are exported. The JavaScript client is meant to call these exported functions with JavaScript objects. The translation from JavaScript values to Rust values is handled by wasm_bindgen and is not something that the Rust library developer has to worry about. For exported Rust functions, the compiler produces a WebAssembly binary along with a JavaScript wrapper file. For libraries being used by Node.js applications, these files (without the Rust source) are bundled in an NPM[2] package. A JavaScript client imports this NPM package to use the library from JavaScript. In Figure 1, the Rust library is bundled in the NPM package "npm-lib".

2.1.2 The JavaScript Client Perspective. A JavaScript client imports the library NPM package as an object lib (line C01). It then calls an exported function with JavaScript objects. On line C02, we see a call to arr_add with an array. The client does not translate this JavaScript object into a WebAssembly value. In fact, they do not interact with WebAssembly directly at all!

2.1.3 The JavaScript Wrapper. In real-world applications, JavaScript and WebAssembly interoperate through a wrapper: a JavaScript file generated by the compiler alongside the binary. This wrapper is used to *lower* a JavaScript object to WebAssembly before a call to a WebAssembly function and to *lift* a WebAssembly value to a JavaScript object after a call to a WebAssembly function. The code that does the lifting and lowering is often referred to as glue code. A WebAssembly module exposes certain functions to JavaScript through exports (lines W22-W26). The wrapper file contains JavaScript functions that have glue code around calls to exported WebAssembly functions.

To understand the glue code that mediates the interoperation between WebAssembly and JavaScript, let us go back to the example in Figure 1. In function num_add (line J01), we see no glue code before the call to num_add. This is because the JavaScript engine internally translates a JavaScript number to a WebAssembly i32 value. However, we see that in function arr_add (line J06), the Int32Array object passed into the function is passed to function passArray32ToWasm along with another exported WebAssembly function, __wbindgen_malloc. In the body of the function, we first see a call to __wbindgen_malloc with the size of the array in bytes. __wbindgen_malloc allocates a block of WebAssembly memory of the required size and returns a pointer to this block. Then a call to the function getUint32Memory accesses the WebAssembly memory[3] and updates the block at the pointer with the contents of the array. The pointer and the array length, both WebAssembly i32 values, are then passed in as arguments to arr_add (line J13). Since the WebAssembly type system is fairly low-level, this is an expected pattern. JavaScript objects that cannot be passed to Wasm directly are stored in memory and the pointer to memory is passed to Wasm instead.

2.2 Call-Graph Analysis Over the WebAssembly Binary

A call-graph analysis over the source code of the Rust library will resolve the targets for the indirect calls at R11 and R25 as num_add, num_mul and arr_add, arr_mul respectively. Unfortunately, a call-graph analysis over the corresponding WebAssembly binary produced by the compiler is not as precise. Indirect calls in WebAssembly occur using a call_indirect instruction and are mediated through a WebAssembly table that contains a list of functions (line W19). WebAssembly is a stack machine and instructions push and pop values from the stack. At runtime, the call_indirect instruction pops an i32 value from the stack which it uses to index into the table, thus determining

[1]wasm-bindgen is a popular tool that enables Rust-JavaScript interoperation by generating bindings and glue code for JavaScript clients looking to use a Rust library through WebAssembly. We discuss this glue code in detail shortly.
[2]NPM is the canonical package manager for Node.js applications.
[3]Note that the WebAssembly memory is also exported to JavaScript in W22.

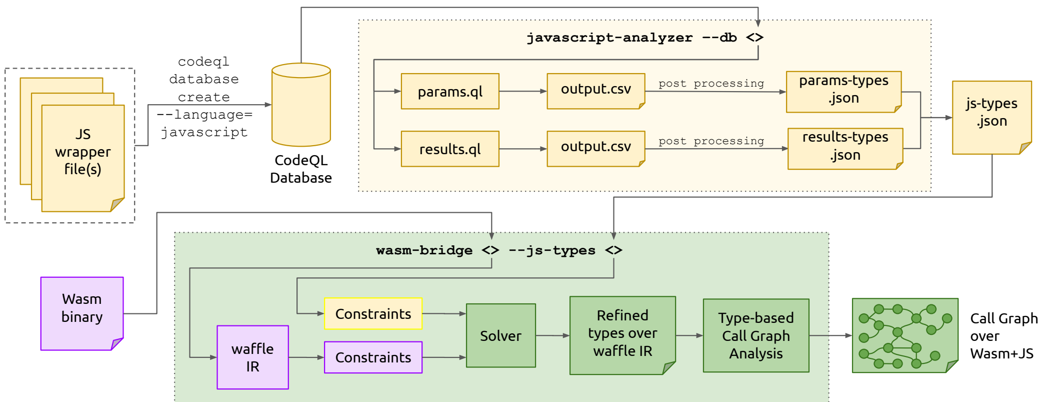

Fig. 2. System Diagram for Multi-Language Static Analysis for WebAssembly and JavaScript.

which function to call. The precision of a call-graph analysis depends on how indirect calls are handled. Current state-of-the-art analysis tools handle indirect calls in one of three ways:

(1) *Naive Analysis:* The target of an indirect call could be any function in the WebAssembly function table. Industry tools like Wasm-Opt and Wasm-Metadce[41] take this approach.
(2) *Type-based Analysis:* The syntax of a call_indirect instruction contains a function type annotation (line W11) which is guaranteed to match the function type of the indirect call target. State-of-the-art analysis tools like Wassail[36] and Wasma[5] restrict the set of possible targets to be the functions in the table whose type matches the type annotation.
(3) *Index Analysis*: Since an i32 value is used to index into the function table, tools like Wassilly [31] and Sturdy [22] perform a sophisticated value analysis to determine the set of possible targets.

Unfortunately, for the example in Figure 1, all three strategies yield the same result. The indirect calls at W11 and W16 have the same type annotation: type0, a function type that takes two i32s as arguments and returns an i32. The number and array add and multiply operations all compile down to have the same function type type0 and the WebAssembly table contains functions for all of these operations. Hence, a static analysis over this binary would resolve all four operations to be potential targets for both indirect calls. With a closed-world (WebAssembly only) analysis, there is no way to distinguish num_add from arr_add since they both have the same type. However, the JavaScript num_add and arr_add functions do have different types. We can see that num_add takes two numbers and passes them to num_add as two i32s, whereas arr_add takes an Int32Array and then calls arr_add with two i32s that actually represent, respectively, a pointer to the array allocated in WebAssembly memory and the length of that array. If our analysis included an analysis of the JavaScript wrapper, we could differentiate between these two Wasm functions more precisely!

2.3 Refining the WebAssembly Type System With Pointers

WebAssembly does not distinguish between numbers and pointers. The WebAssembly memory is only indexed by i32 values, or 32-bit integers, while i32 values are also used as conditionals for if-then-else expressions and as indices into the function table. Pointers to the WebAssembly memory can be distinguished in the JavaScript wrapper from regular numbers by virtue of being passed into WebAssembly through a different pattern, and so, we can refine i32 to two new types, ptr and num, both subtypes of i32. However, we must propagate these refined types through the WebAssembly code to have any hope of improving the precision at an indirect call site. To this end, we modify the WebAssembly typing rules, change the type system to be a refinement type system [9, 16, 19, 33] and type the WebAssembly memory to be able to track reads and writes to memory. The changes to the type system are described in Section 3. The changes we make to the type system are not extensive. We only refine i32's and do not refine the other base WebAssembly types, i64, f32 and f64. We also do not incorporate any of the rich JavaScript types for data allocated in the WebAssembly memory into the type system, leaving that to future work. Our hypothesis is that in working with even a simple set of refined types for WebAssembly, *we are able to be more precise in our estimation of indirect call targets simply by virtue of richer type information.*

Figure 2 illustrates how we discover these richer refinement types for WebAssembly. In order to type the arguments and returns for calls to WebAssembly exported functions in the JavaScript wrapper, we write a CodeQL [1] analysis that infers a refined type by checking for specific patterns of accessing the WebAssembly memory. CodeQL is a framework for writing static analyses with datalog-like queries. We then pass these types into our tool WasmBridge that encodes them as constraints. We also generate constraints over the WebAssembly binary under inspection. This allows us to discover refined types — for instance, if a WebAssembly instruction loads from or stores to an i32 then that type should be refined to a pointer. Instead of working over the binary directly, we generate constraints over waffle [8], an intermediate representation for WebAssembly that uses Static Single Assignment Form [10]. We then solve the constraints, while doing a value and pointer analysis, to get refinement types over the Wasm binary. We then apply a type-based call-graph analysis over the refined type system. We explain the details of this process in Section 3, 4 and 5. Note that we only generate and solve constraints over WebAssembly 1.0 and support analysis over JavaScript wrapper files generated by the Rust compiler.

3 A REFINEMENT TYPE SYSTEM FOR WEBASSEMBLY

3.1 Background on WebAssembly Instructions and Typing

Below we discuss a subset of WebAssembly instructions and their typing. For a description of the complete instruction set and module typing, please refer to Haas et. al.[20] and the WebAssembly 1.0 specification [40]. Instructions in WebAssembly operate on an implicit value stack by popping argument values and pushing computed results. As shown in Figure 3, the instruction typing judgment in WebAssembly has the form $C^w \vdash e^* : \tau_w^m \rightarrow \tau_w'^n$ which says that in a context C^w, the one-or-more instructions e^* expect a sequence of m values of types τ_w^i (for $i \in \{1, \ldots, m\}$) at the top of the stack and replace these with a sequence of n values of types $\tau_w'^j$ (for $j \in \{1, \ldots, n\}$) at the top of the stack. Note that here we use subscripts and superscripts w for grammars of original WebAssembly elements, while later we will use r for our refinement-typed WebAssembly.

3.1.1 Values and Arithmetic Instructions. WebAssembly has four value types: i32, i64, f32, f64. They represent 32- and 64-bit integers and 32- and 64-bit floating-point numbers. Values in WebAssembly are numeric constants tagged with the appropriate value type, e.g., the 32-bit integer 42 is represented as i32.const 42. The i32.const 42 instruction pops nothing from the stack and pushes the i32.const 42 value onto the stack. This is reflected in the typing rule for constants, as shown in Figure 3. The figure also contains the typing rule for binary operations, which expect two values of the type they are annotated with on the stack and produce a value for the same type. For example, i32.add expects two i32's on the stack and pushes a i32 onto the stack.

3.1.2 Control Constructs and Breaks. WebAssembly has control constructs such as blocks and loops and provides structured control flow with break instructions that are annotated with an index: br i. Here, i is a de-bruijn index [11] out of n labels that are associated with n enclosing control constructs. The target construct at i determines how the br instruction behaves —if it is a block, control jumps to the *end* of the block and if it is a loop, control jumps to the *start* of a loop. This is

$$C^w ::= \{\text{func } tf_w{}^*,\ \text{local } \tau_w{}^*,\ \text{global } \tau_w{}^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{label } (\tau_w^*)^*,\ \text{return } (\tau_w^*)^?\}$$

Typing Instructions With the Original WebAssembly Value Types. $\boxed{C^w \vdash e^* : tf_w}$

$$\tau_w ::= \text{i32} \mid \text{i64} \mid \text{f32} \mid \text{f64}$$
$$tf_w ::= \tau_w^* \rightarrow \tau_w^*$$

$$\frac{}{C^w \vdash \tau_w.\mathbf{const}\ c : \epsilon \rightarrow \tau_w}\ \textsc{Constant} \qquad \frac{}{C^w \vdash \tau_w.binop : \tau_w\tau_w \rightarrow \tau_w}\ \textsc{Binary Ops}$$

$$\frac{tf_w = \tau_w^m \rightarrow \tau_w^n \qquad C^w, \text{label}(\tau_w^n) \vdash e^* : tf_w}{C^w \vdash \mathbf{block}\ tf_w\ e^*\ \mathbf{end} : tf_w}\ \textsc{Block} \qquad \frac{C^w_{\text{label}}(i) = \tau_w^m}{C^w \vdash \mathbf{br}\ i : \tau_w^*\ \tau_w^m \rightarrow \tau_w^*}\ \textsc{Break}$$

$$\frac{C^w_{\text{func}}(i) = tf_w}{C^w \vdash \mathbf{call}\ i : tf_w}\ \textsc{Call} \qquad \frac{C^w_{\text{memory}} = n \qquad 2^a \leq (|tp| <)^? |\tau_w| \qquad (tp\_sz)^? = \epsilon \vee \tau_w = \mathbf{i}m}{C^w \vdash \tau_w.\mathbf{load}(tp\_sx)^?\ a\ o : \text{i32} \rightarrow \tau_w}\ \textsc{Load}$$

Fig. 3. Typing of a subset of WebAssembly Instructions in the Original Type System.

evident in the typing of the block and br instructions. Blocks are annotated with a signature tf_w that expects τ_w^m types on the top of the stack at the start of the block, and expects τ_w^n types on the top of the stack at the end of a block. The sequence of instructions in a block is type checked under a label that states that this enclosing block expects τ_w^n values on the stack. A br instruction finds the label associated with the i^{th} enclosing target control construct and before jumping to the end or start of the target, ensures it has the right types on the stack. If the target construct is a block, it would expect τ_w^n on the top of the stack.

3.1.3 Loads and Stores to Linear Memory. Loads and stores to WebAssembly memory are done with load and store instructions that are annotated with the type of data loaded from memory and stored in memory respectively. For example, i64.load loads a i64 value from memory. WebAssembly memory is only indexed by i32 values and so the i64.load expects an i32 value on the stack and pushes an i64 value to the stack.

3.1.4 Function Calls. Functions in WebAssembly are referenced using an immediate index into the function section of a module. Hence, a call instruction is annotated with a index i, identifying the function to be called. This index is used to look up the function type of the called function in the context C^w. The function type specifies the types at the top of the stack before and after the call.

3.2 Refinement Type System

We present a refinement over the original WebAssembly (version 1.0) type system in Figure 4. Here, R is a refined store context, analogous to the store context S in the standard WebAssembly type system. We discuss each change to the type system below:

(1) *Refinement Types, τ_r*: We introduce refinements to each base WebAssembly value type τ_w. For example, if the top of the stack is typed to be τ_w in WebAssembly, it is ascribed the refinement type $\tau_w(n)$ in our type system. A refinement type such as i32(42), in a more traditional presentation of refinement types, might be written as $\{n\colon \beta \mid \beta <: \text{i32} \wedge n = 42\}$. For types i32($n$) and f32($n$), $n \in \{\mathbb{N}_{32} \cup \top\}$, while for i64($n$) and f64($n$), $n \in \{\mathbb{N}_{64} \cup \top\}$. Floating point numbers are represented by their bits.

(2) *Pointers and Numbers*: Since i32s are used both as pointers into the WebAssembly memory and numbers, we add two new types, ptr(n) and num(n), where ptr and num are subtypes of i32. The types i32(n), ptr(n), and num(n) form a lattice, the subtyping relation and

$$
\begin{array}{lcl}
R & ::= & \{\text{inst } {C^r}^*,\ \text{tab } n^*,\ \text{mem } \Psi^*\} \\
C^r & ::= & \{\text{func } {tf_r}^*,\ \text{local } {\tau_r}^*,\ \text{global } {\tau_r}^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{label } (\tau_r^*, \Psi)^*,\ \text{return } (\tau_r^*, \Psi)^?\}
\end{array}
$$

Typing WebAssembly Instructions with Refinement Types $\boxed{R; C^r \vdash {e_r}^* : tf_r}$

$$
\begin{array}{lcl}
\tau_{r_{32}} & ::= & \mathsf{i32}(n) \mid \mathsf{ptr}(n) \mid \mathsf{num}(n) \mid \mathsf{f32}(n) \\
\tau_r & ::= & \tau_{r_{32}} \mid \mathsf{i64}(n) \mid \mathsf{f64}(n) \\
\tau & ::= & \mathsf{i32} \mid \mathsf{ptr} \mid \mathsf{num} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\Psi & ::= & \{(\mathsf{ptr}(n) \mapsto \tau_{r_{32}})^*\} \\
tf_r & ::= & \tau_r^*, \Psi \to \tau_r^*, \Psi'
\end{array}
$$

$$
\frac{}{R; C^r \vdash \tau.\mathbf{const}\ c : \tau(c)}\ \textsc{Constant}
\qquad
\frac{\tau_r^3 = \mathrm{binop}(\tau_r^1, \tau_r^2) \qquad \tau_r^3 <: \tau(\top)}{R; C^r \vdash \tau.binop : \tau_r^1 \tau_r^2 \to \tau_r^3}\ \textsc{Binary Ops}
$$

$$
\frac{\begin{array}{c} tf_r = \tau_r^m, \Psi_{\mathrm{pre}} \to \tau_r^n, \Psi_{\mathrm{post}} \qquad R_\Psi = \Psi_{\mathrm{pre}} \\ R; C^r, \mathrm{label}(\tau_r^n, \Psi_{\mathrm{post}}) \vdash e^* : \tau_r^m, \Psi_{\mathrm{pre}} \to (\tau_r')^n, \Psi'_{\mathrm{post}} \\ (\tau_r' <: \tau_r)^n \qquad \Psi'_{\mathrm{post}} <: \Psi_{\mathrm{post}} \end{array}}{R; C^r \vdash \mathbf{block}\ tf_r\ e^*\ \mathbf{end} : tf_r}\ \textsc{Block}
\qquad
\frac{\begin{array}{c} C^r_{\mathrm{label}}(i) = (\tau_r')^m, \Psi' \\ (\tau_r <: \tau_r')^m \qquad R_\Psi <: \Psi' \end{array}}{R; C^r \vdash \mathbf{br}\ i : \tau_r^*\ \tau_r^m \to \tau_r^*}\ \textsc{Break}
$$

$$
\frac{\begin{array}{c} C^r_{\mathrm{func}}(i) = (\tau_r')^m, \Psi'_{\mathrm{pre}} \to (\tau_r')^n, \Psi'_{\mathrm{post}} \\ tf_r = \tau_r^m, \Psi_{\mathrm{pre}} \to \tau_r^n, \Psi_{\mathrm{post}} \\ R_\Psi = \Psi_{\mathrm{pre}} \qquad (\tau_r <: \tau_r')^m \qquad (\tau_r' <: \tau_r)^n \\ \Psi_{\mathrm{pre}} <: \Psi'_{\mathrm{pre}} \qquad \Psi'_{\mathrm{post}} <: \Psi_{\mathrm{post}} \end{array}}{R; C^r \vdash \mathbf{call}\ i\ tf_r : \tau_r^m, \Psi_{\mathrm{pre}} \to \tau_r^n}\ \textsc{Call}
\qquad
\frac{\begin{array}{c} C^r_{\mathrm{memory}} = n \\ \tau_r = \mathrm{load\_and\_extend}(c + o, \tau, (tp\_sx)^?, R_\Psi) \\ 2^a \le (|tp| <)^? |\tau_r| \qquad (tp\_sz)^? = \epsilon \lor \tau_r = \mathbf{im} \end{array}}{\Psi; C^r \vdash \tau.\mathbf{load}(tp\_sx)^?\ a\ o : \mathrm{ptr}(c) \to \tau_r}\ \textsc{Load}
$$

Fig. 4. Typing of a subset of WebAssembly Instructions in the Refinement Type System.

meet and join operations for which are discussed in Section C. Refinements of the other WebAssembly value types, $\mathsf{i64}(n)$, $\mathsf{f32}(n)$, and $\mathsf{f64}(n)$, are not refined further since they are not used to index memory.

(3) *Memory Typing,* Ψ: Unlike the original WebAssembly type system, we type the flat contiguous array of bytes that serves as the WebAssembly linear memory. Since we hope to propagate types from JavaScript to indirect call sites, and since memory operations are ubiquitous in WebAssembly, we need to be able to recover the (refined) types of values that are stored in memory at a specific address. A memory typing Ψ is a mapping from pointer addresses a, which are 32-bit integers, to refinement types τ_r.

(4) *Function Types*: Function types tf_w in WebAssembly are used to encode the types of functions, blocks, loops, and calls. They specify the (input and output) base WebAssembly types τ_w for all of these constructs. In our refinement type system, a function type specifies not only the lists of (input and output) refinement types τ_r, but also the shape of the (input and output) memory typing Ψ. We must track how the memory typing changes over the course of execution since WebAssembly programs perform strong updates that can change the type of data stored in memory. For instance, a function that expects a pointer at location a, because it loads from a, might later store an f32 at location a.

(5) *Label Typing*: In WebAssembly, labels of control constructs contain the shape of the stack expected when there is a jump to the control construct. In our refinement type system, we must specify not only the expected refined types τ_r on the stack but also the expected memory typing Ψ for the control construct in the label.

Next, we describe the refinement typing rules for a subset of WebAssembly instructions (shown in Figure 4). Refinement typing rules for the entire instruction set can be found in Appendix A.

3.2.1 Arithmetic Instructions. We restrict the refinement typing of arithmetic instructions to only allow certain combinations of i32, ptr and num. For instance, in the Binary Ops rule in Figure 4, the result type is computed using the function binop which takes two input refinement types and computes a result refinement type, but only considers certain pairs of input types valid for each specific binary instruction. The instruction will fail to type check if a invalid pair of input types is provided for a given binary operation. We discuss a few specific binary operations below.

- *Comparison*: Comparison of i32s and all its subtypes result in a num. All combinations of i32, ptr and num can be compared.
- *Addition*: WebAssembly allows addition of two i32s, either of which, or the result of which, can be used to index into the WebAssembly memory. We do not allow addition of two pointers, but allow addition of every other combination of i32, ptr and num. Moreover, addition of a ptr and a num yields a ptr.
- *Subtraction*: We do not allow subtraction of ptr from a num, but we do allow all other i32, ptr and num combinations for the subtraction operation. Subtracting a ptr from a ptr yields a num (an offset).

3.2.2 Control Constructs and Breaks. The Block rule is annotated with a refined function type tf_f which specifies the stack shape and memory typing expected before the block instruction and at the end of the block instruction. Since the end of a block is the meet of several breaks out of the block, we ensure that the shape of the stack and memory after type checking the instructions in the block should be subtypes of the types expected on the top of the stack and of the expected memory typing. This is necessary since it is valid in WebAssembly for one control path to the end of the block to return a ptr and another to return a i32, since they would both be integers in the WebAssembly type system. br instructions are typed similarly.

3.2.3 Loads from Linear Memory. While in WebAssembly, i32s are used to index into memory, our refinement type system requires that only values of ptr type be used to index into memory. We modify the typing rule for the Load instruction to expect a $\mathtt{ptr}(n)$ on the stack, as shown in Figure 4. The load instruction is optionally annotated with a packed type and size, tp_sx, which is used to pack and sign extend the data stored in memory. Addtionally, our memory typing Ψ only stores types of 32-bit size, $\tau_{r_{32}}$. If the load instruction expects a type of size 64 bits, we load the sequence of bytes from address $n + o$, where o is the offset provided to the load instruction, and address $n + o + 4$ as a 64-bit value as directed by the τ annotation on the load instruction which specifies the type of data we want to load. All this is done by the load_and_extend function, in Appendix A.

3.2.4 Function Calls. Typing a call instruction is more complex than in WebAssembly since our refinement type system has subtyping and since it keeps track of the expected memory typing at various points in the program. We change the WebAssembly **call** i instruction to **call** $i\ tf_r$ where tf_r specifies the stack shape and memory typing expected before and after the call. When typing the call instruction, we need to ensure that the stack shape and memory typing before the call are subtypes of the stack shape and memory typing expected by the function being called, and that the stack shape and memory typing when the function returns are subtypes of the stack shape and memory typing expected after the call by the callee.

Refinement Type System

$$
\begin{array}{lll}
\textit{Symbolic Number} & n & \\
\textit{Symbolic Location} & l & \\
\textit{Abstract Refinement Type} & \alpha & \\
\textit{Symbolic Refinement Type} & \hat{\tau}_r & ::= \mathsf{i32}(n) \mid \mathsf{ptr}(l, n) \mid \mathsf{num}(n) \mid \mathsf{i64}(n) \mid \mathsf{f32}(n) \mid \mathsf{f64}(n) \\
\textit{Type Tag} & \tau & ::= \mathsf{i32} \mid \mathsf{ptr} \mid \mathsf{num} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\textit{Original Value Types} & \tau_w & ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\textit{Symbolic Function Type} & tf_\alpha & ::= \alpha^* \rightarrow \alpha^* \\
\textit{Symbolic Global Type} & tg_\alpha & ::= \mathsf{mut}^? \ \alpha^* \\
\textit{Abstract Memory Typing} & \varsigma & \\
\textit{Symbolic Memory Typing} & \Sigma & ::= \cdot \mid \Sigma, (l, n) \mapsto \hat{\tau}_r
\end{array}
$$

Constraint System

$$
\begin{array}{lll}
\textit{Operations} & op & ::= unop_{iN} \mid unop_{fN} \mid cvtop \mid binop_{iN} \mid binop_{fN} \mid testop_{iN} \mid \\
 & & \quad relop_{iN} \mid relop_{fN} \\
\textit{Type Constraint} & t & ::= \alpha \doteq \hat{\tau}_r \mid \alpha <: \tau \mid \alpha \doteq \alpha \mid l \neq l \mid \alpha \doteq \bigsqcup \alpha^+ \mid \alpha \doteq \varsigma[\alpha]_{(tp\_sx^?,\ a,\ o)} \mid \alpha \doteq op((\alpha^?)^+) \\
\textit{Memory Constraint} & m & ::= \varsigma \doteq \Sigma \mid \varsigma \doteq \varsigma \mid \varsigma \doteq \bigsqcup \varsigma^+ \mid \varsigma \doteq \varsigma[\alpha \mapsto \alpha]_{(tp^?,\ a,\ o)} \\
\textit{Constraint Set} & S & ::= \cdot \mid S, t \mid S, m
\end{array}
$$

Fig. 5. Syntax for Symbolic Refinement Types and Constraints Generated Over WebAssembly.

3.3 Discovering Refinement Types using Type and Memory Constraints

In this section, we explain how we discover refinement types for WebAssembly code. WebAssembly's type validation algorithm, described in the specification [40], performs a single forward pass over the instructions in a function body to validate that WebAssembly function. Unfortunately, a forward-only analysis isn't possible when we wish to discover refinement types. Instead, we first generate constraints over a WebAssembly function in order to later discover refinement types for each function, instruction, and variable via constraint solving. Accumulating constraints is necessary because it is often not immediately obvious if an i32 value is a pointer or a number.

Another point of interest is that pointers in WebAssembly don't often have a concrete memory address that can be statically known. Our refinement type system expects the pointer type to have a concrete address a as its value — written $\mathsf{ptr}(a)$ — and for memory typing Ψ to be a mapping from concrete addresses a to values. Note that if the address is unknown, we would have to resort to $\top$ as the address value, which reduces precision. Thus, when discovering refinement types, we introduce *symbolic* pointer types $\mathsf{ptr}(l, n)$ that have a symbolic base address l and a symbolic offset n. With symbolic pointers in hand, we define a grammar for symbolic refinement types $\hat{\tau}_r$, given in Figure 5. We also introduce symbolic memory typing to be a map from symbolic pointers $\mathsf{ptr}(l, n)$ to symbolic refinement types $\hat{\tau}_r$. We generate constraints over WebAssembly functions and solve them to get *symbolic* refinement types $\hat{\tau}_r$ for every stack slot in the value stack, and local and global variables, and *symbolic* memory typing Σ. Our call-graph analysis is over the symbolic refinement type system. We will later use the symbolic refinement types we discover for WebAssembly code e^* to guide the construction and typing of a concrete-refinement-typed WebAssembly program $e*_r$ for which we prove type safety in Section 3.4.

The symbolic refinement types and memory typing, as well as the constraints that are generated over WebAssembly instructions can be found in Figure 5. We associate every stack slot in the value stack and local and global variables, with a unique abstract refinement type variable α and every memory state with a unique abstract memory variable ς. We generate constraints over

$$C^\alpha ::= \{\text{func } tf_w{}^*,\ \text{local } \alpha^*,\ \text{global } tg_\alpha{}^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{label } (\alpha^*, \varsigma)^*,\ \text{return } (\alpha^*, \varsigma)^?\}$$

Constraint Generation for Instructions $\boxed{S; \varsigma; C^\alpha \vdash e : \alpha^* \to \alpha^*; S'; \varsigma'}$

$$\frac{\alpha \text{ fresh} \qquad S' = S :: [\alpha \doteq \tau_w(c)]}{S; \varsigma; C^\alpha \vdash \tau_w.\mathbf{const}\ c : \epsilon \to \alpha; S'; \varsigma}\ \textsc{Constant}$$

$$\frac{\begin{array}{c}\alpha_1\ \alpha_2 \in dom(S) \qquad \alpha_3 \text{ fresh} \\ S' = S :: [\alpha_1 \doteq binop(\_, \alpha_2, \alpha_3) \wedge \alpha_2 \doteq binop(\alpha_1, \_, \alpha_3) \wedge \alpha_3 \doteq binop(\alpha_1, \alpha_2, \_)]\end{array}}{S; \varsigma; C^\alpha \vdash \tau_w.binop : \alpha_1\ \alpha_2 \to \alpha_3; S'; \varsigma}\ \textsc{Binary Ops}$$

$$\frac{\begin{array}{c}tf_w = \tau_w^n \to \tau_w^m \qquad \alpha^n \in dom(S) \qquad \alpha^m, \varsigma' \text{ fresh} \\ S; \varsigma; C, \text{label}(\alpha^m, \varsigma') \vdash e^* : \alpha^n \to (\alpha')^m; S'; \varsigma'' \\ S'' = S' :: [(\alpha \doteq \alpha')^m \wedge \varsigma' \doteq \varsigma'']\end{array}}{S; \varsigma; C^\alpha \vdash \mathbf{block}\ tf_w\ e^*\ \mathbf{end} : \alpha^n \to \alpha^m; S''; \varsigma'}\ \textsc{Block}$$

$$\frac{\begin{array}{c}\alpha^* \alpha^n \in dom(S) \qquad C^\alpha_{\text{label}}(i) = \alpha'_n, \varsigma' \\ S' = S :: [\varsigma \doteq \sqcup \varsigma \varsigma' \wedge (\alpha \doteq \sqcup \alpha \alpha')^n]\end{array}}{S; \varsigma; C^\alpha \vdash \mathbf{br}\ i : \alpha^* \alpha^n \to \alpha^*; S'; \varsigma}\ \textsc{Break}$$

$$\frac{\begin{array}{c}C^\alpha_{\text{func}} = \tau_w^m \to \tau_w^n \qquad \alpha^n \in dom(S) \\ S' = S :: [(\alpha <: \tau_w)^m \wedge (\alpha' <: \tau_w)^n]\end{array}}{S; \varsigma; C^\alpha \vdash \mathbf{call} : \alpha^m \to (\alpha')^n; S'; \varsigma}\ \textsc{Call}$$

$$\frac{\begin{array}{c}\alpha_1 \in dom(S) \\ S' = S :: [\alpha_1 <: \text{ptr} \wedge \alpha_2 \doteq \varsigma[\alpha_1]_{(tp\_sx^?,\ a,\ o)} \wedge \alpha_2 <: \tau_w]\end{array}}{S; \varsigma; C^\alpha \vdash \tau_w.\mathbf{load}(tp\_sx)^?\ a\ o : \alpha_1 \to \alpha_2; S'; \varsigma}\ \textsc{Load}$$

Fig. 6. Constraint-Generation Rules for a Subset of WebAssembly Instructions.

WAFFLE [8], an SSA IR over WebAssembly. For each instruction, we generate *type constraints* over αs and *memory constraints* over ςs, the syntax of which can be seen in Figure 5. A subset of the constraint-generation rules for WebAssembly instructions are described in Figure 6, with all the rules in Appendix D. We discuss the interesting constraint-generation cases below.

3.3.1 Arithmetic Instructions. Let us consider the example of i32.add, which expects two values on the stack, of types α_1 and α_2 and produces a single value of type α_3, i.e., $\alpha_3 = \alpha_1 + \alpha_2$. We know that the only disallowed case for this operation is the addition of two pointers. This has interesting implications for constraint solving, since if $\alpha_1 = \text{ptr}(l, n_1)$ and $\alpha_2 = \text{i32}(n_2)$, α_2 can be refined to be $\text{num}(n_2)$, since the alternative is not permitted. In fact, the types of each argument and result can affect each other. If $\alpha_1 = \text{num}(n_1)$, $\alpha_2 = \text{i32}(n_2)$ and $\alpha_3 = \text{ptr}(l, n_3)$, α_2 is refined to be ptr. Meanwhile, if $\alpha_3 = \text{num}(n_3)$, α_2 is refined to num. We leave discovering the appropriate refinements to constraint solving and at the time of constraint generation, generate constraints to tie each α to each other α, with the relevant arithmetic instruction. For i32.add, we generate the constraints, $\alpha_1 \doteq \text{i32.add}(\_, \alpha_2, \alpha_3)$, $\alpha_2 \doteq \text{i32.add}(\alpha_1, \_, \alpha_3)$ and $\alpha_3 \doteq \text{i32.add}(\alpha_1, \alpha_2, \_)$. The underscore in the constraint holds the place of the α current being refined. A similar strategy is followed for all arithmetic instructions.

3.3.2 Memory Instructions. The i64.store instruction expects a pointer type α_{ptr} on the stack and some data type α_{data}. α_{ptr} is constrained to be a pointer by generating a subtyping constraint $\alpha_{\text{ptr}} <: \text{ptr}$, while the data is constrained to be a subtype of the type expected by the store instruction, $\alpha_{\text{data}} <: \text{i64}$. The abstract memory typing represented by ς, an input to the rule, is updated to record a mapping from $\alpha_{\text{ptr}} \mapsto \alpha_{\text{data}}$, and is copied into a new ς', which is returned by the rule. The constraint on the abstract memory typing is $\varsigma' = \varsigma[\alpha_{\text{ptr}} \mapsto \alpha_{\text{data}}]$. The i64.load instruction is similarly constrained to expect a pointer on the stack and a subtype of i64 as the result. Here, the load constraint is represented as $\alpha_{\text{data}} \doteq \varsigma[\alpha_{\text{ptr}}]$. Similar constraints are generated for all τ_w.load and τ_w.store instructions.

3.3.3 Blocks, Loops. Let us consider the sequence of instructions, block tf_w e^* end. Let the block expect n values on the stack and return m values. m α_1's are freshly generated and added to the context with a label. This is done so that break instructions out of blocks know the number of values required on the output stack of the target block. We generate constraints for the body of the block, with this new context, which gives us m α_2s. The m α_1 and α_2s are now equated with equality constraints, and m α_1s are returned by the instruction on the result stack. A similar scenario arises for loop instructions. For loops, the n input αs are added to the context instead of the m result αs that are added to the context for blocks. This is because break instructions in loops restart the loop and have to know the number of values required on the input stack of the loop.

3.3.4 Break Instructions. br instructions are used to break out of blocks and restart loops. In the case of blocks, the values on the stack at the br instruction are returned by the target block. Since our analysis is flow-insensitive, we join all possible result stacks for a block. The context carries the expected value stack at the target block, so we generate join constraints for the αs on the stack and the αs expected by the target block. In the case of loops, the values on the stack at the br instruction are used to restart the loop. Constraint generation is the same regardless of whetherthe target of a br instruction is a loop or a block. Note that generating join constraints in this fashion embeds the looping structure of the program into the constraints. For example, consider the following sequence of instructions: loop tf_w e^* end. If $tf_w = \text{i32} \rightarrow ()$, we would generate constraints for the loop inputs as $\alpha_0 <: \text{i32}$. We would then generate constraints for the body of the loop with $\text{label}(\alpha_0)$ added to the context. If we came across the instruction br 0, in the loop body, with α_1 on the stack, we would generate the constraint $\alpha_0 \doteq \bigsqcup \alpha_0\alpha_1$.

3.4 Refining WebAssembly and Type Safety for the Refinement Type System

In this section, we show two results. First, we explain how, given a WebAssembly program, potentially in a JavaScript context, we can discover symbolic refinement types and use them to arrive at a concrete-refinement-typed WebAssembly program. Second, we prove type safety for our refinement type system using progress and preservation.

From WebAssembly to Refinement-Typed WebAssembly. We've seen how to generate constraints over a given WebAssembly function to discover *symbolic* refinement types $\hat{\tau}_r$. However, our typing rules are over concrete refinement types τ_r and to show that a WebAssembly program is refinement-typed we need to use the symbolic refinement types and symbolic memory typing we have discovered to guide the annotation of the WebAssembly program with refinement types so that type-checks using the refinement type system. In order to obtain the concrete refinement types for a module, we first generate constraints for a given function. During constraint generation, we map each expression at every point of the program to the αs and ςs it generates on the stack. We elide this in Figure 6 for conciseness. Section 5 describes how we solve constraints over αs and ςs to get a least solution ρ that maps αs to symbolic refinement types $\hat{\tau}_r$ and ςs to symbolic memory typing Σ. However, these refinement types are still symbolic. After constraint solving, we posit that there exists a mapping η from symbolic addresses to concrete addresses for every Σ, where, for a Σ_{pre} before and Σ_{post} after an instruction, $\eta_{\text{post}} \supseteq \eta_{\text{pre}}$. Hence, for every α and ς, we recover a τ_r and Ψ. Since Ψ is only typed with 32-bit types, we split each 64-bit sized type $\text{i64}(n)$ or $\text{f64}(n)$, into two i32 values from the high and low 32 bits of n. We now make several transformations to the WebAssembly function, in order to type check it:

- Instructions tagged with a WebAssembly type annotation τ_w are transformed to have a refined type tag τ. E.g. i32.const 42 $\rightsquigarrow$ num.const 42, where the latter expects a num on the stack rather than a i32.

- Instructions annotated with WebAssembly function type annotations tf_w are instead annotated with concrete refinement function types tf_r that include Ψ_{pre} and Ψ_{post} memory typing before and after the instruction.
- Function types are also transformed to have concrete refinement function types tf_r.
- The call instruction is typed to have a tf_r annotation (shown in Figure 4) to denote the shape of the stack before and after the call.

Type Safety for the Refinement Type System. The operational semantics of this transformed WebAssembly program $e_r{}^*$ remains largely unchanged from the original WebAssembly operational semantics, but there are a few changes. For example, for the binary operation rule, the type annotation on the $\tau.binop$ instruction no longer tells us the types expected on the WebAssembly stack. Instead of, $(\tau_w.\text{const } c_1)(\tau_w.\text{const } c_2)\ \tau_w.binop \hookrightarrow \tau_w.\text{const}(\text{binop }(c_1, c_2))$, in the refinement type system, the operational semantics of binop steps as follows, $(\tau_1.\text{const } c_1)\ (\tau_2.\text{const } c_2)\ \tau_3.binop \hookrightarrow binop\ ((\tau_1.\text{const } c_1), (\tau_2.\text{const } c_2))$.

We now prove type safety as the standard progress and preservation theorems.

Theorem 1. ***Progress**: If $R; C^r \vdash_i e_r{}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^* \wedge (v : \tau_r)^m \wedge \vdash_i s : R$ then $e_r{}^* = v^*$ or $e_r{}^* = \text{trap}$ or $s; v_r{}^*; e_r{}^* \hookrightarrow s'; v'^*; e_r'^*$*

Theorem 2. ***Preservation**: If $R; C^r \vdash_i e_r{}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^* \wedge \vdash_i (v : \tau_r)^m \wedge \vdash_i s : R \wedge s, v^*, e_r{}^* \hookrightarrow s', v'^*, e_r'^*$ then $\exists\, R'$ such that $dom(R_\Psi) \subseteq dom(R'_\Psi) \wedge \vdash_i s' : R' \wedge R'; C^r \vdash_i e_r'^* : \tau_r^*$*

Proofs of the above theorems are very similar to the WebAssembly progress and preservation proofs except for minor changes to account for the refined types. The main interesting aspect of our refinement type system is that, unlike WebAssembly, we also type the linear memory, keeping track of the memory typing Ψ in the context R. We present the proof of progress and preservation for the load instruction in Appendix B.

4 GENERATING CONSTRAINTS OVER THE JAVASCRIPT WRAPPER

We've seen how to recover refinement types over WebAssembly, but we can also type the parameters passed into WebAssembly exported function calls and the results of these calls to have refinement types. We can analyze the glue code before and after a call to an exported WebAssembly function to type exported functions. Specifically, we want to type arguments as pointers when a JavaScript object is *lowered* to WebAssembly by fetching a pointer from malloc and writing the object's contents to memory at that pointer. We also want to be able to type the data being put into the WebAssembly memory. Similarly, we want to type results as pointers when a WebAssembly value is *lifted* to JavaScript by reading data out of WebAssembly memory, into a JavaScript object. We also want to type the data that is expected to be at that WebAssembly memory location.

To this end, we write a CodeQL analysis to infer refinement types for exported WebAssembly function calls. CodeQL [1] is a declarative semantic code analysis framework maintained by GitHub, in which analysis writers can write datalog-like queries for languages like JavaScript, Python, etc. It is particularly helpful in analyzing code to check for specific patterns. The CodeQL analysis consists of two parts: an analysis to identify calls to the exported `malloc` function and accesses to WebAssembly memory, and a type inference engine that infers a type for a given JavaScript expression. The former is a series of rules that check for a sequence of patterns through data-flow and the latter is a type-inference engine. We rely on type annotations provided by the wrapper file, annotated TypeScript library functions and typing through dataflow expressions to infer a type. For example, to infer the type for a if-then-else expression, we type the then and else branches. We also infer the type of an expression, by trying to infer the type of its parent or, more generally,

Application	Size (KB)	Functions	Exported Functions	Imported Functions	Direct Calls	Indirect Calls	Functions in Table
webcam	28.61	125	6	33	270	79	58
leaflet-rs	20.16	110	8	45	228	39	39
wasm-rsa	369.00	785	25	5	6058	107	237
blake3	34.41	81	14	1	206	26	21
epqueue	23.26	123	17	9	471	25	29
serde	13.61	44	0	1	67	25	31
snowpack-ts	35.11	107	4	37	236	55	35
warp	162.09	283	10	14	1856	112	141
wasm-pack	13.78	43	1	6	69	19	17
source-map	48.53	68	10	1	395	1	18

Table 1. Description of Subject Applications

its predecessor (an expression whose data flows into the current expression) or successor (an expression into which the current expression's data flows).

The inferred types for each argument and result of exported WebAssembly functions are passed into WasmBridge which generates constraints from them, as shown in Figure 2. Generating constraints from the types is fairly straightforward. If an argument or result is typed to be a pointer with that memory block containing a i64, we generate the subtype constraints $\alpha_1 \doteq \mathrm{ptr}(l_1, 0)$, $\alpha_2 <: \mathrm{i64}$ and the memory constraint $\varsigma_1 \doteq \varsigma_0[\alpha_1 \mapsto \alpha_2]$. Note that since we do not analyze the JavaScript client, we do not have any data to associate with these types. If a function has two arguments that have a pointer type and we assume that a call to malloc gives us fresh, disjoint labels, we can generate a constraint that encodes that the two pointers' symbolic base addresses are not equal to each other. For example, if func \$foo (α_1, α_2) where $\alpha_1 \doteq \mathrm{ptr}(l_1, 0)$ and $\alpha_2 \doteq \mathrm{ptr}(l_2, 0)$, we generate the constraint $l_1 \neq l_2$ since our analysis over the JavaScript wrapper tells us that α_1 and α_2 came from two separate malloc calls. We solve these constraints along with the constraints generated over the associated WebAssembly binary to obtain refinement types for each WebAssembly function, stack slot and local variable.

5 CONSTRAINT SOLVING

We solve a set of constraints generated over a WebAssembly function and its associated JavaScript glue code using a standard fixpoint algorithm, described in detail in Appendix F. At the end of constraint generation, we have a set S of constraints over a function body. We create a mapping *Constraint*, that maps each α in S to the type constraints on α and similarly, maps each ς in S to the memory constraints on ς. We resolve the equality constraints for α's and ς's when creating this map. We also record dependencies between α's and ς's using the influence vector *infl*. This is used to recompute the constraints of a certain α or ς when its dependencies have changed. We pass the *Constraint* and *infl* maps as inputs to constraint solving and compute a least solution for every α and ς in S. The constraints on α's are solved to get a refinement type, τ_r, and the constraints on ς's are solved to get a memory state, Σ. We obtain the least solution using a standard worklist algorithm. If there exists a $\alpha \doteq \tau_r$ in the constraints for α, we initialize the solution for the α to be τ_r. Otherwise, we initialize the solution to be the $\top$ type. We initialize the solution of all ς's to be an empty memory state. We initialize the worklist with all α's and ς's in the domain of the *Constraint* map, and continue solving till the worklist is empty. If the solution of an α or ς is found to be different from its solution in the previous iteration, we add all α's and ς's that depend on it into the worklist. We briefly discuss solving type and memory constraints below.

	WebAssembly			Refined Type System						WebAssembly + JavaScript					
Applications	$\lvert E\rvert$	$\lvert U\rvert$	$\lvert M\rvert$	$\lvert E\rvert$	$\lvert U\rvert$	$\lvert M\rvert$	**% $\lvert E\rvert$ Red.**	**% Call Sites w/ Red.**	**Avg Red.**	$\lvert E\rvert$	$\lvert U\rvert$	$\lvert M\rvert$	**% $\lvert E\rvert$ Red.**	**% Call Sites w/ Red.**	**Avg Red.**
webcam	526	7	4	499	7	5	**9.28**	**50.00**	**16.64**	499	7	5	**9.28**	50.00	16.64
leaflet-rs	264	9	3	229	11	6	**17.78**	**62.86**	**36.74**	229	11	6	**17.78**	62.86	36.74
wasm-rsa	3896	8	2	3696	9	2	**6.15**	**35.35**	**12.41**	3696	9	2	**6.15**	35.35	12.41
blake3	202	5	13	195	16	14	**5.03**	**19.23**	**13.00**	195	16	14	**5.03**	19.23	13.00
epqueue	287	14	3	284	14	5	**1.51**	**15.00**	**9.50**	284	14	5	**1.51**	15.00	9.50
serde	96	43	3	90	43	5	**12.50**	**45.00**	**23.12**	90	43	5	**12.50**	45.00	23.12
snowpack-ts	352	45	4	338	45	21	**7.37**	**51.92**	**30.88**	338	45	21	**7.37**	51.92	30.88
warp	1832	9	3	1707	11	3	**8.73**	**34.86**	**13.04**	1707	11	3	**8.73**	34.86	13.04
wasm-pack	88	35	9	83	35	9	**11.11**	**42.11**	**14.04**	83	35	9	**11.11**	42.11	14.04
source-map	126	7	0	126	7	0	**0.00**	**0.00**	**0.00**	126	7	0	**0.00**	0.00	0.00
Averages	766.9	18.2	4.4	724.7	19.8	7	7.946	35.633	16.937	724.7	19.8	7	7.946	35.633	16.937

Table 2. Comparing callgraphs generated over the original WebAssembly type system, the refinement type system, and the combined WebAssembly and JavaScript analysis. $|E|$ denotes the number of edges in the callgraph, $|M|$ denotes the number of monomorphic calls, and $|U|$ denotes the number of unreachable nodes. Avg Red. is average reduction at call sites.

5.1 Solving Type Constraints

To solve the type constraints for a given α we iterate over each type constraint and do a meet operation between the currently computed type for the α and the type computed with a specific constraint. The solution of most type constraints is standard: we simply evaluate the arithmetic operations or perform joins as dictated by the join constraints. However, the memory load constraint is more complicated in the face of possible aliasing between pointers. Let us suppose that we are trying to load from a memory state, Σ, at $\mathrm{ptr}(l, n)$. If Σ contains $\mathrm{ptr}(\top, \top)$, $\mathrm{ptr}(l, \top)$ or $\mathrm{ptr}(l', n')$ where there does not exist a $l \neq l'$ constraint, we make the conservative assumption that all these pointers might alias with $\mathrm{ptr}(l, n)$ and join the data pointed to by these potential aliased pointers with τ_r where $\Sigma[(l, n) \mapsto \tau_r]$.

5.2 Solving Memory Constraints

Solving the memory constraints for a given ς is similar to solving type constraints. Here too, when evaluating a memory store constraint $\Sigma[\mathrm{ptr}(l, n) \mapsto \tau_r]$, we have to be conservative in the face of aliasing and join the data at a potential alias site, i.e., $\mathrm{ptr}(\top, \top)$, $\mathrm{ptr}(l, \top)$ or $\mathrm{ptr}(l', n')$, with the type currently being stored in the memory state, τ_r.

6 EVALUATION ON REAL-WORLD SUBJECT APPLICATIONS

We evaluate a call graph analysis over the refinement type system on ten real-world WebAssembly/-JavaScript applications that have been compiled from Rust. Our approach only supports generating constraints over WebAssembly 1.0, the core WebAssembly language introduced in 2017, and only supports an analysis of JavaScript wrappers generated by the Rust compiler. We discuss extending our analysis to include other compilers and extensions to WebAssembly in Section 8. We pick subject applications from two datasets [28] [38] and identify 29 NPM packages that have been compiled from Rust. Of these, we find 10 libraries which satisfy our criteria of only using WebAssembly 1.0. Table 1 shows the characteristics of the subject applications, including the size of the binary and the number of direct and indirect call sites in the binary. We run WasmBridge over the chosen subject applications in order to evaluate the precision of the refinement types, the analysis over JavaScript and how WasmBridge compares to other state-of-the-art call graph analysis tools.

Applications	WebAssembly Type System Time (s)	**Refined Type System Time (s)**	Wasm+JS Analysis CodeQL DB Creation	CodeQL Query Execution	Constraint Solving	**Total Time (s)**
webcam	0.005	**0.053**	4.80	8.33	0.06	**13.20**
leaflet-rs	0.005	**0.024**	5.36	7.81	0.03	**13.19**
wasm-rsa	0.032	**1.330**	5.14	7.78	1.44	**14.36**
blake3	0.005	**0.045**	4.93	7.68	0.05	**12.66**
epqueue	0.007	**0.042**	4.29	7.82	0.08	**12.18**
serde	0.004	**0.027**	4.82	8.49	0.03	**13.34**
snowpack-ts	0.007	**0.052**	5.58	7.38	0.06	**13.01**
warp	0.023	**1.184**	4.54	7.97	1.32	**13.83**
wasm-pack	0.004	**0.025**	4.46	6.90	0.03	**11.39**
source-map	0.010	**0.189**	5.04	7.28	0.20	**12.52**

Table 3. Runtime performance comparison (in seconds) for analysis over the original WebAssembly type system, the refinement type system, and the combined Wasm+JS analysis for each subject application.

RQ1: How much more precise is the call graph over the refinement type system when compared to the original WebAssembly type system?

In order to determine the increase in precision afforded by the refinement type system, we compare the same type-based call graph analysis on the refinement type system and the original WebAssembly type system. A type-based call graph analysis restricts the set of targets at an indirect call sites with the type annotation at the call_indirect instruction as described in Section 2.2. The results of this evaluation are shown in the first two rows of Table 2. The call graph over the refinement type system is more precise than the call graph over the original WebAssembly type system:

(1) *Edge Reduction*: We reduce the total number of call graph edges by up to 17.50% (leaflet-rs) compared to the baseline WebAssembly analysis. This reduction occurs at indirect call sites where precision matters most; we observe a reduction at 62.9% of indirect call sites.
(2) *Call Site Level Improvements*: The precision gains are more apparent at indirect call sites. For all indirect call sites, we achieve an average edge reduction ranging from 9.5% to 36.74% per call site and an average reduction of 16.9%.
(3) *Monomorphic Call site Discovery*: Our refinement type system enables us to identify several additional monomorphic call sites—indirect calls that resolve to exactly one target. In snowpack-ts, we discover 17 additional monomorphic call sites (from 4 to 21 call sites); leaflet-rs has 3 additional monomorphic call sites (from 3 to 6); webcam-stream has 1 additional monomorphic call sites (from 4 to 5); epqueue has 2 additional monomorphic call sites (from 3 to 5); serde-bindgen has 2 additional monomorphic call sites (from 3 to 5) and blake3 shows 1 additional monomorphic call site (from 13 to 14). These discoveries enable subsequent compiler optimizations and improve the precision of downstream analyses.
(4) *Unreachable Function Detection*: The refinement type information also improves our ability to identify unreachable functions. Blake3 shows the most dramatic improvement, with unreachable nodes increasing from 5 to 16—indicating that there are functions in the WebAssembly binary that are never called in practice.

Notably, source-map shows no improvement in our analysis because it is a highly optimized binary with only one indirect call. This result is expected in the case of call graph analysis—applications with simple control flow gain little from the refinement type analysis. However, the richer types might improve precision in other analyses.

	WasmBridge			Wasm-Opt				Wasma				Wassail			
Applications	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert U\rvert$	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert U\rvert$	**Edge Diff**	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert U\rvert$	**Edge Diff**	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert U\rvert$	**Edge Diff**
webcam	125	499	7	125	1903	0	**73.78%**	125	526	7	**5.13%**	125	526	7	**5.13%**
leaflet-rs	110	229	11	110	845	0	**72.90%**	110	264	9	**13.26%**	110	264	9	**13.26%**
wasm-rsa	785	3696	9	785	9501	0	**61.10%**	785	3896	8	**5.13%**	785	3896	8	**5.13%**
blake3	81	195	16	81	269	0	**27.51%**	78	202	2	**3.47%**	81	202	5	**3.47%**
epqueue	123	284	14	123	602	0	**52.82%**	119	287	10	**1.05%**	123	287	14	**1.05%**
serde	44	90	43	44	405	43	**77.78%**	-	-	-	-	44	96	43	**6.25%**
snowpack-ts	107	338	45	107	846	0	**60.05%**	107	352	45	**3.98%**	107	352	45	**3.98%**
warp	283	1707	11	283	5537	0	**69.15%**	280	1832	6	**6.82%**	283	1832	9	**6.82%**
wasm-pack	43	83	35	43	153	35	**45.75%**	43	88	35	**5.68%**	43	88	35	**5.68%**
source-map	68	126	7	68	133	0	**5.26%**	62	126	1	**0.00%**	68	126	7	**0.00%**
Average	**176.90**	**724.70**	**19.80**	176.90	2019.40	7.80	54.61%	189.89	841.44	13.67	4.94%	176.90	766.90	18.2	5.08%

Table 4. Call Graph Comparison across different WebAssembly analysis tools. Edge difference shows the % change in callgraph edges compared to our WasmBridge analysis. Negative values indicate the tool produces fewer edges than WasmBridge. $|N|$ denotes the number of nodes (functions) in the callgraph, $|E|$ denotes the number of edges (calls), and $|U|$ denotes the number of unreachable nodes in the callgraph.

RQ2: How much more precise is the call graph over WebAssembly and JavaScript when compared to WebAssembly alone?

We compare call graphs generated over the refinement type system with constraints generated over WebAssembly alone versus constraints generated over WebAssembly and its associated JavaScript wrapper in the second and third row of Table 2. We find no change in the precision at indirect call sites and no reduction in the number of edges. This is not too surprising since the refinements to the type system are not discovered through the JavaScript wrapper alone. WebAssembly instructions also provide these refinements, such as the load instruction which expects a ptr on the stack rather than just an i32. The one constraint that is unique to the JavaScript wrapper is one that specifies the disjointedness of symbolic base addresses of pointers that have come from different calls to malloc ($l_1 \neq l_2$). It seems as though the imprecision of pointer analysis leads to this constraint not improving precision at call sites.

Note that analyzing the wrapper cannot tell us the JavaScript values that are being passed to WebAssembly exported functions. We would need a sophisticated JavaScript client analysis that performs a value and call graph analysis over the entire JavaScript application and all its dependencies, in order to see more precision improvements. No such tool exists off-the-shelf and we leave it to future work. However, the CodeQL analysis of the JavaScript wrapper makes our analysis extensible to such a client analysis. Other state-of-the-art tools cannot support this.

RQ3: What is the cost of the analysis over WebAssembly and JavaScript?

Table 3 shows the runtime overhead of each component of our multi-language analysis and how it compares to the analysis over constraints generated from WebAssembly alone versus the call graph analysis over the original WebAssembly type system. The latter is our baseline. The call graph analysis over the refinement type system over WebAssembly alone shows a maximum of a 1.33 second increase over the baseline. The multi-language analysis, however, is dominated by the time it takes to create a CodeQL database and run the analysis (or query) over the database. Even so, the total time of the multi-language analysis does not exceed 14.36s in the worst case and constraint generation and solving is relatively inexpensive.

Metric		blake3			wasm-rsa	
		Baseline	Client 1	Client 2	Baseline	Client 1
Binary Before DCE	Functions	81	81	81	785	785
	Edges	269	269	269	9501	9501
Wasm-Metadce	Functions	81	63	65	785	605
	Edges	269	221	228	9501	8840
	Reduction%	**0.0**	**22.22**	**19.75**	**0.00**	**22.93**
WasmBridge	Functions	65	47	49	776	596
	Edges	151	103	110	3652	2991
	Reduction%	**19.75**	**41.98**	**39.51**	**1.15**	**24.08**

Table 5. Comparison of Dead Code Elimination performed by Wasm-Metadce and WasmBridge. Reduction% shows the reduction in the number of functions after dead-code elimination.

RQ4: How does our approach compare to other state-of-the-art call graph analysis tools?

We compare our approach with other state-of-the-art call graph analysis tools, Wasm-Opt[41], Wassail[37] and Wasma[5]. Several tools were excluded from our evaluation: Wassilly [31] timed out on each subject application while generating a call graph, Sturdy [22] does not support analysis over binaries that interoperate with JavaScript (the framework throws an error), and Wasmati [6] does not support WebAssembly functions that return more than one result (multi-value proposal). Table 4 compares the different analysis tools and we discuss their call graph analyses below.

- *Wasm-Opt* performs a naive call graph analysis. At every indirect call site, it assumes that all the functions in the function table may be called. This results in a call graph with up to 3.8× more edges than our approach (e.g., 1,903 edges vs. 504 for webcam-stream). While this is the most sound analysis possible, the resulting call graph is very imprecise.
- *Wasma* performs a closed-world type-based call graph analysis. At every indirect call site, it considers all the functions in the table with the same function type as the type annotation at the call site to be a potential target of this call site. As expected from the results of RQ1 and 2, we see a reduction in the number of edges compared to Wasma. Note that Wasma removes functions that do not have any incoming or outgoing edges, which leads to a reduction in the number of nodes for subjects like epqueue. Our analysis does not remove these functions. Additionally, Wasma fails to run on one subject.
- *Wassail* performs a closed-world type-based call graph analysis like Wasma but like us, does not remove functions that do not have any incoming or outgoing edges.

RQ5: What is the effect of the increased precision of our approach on a downstream analysis like dead-code elimination?

To demonstrate the potential impact of our analysis technique on downstream analyses and how the addition of a JavaScript client analysis performs, we perform dead-code elimination to specialize a Wasm binary to a JavaScript client. For two subject applications, blake3 and wasm-rsa, the dataset we obtain them from has mock JavaScript clients for these subjects. The JavaScript clients call a subset of exported functions from the JavaScript wrapper of the WebAssembly binary. Through manual inspection, we determine the set of WebAssembly exported functions that are called by the mock client. We pass this information to Wasm-Metadce, an industry state-of-the-art WebAssembly dead-code elimination tool, which, given a WebAssembly binary and a list of exported functions that are called by a client, performs dead-code elimination and produces a smaller binary that has been specialized to that client. We also use the list of exported functions called by the client to perform dead-code elimination based on our call graph analysis. We then compare the number of functions that have been removed by Wasm-Metadce versus WasmBridge in Table 5. As a baseline, we pass in all the exported functions to both tools, to measure dead code elimination for a client that uses all exported functions.

- *Function Reduction*: WasmBridge identifies more functions to be dead than Wasm-Metadce, across both subjects and their clients. For blake3, we achieve function reductions of 19.75%, 41.98%, and 39.51% compared to metadce's 0%, 22.22%, and 19.75% respectively. The reduction difference is not as high in wasm-rsa, but we still outperform Wasm-Metadce.
- *Edge Reduction*: WasmBridge continues to produce more precise call graphs with fewer edges, which can enable further optimizations and analyses.
- *Baseline*: Even in the baseline case, where a mock client uses all exported functions of a binary, our analysis identifies dead code that Wasm-Metadce misses entirely (19.75% reduction vs. 0% for blake3).

7 RELATED WORK

Inferring Richer Types for Binary Analysis. Lehmann et al. [27] present SnowWhite, a learning-based approach that recovers high-level function types for WebAssembly functions using DWARF information in binaries. Zhao et al. [43] present WasmHint, which infers 'Rust-like' function types for WebAssembly functions using semantic learning in combination with program slicing. We do not compare against these approaches since they infer much more complicated types like structs and arrays and don't provide an estimation of the soundness of their type systems or how the inferred types affect program analysis. However, both systems do infer a ptr type.

There has also been interest in recovering high-level type information for x86 binaries. Lee et al. [25] recover high-level types for x86 binaries by inferring types based on how code operates on data in the binary, similar to our inference algorithm over WebAssembly, while Bosamiya et al. [4] use a deductive type reconstruction strategy. ElWazeer et al. [12] use a "best-effort" flow and context-insensitive pointer analysis to recover type and variable information from x86 binaries.

Multi-Language Static Analysis. We now summarize alternate approaches to multi-language static analysis for other language pairs. Lee et al. [26] analyze JNI interoperation between Java and C by generating semantic summaries for C functions called by Java code. These summaries are translated into Java and calls to C functions are replaced with their Java summaries. Liang et al. [29] use LLVM IR as a common intermediate from multiple languages and detect bugs using symbolic execution coupled with Z3. Fornaia et al. [14] translates C, C++ and Java code into LLVM. Roth[34] reuses existing single-language analyses and combines them to support multiple languages by developing architecture that can be used to combine analyses. Monat et al. [30] implement an extensive C-Python abstract interpretation framework using transfer functions to convert C values to Python and vice-versa. Buro et al. [7] provide a general, theoretical framework combining abstract interpretations of different languages. Furr and Foster [17] also perform multilingual type inference for C code using client OCaml and Java types for checking the type-safety of FFI calls.

WebAssembly Static Program Analysis. There are several static analysis frameworks for WebAssembly. We compare against Wassail[36], Wasma[5], Wasm-Opt[41] and Wasm-Metadce[41]. The latter two are part of the Binaryen toolchain. Wassilly [31] and Sturdy [22] improve the precision of indirect calls by relying on a better value analysis to determine the index into the function table. They are both abstract interpretation frameworks. We do not compare against them because the former times out on all our subjects and the latter does not support analysis of binaries that interoperate with JavaScript. Wasmati [6] uses code property graphs to statically detect vulnerabilities in WebAssembly binaries and use a type-based call graph analysis.

Refinement Type Systems. Refinement types have long been used to enhance type systems with logical predicates that constrain the set of values described by a type [15, 16, 32, 39, 42], allowing programmers to express more precise program properties and enabling program verification. The most closely related work is WasmPrecheck [18], a richer type system for WebAssembly that uses *indexed types* to express static constraints that enable safe removal of dynamic checks for type and memory safety. WasmPrecheck supports general constraints on index terms, while we support only singleton refinements and refining `i32` to $\texttt{ptr}(a)$ or $\texttt{num}(n)$. Another point of comparison is that both WasmPrecheck and our work shows how to embed a Wasm module into a refined type system. However, Geller et al. give only a naive embedding — they show that a Wasm module can be (automatically) embedded into WashPrecheck by replacing all type annotations in the Wasm module with indexed types that have no constraints. This does not enhance the performance of the program, while our analysis considers information from interoperation with JavaScript to discover refinement types that in turn lead to improved static analysis.

8 CONCLUSION AND FUTURE WORK

We have presented the first multi-language static analysis for WebAssembly and JavaScript. Our work shows how a small refinement to WebAssembly's type system, supported by value and pointer analysis, improves the precision of a call graph analysis by reducing the number of estimated call targets by an average of 17% at an average of 35.63% of indirect call sites. We also estimate the effect of this improved precision on downstream dead-code elimination analyses and outperform an industry state-of-the-art tool. However, our analysis only supports WebAssembly version 1.0 and JavaScript wrapper files generated by the Rust compiler. We plan to extend our analysis to support other versions of WebAssembly as well as binaries and wrappers produced by other toolchains.

Supporting WebAssembly 2.0 and 3.0. WebAssembly's first verson update introduced vector instructions, bulk memory instructions, multi-value results, non-trapping conversions and sign-extension instructions. We already support multi-value results and do not anticipate trouble when including these new instructions. WebAssembly 3.0 introduced Garbage Collection (GC), which has aggregate types like structures and arrays and linear memory is now 32- and 64-bit addressable. To support the GC proposal, we would first refine `i64` types to have a similar refinement to `i32` and extend the type system to include aggregate types. Since our refinement introduces subtype constraints between pointers and integers, we would still see precision improvements.

JavaScript Wrapper Analysis. Currently, our analysis only supports a CodeQL analysis of JavaScript wrappers generated by `wasm-bindgen`. However, this is not the only option to support an FFI between WebAssembly and JavaScript. C/C++ library developers typically use Emscripten to compile code to WebAssembly and JavaScript. While Emscripten provides several interoperation mechanisms between the two languages [13], `Embind` is the most popular mechanism and employs the same lowering and lifting as the wrappers produced by `wasm-bindgen` do.

Improvements to Precision. Our analysis discovers refinement types that are flow-insensitive, context-insensitive and intra-procedural. Additionally, the domain of values is fairly simple —a flat lattice of natural numbers. Considering the relative simplicity of our analysis we plan on employing several traditional techniques to increase precision of the analysis. We also plan to extend our CodeQL analysis to the JavaScript client to determine the values being passed to WebAssembly or being stored into the WebAssembly linear memory and the subset of exported functions being used by the client. The former would improve precision and the latter would enable a dead-code elimination analysis that specializes a WebAssembly binary to a JavaScript client.

REFERENCES

[1] 2023. *CodeQL: A Semantic Code Analysis Engine*. https://codeql.github.com/

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3359789.3359823

[3] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. 2020. *Binary code fingerprinting for cybersecurity: Application to malicious code fingerprinting*. Springer.

[4] Jay Bosamiya, Maverick Woo, and Bryan Parno. 2025. TRex: practical type reconstruction for binary code. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 354, 19 pages.

[5] Florian Breitfelder, Tobias Roth, Lars Baumgärtner, and Mira Mezini. 2023. WasmA: A Static WebAssembly Analysis Framework for Everyone. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 753–757. https://doi.org/10.1109/SANER56733.2023.00085

[6] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. [n. d.]. Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly. ([n. d.]), 102745. https://doi.org/10.1016/j.cose.2022.102745

[7] Samuele Buro, Roy L Crole, and Isabella Mastroeni. 2020. On multi-language abstraction: Towards a static analysis of multi-language programs. In *Static Analysis: 27th International Symposium, SAS 2020, Virtual Event, November 18–20, 2020, Proceedings 27*. Springer, 310–332.

[8] Chris Fallin. 2025. *Waffle: WebAssembly Analysis Framework for Lightweight Experimentation*. Bytecode Alliance. https://github.com/bytecodealliance/waffle

[9] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 587–606.

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35.

[11] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

[12] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 51–60.

[13] Emcc Docs Interacting with Code [n. d.]. *Emscripten Docs: Connecting C++ and JavaScript*. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html

[14] Andrea Fornaia, Stefano Scafiti, and Emiliano Tramontana. 2019. JSCAN: Designing an Easy to use LLVM-Based Static Analysis Framework. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 237–242. https://doi.org/10.1109/WETICE.2019.00058

[15] Jeremy Freeman, Catalin Hritcu, Marco Gaboardi, Vincent Laporte, Sergey Firsov, Gregory Malecha, Dustin Swasey, Conor McBride Watt, and Benjamin C. Pierce. 2020. RefinedC: Automating the Foundational Verification of C Code with Refined Types. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, Vol. 4. 1–29. https://doi.org/10.1145/3428198

[16] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

[17] Michael Furr and Jeffrey S Foster. 2005. Checking type safety of foreign function calls. *ACM SIGPLAN Notices* 40, 6 (2005), 62–72.

[18] Adam T. Geller, Justin P. Frank, and William J. Bowman. 2024. Indexed Types for a Statically Safe WebAssembly. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2395–2424. https://doi.org/10.1145/3632922

[19] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. 2006. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, Vol. 6. 93–104.

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

[21] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[22] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 176 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360602
[23] Johannes Kinder. 2010. *Static analysis of x86 executables.* Ph. D. Dissertation. Technische Universität Darmstadt.
[24] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, Vol. 13. 18–18.
[25] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).
[26] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 127–137.
[27] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA) *(PLDI '22)*. Association for Computing Machinery. https://doi.org/10.1145/3519939.3523449
[28] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597926.3598104
[29] Hongliang Liang, Lei Wang, Dongyang Wu, and Jiuyun Xu. 2016. MLSA: A static bugs analysis tool based on LLVM IR. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 407–412. https://doi.org/10.1109/SNPD.2016.7515932
[30] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A multilanguage static analysis of python programs with native C extensions. In *International Static Analysis Symposium*. Springer, 323–345.
[31] Mattia Paccamiccio, Franco Raimondi, and Michele Loreti. 2024. Building Call Graph of WebAssembly Programs via Abstract Semantics. arXiv:2407.14527 [cs.SE] https://arxiv.org/abs/2407.14527
[32] Patrick M. Rondon, Kohei Kawaguchi, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 159–169.
[33] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. *ACM Sigplan Notices* 45, 1 (2010), 131–144.
[34] Tobias Roth. 2023. Reusing Single-Language Analyses for Static Analysis of Multi-language Programs. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 16–18.
[35] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4*. Springer, 1–25.
[36] Quentin Stiévenart and Coen De Roover. 2021. Wassail: a WebAssembly Static Analysis Library *(ProWeb21)*. https://2021.programming-conference.org/home/proweb-2021 Fifth International Workshop on Programming Technology for the Future Web.
[37] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 13–24. https://doi.org/10.1109/SCAM51674.2020.00007
[38] Michelle Thalakottur, Max Bernstein, Daniel Lehmann, Michael Pradel, and Frank Tip. 2026. An Empirical Study of WebAssembly Usage in Node.js. In *48rd IEEE/ACM International Conference on Software Engineering, ICSE 2026.*
[39] Niki Vazou, Eric Seidel, and Ranjit Jhala. 2014. Refinement Types for Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 269–282.
[40] Andreas Rossberg (Ed.). 2022-03-31. *WebAssembly 1.0 Core Specification.* World Wide Web Consortium (W3C). https://www.w3.org/TR/wasm-core-1/
[41] WebAssembly Community Group. 2025. Binaryen: Optimizer and compiler/toolchain library for WebAssembly. https://github.com/WebAssembly/binaryen.
[42] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 214–227.
[43] Kunsong Zhao, Zihao Li, Weimin Chen, Xiapu Luo, Ting Chen, Guozhu Meng, and Yajin Zhou. 2025. Recasting Type Hints from WebAssembly Contracts. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE118 (June 2025), 24 pages. https://doi.org/10.1145/3729388

Appendices

A INSTRUCTION TYPING FOR THE REFINEMENT TYPE SYSTEM

$$C^r ::= \{\text{func } tf_r{}^*,\ \text{local } \tau_r{}^*,\ \text{global } \tau_r{}^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{label } (\tau_r^*, \Psi)^*,\ \text{return } (\tau_r^*, \Psi)^?\}$$

Typing WebAssembly Instructions with Refinement Types $\boxed{R; C^r \vdash e_r{}^* : tf_r}$

$$\frac{R = \{\text{inst } C^{r*}, \text{tab } n^*, \text{mem } \Psi^*\} \qquad (R \vdash inst : C^r)^* \qquad (n \leq |cl|^*)^* \qquad \vdash s.\text{mem} : R_\Psi}{\vdash \{\text{inst } inst^*, \text{tab } (cl^*)^*, \text{mem } (b^*)^*\} : R}$$

$$\frac{\forall a \in 0 \ldots \text{size}(s.\text{mem}) \qquad s.\text{mem}(a) = R_\Psi(a)}{\vdash s.\text{mem} : R_\Psi}$$

$$\frac{\begin{array}{c} \varsigma \text{ fresh} \qquad \varsigma; \cdot; C^\alpha \vdash ex^* \ \textbf{func } tf_w \ \textbf{local } \tau_w^\ell \ e^* : \alpha^m \to \alpha^n; S'; \varsigma' \\ \rho = \text{solve}(S') \\ \tau_r^m = \rho[\alpha^m] \qquad \tau_r^n = \rho[\alpha^n] \qquad \Sigma_{pre} = \rho[C] \qquad \Sigma_{post} = \rho[C'] \\ \exists \Psi_{\text{pre}}, \Psi_{\text{post}}.\ \exists \eta_{\text{pre}}, \eta_{\text{post}}. \\ \text{Bij}(\eta_{\text{pre}}) \land \text{Bij}(\eta_{\text{post}}) \land \eta_{\text{pre}}(\Sigma_{\text{pre}}) = \Psi_{\text{pre}} \land \eta_{\text{post}}(\Sigma_{\text{post}}) = \Psi_{\text{post}} \land \eta_{\text{post}} \supseteq \eta_{\text{pre}} \\ tf_r = \tau_r^m, \Psi_{\text{pre}} \to \tau_r^n, \Psi_{\text{post}} \end{array}}{C^\alpha \vdash ex^* \ \textbf{func } tf_w \ \textbf{local } \tau_w^\ell \ e^* : ex^* \ tf_r; \rho}$$

$$\frac{\begin{array}{c} \alpha_w^g \text{ fresh} \\ C^\alpha = \{\textbf{func } tf_w{}^*, \textbf{global } \alpha_w^g, \textbf{table } n^?, \textbf{memory } n^?\} \\ (C^\alpha \vdash ex^* \ \textbf{func } tf_w \ \textbf{local } \tau_w^l \ e^* : ex^* \ tf_r; \rho)^* \\ C^r = \{\textbf{func } tf_r{}^*, \textbf{global } \tau_r^g, \textbf{table } n^?, \textbf{memory } n^?\} \\ (\textbf{module } f^* \ \textbf{global } \tau_w^g \ \textbf{tab}^? \ \textbf{mem}^?) \overset{\rho}{\rightsquigarrow} (\textbf{module } f^* \ \textbf{global } \tau_r^g \ \textbf{tab}^? \ \textbf{mem}^?) \\ (R; C^r \vdash f : ex^* \ tf_r)^* \end{array}}{R \vdash \textbf{module } f^* \ \textbf{global } \tau_w^g \ \textbf{tab}^? \ \textbf{mem}^?} \textsc{Module}$$

$$\frac{\begin{array}{c} tf_r = \tau_r^m, \Psi_{\text{pre}} \to \tau_r^n, \Psi_{\text{post}} \\ R, C^r, \textbf{local}(\tau_r^m, \tau_r^\ell),\ \textbf{label}(\tau_r^n, \Psi_{\text{post}}),\ \textbf{return}(\tau_r^n, \Psi_{\text{post}}) \vdash e^* : \epsilon \to \tau_r^n \end{array}}{R; C^r \vdash ex^* \textbf{func } tf_r \ \textbf{local } \tau_r^\ell \ e^* : ex^* \ tf_r} \textsc{Function}$$

$$\frac{\begin{array}{c} tf_r = \tau_r^m, \Psi_{\text{pre}} \to \tau_r^n, \Psi_{\text{post}} \qquad R_\Psi = \Psi_{\text{pre}} \\ R; C^r, \text{label}(\tau_r^n, \Psi_{\text{post}}) \vdash e^* : \tau_r^m, \Psi_{\text{pre}} \to (\tau_r')^n, \Psi_{\text{post}}' \\ (\tau_r' <: \tau_r)^n \qquad \Psi_{\text{post}}' <: \Psi_{\text{post}} \end{array}}{R; C^r \vdash \textbf{block } tf_r \ e^* \ \textbf{end} : tf_r} \textsc{Block}$$

$$\frac{\begin{array}{c} tf_r = \tau_r^m, \Psi_{\text{pre}} \to \tau_r^n, \Psi_{\text{post}} \qquad R_\Psi = \Psi_{\text{pre}} \\ R; C^r, \text{label}(\tau_r^m, \Psi_{\text{pre}}) \vdash e^* : \tau_r^m, \Psi_{\text{pre}} \to (\tau_r')^n, \Psi_{\text{post}}' \\ (\tau_r' <: \tau_r)^n \qquad \Psi_{\text{post}}' <: \Psi_{\text{post}} \end{array}}{R; C^r \vdash \textbf{loop } tf_r \ e^* \ \textbf{end} : tf_r} \textsc{Loop}$$

Fig. 7. Typing Rules for the Refinement Type System

def load_and_extend$(a, \tau, tp_sz^?, \Psi) :=$
  let $\tau_r(n_1) = \Psi(a)$
  let $\tau_r(n_2) = \Psi(a-4)$
  if tp given then let $N = |tp|$ else let $N = |\tau|$
  if $wasm\_type(\tau_r(n_1)) == \tau$ then let $\tau_r(n) = \tau_r(n_1)$
  if $|\tau_r(n_1)| < N$ then let $\tau_r(n) = \tau(\mathrm{val}(\mathrm{bits}(n_2)\,\mathrm{bits}(n_1)))$
  if $|\tau_r(n_1)| > N$ then let $\tau_r(n) = \tau(\mathrm{val}(\mathrm{bits}(n_1)_{0..N}))$
  if $(\tau_r(n_1) = (\mathrm{i32}(n_1) \vee \mathrm{ptr}(n_1) \vee \mathrm{num}(n_1)) \wedge \tau = \mathrm{f32})$ then let $\tau_r(n) = \mathrm{f32}(bits(n_1 \text{ as } \mathrm{f32}))$
  if $(\tau_r(n_1) = \mathrm{f32}(n_1) \wedge \tau = (\mathrm{i32} \vee \mathrm{ptr} \vee \mathrm{num}))$ then let $\tau_r(n) = \mathrm{i32}(\mathrm{f32}(n_1) \text{ as } \mathrm{i32})$
  if tp given
    then return $\tau_r(\mathrm{extend\_sx}_{\mathrm{N},|\tau_w|}(n))$
    else return $\tau_r(n)$

$$\frac{C^r_{\mathrm{memory}} = n \qquad \tau_r^1 <: \mathrm{ptr}(c) \qquad \tau_r^2 = \mathrm{load\_and\_extend}(c+o, \tau, (tp\_sx)^?, R_\Psi) \qquad 2^a \le (|tp| <)^? |\tau_r^2| \qquad (tp\_sz)^? = \epsilon \vee \tau_r^2 = \mathbf{im}}{R; C^r \vdash \tau.\mathbf{load}(tp\_sx)^?\ a\ o : \tau_r^1 \to \tau_r^2}\ \textsc{Load}$$

def wrap_and_store$(p, \tau_r(n), \tau, tp^?, R) :=$
  if tp then let $N = |tp|$ else let $N = |\tau|$
  if $|\tau| == 64$ then $R_\Psi[p \mapsto \mathrm{i32}(\mathrm{val}(\mathrm{bits}(n)_{0..32})) \wedge (p-4) \mapsto \mathrm{i32}(\mathrm{val}(\mathrm{bits}(n)_{32..64}))]$
  if $|\tau_r(n)| > N$ then
    $R_\Psi[p \mapsto \tau_r(\mathrm{wrap}_{|\tau|,N}(n))]$
    else $R_\Psi[p \mapsto \tau_r(n)]$

$$\frac{C^r_{\mathrm{memory}} = n \qquad 2^a \le (|tp| <)^? |\tau_r^2| \qquad tp^? = \epsilon \vee \tau_r^2 = \mathbf{im} \qquad \tau_r^1 <: \mathrm{ptr32}(c) \qquad \mathrm{wrap\_and\_store}(c+o, \tau_r^2, \tau_w, tp^?, R) \qquad \tau_r^2 <: \tau(\top)}{R, C^r \vdash \tau.\mathbf{store}\ tp^?\ a\ o : \tau_r^1\ \tau_r^2 \to \epsilon}\ \textsc{Store}$$

$$\frac{C^r_{\mathrm{memory}} = n \qquad \mathrm{num32}(n_1) <: \tau_r^1 \qquad \mathrm{num32}(n_2) <: \tau_r^2}{R; C^r \vdash \mathbf{grow\_memory} : \tau_r^1 \to \tau_r^2}\ \textsc{Grow-memory}$$

$$\frac{C^r_{\mathrm{memory}} = n \qquad \mathrm{num32}(n_1) <: \tau_r \qquad |R_\Psi| = n}{R; C^r \vdash \mathbf{current\_memory} : \epsilon \to \tau_r}\ \textsc{Current-memory}$$

$$\frac{C^r_{\mathrm{func}}(i) = (\tau_r')^m, \Psi'_{\mathrm{pre}} \to (\tau_r')^n, \Psi'_{\mathrm{post}} \qquad tf_r = \tau_r^m, \Psi_{\mathrm{pre}} \to \tau_r^n, \Psi_{\mathrm{post}} \qquad (\tau_r <: \tau_r')^m \qquad (\tau_r' <: \tau_r)^n \qquad \Psi'_{\mathrm{pre}} <: \Psi_{\mathrm{pre}} \qquad \Psi_{\mathrm{post}} <: \Psi'_{\mathrm{post}}}{R; C^r \vdash \mathbf{call}\ i\ tf_r : tf_r}\ \textsc{Call}$$

$$\frac{tf_r = \tau_r^m, \Psi_{\mathrm{pre}} \to \tau_r^n, \Psi_{\mathrm{post}} \qquad C^r_{\mathrm{table}} = n \qquad \mathrm{num32}(c) <: \tau_r^1}{R; C^r \vdash \mathbf{call\_indirect}\ tf_r : \tau_r^m, \tau_r^1 \to \tau_r^n, \Psi_{\mathrm{post}}}\ \textsc{Call-Indirect}$$

Fig. 8. Typing Rules for Refinement Type System continued.

$$\frac{}{R; C^r \vdash \tau.\mathbf{const}\ c : \epsilon \rightarrow \tau(c)}\ \textsc{Const} \qquad \frac{\tau_r^2 = \mathrm{unop}(\tau_r^1)}{R; C^r \vdash \tau.\mathbf{unop} : \tau_r^1 \rightarrow \tau_r^2}\ \textsc{Unop}$$

$$\frac{\tau_r^3 = \mathrm{binop}(\tau_r^1, \tau_r^2) \qquad \tau_r^3 <: \tau(\top)}{R; C^r \vdash \tau.\mathbf{binop} : \tau_r^1\ \tau_r^2 \rightarrow \tau_r^3}\ \textsc{Binop} \qquad \frac{\tau_r^2 = \mathrm{testop}(\tau_r^1) \qquad \tau_r^2 <: \mathrm{num}(\top)}{R; C^r \vdash \tau.\mathbf{testop} : \tau_r^1 \rightarrow \tau_r^2}\ \textsc{Testop}$$

$$\frac{\tau_r^3 = \mathrm{relop}(\tau_r^1, \tau_r^2) \qquad \tau_r^3 <: \mathrm{num}(\top)}{R; C^r \vdash \tau.\mathbf{relop} : \tau_r^1\ \tau_r^2 \rightarrow \tau_r^3}\ \textsc{Relop} \qquad \frac{}{R; C^r \vdash \mathbf{unreachable} : \tau_r^n \rightarrow \tau_r^m}\ \textsc{Unreachable}$$

$$\frac{}{R; C^r \vdash \mathbf{nop} : \epsilon \rightarrow \epsilon}\ \textsc{Nop} \qquad \frac{}{R; C^r \vdash \mathbf{drop} : \tau_r \rightarrow \epsilon}\ \textsc{Drop}$$

$$\frac{}{R; C^r \vdash \mathbf{select} : \tau_r^1\ \tau_r^2\ \mathrm{num32}(n) \rightarrow \tau_r^3}\ \textsc{Select}$$

$$\frac{\begin{array}{c} tf_r = \tau_r^m, \Psi_{\mathrm{pre}} \rightarrow \tau_r^n, \Psi_{\mathrm{post}} \qquad R_\Psi = \Psi_{\mathrm{pre}} \\ R; , C^r, \mathrm{label}(\tau_r^n) \vdash {e_1}^* : \tau_r^m, \Psi_{\mathrm{pre}} \rightarrow (\tau_r')^n, \Psi_{\mathrm{post}}' \\ R; C^r, \mathrm{label}(\tau_r^n) \vdash {e_2}^* : \tau_r^m, \Psi_{\mathrm{pre}} \rightarrow (\hat{\tau}_r'')^n, \Psi_{\mathrm{post}}'' \\ (\tau_r' <: \tau_r)^n \qquad (\hat{\tau}_r'' <: \tau_r)^n \qquad \Psi_{\mathrm{post}}' <: \Psi_{\mathrm{post}} \qquad \Psi_{\mathrm{post}}'' <: \Psi_{\mathrm{post}} \end{array}}{R; C^r \vdash \mathbf{if}\ tf_r\ \mathbf{then}\ {e_1}^*\ \mathbf{else}\ {e_2}^*\ \mathbf{end} : tf_r}\ \textsc{If-Else}$$

$$\frac{C^r_{\mathrm{label}}(i) = (\tau_r')^m, \Psi' \qquad (\tau_r <: \tau_r')^m \qquad R_\Psi <: \Psi'}{R; C^r \vdash \mathbf{br}\ i : \tau_r^*\ \tau_r^m \rightarrow \tau_r^*}\ \textsc{Br}$$

$$\frac{\begin{array}{c} C^r_{\mathrm{label}}(i) = (\tau_r')^m, \Psi' \qquad \mathrm{num32}(n) <: \tau_r^1 \\ (\tau_r <: \tau_r')^m \qquad R_\Psi <: \Psi' \end{array}}{R; C^r \vdash \mathbf{br\_if}\ i : \tau_r^m\ \tau_r^1 \rightarrow \tau_r^m}\ \textsc{Br-if} \qquad \frac{\begin{array}{c} C^r_{\mathrm{label}}(i) = ((\tau_r')^m, \Psi)^+ \\ ((\tau_r <: \tau_r')^m)^+ \qquad (R_\Psi <: \Psi)^+ \end{array}}{R; C^r \vdash \mathbf{br\_table}\ i^+ : \tau_r^*\ \tau_r^m\ \tau_r^1 \rightarrow \tau_r^*}\ \textsc{Br-table}$$

$$\frac{\begin{array}{c} C^r_{\mathrm{label}}(i) = (\tau_r')^m, \Psi \\ (\tau_r <: \tau_r')^m \qquad R_\Psi <: \Psi \end{array}}{R; C^r \vdash \mathbf{return} : \tau_r^*\ \tau_r^m \rightarrow \tau_r^*}\ \textsc{Return} \qquad \frac{C^r_{\mathrm{local}}(i) = \tau_r}{R; C^r \vdash \mathbf{get\_local}\ i : \epsilon \rightarrow \tau_r}\ \textsc{Get-local}$$

$$\frac{C^r_{\mathrm{local}}(i) = \tau_r^2 \qquad \tau_r^1 <: \tau_r^2}{R; C^r \vdash \mathbf{set\_local}\ i : \tau_r^1 \rightarrow \epsilon}\ \textsc{Set-local} \qquad \frac{C^r_{\mathrm{local}}(i) = \tau_r^2 \qquad \tau_r^1 <: \tau_r^2}{R; C^r \vdash \mathbf{tee\_local}\ i : \tau_r^1 \rightarrow \tau_r^1}\ \textsc{Tee-local}$$

$$\frac{C^r_{\mathrm{global}}(i) = \tau_r}{R; C^r \vdash \mathbf{get\_global}\ i : \epsilon \rightarrow \tau_r}\ \textsc{Get-global} \qquad \frac{C^r_{\mathrm{global}}(i) = \tau_r^2 \qquad \tau_r^1 <: \tau_r^2}{R; C^r \vdash \mathbf{set\_global}\ i : \tau_r^1 \rightarrow \epsilon}\ \textsc{Set-global}$$

$$\frac{}{R; C^r \vdash \epsilon : \epsilon \rightarrow \epsilon}\ \textsc{EmptyStack} \qquad \frac{R; C^r \vdash {e_1}^* : \tau_r^a \rightarrow \tau_r^b \qquad R; C^r \vdash e_2 : \tau_r^b \rightarrow \tau_r^c}{R; C^r \vdash {e_1}^*\ e_2 : \tau_r^a \rightarrow \tau_r^c}\ \textsc{Sequencing}$$

$$\frac{R; C^r \vdash e^* : \tau_r^b \rightarrow \tau_r^c}{R; C^r \vdash e^* : \tau_r^a\ \tau_r^b \rightarrow \tau_r^a\ \tau_r^c}\ \textsc{TopOfStack} \qquad \frac{}{R; C^r \vdash \mathrm{trap} : tf_r}\ \textsc{Trap}$$

Fig. 9. Typing Rules for the Refinement Type System continued.

B TYPE SAFETY PROOF

Theorem 3. ***Progress**: If $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^* \wedge (v : \tau_r)^m \wedge \vdash_i s : R$ then $e^* = v^*$ or $e^* = \mathsf{trap}$ or $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$*

Proof. Proof by induction of the deriviation of $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^*$

Case Load:

$$\frac{C^r_{\text{memory}} = n \qquad \tau_r^1 <: \text{ptr}(c) \qquad \tau_r^2 = \text{load\_and\_extend}(c+o, \tau_w, (tp\_sx)^?, R_\Psi) \\ 2^a \leq (|tp| <)^? |\tau_r^2| \qquad (tp\_sz)^? = \epsilon \vee \tau_r^2 = \mathbf{im}}{R; C^r \vdash \tau.\mathbf{load}(tp\_sx)^? \ a \ o : \tau_r^1 \rightarrow \tau_r^2}\ \textsc{Load}$$

The load instruction has to take a step since only end instructions signify the end of a function. We now case on the shape of a well typed stack. The only possible case is that the load instruction has a ptr(a) on the stack, since this is a premise of our typing rule. The operational semantics of the load instruction matches this stack shape. If the side conditions are not met, it steps to a trap. □

Theorem 4. ***Preservation**: If $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^* \wedge \vdash_i (v : \tau_r)^m \wedge \vdash_i s : R \wedge s, v^*, e^* \hookrightarrow s', v'^*, e'^*$ then $\exists\, R'$ such that $dom(R_\Psi) \subseteq dom(R'_\Psi) \wedge \vdash_i s' : R' \wedge R'; C^r \vdash_i e'^* : \tau_r^*$*

Proof. Proof by induction of the deriviation of $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \rightarrow \tau_r^*$

Case Load:

$$\frac{C^r_{\text{memory}} = n \qquad \tau_r^1 <: \text{ptr}(c) \qquad \tau_r^2 = \text{load\_and\_extend}(c+o, \tau_w, (tp\_sx)^?, R_\Psi) \\ 2^a \leq (|tp| <)^? |\tau_r^2| \qquad (tp\_sz)^? = \epsilon \vee \tau_r^2 = \mathbf{im}}{R; C^r \vdash \tau.\mathbf{load}(tp\_sx)^? \ a \ o : \tau_r^1 \rightarrow \tau_r^2}\ \textsc{Load}$$

From our typing rule for the load instruction we have that it expects τ_r^1 on the stack and pushes τ_r^2 on the stack. From the operational semantics of the load instruction we know that there are three possible cases:

Case 1: s; ptr.const(c); τ.load $a\ o \hookrightarrow \tau$.const$(b*)$ if $s_{\text{mem}}(i, c+o, |\tau|) = b^*$

From $\vdash_i s : R$, we know that $\vdash s_{\text{mem}} : R_\Psi$, from which we know that $s_{\text{mem}}(c+o) : R_\Psi(c+o)$. Since our memory typing R_Ψ maps addresses to types of size 32, we have to now case on the data expected after the load instruction. This is annotated on the load instruction itself as τ.

Case 1.1: τ.load $a\ o \wedge R_\Psi[(c+o) \mapsto \tau_{r_{32}}(n)]$

This is the case where the data in memory is of the type that the load expects to produce on the stack. We now have to show that $b^* : \tau(n)$. From the typing rule, we have that load expects τ_r^2. On inspecting the load_and_extend function, we see that for the case where the data in memory is the same type as is expected on the stack, $\tau_r^2 = \tau_r(n_1)$ where $\tau_r(n_1) = \Psi(c+o)$. Since $s_{\text{mem}}(c+o) : R_\Psi(c+o)$, we know that $s_{\text{mem}}(c+o) : \tau_r^2$ and since $s_{\text{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

Case 1.2: i64.load $a\ o \wedge R_\Psi[(c+o) \mapsto \tau_{r_{32}}(n_1), (c+o-4) \mapsto \tau_{r_{32}}(n_2)]$

In this case, the load expects a i64 on the stack and the memory has $\tau_{r_{32}}(n_1)$ at $(c+o)$ and $\tau_{r_{32}}(n_2)$ at $(c+o-4)$. We now have to show that $b^* :$ i64(n). Note that b^* is the byte sequence from $s_{\text{mem}}[c+o : 8]$, as specified in the detailed Wasm specification [40]. From the typing rule, we know that load expects τ_r^2 on the stack. On inspecting the load_and_extend function, we see that this case is the one that matches $|\tau_r(n_1)| < N$, where $N = |\tau| = 64$. We see then that $\tau_r^2 = \tau(\text{val}(\text{bit}(n_1), \text{bit}(n_2)))$. n_1 and n_2 are the values in memory at $c+o$ and $c+o-4$. Since $s_{\text{mem}}(c+o) : R_\Psi(c+o) \wedge s_{\text{mem}}(c+o-4) : R_\Psi(c+o-4)$, we know that $s_{\text{mem}}(c+o : 8) : \tau_r^2$ and since $s_{\text{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

Case 1.3: f64.load $a\ o \wedge R_\Psi[(c+o) \mapsto \tau_{r_{32}}(n_1), (c+o-4) \mapsto \tau_{r_{32}}(n_2)]$
In this case, the load expects a f64 on the stack and the memory has $\tau_{r_{32}}(n_1)$ at $(c+o)$ and $\tau_{r_{32}}(n_2)$ at $(c+o-4)$. We now have to show that $b^* : \mathsf{f64}(n)$. The proof proceeds exactly as the case above.

Case 1.4: ptr.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{f32}(n)]$
Case 1.5: num.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{f32}(n)]$
Case 1.6: i32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{f32}(n)]$
In this case, the load expects a i32 on the stack and Ψ has $\mathsf{f32}(n)$ at $c+o$. We now have to show that $b^* : \mathsf{f32}(n)$. From the typing rule, we have that load expects τ_r^2. On inspecting the load_and_extend function, we see that for this case, $\tau_r^2 = \mathsf{f32}(\mathsf{bits}(n_1 \text{ as } \mathsf{f32}))$ where $\tau_r(n_1) = \Psi(c+o)$. Since $s_{\mathsf{mem}}(c+o) : R_\Psi(c+o)$, we know that $s_{\mathsf{mem}}(c+o) : \tau_r^2$ and since $s_{\mathsf{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

Case 1.7: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{ptr}(n)]$
Case 1.8: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{num}(n)]$
Case 1.9: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \mathsf{i32}(n)]$
In this case, the load expects a f32 on the stack and Ψ has $\mathsf{i32}(n)$ at $c+o$. We now have to show that $b^* : \mathsf{i32}(n)$. From the typing rule, we have that load expects τ_r^2. On inspecting the load_and_extend function, we see that for this case, $\tau_r^2 = \mathsf{i32}(\mathsf{f32}(n_1) \text{ as } \mathsf{i32})$ where $\tau_r(n_1) = \Psi(c+o)$. Since $s_{\mathsf{mem}}(c+o) : R_\Psi(c+o)$, we know that $s_{\mathsf{mem}}(c+o) : \tau_r^2$ and since $s_{\mathsf{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

Case 2: s; ptr.const(c); τ.load $tp\_sx\ a\ o \hookrightarrow \tau$.const$(b*)$ if $s_{\mathsf{mem}}(i, c+o, |tp|) = b^*$
From $\vdash_i s : R$, we know that $\vdash s_{\mathsf{mem}} : R_\Psi$, from which we know that $s_{\mathsf{mem}}(c+o) : R_\Psi(c+o)$. Since our memory typing R_Ψ maps addresses to types of size 32, we have to now case on the data expected after the load instruction. The proof proceeds as above except with a small change introduced by the tp_sx annotation on the load. In the WebAssembly operational semantics [40], when a load has this packed type annotation, the bits are read from memory upto $|tp|$ and the value read from memory is size extended with the $extend\_sx_{N,|\tau|}$ function. Both of these cases are handled in the load_and_extend function, in case $|\tau_r(n_1)| > N$ and size extending function extend $\_sx_{N,|\tau|}$.

Case 3: s; ptr.const(c); τ.load $(tp\_sx)^?\ a\ o \hookrightarrow$ trap otherwise
The trap instruction is well typed. □

C MEET AND JOIN OPERATIONS OVER THE INTEGER SUB-LATTICE

$\sqcup$	i32($\top$)	i32(n)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	num($\top$)	num(n)	num($\bot$)	i32($\bot$)
i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)
i32(n')	i32($\top$)	i32($n \sqcup n'$)	i32(n')	i32($n' \sqcup n_0$)	i32($\top$)	i32(n')	i32($\top$)	i32($n' \sqcup n$)	i32(n')	i32(n')
ptr($\top,\top$)	i32($\top$)	i32(n)	ptr($\top,\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr($\top,\top$)	i32($\top$)	i32($\top$)	i32($\top$)	ptr($\top,\top$)
ptr(l_0,n_0')	i32($\top$)	i32($n \sqcup n_0'$)	ptr($\top,\top$)	ptr($l_0,n_0' \sqcup n_0$)	ptr($\top,\top$)	ptr(l_0,n_0')	i32($\top$)	i32($\top$)	i32(n_0')	ptr(l_0,n_0')
ptr(l_1,n_1')	i32($\top$)	i32($\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr($l_1,n_1 \sqcup n_1'$)	ptr(l_1,n_1')	i32($\top$)	i32($\top$)	i32($\top$)	ptr(l_1,n_1')
ptr(l_2,n_2')	i32($\top$)	i32($\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr(l_2,n_2')	i32($\top$)	i32($\top$)	i32($\top$)	ptr(l_2,n_2')
ptr($\bot,\bot$)	i32($\top$)	i32($\top$)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	i32($\top$)	i32($\top$)	i32($\top$)	ptr($\bot,\bot$)
num($\top$)	i32($\top$)	i32(n)	i32($\top$)	i32($\top$)	i32($\top$)	i32($\top$)	num($\top$)	num($\top$)	num($\top$)	num($\top$)
num(n')	i32($\top$)	i32($n \sqcup n'$)	i32($\top$)	i32($n_0 \sqcup n'$)	i32($\top$)	i32($\top$)	num(n')	num($n' \sqcup n$)	num(n')	num(n')
num($\bot$)	i32($\top$)	i32(n)	i32($\top$)	i32(n_0)	i32($\top$)	i32($\top$)	num($\top$)	num(n)	num($\bot$)	num($\bot$)
i32($\bot$)	i32($\top$)	i32(n)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	num($\top$)	num(n)	num($\bot$)	i32($\bot$)

Table 6. Join $\sqcup$ opertation for the i32 sub-lattice.

$\sqcap$	i32($\top$)	i32(n)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	num($\top$)	num(n)	num($\bot$)	i32($\bot$)
i32($\top$)	i32($\top$)	i32(n)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	num($\top$)	num(n)	num($\bot$)	i32($\bot$)
i32(n')	i32(n')	i32($n \sqcap n'$)	ptr($\top,\top$)	ptr($l_0,n_0 \sqcap n'$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	num(n')	num($n \sqcap n'$)	num($\bot$)	i32($\bot$)
ptr($\top,\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr($\top,\top$)	ptr(l_0,n_0)	ptr(l_1,n_1)	ptr($\bot,\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)
ptr(l_0,n_0')	ptr(l_0,n_0')	ptr($l_0,n_0' \sqcap n$)	ptr(l_0,n_0')	ptr($l_0,n_0 \sqcap n_0'$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)
ptr(l_1,n_1')	ptr(l_1,n_1')	ptr($\bot,\bot$)	ptr(l_1,n_1')	ptr($\bot,\bot$)	ptr($l_1,n_1 \sqcap n_1'$)	ptr($\bot,\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)
ptr(l_2,n_2')	ptr(l_2,n_2')	ptr($\bot,\bot$)	ptr(l_2,n_2')	ptr($\bot,\bot$)	ptr($l_3,0$)*	ptr($\bot,\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)
ptr($\bot,\bot$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	ptr($\bot,\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)
num($\top$)	num($\top$)	num($\top$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	num($\top$)	num(n)	num($\bot$)	i32($\bot$)
num(n')	num(n')	num($n \sqcap n'$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	num(n')	num($n \sqcap n'$)	num($\bot$)	i32($\bot$)
num($\bot$)	num($\bot$)	num($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	num($\bot$)	num($\bot$)	num($\bot$)	i32($\bot$)
i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)	i32($\bot$)

*l_3 fresh $\wedge\, l_1 = l_3 - n_1 \wedge l_2 = l_3 - n_2'$

Table 7. Meet $\sqcap$ opertation for the i32 sub-lattice.

C.1 Join of two pointers

The canonical example for joins in static analysis is the if-then-else expression. Let us suppose that the then branch of such an expression returns $\mathsf{ptr}(l_1, n_1)$, and the else branch returns $\mathsf{ptr}(l_2, n_2)$. What pointer does the if-then-else expression return? Since it could be either of the two, we do not presume to know what the pointer could be and instead return $\mathsf{ptr}(\top, \top)$. Instead, if the then branch returned $\mathsf{ptr}(l, n_1)$ and the else branch returned $\mathsf{ptr}(l, n_2)$, we say that the if-then-else expression returns $\mathsf{ptr}(l, n_1 \sqcap n_2)$.

C.2 Meet of two pointers

The canonical example for meet operations in static analysis is values during several loop iterations. Say, at loop iteration n, the type of a stack slot is $\mathsf{ptr}(l_1, n_1)$, a pointer with a symbolic address, and at loop iteration $n + 1$, its type is $\mathsf{ptr}(l_c, n_0)$, a constant pointer. Since it is impossible for a stack slot to have a symbolic and constant pointer, we return $\mathsf{ptr}(\bot, \bot)$. On the other hand, say that at loop iteration n, the type of a stack slot is $\mathsf{ptr}(l_1, n_1)$ and at loop iteration $n + 1$, its type is $\mathsf{ptr}(l_2, n_2)$. This means that the type at this stack slot is both $\mathsf{ptr}(l_1, n_1)$ and $\mathsf{ptr}(l_2, n_2)$. We equate these two pointers to a third pointer $\mathsf{ptr}(l_3, 0)$, where l_3 is a fresh symbolic location and $\mathsf{ptr}(l_1, n_1) = \mathsf{ptr}(l_3, 0) \wedge \mathsf{ptr}(l_2, n_2) = \mathsf{ptr}(l_3, 0)$. Pointers can be written as polynomials (since the offset is usually added into the base address) and we get that $l_1 + n_1 = l_3$ or that, $l_1 = l_3 - n_1$ and $l_2 = l_3 - n_2$. We say that the type of this stack slot is $\mathsf{ptr}(l_3, 0)$.

C.3 Join and Meet of a pointer and a number

The join of a pointer with a number results in an i32($\top$), except in the case of constant pointers $\mathsf{ptr}(l_c, n_0)$. For a constant pointer, we know that the base address l_c equates to 0 and so the operation $\mathsf{ptr}(l_c, n_0) \sqcup \mathsf{num}(n_1)$, produces $\mathsf{i32}(n_0 \sqcup n_1)$ as its result. The meet of a pointer and number unequivocally results in a i32($\bot$), as per our relation definition shown in Figure ??.

C.4 Join and meet of a i32 with a pointer

If the then branch of a if-then-else expression returned $\mathsf{i32}(n_1)$ and the else branch returned a constant pointer $\mathsf{ptr}(l_c, n_0)$, the if-then-else expression would return $\mathsf{i32}(n_0 \sqcup n_1)$. If the else branch returned a pointer with a symbolic address $\mathsf{ptr}(l_2, n_2)$ instead, the if-then-else expression would return i32($\top$), since the base address l_1 is unknown. For the meet operation between i32's and ptr's, let us imagine that the type of a stack slot at loop iteration n is $\mathsf{i32}(n_1)$ and that at loop iteration $n + 1$, the type of the stack slot is a constant pointer $\mathsf{ptr}(l_c, n_0)$. The type of the stack slot is then both these types and so, $\mathsf{ptr}(l_c, n_0 \sqcap n_1)$. If instead, at loop iteration $n + 1$, the type of the stack slot is a symbolic pointer $\mathsf{ptr}(l_2, n_2)$, the type of the stack slot would be $\mathsf{ptr}(l_c, n_1) \sqcap \mathsf{ptr}(l_2, n_2)$, which results in $\mathsf{ptr}(\bot, \bot)$.

C.5 Join and meet of a i32 with a number

If the then branch of a if-then-else expression returned a $\mathsf{i32}(n_0)$ and the else branch returned $\mathsf{num}(n_1)$, the if-then-else expression would return $\mathsf{i32}(n_0 \sqcup n_1)$. For the meet operation between i32's and num's, let us imagine that the type of a stack slot at loop iteration n is $\mathsf{i32}(n_0)$ and that at loop iteration $n + 1$, the type of the stack slot is a constant pointer $\mathsf{num}(n_1)$. The type of the stack slot is then both these types and so, $\mathsf{num}(n_0 \sqcap n_1)$.

Note that we never explicitly join or meet locations separate from their offsets. When the meet operation is performed over two pointers with the same symbolic base address l, the result is a pointer with that base address l and the join of their offsets. When the meet operation is performed on two pointers with different symbolic locations, we produce a pointer with a $\top$ location and $\top$ offset. This is analogous to $\mathsf{ptr}(\top)$, since we never construct $\mathsf{ptr}(\top, n)$. Similarly, for join operations, we never construct $\mathsf{ptr}(\bot, n)$ and $\mathsf{ptr}(\bot, \bot)$ is analogous to $\mathsf{ptr}(\bot)$. However, in both cases, we do construct $\mathsf{ptr}(l, \top)$ and $\mathsf{ptr}(l, \bot)$.

D CONSTRAINT GENERATION FOR ALL WEBASSEMBLY INSTRUCTIONS

$$\textit{context} \quad C^{\alpha} \ ::= \ \{\text{func } \textit{tf}^{*},\ \text{local } \alpha^{*},\ \text{global } \alpha^{*},\ \text{table } n^{?},\ \text{memory } n^{?},\ \text{label } (\alpha^{*}, \varsigma)^{*},\ \text{return } (\alpha^{*}, \varsigma)^{*}\}$$

Constraints for Instructions $\boxed{S; \varsigma; C^{\alpha} \vdash e : \alpha^{*} \rightarrow \alpha^{*}; S'; \varsigma'}$

$$\frac{\alpha \text{ fresh} \qquad S' = S :: [\alpha \doteq \tau_w(c)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathbf{const}\ c : \epsilon \rightarrow \alpha; S'; \varsigma}\ \textsc{Constant}$$

$$\frac{\alpha_1 \in \mathit{dom}(S) \qquad \alpha_2 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq \mathit{unop}(\alpha_2)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathit{unop} : \alpha_1 \rightarrow \alpha_2; S'; \varsigma}\ \textsc{Unary-Ops}$$

$$\frac{\alpha_1\ \alpha_2 \in \mathit{dom}(S) \qquad \alpha_3 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq \mathit{binop}(\_, \alpha_2, \alpha_3) \wedge \alpha_2 \doteq \mathit{binop}(\alpha_1, \_, \alpha_3) \wedge \alpha_3 \doteq \mathit{binop}(\alpha_1, \alpha_2, \_)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathit{binop} : \alpha_1\ \alpha_2 \rightarrow \alpha_3; S'; \varsigma}\ \textsc{Binary-Ops}$$

$$\frac{\alpha_1\ \alpha_2 \in \mathit{dom}(S) \qquad \alpha_3 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq \mathit{testop}(\_, \alpha_2, \alpha_3) \wedge \alpha_2 \doteq \mathit{testop}(\alpha_1, \_, \alpha_3) \wedge \alpha_3 \doteq \mathit{testop}(\alpha_1, \alpha_2, \_)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathit{testop} : \alpha_1\ \alpha_2 \rightarrow \alpha_3; S'; \varsigma}\ \textsc{Test-Ops}$$

$$\frac{\alpha_1\ \alpha_2 \in \mathit{dom}(S) \qquad \alpha_3 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq \mathit{relop}(\_, \alpha_2, \alpha_3) \wedge \alpha_2 \doteq \mathit{relop}(\alpha_1, \_, \alpha_3) \wedge \alpha_3 \doteq \mathit{relop}(\alpha_1, \alpha_2, \_)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathit{relop} : \alpha_1\ \alpha_2 \rightarrow \alpha_3; S'; \varsigma}\ \textsc{Rel-Ops}$$

$$\frac{\alpha_1 \in \mathit{dom}(S) \qquad \alpha_2 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq \mathit{cvtop}\ \tau_w\_sx^{?}(\_, \alpha_2) \wedge \alpha_2 \doteq \mathit{cvtop}\ \tau_w\_sx^{?}(\alpha_1, \_)]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathit{cvtop}\ \tau_w\_sx^{?} : \alpha_1 \rightarrow \alpha_2; S'; \varsigma}\ \textsc{Convert-Ops}$$

$$\frac{\alpha_1 \in \mathit{dom}(S) \qquad S' = S :: [\alpha_1 <: \text{ptr} \wedge \alpha_2 \doteq \varsigma[\alpha_1]_{(\mathit{tf\_sx}^{?},\ a,\ o)} \wedge \alpha_2 <: \tau_w]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathbf{load}(\mathit{tp\_sx})^{?}\ a\ o : \alpha_1 \rightarrow \alpha_2; S'; \varsigma}\ \textsc{Load}$$

$$\frac{\alpha_1\ \alpha_2 \in \mathit{dom}(S) \qquad S' = S :: [\alpha_1 <: \text{ptr} \wedge \alpha_2 <: \tau_w \wedge \varsigma' \doteq \varsigma[\alpha_1 \mapsto \alpha_2]_{(\mathit{tp}^{?},\ a,\ o)}]}{S; \varsigma; C^{\alpha} \vdash \tau_w.\mathbf{store}\ \mathit{tp}^{?}\ a\ o : \alpha_1\ \alpha_2 \rightarrow \epsilon; S'; \varsigma'}\ \textsc{Store}$$

Fig. 10. Constraint Generation Rules.

$$\dfrac{\begin{array}{c} tf_w = \tau_w^m \to \tau_w^n \qquad \alpha^m \in dom(S) \qquad \alpha^n \text{ fresh} \\ S;\varsigma;C^\alpha, \text{label}(\alpha^n) \vdash e^* : \alpha^m \to (\alpha')^n; S';\varsigma' \qquad S'' = S' :: [(\alpha \doteq \alpha')^n] \end{array}}{S;\varsigma;C^\alpha \vdash \textbf{block}\ tf\, e^*\ \textbf{end} : \alpha_1^n \dots \alpha_n^n \to \alpha_1^m \dots \alpha_m^m; S''; \varsigma'}\ \textsc{Block}$$

$$\dfrac{\begin{array}{c} tf_w = \tau_w^m \to \tau_w^n \qquad \alpha^m \in dom(S) \qquad \alpha^n \text{ fresh} \\ S;\varsigma;C^\alpha, \text{label}(\alpha^m) \vdash e^* : \alpha^m \to (\alpha')^n; S';\varsigma' \qquad S'' = S' :: [(\alpha \doteq \alpha')^n] \end{array}}{S;\varsigma;C^\alpha \vdash \textbf{loop}\ tf_w\ e^*\ \textbf{end} : \alpha^m \to \alpha^n; S''; \varsigma'}\ \textsc{Loop}$$

$$\dfrac{\begin{array}{c} tf_w = \tau_w^m \to \tau_w^n \qquad \alpha^m\ \alpha \in dom(S) \qquad \alpha_t^n, \alpha_e^n, \alpha^n \text{ fresh} \qquad S' = S :: [\alpha <: \text{num}] \\ S';\varsigma;C^\alpha, \text{label}(\alpha_t^n) \vdash {e_t}^* : \alpha_m \to \alpha_t^n; S_e; \varsigma_e \qquad S';\varsigma;C^\alpha, \text{label}(\alpha_e^n) \vdash {e_e}^* : \alpha_m \to \alpha_e^n; S_t; \varsigma_t \\ S'' = S' :: S_e :: S_t :: [\varsigma' \doteq \bigsqcup \varsigma_e \varsigma_t \wedge (\alpha' \doteq \bigsqcup \alpha_t \alpha_e)^n] \end{array}}{S;\varsigma;C^\alpha \vdash \textbf{if}\ tf\, {e_t}^*\ \textbf{else}\ {e_e}^*\ \textbf{end} : \alpha^m\ \alpha \to \alpha^n; S''; \varsigma'}\ \textsc{If-Else}$$

$$\dfrac{\alpha^*\alpha^n \in dom(S) \qquad C^\alpha_{\text{label}}(i) = (\alpha')^n, \varsigma' \qquad S' = S :: [\varsigma \doteq \bigsqcup \varsigma\varsigma' \wedge (\alpha \doteq \bigsqcup \alpha\alpha')^n]}{S;\varsigma;C^\alpha \vdash \textbf{br}\ i : \alpha^*\alpha_1 \dots \alpha_n \to \alpha^*; S'; \varsigma}\ \textsc{Br}$$

$$\dfrac{\alpha^m\alpha \in dom(S) \qquad C^\alpha_{\text{label}}(i) = (\alpha')^n, \varsigma' \qquad S' = S :: [\varsigma \doteq \bigsqcup \varsigma\varsigma' \wedge \alpha <: \text{num} \wedge (\alpha \doteq \bigsqcup \alpha\alpha')^n]}{S;\varsigma;C^\alpha \vdash \textbf{br\_if}\ i : \alpha^m\alpha \to \alpha^n; S'; \varsigma}\ \textsc{Br-If}$$

$$\dfrac{\begin{array}{c} \alpha^*\alpha^m\alpha \in dom(S) \qquad (C^\alpha_{\text{label}}(i) = (\alpha')^n)^+, \varsigma' \\ S' = S :: ([\varsigma \doteq \bigsqcup \varsigma\varsigma' \wedge \alpha_{n+1} <: \text{num} \wedge \alpha_1 \doteq \bigsqcup \alpha_1\alpha_1' \wedge \cdots \wedge \alpha_n \doteq \bigsqcup \alpha_n\alpha_n'])^+ \end{array}}{S;\varsigma;C^\alpha \vdash \textbf{br\_table}\ i^+ : \alpha^*\alpha_1 \dots \alpha_n\alpha_{n+1} \to \alpha^*; S'; \varsigma}\ \textsc{Br-Table}$$

$$\dfrac{\begin{array}{c} \alpha^*\alpha_1 \dots \alpha_n \in dom(S) \qquad C_{\text{return}}(i) = \alpha_1' \dots \alpha_n' \\ S' = S :: [\alpha_1 \doteq \bigsqcup \alpha_1\alpha_1' \wedge \cdots \wedge \alpha_n \doteq \bigsqcup \alpha_n\alpha_n'] \end{array}}{S;\varsigma;C \vdash \textbf{return} : \alpha^*\alpha_1 \dots \alpha_n \to \alpha^*; S'; \varsigma}$$

$$\dfrac{\begin{array}{c} C_{\text{func}} = tf \qquad tf = \tau_w^1 \dots \tau_w^n \to \tau_w^1 \dots \tau_w^m \\ \alpha_1 \dots \alpha_n \in dom(S) \qquad S' = S :: [\alpha_1' <: \tau_1 \wedge \cdots \wedge \alpha_m' <: \tau_m] \end{array}}{S;\varsigma;C \vdash \textbf{call} : \alpha_1 \dots \alpha_n \to \alpha_1' \dots \alpha_m'; S'; \varsigma}$$

$$\dfrac{\begin{array}{c} C_{\text{table}} = n \qquad tf = \tau_w^1 \dots \tau_w^n \to \tau_w^1 \dots \tau_w^m \qquad \alpha_1 \dots \alpha_n\alpha_{n+1} \in dom(S) \\ S' = S :: [\alpha_{n+1} <: \text{num} \wedge \alpha_1' <: \tau_1 \wedge \cdots \wedge \alpha_m' <: \tau_m] \end{array}}{S;\varsigma;C \vdash \textbf{call\_indirect}\ tf : \alpha_1 \dots \alpha_n\alpha_{n+1} \to \alpha_1' \dots \alpha_m'; S'; \varsigma}$$

$$\dfrac{\alpha \text{ fresh} \qquad S' = S :: [\alpha <: \text{num}]}{S;\varsigma;C^\alpha \vdash \textbf{current\_memory} : \epsilon \to \alpha; S'; \varsigma}\ \textsc{Current-Memory}$$

$$\dfrac{\alpha_1 \in dom(S) \qquad \alpha_2 \text{ fresh} \qquad S' = S :: [\alpha_1 <: \text{num} \wedge \alpha_2 <: \text{num}]}{S;\varsigma;C^\alpha \vdash \textbf{grow\_memory} : \alpha_1 \to \alpha_2; S'; \varsigma}\ \textsc{Grow-Memory}$$

$$\dfrac{\alpha_1\ \alpha_2; \alpha_3 \in dom(S) \qquad \alpha_4 \text{ fresh} \qquad S' = S :: [\alpha_3 <: \text{num} \wedge \alpha_4 \doteq \bigsqcup \alpha_1\alpha_2]}{S;\varsigma;C^\alpha \vdash \textbf{select} : \alpha_1\ \alpha_2\ \alpha_3 \to \alpha_4; S'; \varsigma}\ \textsc{Select}$$

Fig. 11. Constraint Generation Rules cont.

$$\frac{{\alpha_1}^* \in dom(S) \qquad {\alpha_2}^* \text{ fresh}}{S;\varsigma;C^{\alpha} \vdash \textbf{unreachable} : {\alpha_1}^* \rightarrow {\alpha_2}^*; S;\varsigma}\textsc{Unreachable} \qquad \frac{}{S;\varsigma;C \vdash \textbf{nop} : \epsilon \rightarrow \epsilon; S;\varsigma}\textsc{Nop}$$

$$\frac{\alpha \in dom(S)}{S;\varsigma;C \vdash \textbf{drop} : \alpha \rightarrow \epsilon; S;\varsigma}\textsc{Drop} \qquad \frac{\alpha \text{ fresh} \qquad S' = S :: [\alpha \doteq C_{\text{local}}(i)]}{S;\varsigma;C \vdash \textbf{get\_local}\ i : \epsilon \rightarrow \alpha; S';\varsigma}\textsc{Get-Local}$$

$$\frac{\alpha \in dom(S) \qquad C_{\text{local}}(i) = \alpha' \qquad S' = S :: [\alpha' \doteq \alpha]}{S;\varsigma;C \vdash \textbf{set\_local}\ i : \alpha \rightarrow \epsilon; S';\varsigma}\textsc{Set-Local} \qquad \frac{\alpha \in dom(S) \qquad C_{\text{local}}(i) = \alpha' \qquad S' = S :: [\alpha' \doteq \alpha]}{S;\varsigma;C \vdash \textbf{tee\_local}\ i : \alpha \rightarrow \alpha; S';\varsigma}\textsc{Tee-Local}$$

$$\frac{\alpha \text{ fresh} \qquad S' = S :: [\alpha \doteq C_{\text{global}}(i)]}{S;\varsigma;C \vdash \textbf{get\_global}\ i : \epsilon \rightarrow \alpha; S';\varsigma}\textsc{Get-Global} \qquad \frac{\alpha \in dom(S) \qquad C_{\text{global}}(i) = \alpha' \qquad S' = S :: [\alpha' \doteq \alpha]}{S;\varsigma;C \vdash \textbf{set\_global}\ i : \alpha \rightarrow \epsilon; S';\varsigma}\textsc{Set-Global}$$

$$\frac{\begin{array}{c} tf = \tau_w^1 \ldots \tau_w^n \rightarrow \tau_w^1 \ldots \tau_w^m \qquad \tau^f = \alpha_{n_1} \ldots \alpha_{n_n} \rightarrow \alpha_{m_1} \ldots \alpha_{m_m} \\ \alpha_{n_1} \ldots \alpha_{n_n}, \alpha_{m_1} \ldots \alpha_{m_m}, \alpha_{l_1} \ldots \alpha_{l_l} \text{ fresh} \\ S = [\varsigma \doteq \cdot \wedge \alpha_{n_1} <: \tau_w^1 \wedge \cdots \wedge \alpha_{n_n} <: \tau_w^n \wedge \alpha_{m_1} <: \tau_w^1 \wedge \cdots \wedge \alpha_{m_m} <: \tau_w^m \wedge \alpha_{l_1} <: \tau_w^1 \wedge \cdots \wedge \alpha_{l_l} <: \tau_w^l] \\ S;\varsigma;C, \text{local}(\alpha_{n_1},\ldots\alpha_{n_n},\alpha_{l_1},\ldots\alpha_{l_l}),\ \text{label}(\alpha_{m_1},\ldots\alpha_{m_m}),\ \text{return}(\alpha_{m_1},\ldots\alpha_{m_m}) \vdash e^* : \tau^f; S';\varsigma' \end{array}}{[];\varsigma;C \vdash ex^*\ \textbf{func}\ tf\ \textbf{local}\ \tau_w^1 \ldots \tau_w^l\ e^* : \tau^f; S';\varsigma'}$$

Fig. 12. Constraint Generation Rules cont.

E CONSTRAINTS GENERATED FOR AN EXAMPLE WEBASSEMBLY FUNCTION

Consider the WebAssembly function in Figure 13. At the start of the function it is not immediately obvious if either parameter is a pointer. Line 8 has a memory load operation from the addition of the two parameter, so we can infer that one of the parameters must be a pointer —a careful reader will remember that we do not allow the addition of two pointers in our type system. However, we do not know which one is a pointer. It is only at line 17, where the first parameter is multiplied by 42, that we know that the first parameter is a pointer! However, we would now like to reflect this type back up to the function parameter and check that every instruction that uses this parameter does so is in accordance with its type. Hence, a forward analysis by itself would be insufficient since it would only propagate type information forwards, when we need the analysis to also propagate types backwards.

F CONSTRAINT SOLVING ALGORITHM

Algorithm 1 Algorithm for solving constraints described in Figure 5

Require: $\mathit{Constraint} = [\alpha \mapsto t^{+}; \varsigma \mapsto m^{+}]$, $\mathit{infl} = [\alpha \mapsto \{\alpha^{*}, \varsigma^{*}\}; \varsigma \mapsto \{\alpha^{*}, \varsigma^{*}\}]$
Ensure: The least Solution $\rho = [\alpha \mapsto \hat{\tau}_r; \varsigma \mapsto \Sigma]$
$\quad \mathit{Worklist} \longleftarrow \mathit{dom}(\mathit{Constraint})$
$\quad$ **for all** $\alpha \in \mathit{dom}(\mathit{Constraint})$ **do**
$\quad\quad$ **if** $\exists\, \alpha \doteq \hat{\tau}_r \in \mathit{Constraint}[\alpha]$ **then** $\rho[\alpha] \doteq \hat{\tau}_r$ **else** $\rho[\alpha] = \top$
$\quad$ **for all** $\varsigma \in \mathit{dom}(\mathit{Constraint})$ **do** $\rho[\varsigma] = \cdot$
$\quad$ **while** $\mathit{Worklist} \neq \emptyset$ **do**
$\quad\quad v := \mathit{Worklist}.\mathrm{pop}()$
$\quad\quad \mathit{Old}_v := \rho[v]$
$\quad\quad$ **if** $v = \alpha$ **then** $\textsc{Eval}_\alpha(\mathit{Constraint}, \rho, \alpha)$
$\quad\quad$ **if** $v = \varsigma$ **then** $\textsc{Eval}_\varsigma(\mathit{Constraint}, \rho, \varsigma)$
$\quad\quad$ **if** $\mathit{Old}_v \neq \rho[v]$ **then** $\mathit{Worklist}.\mathrm{append}(\mathit{infl}[v])$
$\quad$ **procedure** $\textsc{Eval}_\alpha(\mathit{Constraint}, \rho, \alpha)$
$\quad\quad \hat{\tau}_r^{\alpha} := \rho[\alpha]$
$\quad\quad$ **for all** $t \in \mathit{Constraint}[\alpha]$ **do**
$\quad\quad\quad$ **if** $t = \alpha <: \tau$ **then** $\hat{\tau}_r^{\alpha} := \hat{\tau}_r^{\alpha} \sqcap \tau(\top)$
$\quad\quad\quad$ **if** $t = \alpha \doteq op(\alpha'_1, \ldots, \alpha'_n)$ **then** $\hat{\tau}_r^{\alpha} := \hat{\tau}_r^{\alpha} \sqcap \mathit{eval}_{op}(\rho[\alpha'_1], \ldots, \rho[\alpha'_n])$
$\quad\quad\quad$ **if** $t = \alpha \doteq \bigsqcup \alpha'_1 \ldots \alpha'_m$ **then** $\hat{\tau}_r^{\alpha} := \hat{\tau}_r^{\alpha} \sqcap (\rho[\alpha'_1] \sqcup \ldots \sqcup \rho[\alpha'_m])$
$\quad\quad\quad$ **if** $t = \alpha \doteq \varsigma[\alpha']$ **then** $\Sigma := \rho[\varsigma] \wedge \hat{\tau}_r^{\mathrm{ptr}} \doteq \rho[\alpha']$
$\quad\quad\quad\quad$ **if** $\hat{\tau}_r^{\mathrm{ptr}} = \mathrm{ptr}(l, n) \wedge (l, n) \mapsto \hat{\tau}_r \in \Sigma$ **then** $\hat{\tau}_r^{(l,n)} := \hat{\tau}_r$ **else** $\hat{\tau}_r^{(l,n)} := \top$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(\top, \top) \mapsto \hat{\tau}_r \in \Sigma$ **then** $\hat{\tau}_r^{(\top,\top)} := \hat{\tau}_r$ **else** $\hat{\tau}_r^{(\top,\top)} := \bot$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(l, \top) \mapsto \hat{\tau}_r \in \Sigma$ **then** $\hat{\tau}_r^{(l,\top)} := \hat{\tau}_r$ **else** $\hat{\tau}_r^{(l,\top)} := \bot$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(l', n') \mapsto \hat{\tau}_r \in \Sigma$ **then**
$\quad\quad\quad\quad\quad$ **if** $\nexists\, l \neq l' \in \mathit{Constraint}[\alpha']$ **then** $\hat{\tau}_r^{(l',n')} := \hat{\tau}_r$
$\quad\quad\quad\quad\quad$ **else if** $\exists\, l \neq l' \in \mathit{Constraint}[\alpha'] \wedge n' = \top$ **then** $\hat{\tau}_r^{(l',n')} := \hat{\tau}_r$
$\quad\quad\quad\quad\quad$ **else** $\hat{\tau}_r^{(l',n')} := \bot$
$\quad\quad\quad\quad \hat{\tau}_r^{\alpha} := \hat{\tau}_r^{\alpha} \sqcap \hat{\tau}_r^{(l,n)} \sqcup \hat{\tau}_r^{(\top,\top)} \sqcup \hat{\tau}_r^{(l,\top)} \sqcup \hat{\tau}_r^{(l',n')}$
$\quad\quad \rho[\alpha] \doteq \hat{\tau}_r^{\alpha}$
$\quad$ **procedure** $\textsc{Eval}_\varsigma(\mathit{Constraint}, \rho, \varsigma)$
$\quad\quad \Sigma \doteq \rho[\varsigma]$
$\quad\quad$ **for all** $m \in \mathit{Constraint}[\varsigma]$ **do**
$\quad\quad\quad$ **if** $m = \varsigma \doteq \Sigma'$ **then** $\Sigma := \Sigma' \sqcap \Sigma$
$\quad\quad\quad$ **if** $m = \varsigma \doteq \bigsqcup \varsigma'_1 \ldots \varsigma'_m$ **then** $\Sigma := (\rho[\varsigma'_1] \sqcup \ldots \sqcup \rho[\varsigma'_m]) \sqcap \Sigma$
$\quad\quad\quad$ **if** $m = \varsigma \doteq \varsigma'[\alpha_1 \mapsto \alpha_2]$ **then** $\Sigma' := \rho[\varsigma'] \wedge \hat{\tau}_r^{\mathrm{ptr}} := \rho[\alpha_1] \wedge \hat{\tau}_r^{\mathrm{data}} := \rho[\alpha_2]$
$\quad\quad\quad\quad$ **if** $\hat{\tau}_r^{\mathrm{ptr}} = \mathrm{ptr}(l, n)$ **then** $\Sigma := \Sigma \sqcap \Sigma'[(l, n) \mapsto \hat{\tau}_r^{\mathrm{data}}]$ **else** $\Sigma := \Sigma \sqcap \Sigma'$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(\top, \top) \mapsto \hat{\tau}'_r \in \Sigma$ **then** $\Sigma := \Sigma[(\top, \top) \mapsto \hat{\tau}'_r \sqcup \hat{\tau}_r^{\mathrm{data}}]$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(l, \top) \mapsto \hat{\tau}'_r \in \Sigma$ **then** $\Sigma := \Sigma[(l, \top) \mapsto \hat{\tau}'_r \sqcup \hat{\tau}_r^{\mathrm{data}}]$
$\quad\quad\quad\quad$ **if** $\exists\, \mathrm{ptr}(l', n') \mapsto \hat{\tau}'_r \in \Sigma$ **then**
$\quad\quad\quad\quad\quad$ **if** $\nexists\, l \neq l' \in \mathit{Constraint}[\alpha']$ **then** $\Sigma := \Sigma[(l', o') \mapsto \hat{\tau}'_r \sqcup \hat{\tau}_r^{\mathrm{data}}]$
$\quad\quad\quad\quad\quad$ **else if** $\exists\, l \neq l' \in \mathit{Constraint}[\alpha'] \wedge n' = \top$ **then** $\Sigma := \Sigma[(l', o') \mapsto \hat{\tau}'_r \sqcup \hat{\tau}_r^{\mathrm{data}}]$
$\quad\quad \rho[\varsigma] \doteq \Sigma$

WebAssembly Function	*Value Stack*	*Generated Constraints*
`(func $foo`		
`(param i32 i32)`		$\alpha_0 <: \text{i32} \land \alpha_1 <: \text{i32} \land \varsigma \doteq [\,]$
`(result i32)`		$\alpha_5 <: \text{i32}$
`(local i32 i32 i32)`		$\alpha_2 <: \text{i32} \land \alpha_3 <: \text{i32} \land \alpha_4 <: \text{i32}$
`local.get 0`	$[\alpha_0]$	
`local.get 1`	$[\alpha_0\alpha_1]$	
`i32.add`	$[\alpha_6]$	$\alpha_6 \doteq \text{i32.add}(\alpha_0, \alpha_1, \_) \land \alpha_0 \doteq \text{i32.add}(\_, \alpha_1, \alpha_6) \land \alpha_1 \doteq \text{i32.add}(\alpha_0, \_, \alpha_6)$
`i32.load`	$[\alpha_7]$	$\alpha_6 <: \text{ptr} \land \alpha_7 <: \text{i32} \land \alpha_7 \doteq \varsigma[\alpha_6]$
`local.set 2`	$[\,]$	$\alpha_2 \doteq \alpha_7$
`local.get 1`	$[\alpha_1]$	
`i32.const 4`	$[\alpha_1\alpha_8]$	$\alpha_8 \doteq \text{i32}(4)$
`i32.add`	$[\alpha_9]$	$\alpha_9 \doteq \text{i32.add}(\alpha_1, \alpha_8, \_) \land \alpha_1 \doteq \text{i32.add}(\_, \alpha_8, \alpha_9) \land \alpha_8 \doteq \text{i32.add}(\alpha_1, \_, \alpha_9)$
`i32.load`	$[\alpha_{10}]$	$\alpha_9 <: \text{ptr} \land \alpha_{10} <: \text{i32} \land \alpha_{10} \doteq \varsigma[\alpha_9]$
`local.set 3`	$[\,]$	$\alpha_3 \doteq \alpha_{10}$
`local.get 0`	$[\alpha_0]$	
`i32.const 42`	$[\alpha_0\alpha_{11}]$	$\alpha_{11} \doteq \text{i32}(42)$
`i32.mul`	$[\alpha_{12}]$	$\alpha_{12} \doteq \text{i32.mul}(\alpha_0,\alpha_{11},\_) \land \alpha_0 \doteq \text{i32.mul}(\_,\alpha_{11},\alpha_{12}) \land \alpha_{11} \doteq \text{i32.mul}(\alpha_0,\_,\alpha_{12})$
`local.set 4`	$[\,]$	$\alpha_4 \doteq \alpha_{12}$
`local.get 2`	$[\alpha_2]$	
`i32.eqz`	$[\alpha_{13}]$	$\alpha_{13} \doteq \text{i32.eqz}(\alpha_2)$
`(if (param) (result i32)`		
`(then`		
`local.get 3)`	$[\alpha_3]$	
`(else`		
`local.get 4)`	$[\alpha_4]$	$\alpha_{14} \doteq \sqcup\alpha_3\alpha_4$
`))`	$[\alpha_{14}]$	$\alpha_{14} \doteq \alpha_5$

High-Level Pseudocode for WebAssembly:

```
int foo (x, y) {
  a = *(x + y);
  b = *(y + 4);
  c = x * 42;
  if a == 0 { b }
  else { c }
}
```

Fig. 13. WebAssembly function with generated constraints.