Constructing Call Graphs for WebAssembly using Refined Numeric Type Analysis

ANONYMOUS AUTHOR(S)

WebAssembly has become a widely used compilation target for languages such as C/C++, Rust, and Go, enabling high-performance applications to run in browsers and other sandboxed environments. Static analysis of WebAssembly is important for tasks such as code debloating, performance optimization, and detection of security vulnerabilities. We present *Refined Numeric Type Analysis (RNTA)*, a refinement-type-based technique for constructing WebAssembly call graphs that are more precise than those obtained using previous techniques. WebAssembly's low-level nature creates two significant challenges for static analysis: (i) memory is represented as an unstructured array of bytes, and (ii) indirect calls are encoded as numeric i32 values indexing function tables. Traditional pointer analysis techniques rely on associating sets of objects with pointers and cannot be applied to WebAssembly, where the concepts of pointers and objects do not exist. Existing call graph construction algorithms for WebAssembly associate a set of functions with each `call_indirect` instruction using function signatures, but suffer from the limited nature of WebAssembly's type system, which reduces opportunities for ruling out functions as call targets. The foundation of RNTA is a refinement type system that distinguishes i32 values that are used to index memory—i.e., pointers—from other i32 values and accordingly infers a refinement signature for each function. RNTA computes call graphs on-the-fly by using the refinement types inferred at indirect calls to approximate possible call targets. We prove the type system sound and show that RNTA produces call graphs that are significantly more precise than those computed by the previous state-of-the-art. Across a corpus of 20 real-world WebAssembly binaries, RNTA reduces reachable functions by 9.9% and call graph edges by 14.6% while increasing indirect calls resolved to a unique target by 12.6%.

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

Additional Key Words and Phrases: WebAssembly, Static Analysis, Call Graph Analaysis, Pointer Analysis

1 INTRODUCTION

WebAssembly [24] is a portable, low-level bytecode format that was designed to run computationally intensive tasks in browsers alongside JavaScript host code. It has become widely used as a compilation target for C/C++, Rust, and Go, so that libraries written in those languages can be invoked directly from within a web application. Our long-term goal is to develop static analysis techniques for WebAssembly that can be used for code-size reduction, optimization, and detection of security vulnerabilities. As a first step towards this goal, this paper presents a *Refined Numeric Type Analysis (RNTA)*, a refinement-type-based technique for constructing WebAssembly call graphs that are more precise than those obtained using previous techniques.

During the past 30+ years, a vast literature of algorithms for pointer analysis and call graph construction for high-level object-oriented and imperative programming languages has been developed (see, e.g., [5–7, 10, 17, 22, 23, 38, 40, 45]. In essence, these techniques involve associating sets of abstract objects with abstract pointers. Unfortunately, since WebAssembly is a low-level language that features neither objects nor pointers, these traditional pointer-analysis algorithms cannot be used. Furthermore, analysis at the source level is often insufficient: WebAssembly modules are frequently compiled from libraries, and many optimization opportunities only become apparent once it is known how a host application uses the library's features. Additionally, many web applications only include the binaries of the WebAssembly libraries that they use, leaving out their source code and the provenance of the binaries is often unclear.

WebAssembly's low-level type system comprises only[1] 32- and 64-bit integer (i32, i64) and floating-point (f32, f64) types, presenting significant challenges to call graph construction. Crucially,

[1]In this paper, we focus on WebAssembly 1.0 as the first and most widely deployed version of WebAssembly[25].

i32 values serve multiple purposes: indexing memory, controlling conditionals, indexing function tables, and performing arithmetic operations. As a result of this impoverished type system, many functions share the same type, so type-based filtering rarely identifies "monomorphic" call sites that have a single target, limiting key optimizations like call devirtualization and inlining.

In WebAssembly, indirect calls are mediated through a function table, where an integer index is used to select a specific function from the table. The reader might therefore think that a value analysis could be used to determine the index used in indirect calls. However, Lehmann et al. [29] identify several unique challenges to WebAssembly call graph analysis, validated on 8,461 WebAssembly binaries [25]. They find that, in nearly all cases, the dispatch of function calls involves non-constant values with complex data-flow (interprocedural data-flow or loads from memory addresses determined at runtime), complicating call graph construction. As a result, existing static analyses for WebAssembly have made limited progress beyond leveraging WebAssembly's type system. Wasma [11] uses type-based filtering. Wassail [42] additionally restricts its analysis to constant indices, which empirical evidence shows rarely occur in practice [29]. Sturdy [28] employs abstract interpretation to track the data flow of abstract integers inter-procedurally as well as through constant memory addresses. However, integer values frequently flow through dynamically assigned memory addresses, leading to significant loss of precision with this approach.

In this paper, we present a refinement type system that refines i32 types to either a singleton pointer type $\mathsf{ptr}(l, n)$ (i.e., a pointer to symbolic location l at abstract offset n) or a singleton number $\mathsf{num}(n)$. Additionally, we type WebAssembly memory, which the original type system leaves untyped, enabling precise tracking of integer values as they flow through memory. Our approach is inspired by recent work on inferring richer types for LLVM IR [31, 32] and binary code [8]. Like those approaches, we enrich the type information of a low-level language. However, our refinement type system is designed specifically for WebAssembly and to enable more precise resolution of indirect calls. This refinement yields more precise function signatures: functions accepting pointers are now distinguished from those accepting numbers. As a result, type-based filtering at indirect call sites becomes significantly more effective because fewer functions share identical signatures. We prove the refinement type system to be type-safe, another distinction from LLVM type analyses.

Our new type system forms the foundation for *Refined Numeric Type Analysis*, an efficient call graph construction algorithm for WebAssembly, which we implement in a tool called RNTA and evaluate on 20 real-world WebAssembly binaries. The results of our experiments show that, on average, when compared against a baseline analysis that uses WebAssembly's standard type system, RNTA reduces reachable functions by 9.9% and call graph edges by 14.6% while increasing indirect calls that are resolved to a unique target by 12.6%.

In summary, this paper makes the following contributions:

(1) a refinement type system for WebAssembly that refines the value type i32 to distinguish between numbers and pointers to WebAssembly memory,
(2) RNTA, a call graph construction algorithm that leverages this refinement type system,
(3) an empirical evaluation of RNTA on 20 binaries showing, on average, reductions in the number of reachable functions and call graph edges by 9.9% and 14.6%, respectively, while increasing indirect calls that are resolved to a unique target by 12.6%, and
(4) a proof that the Refinement Type System is type-safe.

The rest of this paper is organized as follows. Section 2 provides background and conveys the intuition behind our approach. Section 3, 5 and 4 cover the refinement type system. An empirical evaluation of RNTA is presented in Section 6. Section 7 presents related work and Section 8 concludes.

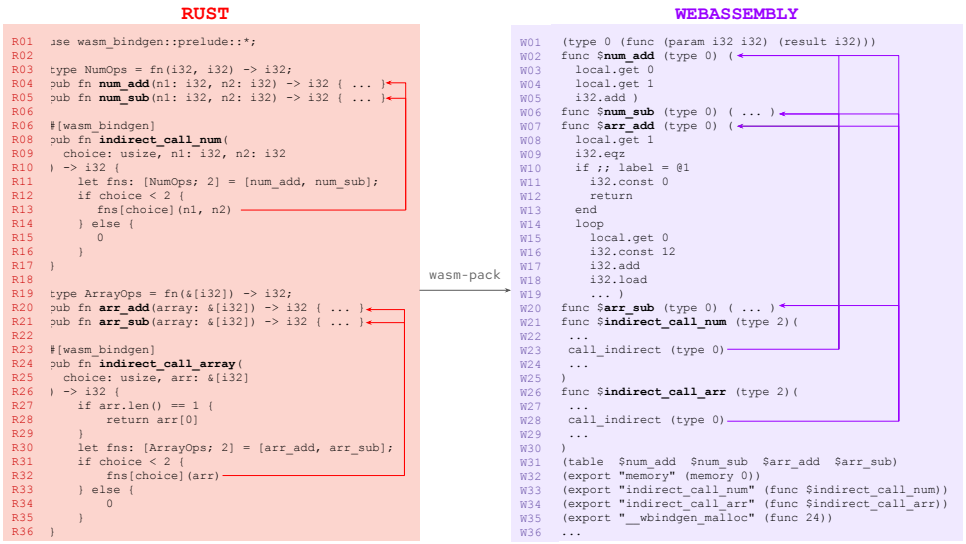


Fig. 1. A Rust Library compiled to WebAssembly.

2 BACKGROUND AND MAIN IDEAS

2.1 Call Graph Analysis Over the WebAssembly Binary

A call graph analysis over the source code of the Rust library in Figure 1 will resolve the targets for the indirect calls at R13 and R32 as num_add, num_sub and arr_add, arr_sub, respectively. Unfortunately, a call graph analysis over the corresponding WebAssembly binary produced by the compiler is not as precise. Indirect calls in WebAssembly make use of the call_indirect instruction and are mediated through a WebAssembly table that contains a list of functions (line W31). 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 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 [48] take this approach.
(2) *Type-based Analysis:* The syntax of a call_indirect instruction contains a function type annotation (line W23) which is guaranteed to match the function type of the indirect call target. State-of-the-art analysis tools like Wassail [42] and Wasma [11] 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 [33] and Sturdy [28] perform a 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 W23 and W28 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 subtract operations all compile down to have the same function type type0 and the WebAssembly table contains functions for all of these operations. Additionally, the index into the table is provided by the user at runtime, and so is unknown. Hence, a static analysis over this binary would resolve all four operations to be potential targets for both indirect calls. In WebAssembly, the type system is not rich enough to distinguish between different uses of i32s. In particular, there is no way to distinguish between i32s

that represent numbers and i32s that represent pointers to the WebAssembly memory and so, there is no way to distinguish num_add from arr_add.

2.2 Refining the WebAssembly Type System With Pointers

WebAssembly does not distinguish between numbers and pointers. All i32 values (32-bit integers) serve multiple purposes: indexing memory, controlling conditionals, indexing the function table, and arithmetic operations. However, we hypothesize that i32 values used as pointers are consistently used to load from and store to memory. For example, in Figure 1, function arr_add treats its first parameter (accessed using local.get 0) as a pointer—adding an offset of 12 and loading from that memory location at W15 – W18. The second parameter (accessed using local.get 1) acts as an array size, triggering an early return at W08 – W13 when zero. This corresponds to how WASM-PACK compiles Rust arrays as a pointer-size pair.

Correspondingly, we refine i32 to two new types, ptr and num, both subtypes of i32, where ptr represents pointers to the WebAssembly memory. We also change the type system to be a refinement type system [14, 19, 21, 35] 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 3.2 and are not extensive. We only refine i32's and do not refine the other base WebAssembly types, i64, f32 and f64. 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.*

To discover these refinement types, we first generate constraints over the WebAssembly binary under inspection. These constraints encode how data is used. For instance, if a WebAssembly instruction loads from or stores to an i32 then that type should be refined to a pointer. For simplicity, we generate constraints over WAFFLE [13], an intermediate representation for WebAssembly that uses Static Single Assignment Form [15]. We then solve the constraints, while performing a value and pointer analysis, to obtain refinement types over the Wasm binary. We then apply a type-based call graph analysis over the refinement type system. We explain the details of this process in Section 3 and 5. Note that we only generate and solve constraints over WebAssembly 1.0.

3 A SYMBOLIC 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.[24] and the WebAssembly 1.0 specification [47]. Instructions in WebAssembly operate on an implicit value stack by popping argument values and pushing computed results. As shown in Figure 2, the instruction typing judgment in WebAssembly has the form $C^{w} \vdash e^{*} : \tau_{w}^{m} \rightarrow \tau_{w}^{m}$ 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 $\tau_{w}^{\prime j}$ (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 2. The figure also contains the typing rule for binary operations, which expect two values of

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

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

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

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

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

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

Fig. 2. Typing of a subset of WebAssembly Instructions in the Original Type System.

the type they are annotated with on the stack and produce a value of 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 [16] 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 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 Symbolic Refinement Type System

We present a refinement over the original WebAssembly (version 1.0) type system in Figure 3. We discuss each change to the type system below:

(1) *Symbolic Refinement Types, $\hat{\tau}_r$*: 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, where n is an abstract number. A refinement

Refinement Type System

Abstract Number	n		
Symbolic Location	l		
Abstract Refinement Type	α		
Symbolic Refinement Type	$\hat{\tau}_\mathrm{r}$	::=	$\mathsf{i32}(n) \mid \mathsf{ptr}(l, n) \mid \mathsf{num}(n) \mid \mathsf{i64}(n) \mid \mathsf{f32}(n) \mid \mathsf{f64}(n)$
Type Tag	τ	::=	$\mathsf{i32} \mid \mathsf{ptr} \mid \mathsf{num} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
Original Value Types	τ_w	::=	$\mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
Refinement Function Type	$tf_{\hat{\tau}_\mathrm{r}}$	::=	$\hat{\tau}_\mathrm{r}^* \to \hat{\tau}_\mathrm{r}^*$
Symbolic Function Type	tf_α	::=	$\alpha^* \to \alpha^*$
Symbolic Global Type	tg_α	::=	$\mathsf{mut}^? \ \alpha^*$
Abstract Memory Typing	ς		
Symbolic Memory Typing	Σ	::=	$\cdot \mid \Sigma, (l, n) \mapsto \hat{\tau}_\mathrm{r}$

Constraint System

Operations	op	::=	$unop_{iN} \mid unop_{fN} \mid cvtop \mid binop_{iN} \mid binop_{fN} \mid testop_{iN} \mid relop_{iN} \mid relop_{fN}$
Type Constraint	t	::=	$\alpha \doteq \hat{\tau}_\mathrm{r} \mid \alpha <: \tau \mid \alpha \doteq \alpha \mid \alpha \doteq \bigsqcup \alpha^+ \mid \alpha \doteq \varsigma[\alpha]_{(tp\_sx^?,\ a,\ o)} \mid \alpha \doteq op((\alpha^?)^+)$
Memory Constraint	m	::=	$\varsigma \doteq \Sigma \mid \varsigma \doteq \varsigma \mid \varsigma \doteq \bigsqcup \varsigma^+ \mid \varsigma \doteq \varsigma[\alpha \mapsto \alpha]_{(tp^?,\ a,\ o)}$
Constraint Set	S	::=	$\cdot \mid S, t \mid S, m$

Fig. 3. Syntax for Symbolic Refinement Types and Constraints Generated Over WebAssembly.

type such as $\mathsf{i32}(42)$, in a more traditional presentation of refinement types, might be written as $\{n\colon \beta \mid \beta <: \mathsf{i32} \wedge n = 42\}$. For types $\mathsf{i32}(n)$ and $\mathsf{f32}(n)$, $n \in \{\mathbb{N}_{32} \cup \{\top, \bot\}\}$, while for $\mathsf{i64}(n)$ and $\mathsf{f64}(n)$, $n \in \{\mathbb{N}_{64} \cup \{\top, \bot\}\}$.

(2) *Pointers and Numbers*: Since i32s are used both as pointers into the WebAssembly memory and numbers, we add two new types ptr and num, where ptr and num are subtypes of i32. Pointers are represented symbolically as $\mathsf{ptr}(l, n)$ with a symbolic base address l and symbolic offset n, allowing us to track pointer arithmetic without knowing concrete addresses. The symbolic address l_c is used for constant memory addresses. Numbers are represented as $\mathsf{num}(n)$ with a symbolic value. The types $\mathsf{i32}(n)$, $\mathsf{ptr}(n)$, and $\mathsf{num}(n)$ form a lattice, the subtyping relation and meet and join operations for which are discussed in Section 3.4. We only refine i32 because the other WebAssembly value types (i64, f32, f64) are not used to index memory.

(3) *Symbolic Memory Typing,* Σ: Unlike the original WebAssembly type system, we type the flat contiguous array of bytes that serves as the WebAssembly linear memory. Since memory operations are ubiquitous in WebAssembly, we need to be able to recover the (refinement) types of values that are stored in memory at a specific address. A memory typing Σ is a mapping from symbolic location, offset pairs (l, n) to symbolic refinement types $\hat{\tau}_\mathrm{r}$.

(4) *Refinement Function Types,* $tf_{\hat{\tau}_\mathrm{r}}$: 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 these constructs. In our refinement type system, a refinement function type $tf_{\hat{\tau}_\mathrm{r}}$ specifies not only the lists of (input and output) refinement types τ_r, but also the shape of the (input and output) memory typing Σ. 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 i32 at location $(l, 0)$, because it reads from $(l, 0)$ using a i32.load instruction, might later store an f32 at the same location.

(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 refinement types $\hat{\tau}_r$ on the stack but also the expected memory typing Σ for the control construct in the label.

3.3 Constraint Language to Discover Refinement Types

In this section, we explain how we discover symbolic refinement types for WebAssembly code. WebAssembly's type validation algorithm, described in the specification [47], 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. For example, in function arr_add (Figure 1), the parameters' types are not immediately clear. Only at line W18, where the first parameter accesses memory, do we discover it is a pointer. We must then propagate this type backward to the function signature and other uses. Thus, a forward-only analysis is insufficient—we need types to propagate backwards too.

We generate constraints over WebAssembly functions and solve them in a fixed-point algorithm to get *symbolic* refinement types $\hat{\tau}_r$ for every stack slot in the value stack, 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 4.3.

To generate constraints, we first 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 WAFFLE [13], an SSA IR over WebAssembly, since SSA form allows us to type different updates to a variable with distinct refinement types. For each instruction, we generate *type constraints* over αs and *memory constraints* over ςs, the syntax of which can be seen in Figure 3. A subset of the constraint-generation rules for WebAssembly instructions are described in Figure 4, 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 $\alpha_1 = \mathsf{ptr}(l, n_1)$ and $\alpha_2 = \mathsf{i32}(n_2)$, α_2 can be refined to be $\mathsf{num}(n_2)$, since the alternative is not permitted. In fact, the types of each argument and result can affect each other. If $\alpha_1 = \mathsf{num}(n_1)$, $\alpha_2 = \mathsf{i32}(n_2)$ and $\alpha_3 = \mathsf{ptr}(l, n_3)$, α_2 is refined to be ptr. Meanwhile, if $\alpha_3 = \mathsf{num}(n_3)$, 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, $\alpha_1 \doteq \mathsf{i32.add}(\_, \alpha_2, \alpha_3)$, $\alpha_2 \doteq \mathsf{i32.add}(\alpha_1, \_, \alpha_3)$ and $\alpha_3 \doteq \mathsf{i32.add}(\alpha_1, \alpha_2, \_)$. 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 $\alpha_{\mathsf{ptr}} <: \mathsf{ptr}$, while the data is constrained to be a subtype of the type expected by the store instruction, $\alpha_{\mathsf{data}} <: \mathsf{i64}$. The abstract memory typing represented by ς, an input to the rule, is updated to record

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

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

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

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

$$\frac{tf_{\text{w}} = \tau_{\text{w}}^{n} \to \tau_{\text{w}}^{m} \qquad \alpha^{n} \in dom(S) \qquad \alpha^{m}, \varsigma' \text{ fresh} \qquad S; \varsigma; C, \text{label}(\alpha^{m}, \varsigma') \vdash e^{*} : \alpha^{n} \to (\alpha')^{m}; S'; \varsigma'' \qquad S'' = S' :: [(\alpha \doteq \alpha')^{m} \wedge \varsigma' \doteq \varsigma'']}{S; \varsigma; C^{\alpha} \vdash \textbf{block } tf_{\text{w}}\ e^{*}\ \textbf{end} : \alpha^{n} \to \alpha^{m}; S''; \varsigma'}\textsc{Block}$$

$$\frac{\alpha^{*}\alpha^{n} \in dom(S) \qquad C^{\alpha}_{\text{label}}(i) = \alpha'_{n}, \varsigma' \qquad S' = S :: [\varsigma \doteq \sqcup\varsigma\varsigma' \wedge (\alpha \doteq \sqcup\alpha\alpha')^{n}]}{S; \varsigma; C^{\alpha} \vdash \textbf{br } i : \alpha^{*}\alpha^{n} \to \alpha^{*}; S'; \varsigma}\textsc{Break}$$

$$\frac{C^{\alpha}_{\text{func}} = \tau_{\text{w}}^{m} \to \tau_{\text{w}}^{n} \qquad \alpha^{n} \in dom(S) \qquad S' = S :: [(\alpha <: \tau_{\text{w}})^{m} \wedge (\alpha' <: \tau_{\text{w}})^{n}]}{S; \varsigma; C^{\alpha} \vdash \textbf{call} : \alpha^{m} \to (\alpha')^{n}; S'; \varsigma}\textsc{Call}$$

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

Fig. 4. Constraint-Generation Rules for a Subset of WebAssembly Instructions.

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 $tf_{\text{w}}\ e^{*}$ 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 whether the 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 $tf_{\text{w}}\ e^{*}$ end. If $tf_{\text{w}} = \text{i32} \to ()$, 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 label(α_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 Constraint Solving

We solve a set of constraints generated over a WebAssembly function using a standard fixpoint algorithm, described in detail in Figure 1. 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 maximum fixed point solution for every α and ς in S using a standard worklist algorithm. The constraints on α's are solved to get a symbolic refinement type, $\hat{\tau}_\mathrm{r}$, and the constraints on ς's are solved to get a memory state, Σ. If there exists a $\alpha \doteq \hat{\tau}_\mathrm{r}$ in the constraints for α, we initialize the solution for the α to be $\hat{\tau}_\mathrm{r}$. 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 discuss solving type and memory constraints below.

3.4.1 Solving Type Constraints. To solve the type constraints for a given α, we iterate over each constraint in *Constraint*$[\alpha]$ and progressively narrow the current solution by taking its meet with the type computed from each constraint.

(1) For a *subtype constraint* $\alpha <: \tau$, the solution is met with $\tau(\top)$, restricting α to the correct base type while keeping its value refinement as precise as possible.
(2) For an *operation constraint* $\alpha \doteq op(\alpha'_1, \ldots, \alpha'_n)$, we evaluate op on the current solutions of the α'_i and meet the result into α's solution.
(3) For a *join constraint* (generated over blocks and loops) $\alpha \doteq \bigsqcup \alpha'_1 \ldots \alpha'_m$, we join the current solutions of all α'_i and meet that join into α's solution, capturing merge points in control flow.
(4) The *memory load constraint* $\alpha \doteq \varsigma[\alpha']$ is the most complex case due to potential aliasing. We resolve the current memory state $\Sigma = \rho[\varsigma]$ and pointer type $\hat{\tau}_\mathrm{r}^{\mathrm{ptr}} = \rho[\alpha']$. If $\hat{\tau}_\mathrm{r}^{\mathrm{ptr}} = \mathrm{ptr}(l, n)$ and $(l, n) \mapsto \tau_r \in \Sigma$, we read τ_r as the primary result. We then conservatively identify all potential aliases in Σ: (1) $\mathrm{ptr}(\top, \top)$, a fully-unknown pointer that could alias anything; (2) $\mathrm{ptr}(l, \top)$, a pointer to the same allocation but an unknown offset; and (3) any other entry $\mathrm{ptr}(l', n')$, which may alias $\mathrm{ptr}(l, n)$ since the relationship between allocation labels cannot be determined statically. The final solution for α is the meet of its current value with the join of all these contributions, ensuring that any type stored at a possible alias location is accounted for.

3.4.2 Solving Memory Constraints. Solving the memory constraints for a given ς iterates over each constraint in *Constraint*$[\varsigma]$, progressively updating the current memory state $\Sigma = \rho[\varsigma]$ via meet operations.

(1) For a *memory equality constraint* $\varsigma \doteq \Sigma'$, we meet the given concrete state Σ' into Σ.
(2) For a *join constraint* $\varsigma \doteq \bigsqcup \varsigma'_1 \ldots \varsigma'_m$, we join the current solutions of all ς'_i and meet the result into Σ, capturing merge points in control flow.

(3) The *memory store constraint* $\varsigma \doteq \varsigma'[\alpha_1 \mapsto \alpha_2]$ requires the most care. We resolve the base state $\Sigma' = \rho[\varsigma']$, the pointer $\hat{\tau}_{\mathrm{r}}^{\mathrm{ptr}} = \rho[\alpha_1]$, and the data type $\hat{\tau}_{\mathrm{r}}^{\mathrm{data}} = \rho[\alpha_2]$. If the pointer is a concrete $\mathrm{ptr}(l, n)$, we perform a **strong update**, meeting Σ with Σ' modified to map (l, n) to $\hat{\tau}_{\mathrm{r}}^{\mathrm{data}}$; otherwise we meet Σ with the unmodified Σ'. Note that a concrete pointer might be a constant memory address with a statically known offset or a symbolic address with a statically known offset. We then handle aliasing conservatively: if Σ already contains an entry for $\mathrm{ptr}(\top, \top)$, $\mathrm{ptr}(l, \top)$, or any other $\mathrm{ptr}(l', n')$ whose relationship to $\mathrm{ptr}(l, n)$ cannot be determined statically, we join $\hat{\tau}_{\mathrm{r}}^{\mathrm{data}}$ into the type at that alias entry, reflecting the possibility that the store may have updated it. This ensures that a store through an ambiguous pointer is reflected at all locations it might affect.

The precision of our memory analysis depends on how much is statically known about the pointer used in a load or store. When a function takes a pointer as a parameter and accesses memory exclusively through statically known immediate offsets (encoded directly in the instruction, eg, i32.load offset = 8), the pointer is represented as $\mathrm{ptr}(l, \mathsf{i32}(8))$, i.e., a pointer with a symbolic location l and concrete offset $\mathsf{i32}(8)$, enabling precise, strong-update semantics for every load and store in the function body. Precision degrades when a dynamic offset is used, or when a function accesses memory through two or more pointers whose relationship cannot be determined statically: since no information is available to distinguish their allocation labels, our analysis must conservatively assume they may alias. Precisely characterising aliasing among dynamically computed addresses is undecidable in general [26], so our analysis must overapproximate when aliases cannot be resolved statically. In such cases, the result type of a load, or the updated memory state after a store, is widened to account for all potential aliases.

3.4.3 Refinement Type Lattice and Soundness. The precision and soundness of the analysis depends on the Refinement Type Lattice, which we now explain. The WebAssembly base types that have not been refined beyond the addition of an abstract value (i64, f32, f64) have sub-lattices defined as follows: $\mathcal{L}_{\tau_{\mathrm{w}}} = \tau_{\mathrm{w}}(\mathbb{N}_{|\tau_{\mathrm{w}}|}) \cup \{\tau_{\mathrm{w}}(\top), \tau_{\mathrm{w}}(\bot)\}$. The i32 sub-lattice is more complex, containing ptr and num as shown in Figure 5 with a portion of its meet and join operations, which we discuss in detail shortly. The entire type lattice is defined as $\mathcal{L} = \{\top, \bot\} \cup \mathcal{L}_{\mathsf{i32}} \cup \mathcal{L}_{\mathsf{i64}} \cup \mathcal{L}_{\mathsf{f32}} \cup \mathcal{L}_{\mathsf{f64}}$. The lattice has finite height, ensuring that constraint solving terminates. Our constraint generation rules guarantee that every α has a valid base WebAssembly type. Since constraint solving computes the greatest fixed point, the analysis is sound: in the worst case, conflicting constraints cause the refinement to collapse to an unrefined base type with no value refinement. For example, if α is constrained to be both ptr and num, their meet loses the refinement but retains the base i32 type (Figure 5, last rule). We discuss this case and other interesting meet and join cases of the i32 sub-lattice below.

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 $\mathrm{ptr}(l_1, n_1)$, and the else branch returns $\mathrm{ptr}(l_2, n_2)$. 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 $\mathrm{ptr}(\top, \top)$. Instead, if the then branch returned $\mathrm{ptr}(l, n_1)$ and the else branch returned $\mathrm{ptr}(l, n_2)$, we say that the if-then-else expression returns $\mathrm{ptr}(l, n_1 \sqcap n_2)$.

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 $\mathrm{ptr}(l_1, n_1)$, a pointer with a symbolic address, and at loop iteration $n+1$, its type is $\mathrm{ptr}(l_c, n_0)$, a constant pointer. Since it is impossible for a stack slot to have a symbolic and constant pointer, we return $\mathrm{ptr}(\bot, \bot)$. On the other hand, say that at loop iteration n, the type of a stack slot is $\mathrm{ptr}(l_1, n_1)$ and at loop iteration $n+1$,

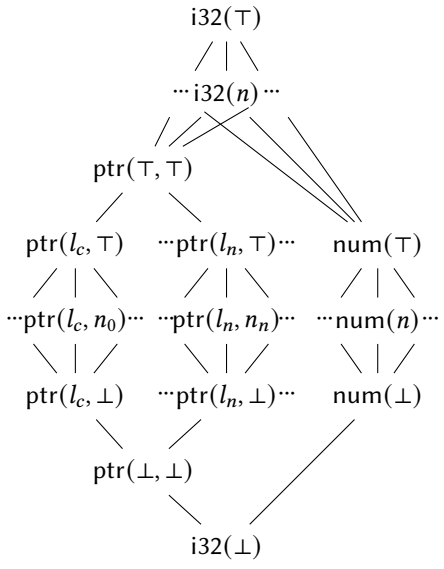
Algorithm 1 Algorithm for solving constraints described in Figure 3

Require: $Constraint = [\alpha \mapsto t^+; \varsigma \mapsto m^+]$, $infl = [\alpha \mapsto \{\alpha^*, \varsigma^*\}; \varsigma \mapsto \{\alpha^*, \varsigma^*\}]$
Ensure: The least Solution $\rho = [\alpha \mapsto \hat{\tau}_\mathrm{r}; \varsigma \mapsto \Sigma]$

```
Worklist ← dom(Constraint)
for all α ∈ dom(Constraint) do
    if ∃ α ≐ τ̂_r ∈ Constraint[α] then ρ[α] ≐ τ̂_r else ρ[α] = ⊤
for all ς ∈ dom(Constraint) do ρ[ς] = ·
while Worklist ≠ ∅ do
    v := Worklist.pop()
    Old_v := ρ[v]
    if v = α then EVAL_α(Constraint, ρ, α)
    if v = ς then EVAL_ς(Constraint, ρ, ς)
    if Old_v ≠ ρ[v] then Worklist.append(infl[v])
procedure EVAL_α(Constraint, ρ, α)
    τ̂_r^α := ρ[α]
    for all t ∈ Constraint[α] do
        if t = α <: τ then τ̂_r^α := τ̂_r^α ⊓ τ(⊤)
        if t = α ≐ op(α'_1, ..., α'_n) then τ̂_r^α := τ̂_r^α ⊓ eval_op(ρ[α'_1], ..., ρ[α'_n])
        if t = α ≐ ⨆ α'_1 ... α'_m then τ̂_r^α := τ̂_r^α ⊓ (ρ[α'_1] ⊔ ... ⊔ ρ[α'_m])
        if t = α ≐ ς[α'] then Σ := ρ[ς] ∧ τ̂_r^ptr ≐ ρ[α']
            if τ̂_r^ptr = ptr(l, n) ∧ (l, n) ↦ τ̂_r ∈ Σ then τ̂_r^(l,n) := τ̂_r else τ̂_r^(l,n) := ⊤
            if ∃ ptr(⊤, ⊤) ↦ τ̂_r ∈ Σ then τ̂_r^(⊤,⊤) := τ̂_r else τ̂_r^(⊤,⊤) := ⊥
            if ∃ ptr(l, ⊤) ↦ τ̂_r ∈ Σ then τ̂_r^(l,⊤) := τ̂_r else τ̂_r^(l,⊤) := ⊥
            if ∃ ptr(l', n') ↦ τ̂_r ∈ Σ then τ̂_r^(l',n') := τ̂_r else τ̂_r^(l,⊤) := ⊥
            τ̂_r^α := τ̂_r^α ⊓ τ̂_r^(l,n) ⊔ τ̂_r^(⊤,⊤) ⊔ τ̂_r^(l,⊤) ⊔ τ̂_r^(l',n')
    ρ[α] ≐ τ̂_r^α
procedure EVAL_ς(Constraint, ρ, ς)
    Σ ≐ ρ[ς]
    for all m ∈ Constraint[ς] do
        if m = ς ≐ Σ' then Σ := Σ' ⊓ Σ
        if m = ς ≐ ⨆ ς'_1 ... ς'_m then Σ := (ρ[ς'_1] ⊔ ... ⊔ ρ[ς'_m]) ⊓ Σ
        if m = ς ≐ ς'[α_1 ↦ α_2] then Σ' := ρ[ς'] ∧ τ̂_r^ptr := ρ[α_1] ∧ τ̂_r^data := ρ[α_2]
            if τ̂_r^ptr = ptr(l, n) then Σ := Σ ⊓ Σ'[(l, n) ↦ τ̂_r^data] else Σ := Σ ⊓ Σ'
            if ∃ ptr(⊤, ⊤) ↦ τ̂'_r ∈ Σ then Σ := Σ[(⊤, ⊤) ↦ τ̂'_r ⊔ τ̂_r^data]
            if ∃ ptr(l, ⊤) ↦ τ̂'_r ∈ Σ then Σ := Σ[(l, ⊤) ↦ τ̂'_r ⊔ τ̂_r^data]
            if ∃ ptr(l', n') ↦ τ̂'_r ∈ Σ then Σ := Σ[(l', o') ↦ τ̂'_r ⊔ τ̂_r^data]
    ρ[ς] ≐ Σ
```

its type is $\mathrm{ptr}(l_2, n_2)$. This means that the type at this stack slot is both $\mathrm{ptr}(l_1, n_1)$ and $\mathrm{ptr}(l_2, n_2)$. We equate these two pointers to a third pointer $\mathrm{ptr}(l_3, 0)$, where l_3 is a fresh symbolic location and $\mathrm{ptr}(l_1, n_1) = \mathrm{ptr}(l_3, 0) \wedge \mathrm{ptr}(l_2, n_2) = \mathrm{ptr}(l_3, 0)$. 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 $\mathrm{ptr}(l_3, 0)$.

Join and Meet of a pointer and a number. The join of a pointer with a number results in an $\mathsf{i32}(\top)$, except in the case of constant pointers $\mathrm{ptr}(l_c, n_0)$. For a constant pointer, we know that the base address l_c equates to 0 and so the operation $\mathrm{ptr}(l_c, n_0) \sqcup \mathsf{num}(n_1)$, produces $\mathsf{i32}(n_0 \sqcup n_1)$ as



$$
\begin{array}{lcll}
\mathsf{i32}(n_1) & \sqcup & \mathsf{i32}(n_1) & = \mathsf{i32}(n_1 \sqcup n_2) \\
\mathsf{i32}(n_1) & \sqcup & \mathsf{ptr}(l_c, n_0) & = \mathsf{i32}(n_1 \sqcup n_2) \\
\mathsf{i32}(n_1) & \sqcup & \mathsf{ptr}(l_2, n_2) & = \mathsf{i32}(\top) \\
\mathsf{i32}(n_1) & \sqcup & \mathsf{num}(n_2) & = \mathsf{i32}(n_1 \sqcup n_2) \\
\mathsf{ptr}(l_1, n_1) & \sqcup & \mathsf{ptr}(l_1, n_2) & = \mathsf{ptr}(l_1, n_1 \sqcup n_2) \\
\mathsf{ptr}(l_1, n_1) & \sqcup & \mathsf{ptr}(l_2, n_2) & = \mathsf{ptr}(\top, \top) \\
\mathsf{num}(n_1) & \sqcup & \mathsf{num}(n_2) & = \mathsf{num}(n_1 \sqcup n_2) \\
\mathsf{num}(n_1) & \sqcup & \mathsf{ptr}(l_c, n_0) & = \mathsf{i32}(n_1 \sqcup n_0) \\
\mathsf{num}(n_1) & \sqcup & \mathsf{ptr}(l_2, n_2) & = \mathsf{i32}(\top)
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{i32}(n_1) & \sqcap & \mathsf{i32}(n_1) & = \mathsf{i32}(n_1 \sqcap n_2) \\
\mathsf{i32}(n_1) & \sqcap & \mathsf{ptr}(l_c, n_0) & = \mathsf{ptr}(l_c, n_1 \sqcap n_0) \\
\mathsf{i32}(n_1) & \sqcap & \mathsf{ptr}(l_2, n_2) & = \mathsf{ptr}(l_c, n_1) \sqcap \mathsf{ptr}(l_2, n_2) \\
\mathsf{i32}(n_1) & \sqcap & \mathsf{num}(n_2) & = \mathsf{num}(n_1) \sqcup \mathsf{num}(n_2) \\
\mathsf{ptr}(l_1, n_1) & \sqcap & \mathsf{ptr}(l_c, n_0) & = \mathsf{ptr}(\bot, \bot) \\
\mathsf{ptr}(l_1, n_1) & \sqcap & \mathsf{ptr}(l_1, n_2) & = \mathsf{ptr}(l_1, n_1 \sqcap n_2) \\
\mathsf{ptr}(l_1, n_1) & \sqcap & \mathsf{ptr}(l_2, n_2) & = \mathsf{ptr}(l_3, 0) \\
\multicolumn{4}{l}{\text{where } l_3 \text{ fresh} \wedge l_1 = l_3 - n_1 \wedge l_2 = l_3 - n_2} \\
\mathsf{num}(n_1) & \sqcap & \mathsf{num}(n_2) & = \mathsf{num}(n_1 \sqcap n_2) \\
\mathsf{num}(n_1) & \sqcap & \mathsf{ptr}(l_c, n_0) & = \mathsf{i32}(\bot) \\
\mathsf{num}(n_1) & \sqcap & \mathsf{ptr}(l_1, n_1) & = \mathsf{i32}(\bot)
\end{array}
$$

Fig. 5. The i32 Sub-Lattice With a Subset of Meet and Join Operations Defined.

its result. The meet of a pointer and number unequivocally results in a $\mathsf{i32}(\bot)$, as per our relation definition shown in Figure 5.

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 $\mathsf{i32}(\top)$, 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 $\mathsf{ptr}(\bot, \bot)$.

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 $\mathsf{ptr}(l, \bot)$.

$$R ::= \{\text{inst } C^{r*},\ \text{tab } n^*,\ \text{mem } \Psi^*\}$$
$$C^r ::= \{\text{func } tf_{\text{r}}^*,\ \text{local } \tau_{\text{r}}^*,\ \text{global } \tau_{\text{r}}^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{label } (\tau_{\text{r}}^*, \Psi)^*,\ \text{return } (\tau_{\text{r}}^*, \Psi)^?\}$$

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

$$\tau_{\text{r}_{32}} ::= \text{i32}(n) \mid \text{ptr}(n) \mid \text{num}(n) \mid \text{f32}(n)$$
$$\tau_{\text{r}} ::= \tau_{\text{r}_{32}} \mid \text{i64}(n) \mid \text{f64}(n)$$
$$\tau ::= \text{i32} \mid \text{ptr} \mid \text{num} \mid \text{i64} \mid \text{f32} \mid \text{f64}$$
$$\Psi ::= \{(\text{ptr}(n) \mapsto \tau_{\text{r}_{32}})^*\}$$
$$tf_{\text{r}} ::= \tau_{\text{r}}^*, \Psi \to \tau_{\text{r}}^*, \Psi'$$

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

$$\frac{tf_{\text{r}} = \tau_{\text{r}}^m, \Psi_{\text{pre}} \to \tau_{\text{r}}^n, \Psi_{\text{post}} \qquad R_\Psi = \Psi_{\text{pre}} \qquad R; C^r, \text{label}(\tau_{\text{r}}^n, \Psi_{\text{post}}) \vdash e^* : \tau_{\text{r}}^m, \Psi_{\text{pre}} \to (\tau_{\text{r}}')^n, \Psi'_{\text{post}} \qquad (\tau_{\text{r}}' <: \tau_{\text{r}})^n \qquad \Psi'_{\text{post}} <: \Psi_{\text{post}}}{R; C^r \vdash \mathbf{block}\ tf_{\text{r}}\ e^*\ \mathbf{end} : tf_{\text{r}}}\ \textsc{Block}$$

$$\frac{C^r_{\text{label}}(i) = (\tau_{\text{r}}')^m, \Psi' \qquad (\tau_{\text{r}} <: \tau_{\text{r}}')^m \qquad R_\Psi <: \Psi'}{R; C^r \vdash \mathbf{br}\ i : \tau_{\text{r}}^* \tau_{\text{r}}^m \to \tau_{\text{r}}^*}\ \textsc{Break}$$

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

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

Fig. 6. Typing of a subset of WebAssembly Instructions in the Concrete Refinement Type System.

4 CONCRETE REFINEMENT TYPES AND TYPE SAFETY

We have seen how to generate constraints over a given WebAssembly function to discover *symbolic* refinement types $\hat{\tau}_{\text{r}}$. To prove this refinement type system sound, we must derive *concrete* refinement types from the symbolic ones. Progress requires demonstrating that well-typed instructions can execute according to the operational semantics, which operates on concrete addresses and values. Unlike the original WebAssembly type system, which does not type memory and relies on type annotations to ensure bytes are interpreted correctly, our refinement type system explicitly types memory contents. This requires concrete types for the progress proof: symbolic types like $\text{ptr}(l, n)$ suffice for static analysis but cannot represent actual runtime values needed to show memory operations execute correctly.

4.1 Concrete Refinement Type System

To prove type safety, we transform the symbolic refinement types $\hat{\tau}_{\text{r}}$ discovered through constraint solving into concrete refinement types τ_{r}. 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 concrete refinement type τ_{r} and concrete memory typing Ψ, defined in Figure 6. 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 6) to denote the shape of the stack before and after the call.

4.2 Typing Rules

Next, we describe the refinement typing rules for a subset of WebAssembly instructions (shown in Figure 6). Here, R is a refined store context and contains the WebAssembly module instances C^r, a shared table of functions and the memory typing Ψ. R is analogous to the store context S in the standard WebAssembly type system, except that S had a linear sequence of bytes as its memory. Refinement typing rules for the entire instruction set can be found in Appendix A.

4.2.1 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 6, 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).

4.2.2 Control Constructs and Breaks. The Block rule is annotated with a refined function type tf_r 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.

4.2.3 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 $\mathsf{ptr}(n)$ on the stack, as shown in Figure 6. 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.

4.2.4 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.

4.3 Type Safety

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, $(\tau_{\mathrm{w}}.\mathrm{const}\ c_1)(\tau_{\mathrm{w}}.\mathrm{const}\ c_2)\ \tau_{\mathrm{w}}.binop \hookrightarrow \tau_{\mathrm{w}}.\mathrm{const}(\mathrm{binop}\ (c_1, c_2))$, in the refinement type system, the operational semantics of binop steps as follows, $(\hat{\tau}^1_{\mathrm{r}}.\mathrm{const}\ c_1)\ (\hat{\tau}^2_{\mathrm{r}}.\mathrm{const}\ c_2)\ \hat{\tau}^3_{\mathrm{r}}.binop \hookrightarrow rt\_binop\ ((\hat{\tau}^1_{\mathrm{r}}.\mathrm{const}\ c_1), (\hat{\tau}^2_{\mathrm{r}}.\mathrm{const}\ c_2))$, where rt_binop calculates the binary operation over refinement types.

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

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

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

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.

5 CALL GRAPH ANALYSIS OVER SYMBOLIC REFINEMENT TYPES

To determine valid targets for an indirect call, RNTA checks type compatibility between the inferred callsite type and each candidate function's signature using standard function-type subtyping: the arrow type is *contravariant* in parameters and *covariant* in return types. A function f is a valid target if (i) each callsite argument type is a subtype of the corresponding parameter type of f, and (ii) the return type of f is a subtype of the return type expected at the callsite. When both the callsite and a candidate have pointer-typed arguments, comparing symbolic pointer values $\mathrm{ptr}(l_1, n_1)$ and $\mathrm{ptr}(l_2, n_2)$ directly is not meaningful, as their labels and offsets are placeholders for statically unknown addresses. Instead, the check is lifted to the data types stored at those pointers: given $\Sigma_1 = \{(l_1, n_1) \mapsto \hat{\tau}^1_{\mathrm{r}}\}$ and $\Sigma_2 = \{(l_2, n_2) \mapsto \hat{\tau}^2_{\mathrm{r}}\}$, we check $\hat{\tau}^1_{\mathrm{r}} <: \hat{\tau}^2_{\mathrm{r}}$.

Figure 7 shows the refinement types discovered for the motivating example from Section 2. `num_add` and `num_sub` remain unrefined at $(\mathrm{i32}(\top), \mathrm{i32}(\top)) \rightarrow \mathrm{i32}(\top)$, as their bodies contain no memory accesses. `arr_add` and `arr_sub` are refined to $(\mathrm{ptr}(l_1, 0), \mathrm{i32}(\top)) \rightarrow \mathrm{i32}(\top)$ with entry memory state $\{(l_1, 0) \mapsto \mathrm{i32}(\top)\}$, since both load from memory through their first parameter. The callsite in `indirect_call_num` has no memory-accessing parameters, giving caller stack $(\mathrm{i32}(\top), \mathrm{i32}(\top)) \rightarrow \mathrm{i32}(\top)$; the callsite in `indirect_call_arr` refines its first parameter to ptr, giving $(\mathrm{ptr}(l_1, 0), \mathrm{i32}(\top)) \rightarrow \mathrm{i32}(\top)$.

Under base WebAssembly type-filtering, both callsites resolve to all four functions. With refined types, `indirect_call_num`'s $\mathrm{i32}(\top)$ first argument cannot satisfy condition (i) for `arr_add` and

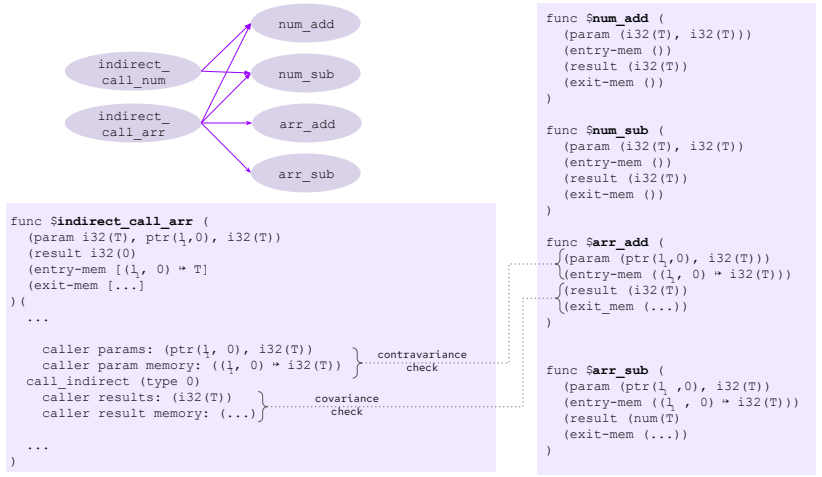


Fig. 7. Call graph over discovered refinement types for the motivating example.

arr_sub, since i32 $\not<:$ ptr, leaving only num_add and num_sub as valid targets. At indirect_call_arr, since ptr <: i32, all four candidates remain valid. The asymmetry is fundamental: a ptr callsite is compatible with callees expecting the more general i32, but an i32 callsite cannot satisfy a callee requiring the stricter ptr.

5.1 Applications to Other Analyses

A more precise call graph is a foundational improvement: virtually every interprocedural static analysis is parameterised by a call graph, and spurious edges propagate imprecision into every downstream client. Beyond this, the ptr/num distinction directly benefits analyses that reason about memory by reducing the aliasing overapproximation inherent in WebAssembly's flat i32 type: a num-typed value cannot alias any ptr-accessed memory location. In the case where a function accesses memory through a single pointer with statically known offsets, each access resolves to a distinct cell $\text{ptr}(l, n)$ with no aliasing ambiguity at all. For taint and dataflow analyses, stores through ptr need not affect num-typed values, and num arithmetic introduces no memory side effects to track. For program slicing, num-typed values can be excluded as candidate memory addresses, substantially reducing slice size.

6 EVALUATION ON REAL-WORLD WEBASSEMBLY BINARIES

We evaluate our call graph analysis on 20 real-world WebAssembly binaries from the NoWaSet dataset [44], a dataset of NPM packages that depend on WebAssembly that has been compiled from diverse source language libraries including Rust, C/C++, Go, etc. NoWaSet has a collection of WebAssembly binaries that is instantiated and executed by JavaScript clients, which they refer to by their module hash. We select 20 Wasm 1.0 binaries with a high number of indirect calls and following NoWaSet's convention, refer to them by their hash, as shown in Table 1. Our evaluation compares a type-based call graph analysis over base WebAssembly types against the same analysis over refinement types inferred by RNTA. We assess how RNTA improves call graph precision, identifies a smaller set of reachable functions, discovers monomorphic callsites that are candidates for further optimizations, and how it compares against other state-of-the-art call graph analysis tools.

Binary	Size (KB)	Function Types	Total Functions	Exported Functions	Imported Functions	Direct Calls	Indirect Calls	Table Funcs
d128e52a5d	360.61	15	544	161	3	2521	5265	160
fd885c2d12	455.67	22	213	13	4	1529	515	50
0f9cbc41a9	666.18	30	143	5	19	1594	169	31
424d9e12e7	1366.61	58	1567	74	9	11939	162	178
04b0c8ef0f	123.52	36	235	49	6	922	116	62
fad0f2d451	57.16	18	167	7	2	463	108	7
b5047ed15e	331.51	31	442	41	8	2936	103	74
47fb4f81d2	86.58	52	248	6	29	761	98	200
f1fb556163	185.58	15	97	22	2	2027	78	6
d1d3df1b33	185.23	38	629	17	9	1177	65	46
e55ee035c7	60.45	23	172	5	0	701	65	51
517f31132d	207.99	15	99	5	18	335	60	35
704bd59a4e	164.80	39	276	58	20	1409	49	30
96e42b64f2	15.84	11	15	7	0	32	31	2
e6171e3491	78.04	18	111	8	0	834	29	23
7c74a56b87	85.72	18	106	8	0	667	26	20
058ca2a32b	65.68	16	99	7	0	606	25	20
1fc3c27d95	19.32	15	73	6	0	132	17	15
b2da8a7b24	17.34	11	66	3	0	289	17	17
477d02ba72	17.03	11	69	3	0	294	17	17

Table 1. Real-World WebAssembly Binary Statistics

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

To evaluate the precision improvement from refinement types, we compare a type-based call graph analysis over the refinement type system against the same analysis over the original WebAssembly type system. We use a type-based analysis as it represents the standard approach employed by other WebAssembly analysis tools. A type-based analysis over base WebAssembly types restricts indirect call targets using only the type annotation at the call_indirect instruction and function signatures. By contrast, RNTA infers refinement function signatures and memory states, constructs a refined caller value stack and memory state, which we use to check contravariance for arguments and covariance for return types. Table 2 shows that refinement types yield significantly more precise call graphs.

Unique Function Types. Refinement types substantially increase the number of distinct function signatures. Across all 20 binaries, RNTA infers on average 3.6× more unique function types than the base WebAssembly type system. The largest increases occur in binaries whose functions make heavy use of i32 parameters that can be split into ptr and num: `fd885c2d12` sees a 7.0× increase (from 22 to 153 types) and `d128e52a5d` a 6.9× increase (from 15 to 103). Conversely, binaries with few functions or limited use of i32 see smaller gains—`96e42b64f2` increases only 1.3× (from 11 to 14) because it contains just 15 total functions and 2 table entries.

Edge Reduction. Edge reduction depends on how effectively refinement splits groups of functions in the WebAssembly table that share a base type. Binaries where refinement produces the greatest increase in type discrimination benefit most: `e55ee035c7` achieves the highest reduction (43.6%) with a 3.2× increase in unique function types (23 base to 73 refined), while `47fb4f81d2` sees only 7.2% with a more modest 2.1× increase (43 to 91). The 4 binaries with no edge reduction either have very few table functions or have refinement types that only distinguish non-table functions. On average, the edge reduction is 14.6%. The primary source of precision improvement is distinguishing pointer-typed arguments from numeric ones: across all 20 binaries, 57.7% of eliminated indirect call edges result from a mismatch where the caller passes a numeric constant

Binary	Type-Based				RNTA			
	Function Types	Reachable Nodes	Edges	Time(s)	Function Types	Reachable Nodes (%Red)	Edges (%Red)	Time(s)
d128e52a5d	15	544	17187	0.17	103	544 (0.0%)	14099 (18.0%)	3.37
fd885c2d12	22	213	1682	0.05	153	209 (1.9%)	1114 (33.8%)	10.99
0f9cbc41a9	30	39	325	0.10	82	39 (0.0%)	325 (0.0%)	21.60
424d9e12e7	57	1566	9474	0.12	370	1459 (6.8%)	6184 (34.7%)	9.16
04b0c8ef0f	34	235	882	0.02	130	234 (0.4%)	681 (22.8%)	1.13
fad0f2d451	18	167	344	0.02	107	148 (11.4%)	306 (11.0%)	0.20
b5047ed15e	31	439	1761	0.02	127	397 (9.6%)	1506 (14.5%)	0.31
47fb4f81d2	43	169	1402	0.05	91	152 (10.1%)	1301 (7.2%)	0.84
f1fb556163	15	91	317	0.02	52	90 (1.1%)	314 (0.9%)	0.51
d1d3df1b33	34	556	1207	0.02	75	519 (6.7%)	1073 (11.1%)	0.18
e55ee035c7	23	165	686	0.02	73	118 (28.5%)	387 (43.6%)	0.16
517f31132d	15	39	162	0.02	58	39 (0.0%)	162 (0.0%)	4.52
704bd59a4e	38	276	797	0.03	97	273 (1.1%)	778 (2.4%)	3.55
96e42b64f2	11	15	16	0.01	14	15 (0.0%)	16 (0.0%)	0.09
e6171e3491	18	105	311	0.01	52	81 (22.9%)	244 (21.5%)	1.09
7c74a56b87	18	100	254	0.01	49	80 (20.0%)	208 (18.1%)	0.35
058ca2a32b	16	93	243	0.01	48	73 (21.5%)	197 (18.9%)	0.21
1fc3c27d95	15	23	92	0.00	34	23 (0.0%)	92 (0.0%)	0.04
b2da8a7b24	11	60	128	0.01	32	43 (28.3%)	106 (17.2%)	0.04
477d02ba72	11	63	130	0.01	32	46 (27.0%)	108 (16.9%)	0.05
Average				0.04		9.9%	14.6%	2.92

Table 2. Comparing the reachable nodes and edges in a type-based call graph analysis over base WebAssembly types versus refinement types.

but the candidate target function expects a pointer argument, or vice versa. The remaining 41.3% of eliminations arise from cases where the source callsite receives a more specific refinement type that excludes certain targets.

Unreachable Function Detection. Removing spurious edges can render functions unreachable from exported function entry points. Of the 16 binaries with edge reduction, 15 also exhibit node reduction, averaging 9.9%. However, the correspondence between edge reduction and node reduction is not uniform. `d128e52a5d` achieves 18.0% edge reduction but no node reduction because this binary has 160 table functions sharing only 15 base type signatures, creating a dense web of indirect call edges. Although refinement types eliminate 3,088 spurious edges, every target of a removed edge retains other incoming edges, and 86 of the 88 affected target functions are also reached via direct calls. Thus, no function becomes unreachable despite the substantial edge reduction. By contrast, `47fb4f81d2` achieves 10.1% node reduction from only 7.2% edge reduction, indicating that the removed edges were the sole incoming paths to those functions.

Analysis Cost. The baseline type-based analysis averages 0.04s, while RNTA averages 2.92s (median 0.43s, maximum 21.60s). The overhead stems from constraint generation over every instruction in every function body and constraint solving, so analysis time scales with binary size and function body complexity rather than with the number of indirect calls. For instance, `0f9cbc41a9` takes 21.60s, while `d128e52a5d` has 5,265 indirect calls but completes in 3.37s. Even in the worst case, the analysis finishes in under 22 seconds, making it practical for real-world binaries.

Binary	Type-Based Polymorphic → RNTA Polymorphic	Type-Based Polymorphic → RNTA Monomorphic	Type-Based Polymorphic → RNTA Unreachable	Type-Based Monomorphic → RNTA Monomorphic	Type-Based Monomorphic → RNTA Unreachable	Both Unreachable
d128e52a5d	5267 → 4884	5267 → 0	5267 → 383	0 → 0	0 → 0	0 → 0
fd885c2d12	501 → 479	501 → 5	501 → 17	13 → 10	13 → 3	1 → 1
0f9cbc41a9	1 → 1	1 → 0	1 → 0	0 → 0	0 → 0	168 → 168
424d9e12e7	151 → 128	151 → 17	151 → 6	2 → 0	2 → 2	0 → 0
04b0c8ef0f	103 → 87	103 → 0	103 → 16	7 → 4	7 → 3	5 → 5
fad0f2d451	38 → 0	38 → 22	38 → 16	15 → 0	15 → 15	55 → 55
b5047ed15e	102 → 56	102 → 6	102 → 40	1 → 1	1 → 0	0 → 0
47fb4f81d2	77 → 75	77 → 1	77 → 1	4 → 3	4 → 1	17 → 17
f1fb556163	3 → 0	3 → 1	3 → 2	63 → 63	63 → 0	8 → 8
d1d3df1b33	38 → 1	38 → 2	38 → 35	0 → 0	0 → 0	19 → 19
e55ee035c7	64 → 22	64 → 5	64 → 37	0 → 0	0 → 0	1 → 1
517f31132d	17 → 17	17 → 0	17 → 0	0 → 0	0 → 0	42 → 42
704bd59a4e	39 → 28	39 → 9	39 → 2	8 → 7	8 → 1	2 → 2
96e42b64f2	0 → 0	0 → 0	0 → 0	31 → 31	31 → 0	0 → 0
e6171e3491	16 → 1	16 → 2	16 → 13	13 → 0	13 → 13	0 → 0
7c74a56b87	15 → 1	15 → 2	15 → 12	10 → 0	10 → 10	0 → 0
058ca2a32b	15 → 1	15 → 2	15 → 12	10 → 0	10 → 10	0 → 0
1fc3c27d95	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	17 → 17
b2da8a7b24	10 → 1	10 → 2	10 → 7	7 → 0	7 → 7	0 → 0
477d02ba72	10 → 1	10 → 2	10 → 7	7 → 0	7 → 7	0 → 0

Table 3. Showcasing the improvement in indirect call resolution for type based call graph analysis on refinement types over an analysis over base WebAssembly types.

RQ2: How does call graph precision improve across callsites?

Table 3 shows how individual indirect callsites transition across three categories when moving from a type-based analysis to RNTA: polymorphic (multiple targets), monomorphic (a single target), and unreachable (no targets).

Monomorphic Callsite Discovery. A key result is the discovery of new monomorphic callsites. Across 14 of the 20 binaries, RNTA identifies 78 indirect callsites as monomorphic that the base type system treats as polymorphic. Among binaries with polymorphic callsites, an average of 12.6% of those callsites are promoted to monomorphic by RNTA. These are direct candidates for call devirtualization (replacing an indirect call with a direct call) which can enable further optimizations such as inlining and specialization. The most notable case is `fad0f2d451`, where all 38 type-based polymorphic callsites are fully resolved: 22 become monomorphic and 16 are identified as unreachable. `424d9e12e7` discovers 17 new monomorphic callsites from 151 polymorphic ones, and `704bd59a4e` resolves 9 of 39.

Polymorphic Callsite Resolution. On a per-binary basis, RNTA resolves a median of 55.4% of type-based polymorphic callsites. Even binaries with modest overall edge reduction benefit from refinement at the callsite level: `f1fb556163` had only 0.9% edge reduction, yet RNTA resolves all 3 of its polymorphic callsites—1 becomes monomorphic and 2 become unreachable—leaving all 64 remaining reachable callsites monomorphic.

RQ3: How does RNTA 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 [48], Wassail [43] and Wasma [11], and with a baseline analysis (Naive) that resolves every indirect call target to be every function in the WebAssembly table. Several tools are excluded from our

Binary	Naive		RNTA		Wasm-Opt		Wasma		Wassail	
	Nodes	Edges	Nodes (Red%)	Edges (Red%)	Nodes (Red%)	Edges (Red%)	Nodes (Red%)	Edges (Red%)	Nodes (Red%)	Edges (Red%)
d128e52a5d	544	32780	544(0.0%)	14099(57.0%)	540(0.7%)	32890(−0.3%)	544(0.0%)	17187(47.6%)	544(0.0%)	17187(47.6%)
fd885c2d12	213	3422	209(1.9%)	1114(67.4%)	208(2.3%)	3418(0.1%)	213(0.0%)	1682(50.8%)	213(0.0%)	1682(50.8%)
0f9cbc41a9	143	776	39(72.7%)	358(53.9%)	123(14.0%)	743(4.3%)	39(72.7%)	375(51.7%)	39(72.7%)	375(51.7%)
424d9e12e7	1567	18759	1459(6.9%)	6184(67.0%)	1558(0.6%)	20044(−6.9%)	1566(0.1%)	9474(49.5%)	1566(0.1%)	9474(49.5%)
04b0c8ef0f	235	2697	234(0.4%)	681(74.7%)	228(3.0%)	2674(0.9%)	235(0.0%)	882(67.3%)	235(0.0%)	882(67.3%)
fad0f2d451	167	459	148(11.4%)	306(33.3%)	164(1.8%)	457(0.4%)	167(0.0%)	344(25.1%)	167(0.0%)	1182(−157.5%)
b5047ed15e	442	3476	397(10.2%)	1508(56.6%)	433(2.0%)	3468(0.2%)	439(0.7%)	1761(49.3%)	439(0.7%)	1761(49.3%)
47fb4f81d2	248	14712	152(38.7%)	1360(90.8%)	217(12.5%)	14653(0.4%)	169(31.9%)	1454(90.1%)	169(31.9%)	1454(90.1%)
f1fb556163	97	461	90(7.2%)	314(31.9%)	94(3.1%)	440(4.6%)	91(6.2%)	321(30.4%)	91(6.2%)	321(30.4%)
d1d3df1b33	629	2538	519(17.5%)	1131(55.4%)	FAIL	FAIL	556(11.6%)	1236(51.3%)	556(11.6%)	1236(51.3%)
e55ee035c7	172	1342	118(31.4%)	388(71.1%)	171(0.6%)	1342(0.0%)	165(4.1%)	690(48.6%)	165(4.1%)	690(48.6%)
517f31132d	99	555	39(60.6%)	194(65.0%)	80(19.2%)	555(0.0%)	39(60.6%)	211(62.0%)	39(60.6%)	211(62.0%)
704bd59a4e	276	1403	273(1.1%)	778(44.5%)	255(7.6%)	1361(3.0%)	276(0.0%)	797(43.2%)	276(0.0%)	797(43.2%)
96e42b64f2	15	16	15(0.0%)	16(0.0%)	14(6.7%)	16(0.0%)	15(0.0%)	16(0.0%)	15(0.0%)	16(0.0%)
e6171e3491	111	451	81(27.0%)	253(43.9%)	111(0.0%)	461(−2.2%)	105(5.4%)	311(31.0%)	105(5.4%)	311(31.0%)
7c74a56b87	106	338	80(24.5%)	211(37.6%)	105(0.9%)	357(−5.6%)	100(5.7%)	254(24.9%)	100(5.7%)	254(24.9%)
058ca2a32b	99	327	73(26.3%)	200(38.8%)	98(1.0%)	327(0.0%)	93(6.1%)	243(25.7%)	93(6.1%)	243(25.7%)
1fc3c27d95	23	167	23(0.0%)	102(38.9%)	23(0.0%)	167(0.0%)	23(0.0%)	122(26.9%)	23(0.0%)	122(26.9%)
b2da8a7b24	66	183	43(34.8%)	106(42.1%)	65(1.5%)	183(0.0%)	60(9.1%)	128(30.1%)	60(9.1%)	128(30.1%)
477d02ba72	69	185	46(33.3%)	108(41.6%)	68(1.4%)	185(0.0%)	63(8.7%)	130(29.7%)	63(8.7%)	130(29.7%)
Average			20.3%	50.6%	4.2%	−0.1%	11.1%	41.8%	11.1%	32.6%

Table 4. Comparison of RNTA against state-of-the-art WebAssembly call graph construction tools. Node and edge reduction percentages are relative to the Naive baseline that considers all functions in the WebAssembly function table to be reachable from an indirect call.

evaluation: Wassilly [33] timed out on each binary while generating a call graph, Sturdy [28] does not support analysis over binaries that interoperate with JavaScript (the framework throws an error), and Wasmati [12] does not support WebAssembly functions that return more than one result (multi-value proposal). Table 4 compares the different analysis tools. We are unable to report on the number of monomorphic callsites discovered by Wasm-Opt, Wassail or Wasma because they generate coarse-grained call graphs without callsite-specific information.

Wasm-Opt. Wasm-Opt's generated call graph does not report the resolution of indirect calls, but rather just direct calls. We estimate the resolution of indirect calls by looking through their codebase and find that Wasm-Opt performs a naive call graph analysis: at every indirect call site, it assumes that all functions in the function table may be called. As a result, Wasm-Opt achieves virtually no edge reduction. They show a node reduction of 4% because they do not report imported functions in the generated call graph. Wasm-Opt also fails to run on `d1d3df1b33`.

Wasma. Wasma performs a type-based call graph analysis, considering all functions in the table with a matching type annotation as potential indirect call targets. This yields substantially better precision than the naive analysis, with 42% average edge reduction and 11% average node reduction. However, RNTA consistently outperforms Wasma, achieving 51% average edge reduction (9 percentage points more) and 20% average node reduction (9 percentage points more). The improvement is most pronounced on binaries where refinement types effectively split function groups that share a base type: `e55ee035c7` (71% vs. 49%), `fd885c2d12` (67% vs. 51%), `424d9e12e7` (67% vs. 50%), and `04b0c8ef0f` (75% vs. 67%).

Wassail. Wassail performs a type-based call graph analysis like Wasma. On 19 of the 20 binaries, Wassail produces identical results to Wasma. The exception is `fad0f2d451`, where Wassail reports 1,182 edges—158% *more* than the naive baseline's 459 edges—while Wasma reports 344 edges (25% reduction) and RNTA achieves 306 edges (33% reduction). This is likely a bug in Wassail's source code. Overall, Wassail averages 33% edge reduction, 18 percentage points less than RNTA's 51%.

Note that if an indirect callsite uses a constant value to index into the function table, Wassail's analysis statically resolves the target to a single function. However, this pattern is not common in real-world binaries and does not improve Wassail precision in our experiments.

7 RELATED WORK

Below, we discuss relevant work in the following four categories: (i) call graph construction techniques for imperative and object-oriented programming languages, (ii) call graph construction for WebAssembly, (iii) call graph construction for other low-level languages, and (iv) work on refinement type systems. Below, we discuss relevant work in each of these categories.

7.1 Call Graph Construction for Imperative and Object-Oriented Languages.

In high-level programming languages, indirect calls are typically performed using function pointers or dynamically dispatched (virtual) method calls. The key challenge in constructing call graphs is to approximate the behavior of indirect calls, which in high-level languages involves reasoning about pointers to objects. The concepts of pointer analysis and call graph construction are closely related because determining the objects that a pointer may point to depends on what functions are reachable, and determining the functions that are reachable depends on what functions a function pointer may point to, or on the type of a reference in which a dynamically dispatched method is invoked.

During the past 30+ years, a vast literature on algorithms for approximating the behavior of indirect calls has been developed, which in essence involve associating sets of abstract objects with abstract pointers. This includes techniques based on: class hierarchies and types [7, 17, 45], context-sensitive algorithms [5, 22, 23, 38, 40], declarative techniques [6, 10], demand-driven and selective algorithms [30, 41], and frameworks for static analysis [2–4]. Overviews of pointer analysis techniques can be found in [26, 27, 36, 39]. Unfortunately, since WebAssembly is a low-level language that features neither objects nor pointers, these traditional pointer-analysis algorithms cannot be used.

7.2 WebAssembly Static Program Analysis

There are several static analysis frameworks for WebAssembly. We compare against Wassail [42], Wasma [11] and Wasm-Opt [48]. Wassilly [33] and Sturdy [28] 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. There are also other analysis tools that have a different focus than that of our work. Wasmati [12] uses code property graphs to statically detect vulnerabilities in WebAssembly binaries and use a type-based call graph analysis. WasmChecker [50] detects bugs in WebAssembly programs using abstract interpretation, instantiating the analysis with multiple abstract domains to find null dereferences, out-of-bounds memory accesses, and division-by-zero errors. Wanilla [37] is a static noninterference analysis for WebAssembly that tracks taints with value-sensitive relational reasoning to capture information flows between a Wasm module and its JavaScript host as well as memory integrity within the module. Like other state-of-the-art tools, they resolve indirect calls via type-based filtering. They also track taints through memory but loose precision in the face of runtime addresses, unlike us.

7.3 Inferring Richer Types for Binary Analysis

In general, inferring higher-level types for LLVM IR and binary code is a popular research direction. LLVM IR encodes some structural type information directly in `getelementptr` (GEP) instructions:

an instruction such as `getelementptr %struct.Foo, ptr %p, i32 0, i32 1` reveals that `%p` points into a `Foo`, providing structural evidence that downstream analyses can exploit. In practice, however, this information is often incomplete or broken. Liu et al. [31] present TFA, which improves indirect-call resolution in LLVM IR for C/C++ programs by first consulting debug metadata nodes embedded in the IR to reconstruct missing struct names and fields, then applying multi-layer type and data-flow analysis. WebAssembly binaries, by contrast, are routinely shipped at minimum size and DWARF debug information is rarely retained, so this recovery strategy is unavailable to us; our refinement type system instead recovers the ptr/num distinction purely from program behaviour. Concurrent work by Nicolosi et al. [32] also uses GEP-encoded structural types to reconstruct transparent pointee types after LLVM 17 collapsed all pointer types to the single opaque type `ptr`. They show that richer type information improves the precision of downstream analyses, something that we believe is true for RNTA as well.

The binary setting is perhaps more closely matched to WebAssembly. Concurrent work by Bosamiya et al. [9] present TRex, which reconstructs C-like types (structs, arrays, unions) from stripped x86/x64 binaries to aid human reverse engineers, explicitly trading soundness for utility via *conditional conservativeness* with toggleable opportunistic inferences. Struct types are inferred via colocation: x86 instructions encode constant-offset memory accesses explicitly (e.g., `mov eax, [rbx+8]`), and a finite set of named registers with well-defined calling conventions means the analysis can attribute semantic roles (pointer, integer, return value) to variables from the outset. WebAssembly is a stack machine with anonymous stack slots and local variables that carry no semantic meaning beyond their base type, so an equivalent colocation type reconstruction does not exist until pointer-typed values have first been identified, which is precisely what our work provides. Our work instead prioritises soundness over human-readable output: by computing a greatest fixpoint over a finite lattice, every inferred refinement type is a safe overapproximation, making our results suitable for downstream static analyses.

7.4 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 [18, 19, 34, 46, 49], allowing programmers to express more precise program properties and enabling program verification. The most closely related work is WasmPrecheck [20], 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 $\mathsf{ptr}(a)$ or $\mathsf{num}(n)$. 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.

8 CONCLUSIONS AND FUTURE WORK

We have presented RNTA, a refinement type system and call graph analysis for WebAssembly that refines i32 types to either a singleton pointer type $\mathsf{ptr}(l, n)$ (i.e., a pointer to symbolic location l at abstract offset n) or a singleton number $\mathsf{num}(n)$. Additionally, we type WebAssembly memory, which the original type system leaves untyped, enabling precise tracking of integer values as they flow through memory. The results of our experiments show that, when compared against a baseline analysis and other state-of-the-art WebAssembly static analysis tools that use WebAssembly's standard type system, RNTA reduces reachable functions by 9.9% and call graph edges by 14.6% while increasing indirect calls that are resolved to a unique target by 12.6%. At present, our analysis only

supports WebAssembly version 1.0. We plan to extend our analysis to support subsequent versions of WebAssembly, and to improve precision through standard analysis techniques as well as through a multi-language analysis with JavaScript, WebAssembly's most popular host language.

8.1 Supporting WebAssembly 2.0 and 3.0

WebAssembly's first version update introduced vector instructions, to support 128-bit wide SIMD functionality; bulk memory instructions, to support copying and initializing regions of memory; multi-value results, to support returning more than one value by functions, blocks and instructions; reference types to support opaque first-class references to functions or pointers; non-trapping conversions to allow seamless conversions from float to integer types; and sign-extension instructions that allow directly extending the width of signed integer values. We already support multi-value results and do not anticipate difficulty extending our refinement type system to include these new instructions.

For WebAssembly 3.0, the WebAssembly working group has already implemented other language extensions [1], the most interesting of which is the Garbage Collection (GC) proposal which introduces aggregate types like structures and arrays with optional mutable fields and packed size fields, reference types, imported types, host types and address types, which are used as offsets into memories and tables. Additionally, linear memory is now 32- and 64-bit addressable, that is, linear memory can be accessed with i32 and i64 values. To add support for the GC proposal, we would first refine i64 types to have a similar refinement to the one presented in the paper for i32, such that, $\{\mathsf{ptr32}(n), \mathsf{num32}(n)\} <: \mathsf{i32}(n) \wedge \{\mathsf{ptr64}(n), \mathsf{num64}(n)\} <: \mathsf{i64}(n)$. Address types (*addrtype* ::= i32 | i64) are a *subset* of number types (*numtype* ::= i32 | i64 | f32 | f64). Since our refinement introduces *subtype* constraints between pointers and integers, we would still see precision improvements.

8.2 JavaScript Client Analysis

WebAssembly was originally designed to interoperate with JavaScript and JavaScript remains its most popular host. Our long-term goal is to develop static-analysis techniques for WebAssembly binaries used in JavaScript clients for code-size reduction (specializing a WebAssembly library to a client's use case), optimization (module splitting to improve run-time performance) and detection of security vulnerabilities. RNTA is a closed-world analysis over WebAssembly, in that it makes worst-case assumptions about the behaviour of a JavaScript client. WebAssembly interoperates with JavaScript through imported and exported functions and a closed-world analysis makes the assumption that all exported functions in the binary can be called by the client. However, Thalakottur et al. [44] have shown that clients often only call a subset of the binary's exported functions. An open-world analysis of WebAssembly and JavaScript that detects the exported functions called by the JavaScript client and can propagate values from JavaScript into the WebAssembly analysis would further improve the precision of the WebAssembly analysis.

8.3 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. For example, we plan to change the domain of values to be the power-set lattice over natural numbers. We also plan to make our analysis flow and context sensitive and propagate types through direct calls to increase precision at callsites.

9 DATA-AVAILABILITY STATEMENT

RNTA along with the binaries and scripts used for evaluation is available as an artifact at https://anonymous.4open.science/r/RNTA. We intend to submit the artifact for Artifact Evaluation.

REFERENCES

[1] Andreas Rossberg (Ed.). [n. d.]. *Current up-to-date WebAssembly Specification.* https://webassembly.github.io/memory64/core/index.html

[2] 2026. Doop: Framework for Java Pointer and Taint Analysis (using P/Taint). https://github.com/plast-lab/doop. Accessed: 2026-03-16.

[3] 2026. SOOT: A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/. Accessed: 2026-03-16.

[4] 2026. WALA. https://github.com/wala/WALA. Accessed: 2026-03-16.

[5] Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 2–26. https://doi.org/10.1007/3-540-49538-X_2

[6] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, Rome, Italy, July 18-22, 2016 (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPICS.ECOOP.2016.2

[7] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, Lougie Anderson and James Coplien (Eds.). ACM, 324–341. https://doi.org/10.1145/236337.236371

[8] Jay Bosamiya, Maverick Woo, and Bryan Parno. 2025. {TRex}: Practical Type Reconstruction for Binary Code. In *34th USENIX Security Symposium (USENIX Security 25).* 6897–6915.

[9] 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.

[10] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[11] 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

[12] 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

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

[14] 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.

[15] 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.

[16] 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.

[17] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[18] 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

[19] 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.

[20] 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

[21] 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.

[22] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[23] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-Oriented Languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[24] 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

[25] 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

[26] Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field and Gregor Snelting (Eds.). ACM, 54–61. https://doi.org/10.1145/379605.379665

[27] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, Debra J. Richardson and Mary Jean Harold (Eds.). ACM, 113–123. https://doi.org/10.1145/347324.348916

[28] 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

[29] 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

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. https://doi.org/10.1145/3381915

[31] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. 2024. Improving {Indirect-Call} analysis in {LLVM} with type and {Data-Flow}{Co-Analysis}. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5895–5912.

[32] Niccolò Nicolosi, Gabriele Magnani, Emilio Corigliano, Davide Baroffio, Federico Reghenzani, and Giovanni Agosta. 2026. Type Deduction Analysis: Reconstructing Transparent Pointer Types in LLVM-IR. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction*. 194–204.

[33] 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

[34] 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.

[35] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. *ACM Sigplan Notices* 45, 1 (2010), 131–144.

[36] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 126–137. https://doi.org/10.1007/3-540-36579-6_10

[37] Markus Scherer, Jeppe Fredsgaard Blaabjerg, Alexander Sjösten, and Matteo Maffei. 2025. Wanilla: Sound Noninterference Analysis for WebAssembly. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 126–140. https://doi.org/10.1145/3719027.3765156

[38] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Ph. D. Dissertation. Carnegie-Mellon University. CMU-CS-91-145.

[39] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[40] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.

1145/1926385.1926390

[41] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. https://doi.org/10.1145/1094811.1094817

[42] 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.

[43] 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

[44] 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*.

[45] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, October 15-19, 2000*, Mary Beth Rosson and Doug Lea (Eds.). ACM, 281–293. https://doi.org/10.1145/353171.353190

[46] 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.

[47] Andreas Rossberg (Ed.). 2022-03-31. *WebAssembly 1.0 Core Specification*. World Wide Web Consortium (W3C). https://www.w3.org/TR/wasm-core-1/

[48] WebAssembly Community Group. 2025. Binaryen: Optimizer and compiler/toolchain library for WebAssembly. https://github.com/WebAssembly/binaryen.

[49] 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.

[50] Junjie Zhuang, Hao Jiang, and Baojian Hua. 2024. WasmChecker: Effectively Detecting WebAssembly Bugs via Static Program Analysis. In *2024 8th International Conference on Electrical, Mechanical and Computer Engineering (ICEMCE)*. 1669–1678. https://doi.org/10.1109/ICEMCE64157.2024.10861926

Appendices

A INSTRUCTION TYPING FOR THE REFINEMENT TYPE SYSTEM

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

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

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

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

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

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

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

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

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

Fig. 8. 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 $\tau_r(n) = \tau(\text{val}(\text{bits}(n_2)\ \text{bits}(n_1)))$
  if $|\tau_r(n_1)| > N$ then let $\tau_r(n) = \tau(\text{val}(\text{bits}(n_1)_{0..N}))$
  if $(\tau_r(n_1) = (\text{i32}(n_1) \vee \text{ptr}(n_1) \vee \text{num}(n_1)) \wedge \tau = \text{f32})$ then let $\tau_r(n) = \text{f32}(bits(n_1 \text{ as f32}))$
  if $(\tau_r(n_1) = \text{f32}(n_1) \wedge \tau = (\text{i32} \vee \text{ptr} \vee \text{num}))$ then let $\tau_r(n) = \text{i32}(\text{f32}(n_1) \text{ as i32})$
  if tp given
    then return $\tau_r(\text{extend\_sx}_{N,|\tau_w|}(n))$
    else return $\tau_r(n)$

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

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 $R_\Psi[p \mapsto \text{i32}(\text{val}(\text{bits}(n)_{0..32})) \wedge (p-4) \mapsto \text{i32}(\text{val}(\text{bits}(n)_{32..64}))]$
  if $|\tau_r(n)| > N$ then
    $R_\Psi[p \mapsto \tau_r(\text{wrap}_{|\tau|,N}(n))]$
    else $R_\Psi[p \mapsto \tau_r(n)]$

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

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

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

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

$$\frac{tf_r = \tau^m_r, \Psi_{\text{pre}} \to \tau^n_r, \Psi_{\text{post}} \qquad C^r_{\text{table}} = n \qquad \text{num32}(c) <: \tau^1_r}{R; C^r \vdash \textbf{call\_indirect}\ tf_r : \tau^m_r, \tau^1_r \to \tau^n_r, \Psi_{\text{post}}}\ \textsc{Call-Indirect}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Typing Rules for the Refinement Type System continued.

B TYPE SAFETY PROOF

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

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

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

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 $\mathsf{ptr}(a)$ 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* $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \to \tau_r^* \wedge \vdash_i (v : \tau_r)^m \wedge \vdash_i s : R \wedge s, v^*, e^* \hookrightarrow s', v'^*, e'^*$ *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 $R; C^r \vdash_i \hat{e}^* : \tau_r^1 \cdots \tau_r^m \to \tau_r^*$
CASE LOAD:

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

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

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. This is annotated on the load instruction itself as τ.

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

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 $s_{\mathsf{mem}}(c+o) : R_\Psi(c+o)$, we know that $s_{\mathsf{mem}}(c+o) : \tau_r^2$ and since $s_{\mathsf{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

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

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 $b^* : \mathsf{i64}(n)$. Note that b^* is the byte sequence from $s_{\mathsf{mem}}[c+o:8]$, as specified in the detailed Wasm specification [47]. 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 $\tau_r^2 = \tau(\mathsf{val}(\mathsf{bit}(n_1), \mathsf{bit}(n_2)))$. n_1 and n_2 are the values in memory at $c+o$ and $c+o-4$. Since $s_{\mathsf{mem}}(c+o) : R_\Psi(c+o) \wedge s_{\mathsf{mem}}(c+o-4) : R_\Psi(c+o-4)$, we know that $s_{\mathsf{mem}}(c+o:8) : \tau_r^2$ and since $s_{\mathsf{mem}}(c+o) = b^*$, $b^* : \tau_r^2$.

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 $b^* : \text{f64}(n)$. The proof proceeds exactly as the case above.

Case 1.4: ptr.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{f32}(n)]$
Case 1.5: num.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{f32}(n)]$
Case 1.6: i32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{f32}(n)]$

In this case, the load expects a i32 on the stack and Ψ has f32(n) at $c+o$. We now have to show that $b^* : \text{f32}(n)$. 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, $\tau_r^2 = \text{f32}(\text{bits}(n_1 \text{ as f32}))$ 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 1.7: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{ptr}(n)]$
Case 1.8: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{num}(n)]$
Case 1.9: f32.load $a\ o \wedge R_\Psi[(c+o) \mapsto \text{i32}(n)]$

In this case, the load expects a f32 on the stack and Ψ has i32(n) at $c+o$. We now have to show that $b^* : \text{i32}(n)$. 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, $\tau_r^2 = \text{i32}(\text{f32}(n_1) \text{ as i32})$ 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 2: $s;\ \text{ptr.const}(c);\ \tau.\text{load}\ tp\_sx\ a\ o \hookrightarrow \tau.\text{const}(b*)$ if $s_{\text{mem}}(i, c+o, |tp|) = b^*$

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. The proof proceeds as above except with a small change introduced by the *tp_sx* annotation on the load. In the WebAssembly operational semantics [47], 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: $s;\ \text{ptr.const}(c);\ \tau.\text{load}\ (tp\_sx)^?\ a\ o \hookrightarrow \text{trap}$ 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 5. 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 6. Meet $\sqcap$ opertation for the i32 sub-lattice.

D CONSTRAINT GENERATION FOR ALL WEBASSEMBLY INSTRUCTIONS

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

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

$$\frac{\alpha \text{ fresh} \qquad S' = S :: [\alpha \doteq \tau_{\mathrm{w}}(c)]}{S;\varsigma;C^{\alpha} \vdash \tau_{\mathrm{w}}.\mathbf{const}\ c : \epsilon \to \alpha;S';\varsigma}\textsc{Constant} \qquad \frac{\alpha_1 \in dom(S) \qquad \alpha_2 \text{ fresh} \qquad S' = S :: [\alpha_1 \doteq unop(\alpha_2)]}{S;\varsigma;C^{\alpha} \vdash \tau_{\mathrm{w}}.unop : \alpha_1 \to \alpha_2;S';\varsigma}\textsc{Unary-Ops}$$

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

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

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

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

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

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

Fig. 11. Constraint Generation Rules.

$$\frac{tf_{\mathrm{w}} = \tau_{\mathrm{w}}^m \to \tau_{\mathrm{w}}^n \qquad \alpha^m \in dom(S) \qquad \alpha^n \text{ fresh} \qquad S; \varsigma; C^\alpha, \mathsf{label}(\alpha^n) \vdash e^* : \alpha^m \to (\alpha')^n; S'; \varsigma' \qquad S'' = S' :: [(\alpha \doteq \alpha')^n]}{S; \varsigma; C^\alpha \vdash \textbf{block}\ tf\ e^*\ \textbf{end} : \alpha_1^n \ldots \alpha_n^n \to \alpha_1^m \ldots \alpha_m^m; S''; \varsigma'}\ \textsc{Block}$$

$$\frac{tf_{\mathrm{w}} = \tau_{\mathrm{w}}^m \to \tau_{\mathrm{w}}^n \qquad \alpha^m \in dom(S) \qquad \alpha^n \text{ fresh} \qquad S; \varsigma; C^\alpha, \mathsf{label}(\alpha^m) \vdash e^* : \alpha^m \to (\alpha')^n; S'; \varsigma' \qquad S'' = S' :: [(\alpha \doteq \alpha')^n]}{S; \varsigma; C^\alpha \vdash \textbf{loop}\ tf_{\mathrm{w}}\ e^*\ \textbf{end} : \alpha^m \to \alpha^n; S''; \varsigma'}\ \textsc{Loop}$$

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

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

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

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

$$\frac{\alpha^* \alpha_1 \ldots \alpha_n \in dom(S) \qquad C_{\mathsf{return}}(i) = \alpha_1' \ldots \alpha_n' \qquad S' = S :: [\alpha_1 \doteq \bigsqcup \alpha_1\alpha_1' \wedge \cdots \wedge \alpha_n \doteq \bigsqcup \alpha_n\alpha_n']}{S; \varsigma; C \vdash \textbf{return} : \alpha^* \alpha_1 \ldots \alpha_n \to \alpha^*; S'; \varsigma}$$

$$\frac{C_{\mathsf{func}} = tf \qquad tf = \tau_{\mathrm{w}}^1 \ldots \tau_{\mathrm{w}}^n \to \tau_{\mathrm{w}}^1 \ldots \tau_{\mathrm{w}}^m \qquad \alpha_1 \ldots \alpha_n \in dom(S) \qquad S' = S :: [\alpha_1' <: \tau_1 \wedge \cdots \wedge \alpha_m' <: \tau_m]}{S; \varsigma; C \vdash \textbf{call} : \alpha_1 \ldots \alpha_n \to \alpha_1' \ldots \alpha_m'; S'; \varsigma}$$

$$\frac{C_{\mathsf{table}} = n \qquad tf = \tau_{\mathrm{w}}^1 \ldots \tau_{\mathrm{w}}^n \to \tau_{\mathrm{w}}^1 \ldots \tau_{\mathrm{w}}^m \qquad \alpha_1 \ldots \alpha_n \alpha_{n+1} \in dom(S) \qquad S' = S :: [\alpha_{n+1} <: \mathsf{num} \wedge \alpha_1' <: \tau_1 \wedge \cdots \wedge \alpha_m' <: \tau_m]}{S; \varsigma; C \vdash \textbf{call\_indirect}\ tf : \alpha_1 \ldots \alpha_n \alpha_{n+1} \to \alpha_1' \ldots \alpha_m'; S'; \varsigma}$$

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

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

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

Fig. 12. Constraint Generation Rules cont.

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

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

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

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

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

Fig. 13. Constraint Generation Rules cont.