WAPPLER: Sound Reachability Analysis for WebAssembly (Technical Report, v24.07.02)

Markus Scherer*†, Jeppe Fredsgaard Blaabjerg‡, Alexander Sjösten†, Magdalena Solitro†, Matteo Maffei*†
*Christian Doppler Lab Blockchain Technologies for the Internet of Things
†TU Wien, Austria
‡Aarhus University, Denmark

Changes from CSF'24

- added appendix with proofs
- fixed various typos in text
- fixed typo in definition of $\alpha_F(\cdot)$
- fixed definition of $\alpha_M(\cdot)$

***Abstract*—WebAssembly (Wasm) is an increasingly deployed low-level language providing near-native performance to security-critical domains such as web browsers, smart contracts, and edge computing. In all of these domains, establishing the absence of bugs and security vulnerabilities is of utmost importance, which motivates the development of sound and automated static analysis techniques. This is, however, a challenging task since the Wasm formal semantics is not directly amenable to efficient static analysis, Wasm code is typically embedded in statically unknown and possibly malicious contexts, and the low-level nature of the language makes it hard to precisely and yet soundly capture memory management and other core features.**

In this work, we present WAPPLER, the first sound and automated static analysis technique for WebAssembly. The core idea is to encode the semantics into Horn clauses so as to make it accessible to automated theorem provers, such as z3. The realization of this approach, however, requires to tackle several challenges. We address the fact that the Wasm semantics is not directly amenable to automation of security proofs by introducing annotations that enable a precise, practical, and yet sound encoding. Furthermore, we devise a formalism to specify embedder behavior and introduce a sound yet precise memory abstraction.

We demonstrate the expressiveness of our logical formalism by encoding several general as well as Wasm-specific security properties. Finally, we implement our static analysis technique and conduct an experimental evaluation over the official Wasm test suite to demonstrate its performance.

I. INTRODUCTION

WebAssembly [29] (Wasm) is a relatively new low-level language that can serve as a compilation target for higher-level languages such as C, C++, and Rust. Its carefully chosen abstractions enable near-native performance while still providing isolation guarantees [9].

While these were designed around the original use case, namely, performant code in web browsers, they soon turned out to be useful in smart contracts [1], [2], plugin systems [26], [30], isolation of untrusted components [17], lightweight containers [6], and edge computing [13], [20].

Since all these application domains are security-critical, it is not surprising that the formal analysis of security properties of Wasm code is a very active research field. In particular, works targeted so far the verification of the semantics of WebAssembly itself [31], [33], the mechanized (but non-automated) verification of program properties [22], [32], and automated (but unsound) bug-finding [3], [12], [27] as well as identification of malicious WebAssembly code [10], [18], [23].

Despite such tremendous progress, there currently exists no sound *and* automated verification technique for Wasm code security: soundness is crucial to provide security guarantees, while automation enables verification at scale and without demanding strong expertise in formal methods.

Devising a sound and automated verification technique for Wasm code is challenging for various reasons.

The first challenge is related to the *rewriting-based semantics*, which operates on evaluation contexts [29], [38]. As described in recent work [34], a direct implementation of this approach in an interpreter comes with several performance issues, chief amongst them the need to rebuild the evaluation context in each step. The same would apply when directly encoding the semantics into a static analysis. Additionally, directly translating the sequence of symbols that is rewritten would necessitate the use of nested data structures, which are less performant in SMT solvers than flat structures.

Secondly, Wasm is meant to be embedded in larger systems that provide application-specific functionalities, and the WebAssembly specification regulates the *interactions with embedders*. These, however, are not known at analysis time and can be possibly malicious: to retain soundness and precision in the analysis, one thus needs to devise careful abstractions.

Lastly, given its low-level nature, WebAssembly provides less information for analysis than higher-level languages. In particular, *memory management* is tricky to statically analyze, since WebAssembly only supports four scalar types natively. Values of other types have to be stored in the linear memory, which is hard to handle soundly, as there are fewer guarantees. Modeling the linear memory soundly is especially important since prior research [11] has identified that WebAssembly lacks certain mitigations for memory safety violations that modern compilers can provide for native code.

```
0  (module
1      (type (func (param i32) (result i32)))
2      (import "env" "table" (table 1 funcref))
3      (func (type 0) (local i32)
4          local.get 0
5          i32.const 0
6          call_indirect (type 0)
7          local.set 1
8          block (result i32)
9              local.get 1
10             local.get 1
11             i32.const 0
12             i32.ge_s
13             br_if 0
14             i32.const −1
15             i32.mul
16         end
17     )
18     (export "abs ∘ f" (func 0))
19 )
```

Fig. 1: Example of a WebAssembly module

In this paper, we present WAPPLER[1], the first sound and automated static analysis technique for reachability, which is based on Horn-clause resolution. Reachability allows us to formalize many interesting security properties, both general and program-specific, such as restricting memory accesses to sensitive arrays, ruling out integer overflows, or generalized assertion checking (encoded via unreachability of functions), as we will demonstrate in our experimental evaluation.

In particular, to counter the aforementioned challenges we

- introduce an annotated program counter-based version of the semantics, which we prove sound against the original term rewriting-based semantics;
- introduce a mechanism to place assumptions on embedder behavior to enhance precision, while soundly overapproximating all cases not covered by the assumptions;
- we devise a precise, yet practical, memory abstraction.

Our contributions can be summarized as follows:

- WAPPLER, the first sound and automated static analysis technique for WebAssembly, which in particular targets reachability properties, whose prototype is available at https://secpriv.wien/wappler;
- A formal soundness proof for WAPPLER against the original WebAssembly semantics (with all details in [24]);
- A formalization of several security properties that rule out general and Wasm-specific unwanted program behaviors;
- An experimental evaluation, demonstrating the performance and precision of our tool, as well as empirically validating its soundness. In particular, we analyzed all 15k+ applicable tests from the official Wasm test suite, with 98% termination rate within 10s.

II. PRELIMINARIES

This section provides an overview of Wasm, focusing in particular on Wasm 1.0 [29]. We refer to the official documentation [38] and the initial publication [9] for more details.

A. WebAssembly

Wasm programs are distributed as *modules*, which describe function types, functions, imports, exports, and an initial store (see below) configuration. A module is instantiated through an embedder in a host environment, and this environment must define imported entities, including host functions and possibly memories. Functions have a type signature specifying their inputs and outputs and may have mutable local variables, which are only in scope for the specific function. Global variables are scoped for the entire module and may be either mutable or immutable.

In Fig. 1, we have an example of a WebAssembly module. It defines a function type (line 1), imports a table (line 2, see below), defines a function with one local variable (lines 3-17) and exports it (line 18). The module provides a function $g(x) = abs(f(x))$ for an embedder-provided function $f(x)$.

1) Types: Global and local variables, function arguments and results, and instructions are all typed in Wasm. This means a Wasm module can be statically type-checked before execution. There are a total of four types: 32- and 64-bit integers (i32, i64), and 32- and 64-bit floating point numbers (f32, f64). Furthermore, there are function types that map between (tuples of) the aforementioned simple types (like **type** 0, defined in line 1 in Fig. 1).

2) Instructions: Wasm's computational model is a stack machine where instructions pop and push values from/to an implicit operand stack. There are six different kinds of instructions in Wasm: *numeric* instructions provide basic numeric operations (e.g., addition), *parametric* instructions can operate on operands of any value (e.g., popping a single operand), *variable* instructions access local and global variables, *memory* instructions manipulate the linear memory, *control* instructions affect the control flow (e.g., nop and branching), and *administrative* instructions structure the computation. In the function in Fig. 1, **local.get** *0*, for example, pushes the argument of the function to the stack, while i32.**const** *0* pushes the constant 0. i32.**ge_s** checks if the second topmost value on the stack is greater or equal (interpreted signed) than the topmost element (and pushes either i32.**const** *0* or i32.**const** *1*), and i32.**mul** calculates the product of the two topmost stack elements.

The most important administrative instructions are **trap**, expressing an unrecoverable error, and the nested control flow instructions. In particular, $\textbf{label}_n\{\epsilon\} \cdot \textbf{end}$ is used to model local control flow and is described in greater detail below, while $\textbf{frame}_n\{F\} \cdot \textbf{end}$, called *activation*, describes a function on the call stack, which returns n values.

3) Small-step Semantics: Wasm has a small-step semantics based on reduction rules from one configuration to another of the form *config* $\hookrightarrow$ *config*, where a configuration is a syntactic

[1]WebAssembly Proven Program Logic Encoding Reachability

description of a program state. The configurations can be described as tuples $(S; F; \mathit{instr}^*)$, consisting of the current store S that contains all globally available objects (e.g., the global variables and the linear memory), the call frame F of the current function that contains the data specific to the current function invocation (e.g., the local variables), and the sequence of instructions instr^*. As mentioned above, instr^* does not only contain instructions to be executed but also administrative instructions and the results of the preceding computations. For the latter, t.**const** instructions are used (when talking about a sequence of such instructions, we use val^*). The combination of a frame object and an instruction sequence sometimes is called a *thread*. A transition rule $S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}'^*$ means the configuration $S'; F'; \mathit{instr}'^*$ can be derived from $S; F; \mathit{instr}^*$ in one step. The overall execution is described in terms of an evaluation context that works as a lens into instr^* and the so-called structural rules. As formalized in the definition below, an evaluation context is an instruction sequence that is preceded by a sequence of already evaluated values and $\mathbf{label}_n\{\mathit{instr}^*\}$ instructions. It has a hole [_] in which arbitrary instruction sequences can be filled in.

$$E = [\_] \mid \mathit{val}^*\ E\ \mathit{instr}^* \mid \mathbf{label}_n\{\mathit{instr}^*\}\ E\ \mathbf{end}$$

The structural rules below have the following semantics: If the instructions of a configuration can be factored into an evaluation context $E[\mathit{instr}^*]$ such that instr^* can be rewritten to $\mathit{instr}^{*\prime}$, we can rewrite instr^* to $\mathit{instr}^{*\prime}$ within the evaluation context while possibly also changing the store and frame data. If we find an activation $\mathbf{frame}_n\{F'\}$ in the evaluation context, we do a similar recursion while changing the frame data to F'. The last two rules allow to "unwrap" evaluation contexts and activations containing **trap** (while disallowing rewriting $S; F; \mathbf{trap}$ as a terminal configuration).

$$\frac{S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}^{*\prime}}{S; F; E[\mathit{instr}^*] \hookrightarrow S'; F'; E[\mathit{instr}^{*\prime}]}\ \text{SR1}$$

$$\frac{S; F'; \mathit{instr}^* \hookrightarrow S'; F''; \mathit{instr}^{*\prime}}{S; F; \mathbf{frame}_n\{F'\}\ \mathit{instr}^*\ \mathbf{end} \hookrightarrow S'; F; \mathbf{frame}_n\{F''\}\ \mathit{instr}^{*\prime}\ \mathbf{end}}\ \text{SR2}$$

$$\frac{E \neq [\_]}{S; F; E[\mathbf{trap}] \hookrightarrow S; F; \mathbf{trap}}\ \text{SR3}$$

$$\mathbf{frame}_n\{F'\}\ \mathbf{trap}\ \mathbf{end} \hookrightarrow S; F; \mathbf{trap}\ \text{SR4}$$

Additionally to the structural rules, there are rules for every possible instruction. As an example, the following rule models the successful execution of an instruction t.**op**[2] that takes the top n values from the stack and applies a function op to them.[3]

$$\frac{c \in \mathit{op}(c_1, \ldots, c_n)}{(\mathtt{t}.\mathbf{const}\ c_1) \ldots (\mathtt{t}_n.\mathbf{const}\ c_n)\ \mathtt{t}.\mathbf{op} \hookrightarrow (\mathtt{t}.\mathbf{const}\ c)}$$

Similarly, the following rule models a trapping execution.

$$\frac{\mathit{op}(c_1, \ldots, c_n) = \emptyset}{(\mathtt{t}_1.\mathbf{const}\ c_1) \ldots (\mathtt{t}_n.\mathbf{const}\ c_n)\ \mathtt{t}.\mathbf{op} \hookrightarrow \mathbf{trap}}$$

[2]In constructions such as t.**op**, t stands for an arbitrary scalar type.

[3]If the transition rule does not touch any component of the configuration, the components are omitted from the transition rule.

By the structural rules, this rule could be applied if the instructions in the configuration can be decomposed in an evaluation context $E[(\mathtt{t}_1.\mathbf{const}\ c_1) \ldots (\mathtt{t}_n.\mathbf{const}\ c_n)\ \mathtt{t}.\mathbf{op}]$.

4) Structured Control-Flow: The instructions of a function in Wasm are organized into blocks that carry a type annotation describing the values that they can put on the stack, and these blocks are created by using either the scoping construct **block** or any of the control flow constructs **if** and **loop**. Depending on which block construct was used, the jump target of a branch differs. For **loop**, a *backward* jump takes place that restarts the loop, whereas a *forward* jump takes place for **block** and **if**, jumping to the end of a block. However, a jump may not jump outside of the function nor further than the maximum depth of the nested blocks. Due to the structured control flow in Wasm, it is not possible to jump to arbitrary destinations like it is with unstructured control flow such as `goto` in C.

The specification describes all of these control flow constructs by rewriting them to the administrative instruction $\mathbf{label}_n\{\mathit{instr}^*\}\ \cdot\ \mathbf{end}$.

$$\mathbf{block}\ t^n\ \mathit{instr}^*\ \mathbf{end} \hookrightarrow \mathbf{label}_n\{\epsilon\}\mathit{instr}^*\mathbf{end}$$

$$\mathbf{loop}\ t^n\ \mathit{instr}^*\ \mathbf{end} \hookrightarrow \mathbf{label}_0\{\mathbf{loop}\ t^n\ \mathit{instr}^*\ \mathbf{end}\}\mathit{instr}^*\mathbf{end}$$

Using labels, we define the so-called *block context* as follows:

$$B^0 = \mathit{val}^*\ [\_]\ \mathit{instr}^*$$
$$B^{k+1} = \mathit{val}^*\ \mathbf{label}_n\{\mathit{instr}^*\}\ B^k\ \mathbf{end}\ \mathit{instr}^*$$

$B[\_]$ is then used to define the behavior of control flow instructions such as **br** and **return**. As shown below, when such a label becomes the target of a branching instruction, the whole construct gets replaced by the n topmost values on the stack and the instructions in instr^* – the so-called continuation.

$$\mathbf{label}_n\{\mathit{instr}^*\}\ B^l[\mathit{val}^n\ \mathbf{br}\ l]\mathbf{end} \hookrightarrow \mathit{val}^n\mathit{instr}^*\mathbf{end}$$

Consider, for example, the function in Fig. 1. When the **block** instruction is encountered in line 8, it gets rewritten to $\mathbf{label}_1\{\epsilon\}(\mathbf{local.get}\ 1) \ldots \mathrm{i32}.\mathbf{mul}\ \mathbf{end}$. The execution continues within the $\mathbf{label}_1\{\epsilon\}$ (see II-A3) and in line 13 encounters $(\mathrm{i32}.\mathbf{const}\ x)$ $(\mathrm{i32}.\mathbf{const}\ c)$ $(\mathbf{br\_if}\ 0)$ with two values on the stack. Depending on c, either the whole $\mathbf{label}_1\{\epsilon\}\cdot\mathbf{end}$ construct is rewritten to $\mathrm{i32}.\mathbf{const}\ x$ (nothing else, since the continuation is ϵ, which means the execution continues in line 17) or the execution continues in the label construct. In this case the whole construct will eventually rewrite to $\mathbf{label}_1\{\epsilon\}(\mathrm{i32}.\mathbf{const}\ y)\mathbf{end}$, which rewrites to $\mathrm{i32}.\mathbf{const}\ y$, where $y = x \cdot (-1)$.

5) Store and Indirection: The store is a runtime structure that holds all data that may be shared between different module instances. As such, it holds functions (funcs), tables (tables), memory instances (mems), and globals (globals). The different module instances hold pointers into these structures to identify them locally.[4] These indirections are called funcaddrs, tableaddrs etc. Of these structures, the concept of table is the most interesting one. In Wasm 1.0, it holds opaque

[4]See sections 4.2.3 and 4.2.5 of [29] for the full definition.

pointers to functions, although later versions will extend the kinds of opaque objects that tables can hold. These pointers allow for a more dynamic control flow and are used in the **call_indirect** instruction (see III-A3). Since every module instance in Wasm 1.0 can only hold at most one table, we will sometimes call this table *the* table of a module instance.

In Fig. 1, we use an imported table to make an (embedder-specified) function available to the module's code. This function is assumed to be stored at index 0 of the table (lines 5 and 6). In this way, our module can calculate $abs(f(x))$ of any embedder-specified function f.

6) Linear Memory: The linear memory is the main storage in Wasm and is a single[5] contiguous mutable vector of raw bytes in little-endian byte order. If the module owns the memory, it is instantiated with an initial size and initialized with zeroes; otherwise, a memory can also be imported from the outside. In any case, WebAssembly provides the possibility to specify so-called data segments. These are static byte sequences that are written to the memory at initialization time (which can be arbitrarily overwritten at a later point).

The memory can be grown using **memory.grow** (unless it will exceed an optionally specified maximum size), and the current size can be queried using **memory.size**. Addresses are unsigned integers of type i32, and the memory can be accessed at arbitrary addresses using **load** and **store** operations. If the access is out of bounds, the program will trap.

B. Challenges in Soundly Analyzing WebAssembly

1) Rewriting-based Semantics: The way WebAssembly's semantics is specified requires factoring the instruction sequence that describes the value stack, the call stack, and the code to be executed into nested evaluation contexts at every execution step. This is very elegant as a specification device but causes performance issues when implemented directly, as the factoring needed at every step is slow for deeply nested evaluation contexts. The official reference interpreter, for example, which aims to keep a close definitional correspondence to the specification, was deemed too slow to be used as a fuzzing oracle in wasmtime's [4] testing infrastructure. This led to the development of WasmRef-Isabelle [34], a verified interpreter that changes the representation of control flow and evaluation contexts to speed up the evaluation.

2) Interaction with Embedders: WebAssembly is meant to be embedded in larger software components such as web browsers. As WebAssembly modules by themselves have no way of interacting with their embedding context, all external components, such as functions, must be imported. The WebAssembly specification guarantees very little regarding these imported functions: they may not alter the call stack, may not add ill-typed constructs to the store, and may not shrink the store (thereby invalidating references). They may, however, change the store's contents, such as globals, linear memories, or tables, and may even add new functions. Without any additional assumptions, this enables practically arbitrary behavior if a **call_indirect** is executed after calling an imported function, which leads either to unsoundness (if an analysis does not consider this possibility) or imprecision (as arbitrary behaviors have to be modeled).

3) Memory Management: Static analyses can take advantage of the fixed structures of the stack, the locals, and the globals. Unfortunately, these regions can only hold the four aforementioned scalar types; all other non-scalar types, like arrays, strings, and structures, have to be stored in the linear memory. The linear memory does not provide type guarantees, can grow during the execution, and has to support unaligned accesses, which makes it challenging to analyze, as every access potentially has to concatenate multiple bytes from the memory to a stack value or split up a stack value to multiple memory cells. In the name of efficiency, some prior analyses forwent modeling the memory at all [3], [27], which is not an option for us, as we aim for soundness.

III. Analysis

Our analysis is based on a Horn-clause-based abstraction in the style of SeaHorn [7]. The principle of a Horn-clause based abstraction is as follows: We define an abstraction function $\alpha(\cdot)$ that translates concrete configurations c into abstract configurations Δ, where abstract configurations are sets of first-order formulas over some analysis-specific domain and signature. On abstract configurations, we define a refinement relation: We write $\Delta \geq \Delta'$ if all concrete configurations abstracted by Δ' are also abstracted by Δ. $\alpha(\cdot)$ does not only generate the facts describing the current configuration but also Horn clauses that describe all possible behaviors of a module. This is done by generating a Horn clause for every possible program point that describes its transition to the next program point. In the abstract analysis, logical derivability $\vdash$ then takes the role of the transition relation $\hookrightarrow$. With these definitions, we can define soundness as follows: if for all concrete executions $c_1 \overset{*}{\hookrightarrow} c_2$ and abstract configurations Δ_1 with $\Delta_1 \geq \alpha(c_1)$ there exists a Δ_2 with $\Delta_1 \vdash \Delta_2$ and $\Delta_2 \geq \alpha(c_2)$, the analysis is sound. From this definition, it follows that if we want to prove that a bad state is not reachable from c_1, it suffices to show that the abstraction of such a state is not logically derivable from Δ_1. This task is well-suited for SMT's *Constrained Horn-Clause* fragment.

The rest of this section is structured as follows: subsection III-A presents the annotated semantics, subsection III-B reviews some security-relevant reachability properties that are desirable for Wasm modules, and subsection III-C overviews our abstract semantics.

A. Annotated Semantics

As mentioned in section II-B1, a naive implementation of the Wasm specification would lead to inefficiencies. Consider, for example, the following configuration, which may appear during the evaluation of the code in Fig. 1.

$S; F_0;$ **frame**$_1\{F\}$ **label**$_1\{\epsilon\}$ **label**$_1\{\epsilon\}$
(i32.**const** *42*) (i32.**const** *1*) (**br_if** *0*) (i32.**const** -1) i32.**mul**
end end end

[5]This is a restriction that may be lifted in future versions.

In order to derive the next configuration, we apply structural rules, which will eventually allow us to apply a non-structural rule via SR1. This traversal through administrative instructions is the first performance burden. The second is, that the instruction sequence in the last application of SR1 can be decomposed into different evaluation contexts e.g. $E[\text{i32.}\textbf{const}\ 42]$ or $E[(\text{i32.}\textbf{const}\ 1)]$, but the possible step is $E[(\text{i32.}\textbf{const}\ 1)\,(\textbf{br\_if}\ 0)] \hookrightarrow E[\textbf{br}\ 0]$. The next evaluation context on which a transition may happen, however, is not $E[\textbf{br}\ 0]$ but $E[\textbf{label}_1\{\epsilon\}\ (\text{i32.}\textbf{const}\ 42)\ (\textbf{br}\ 0)\ (\text{i32.}\textbf{const}\ -1)\ \text{i32.}\textbf{mul}\ \textbf{end}]$ which can be rewritten to $E[\text{i32.}\textbf{const}\ 42]$. To remove the need to traverse activations and evaluation contexts in each step to find the next applicable rewriting rule, we instead identify each point in the execution with a program counter and a function id. This presents us with two obstacles. Firstly, the original semantics does not have a notion of program counters. Secondly, the rewriting may put (administrative) instructions on the stack that do not appear in the code-as-stored (which is a problem when generating the Horn clauses describing a module). Thankfully, the type system of WebAssembly provides strong guarantees on the shape of the program state at any given program point so that we can prove the soundness of our analysis with a minimally annotated version of the original semantics.

An annotated configuration is the same as an original configuration, except that the frame objects carry more information (see below), and any instruction **cmd** may carry an integer that describes its position in the original program code (denoted, if relevant, as $\textbf{cmd}_{\textbf{pc}}$, where constructs like **if** $\cdot$ **then** $\cdot$ **else** can carry one program counter for each part). Since we have to generate all Horn clauses describing possible transitions in advance, and these Horn clauses will make use of the program counter of the currently executed instruction, we have to guarantee that every instruction **cmd** that is executed in an execution context $E[val^*\ \textbf{cmd}]$ actually carries a program counter. For this reason, we introduce $\cdot \hookrightarrow\!\!\!\!\!\rightarrow \cdot$, the transition relation on annotated configurations. $\cdot \hookrightarrow\!\!\!\!\!\rightarrow \cdot$ differs from $\cdot \hookrightarrow \cdot$ in the following regards:

1) It operates on annotated instructions: Any rule that operates on sequences of instructions instead works on annotated instructions. This means, especially, that the continuations $instr^*$ in $\textbf{label}_n\{instr^*\}$ are annotated instructions (which retain their annotation when being the target or source of a rewriting). Other instructions that are the result of rewriting (such as the t.**const** as results of numeric instructions) do not carry a program counter (with one exception in section III-A4).

2) Some rewriting rules are fused: For the sake of clarity, some instructions in the original semantics rewrite to other, non-constant instructions, most notably **br_if** and **br_table** rewrite to **br**. Since we do not want to generate these intermediate configurations, we modify the rules to immediately take a transition step to get to the next configuration. The new transition rule for **br_if** is given below; the corresponding rule for **br_table** looks similar.

$$\frac{c \neq 0 \qquad instr^* = E[(\text{i32.}\textbf{const}\ c)(\textbf{br\_if}\ l)] \qquad instr^* \hookrightarrow instr^{*\prime} \hookrightarrow instr^{*\prime\prime}}{instr^* \hookrightarrow\!\!\!\!\!\rightarrow instr^{*\prime\prime}}$$

$$\frac{c = 0 \qquad instr^* = E[(\text{i32.}\textbf{const}\ c)(\textbf{br\_if}\ l)] \qquad instr^* \hookrightarrow instr^{*\prime}}{instr^* \hookrightarrow\!\!\!\!\!\rightarrow instr^{*\prime}}$$

3) Invocation of functions is used to annotate instructions: In the original semantics, **call** and **call_indirect**[6] rewrite to **invoke** a, which handles invoking a function in a uniform way, as shown in the transition rules below:

$$\frac{S.\text{tables}[F.\text{module.tableaddrs}[0]].\text{elem}[i] = a \qquad S.\text{funcs}[a] = f \qquad F.\text{module.types}[x] = f.\text{type}}{S; F; (\text{i32.}\textbf{const}\ i)(\textbf{call\_indirect}\ x) \hookrightarrow S; F; (\textbf{invoke}\ a)}$$

$$\frac{S.\text{funcs}[a] = f \qquad f.\text{type} = [t_1^n] \mapsto [t_2^m] \qquad m \leq 1 \qquad f.\text{code} = \left\{\text{type}\ x, \text{locals}\ t^k, \text{body}\ instr^*\ \textbf{end}\right\} \qquad F = \left\{\text{module}\ f.\text{module}, \text{locals}\ val^n(\text{t.}\textbf{const}\ 0)^k\right\}}{S; val^n(\textbf{invoke}\ a) \hookrightarrow S; \textbf{frame}_m\{F\}\ \textbf{block}\ t_2^m\ instr^*\ \textbf{end}\ \textbf{end}}$$

This is problematic for the same reasons as presented above for **br_if**. Additionally, this is a good point to add the annotation to our semantics. We, therefore, replace the **call_indirect** rule with the following one (where "..." stands for the preconditions of the last two rules, excluding the restriction on F):

$$\frac{\ldots \qquad F' = \left\{\begin{array}{l}\text{module}\ f.\text{module}, \text{locals}\ val^n(\text{t.}\textbf{const}\ 0)^k, \\ \text{args}\ val^n, \text{stor}\ S, \text{pc}\ pc, \text{fid}\ a, \text{index}\ (\text{i32.}\textbf{const}\ i)\end{array}\right\} \qquad instr^{*\prime} = annotate(\textbf{block}\ t_2^m\ instr^*\ \textbf{end}\ \textbf{end})}{\begin{array}{l}S; F; val^n\ (\text{i32.}\textbf{const}\ i)(\textbf{call\_indirect}_{pc}\ x) \hookrightarrow\!\!\!\!\!\rightarrow \\ \qquad\qquad S; F; \textbf{frame}_m\{F'\}\ instr^{*\prime}\end{array}}$$

In the new rule, a frame object (F') additionally carries a copy of the arguments (args) and the store (stor) at the time of the function call, since our abstract configurations will also carry this information to increase precision; Additionally, we record the program counter (pc), the id of the current function, and the index **call_indirect** used to determine which function to call since this information is important for our abstraction function. The second change is that we annotate the stored instructions before putting them in the configuration. The $annotate(\cdot)$-function is defined in Fig. 6 in the appendix but, at heart, just assigns an increasing counter to each instruction. We apply the same changes to the **call** rule, except that index is set to ϵ.

4) Control instructions maintain program counter annotations: The **end** of **block** and **loop** instructions keeps its program counter when being rewritten, e.g.

$$\textbf{block}\ t^n\ instr^*\ \textbf{end}_{pc} \hookrightarrow\!\!\!\!\!\rightarrow \textbf{label}_n\{\epsilon\} instr^* \textbf{end}_{pc}$$

[6]In **call_indirect** x, x specifies the index of the called function's type.

```
1  static const char* trusted = "TRUSTED";
2  char game_state[64] = ...;
3  extern void eval(const char*);
4
5  void update_game_state(int x, int y, char c) {
6    if (x < 8 && y < 8) {
7      game_state[y * 8 + x] = c;
8    }
9    eval(TRUSTED);
10 }
```

Fig. 2: Example of a vulnerable function (C).

For **if** · **else** · **end** constructions, we use a "gap" introduced by the $annotate(\cdot)$-function to give a program counter to the **end** of the first block.

$$\frac{c \neq 0}{(\text{i32.const } c)\ \textbf{if}\ t^n\ instr_1^*\ \textbf{else}_{pc}\ instr_2^*\ \textbf{end} \hookrightarrow\!\!\!\!\rightarrow \textbf{label}_n\{\epsilon\} instr_1^* \textbf{end}_{pc-1}}$$

$$\frac{c = 0}{(\text{i32.const } c)\ \textbf{if}\ t^n\ instr_1^*\ \textbf{else}\ instr_2^*, \textbf{end}_{pc} \hookrightarrow\!\!\!\!\rightarrow \textbf{label}_n\{\epsilon\} instr_2^* \textbf{end}_{pc}}$$

In section III, we demonstrate that the annotated semantics and the original one are equivalent (cf. Theorem 1).

Note that a naive implementation of the annotated semantics would not be more efficient than the original one, but it could be used to prove equivalence between a program-counter-based semantics that uses a separate stack for activations, labels, and values (similar to WasmRef-Isabelle, but with program counters). This new semantics would lend itself to an efficient implementation (by sacrificing some of the elegance of the current specification). We contemplated introducing such an alternative, concrete semantics but instead merged the translation to this semantics with the translation to the abstract semantics.

B. Reachability Properties for WebAssembly

We will now illustrate how reachability analysis helps to analyze security-relevant properties of Wasm-modules. The program in Fig. 2 models a simple game that is played on an 8x8 grid (such as chess or mine sweeper). The game state is stored in the `char` array `game_state`; `update_game_state` updates the cell at the given coordinates to a value `c` and afterward calls `eval` on a constant string `trusted`. `eval` and `trusted` are stand-ins for a privileged function and data assumed to be trusted and immutable. A concrete instance in a browser would be a function that calls out to JavaScript and to evaluate code stored in WebAssembly (where it is important to preserve `trusted`'s integrity to avoid XSS-attacks).

Unfortunately, `update_game_state` is vulnerable to an attack that allows arbitrary memory writes. When generating code for a native environment, compilers can assure the `trusted`'s integrity, by, e.g., putting it on a read-only page in the RAM, thereby raising an exception when written to.

As observed by Lehmann et al. [11], WebAssembly does not provide such mitigations: all data stored in the linear memory is potentially writeable. The vulnerability in Fig. 2 stems from insufficient restrictions on `x` and `y`: since these values are

```
0  (module
1      ...
2      block
3          local.get 0
4          i32.const 7                  if (x < 8)
5          i32.gt_s
6          br_if 0
7          local.get 1
8          i32.const 7                  if (y < 8)
9          i32.gt_s
10         br_if 0
11         local.get 1
12         i32.const 3
13         i32.shl                      y * 8 + x
14         local.get 0
15         i32.add
16         i32.const 1056               game_state[...]
17         i32.add
18         local.get 2                  ... = c
19         i32.store
20     end
21     ...
22     (data (i32.const 1024) "TRUSTED")
23     ...
24 )
```

Fig. 3: Example of a vulnerable function (Wasm).

treated as signed, they could be negative and still be used in the index of `game_data`. In the compiled version of the function (see Fig. 3), this can be seen in lines 5 and 9, where an unsigned comparison (i32.**gt_u**) should be used. By choosing the correct values for `x` and `y`, an attacker could therefore write to any memory position, including `trusted` (which is initialized with a data section but shares the same mutable memory with `game_state`).

As shown in [11], constant data is not the only sensitive data that is stored in linear memory: Additionally, an attacker could overwrite function pointers, stack frames (that in some cases have to be modeled in linear memory to maintain the semantics of the source language), heap data, and function tables (again, in case the mechanisms Wasm provides are not used). The absence of mitigations such as stack canaries and address space layout randomization in WebAssembly aggravates vulnerabilities like the one in our example program.

We now present three reachability properties that are all violated in the presented version of the program but respected in a fixed version[7].

1) No-i32.add-Overflow: `trusted` is stored before `game_state` in the memory, which means that an attacker must trigger an integer overflow in the address calculation (lines 11 to 17 in Fig. 3) to be able to write to it. The security property *No*-i32.**add**-*Overflow* excludes the vulnerability by ensuring that additions (on i32) do not overflow. A module starting its execution in a configuration c_1 is secure from such integer overflows if for every instance of i32.**add** that

[7]A possible fix is declaring `x` and `y` as `unsigned int`. Another one would be including `0 <= x && 0 <= y` in the condition.

is executed, whose operands have the same sign, the 32-bit-sum of its operands has the same sign as the operands.

$$\forall c_2 \Big(c_1 \overset{*}{\hookrightarrow} c_2 \wedge \\ c_2 = S; F; E[(\mathsf{i32.const}\ c_1)(\mathsf{i32.const}\ c_2)\mathsf{i32.add}] \wedge \\ sgn(c_1) = sgn(c_2) \implies sgn(c_1 + c_2) = sgn(c_1) \Big)$$

Similar properties can be provided for other potentially overflowing instructions such as i32.**sub**, i64.**mul** etc. To increase precision, an instrumented compiler could output locations of unsigned integer computations (whose overflow is defined behavior in, e.g., C) to exclude innocuous instances of i32.**add**. Another approach would be only to include instances whose result ends up in the arguments of memory access operations. To find these, one could run a simple taint analysis before running WAPPLER.

2) No-Sensitive-Overwrite: Besides the potential computation of an illegal memory address, the fundamental issue with our program is that an attacker can trigger a write (line 19 in Fig. 3) to a memory region (line 22 in Fig. 3) that is not meant to be written to. An automated pre-analysis, manual inspection, or an instrumented linker could provide information on such memory regions. In Fig. 3, for instance, addresses 1024 to 1032 should never be written to. Given this information as an interval of memory addresses $[low, high]$, we can formalize a property *No-Sensitive-Overwrite* as follows: A module starting its execution in a configuration c_1 does not overwrite sensitive data, if for every instruction t.**store**N *memarg*, that writes N bits of a t-type value at the effective address *ea*[8] we have that $ea + N/8$ is smaller than *low* and *ea* is greater than *high*.

$$\forall c_2 \Big(c_1 \overset{*}{\hookrightarrow} c_2 \wedge ea = memarg.\mathsf{offset} + i \wedge \\ c_2 = S; F; E[(\mathsf{i32.const}\ i)(\mathsf{t.const}\ c)(\mathsf{t.store}N\ memarg)] \\ \implies ea + N/8 < low \wedge high < ea \Big)$$

Similarly, an instrumented compiler could output legal memory ranges for each store, which could then be used to formulate a dual property (in the given program, the address argument of i32.**store** should, for example, be in the range of $[1056, 1119]$).

3) General Assertion Checking: Assertions provide a developer-friendly way of specifying assumptions on a program in the source language. Assuming a designated function `reach_error` we can define a function `assert` like this:

```
1 void assert(int b) {
2   if(!b) reach_error();
3 }
```

Given these functions, we can verify the absence of assertion violations by ensuring that `reach_error` is never called. In our example program, we could assert that `trusted` indeed contains `"TRUSTED"` before `eval` is called like this:

[8]In WebAssembly, every instruction that accesses the memory comes with a static argument *memarg* that can specify a static offset that is added to the address taken from the stack. This sum is called *effective address*.

```
1 assert(trusted[0] == 'T');
2 assert(trusted[1] == 'R');
3 ...
4 assert(trusted[6] == 'D');
5 assert(trusted[7] == '\0');
```

Assuming `reach_error` has the function id $\mathbf{fid}_{re}$, we can, given the program contains `assert` as defined above, define the absence of assertion violations as follows: a module starting its execution in a configuration c_1 does not violate any assertion, if for any configuration $(S; F; instr^*)$ that can be derived (including sub-derivations), we have that $F.\mathsf{fid} \neq \mathbf{fid}_{re}$.

$$\forall c_2 \Big(c_1 \overset{*}{\hookrightarrow} c_2 \wedge c_2 = S; F; instr^* \implies F.\mathsf{fid} \neq \mathbf{fid}_{re} \Big)$$

An important caveat for source-level specifications of properties is that an overeager compiler might optimize them away even when the problematic behavior can indeed occur. [39][9].

C. Abstract Semantics

1) Assumptions: In the interest of clarity, we restrict our analysis to a *single*, *instantiated* module. Instead of modeling the instantiation within the analysis, we compute the state of the module instance in a preceding step.[10], has already been executed. The restriction on a single analyzed module instance could be easily lifted by adding a parameter identifying the instance to the *MState*- and other predicates to distinguish the different instances and tracking the memories and globals of all modules in *MState*. While we could model the behavior of linked module instances precisely with these changes, we, for now, treat all imported functions (regardless of whether imported from another Wasm-module or the embedder) as overapproximated (see below in section III-C4).

2) Predicates and Types: Our abstraction is defined over the domain of $\mathbb{B}_{64}$, the 64-bit bit vectors, as these are sufficient to hold all kinds of values that appear in WebAssembly modules. *Memory* is a named tuple describing one single memory cell. Its first coordinate holds the index of the memory cell, the second the byte at this position, and the third the current memory size.

$$Memory = Mem(\mathbb{B}_{64} \times \mathbb{B}_{64} \times \mathbb{B}_{64})$$

The signature of predicates over which our analysis is defined is shown in Fig. 7 in the appendix, here we will only present the most important ones. Given a valid WebAssembly module, we can precompute useful information (stack sizes, different variable counts) for the analysis that is provided by a family of functions $\mathsf{SI}.*$. Using $\mathsf{SI}.*$, we describe the $MState_{\mathbf{fid} \times \mathbf{pc}}$-predicate, which (depending on **fid** and **pc**) has the following arguments: a stack of values, two tuples holding the globals and locals, and a memory cell. Additionally, it holds the values

[9]The presented assertions are indeed compiled away by `emcc`. In this particular case, the compiler can be convinced not to apply this optimization by removing the `static` keyword from `trusted`'s declaration, thus keeping the assertions without changing the semantics of the WebAssembly code.

[10]Instantiation and invocation are described in full detail in [29] in sections 4.5.4 and 4.5.5. We additionally assume, for the sake of simplicity, that the `start` function, an optional function that is automatically executed at initialization time that can act as "constructor"

of the arguments, globals, and the memory cell when the function was called. The last three values are kept to provide some form of context sensitivity when calling functions since the call stack itself is not modeled in our analysis.

$$\begin{aligned} \mathit{MState}_{\mathbf{fid}\times\mathbf{pc}}: \quad & \mathbb{B}_{64}{}^{\mathsf{SI.ss}(\mathbf{fid},\mathbf{pc})} \times \mathbb{B}_{64}{}^{\mathsf{SI.gs}()} \times \mathbb{B}_{64}{}^{\mathsf{SI.ls}(\mathbf{fid})} \times \\ & \mathit{Memory} \times \mathbb{B}_{64}{}^{\mathsf{SI.as}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{SI.gs}()} \times \mathit{Memory} \end{aligned}$$

Similarly, $\mathit{Return}_{\mathbf{fid}}$ models that a function **fid** returned the values and (potentially) modified globals and a memory cell (first three arguments) when called with the arguments, globals, and a memory cell in the last three arguments.

$$\begin{aligned} \mathit{Return}_{\mathbf{fid}}: \quad & \mathbb{B}_{64}{}^{\mathsf{SI.rs}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{SI.gs}()} \times \mathit{Memory} \times \\ & \mathbb{B}_{64}{}^{\mathsf{SI.as}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{SI.gs}()} \times \mathit{Memory} \end{aligned}$$

3) Abstraction: The abstraction function $\alpha(\cdot)$ is described in Fig. 4. Due to space restrictions, we refer to appendix A2 for the discussion of all details and here only highlight the overall structure.

$\alpha_T(\cdot)$ generates sets of facts that describe a certain point in the execution, while $\alpha_C(\cdot)$ generates Horn clauses describing all possible behaviors for each function **fid**, by applying the instruction abstraction function $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ to each instruction at **pc**. $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$'s definition is extensive, so we will only show one example here. The full definition is, however, given in appendix A2.

The instruction abstraction for **local.get idx**, an instruction that copies the local value identified by the immediate **idx** to the top of the stack, is given below:

$$\begin{aligned} &(\!|\textbf{local.get idx}|\!)^{\mathbf{fid}}_{\mathbf{pc}} = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0) \\ &\implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\mathit{lt}[\mathbf{idx}] :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0)\} \end{aligned}$$

This rule describes that if an abstract configuration contains an *MState*-fact where a **local.get idx** appears in the code, we can logically derive a similar *MState*-fact, that only differs in the program counter and the contents of the stack, to which $\mathit{lt}[\mathbf{idx}]$, the accessed local, is pushed.

a) Abstracting the Memory: To address the challenge of soundly handling arbitrary memory access (section II-B3) we gave special consideration to our memory abstraction, which we chose to be general and precise and is inspired by Monniaux et al. [15]. For every possible memory cell, an abstract configuration has to contain a separate *MState*-fact. Rules that do not touch the memory, such as the one above, can just propagate the memory cell, which is efficient. Rules that read the memory have to have a separate *MState*-predicate for every byte read in the premises, which is how we can handle unaligned reads. Under the assumption that there are no unaligned writes, one could instead have the memory cells only hold values of a certain size (e.g., 32 or 64 bits), which would increase performance. Furthermore, this memory model makes the future implementation of the bulk memory operations introduced in Wasm 2.0 easy.

4) Overapproximation of Host Functions: As mentioned in section II-B2, host functions can change the store in multiple ways. Of these possible changes to the store, adding functions and modifying the table is the most problematic for the analysis: When generating the Horn clauses, we must already know all possible behaviors of the module. Every execution of a **call_indirect** instruction looks up the function to be executed in the table. Therefore, the Horn clauses for all possibly called functions have to be generated. Since a call to a host function may add functions with arbitrary behavior to the store *and* add these functions to the table, all subsequent executions of **call_indirect** could possibly show this arbitrary behavior.

Overapproximating this case is possible, but doing so for every call to an imported function would be detrimental to our analysis' precision. To mitigate this problem, we introduce the following (partial) functions which allow specification of restrictions on the behavior of host functions.

$$\begin{aligned} \mathsf{SI.bnd}_G &: \mathbb{N} \times \mathbb{N} \mapsto (\mathbb{B}^?_{64} \times \mathbb{B}^?_{64}) \cup \{\circledcirc\} \\ \mathsf{SI.bnd}_R &: \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}^?_{64} \times \mathbb{B}^?_{64} \\ \mathsf{SI.sm} &: \mathbb{N} \mapsto \mathcal{P}(\{\textsc{tm}, \textsc{gm}, \textsc{tt}, \textsc{af}\}) \end{aligned}$$

For every id of an imported function and every global (identified by its index) $\mathsf{SI.bnd}_G$ either assigns (optional) lower and upper bounds on the resulting values or a special value $\circledcirc$ if the function cannot modify the corresponding global (especially if it is immutable). Similarly, $\mathsf{SI.bnd}_R$ returns a valid range of possible return values.

SI.sm returns a set of semantic markers for each imported function: TM means that the function can modify the memory (but not grow it), GM means that it may grow the memory (where new memory will be zero-initialized), TT means that the function table may be changed (but no functions might be added), and AF means that new, unknown functions may be added to the store (this marker also implies that the table is changed, as adding functions without changing the table would be unobservable). The absence of one of these markers means the opposite (e.g., $\textsc{tm} \notin \mathsf{SI.sm}(\mathbf{fid})$ means that the function **fid** may not touch the memory).

In Fig. 5, the abstractions for imported functions **fid** are presented. In ❶, the *MState* for program counter 0 directly implies a corresponding *Return* fact, where the free variables bound in it are restricted in the premises by the specification of host behavior. We use $\overset{?}{\leq}$ to denote "less than or equal" on an optional bound (returning *true* if no bound is given). The return values *rt* and returned globals *rgt* are restricted by $\mathsf{SI.bnd}_R$ and $\mathsf{SI.bnd}_G$ (whereby the contents of *rgt* are propagated unchanged if no interval is returned). The propagated values for memory contents and size depend on the set returned by SI.sm. Rule ❷, allows to derive arbitrary *Table*-facts (see section A1 in the appendix for details), while ❸ implies the *FunctionsAdded* fact, which allows the abstraction of **call_indirect** to derive arbitrary store contents.

5) Soundness Statement: We start by formalizing the equivalence of the original and annotated semantics.

a) Equivalence of Original and Annotated Semantics: The annotated semantics and the original semantics differ in two main aspects, namely, annotations and fused rules. The former have no semantic import, as they are just used by the

$$\alpha_V(x) = \begin{cases} \{zeropad(c)\} & \text{if } x = \mathsf{i32.const}\ c \\ \{c\} & \text{if } x = \mathsf{i64.const}\ c \\ \{zeropad(y) \mid y \in \mathbb{B}_{32}\} & \text{if } x = \mathsf{f32.const}\ c \\ \mathbb{B}_{64} & \text{if } x = \mathsf{f64.const}\ c \end{cases} \qquad \alpha_{seq}(s) = \begin{cases} \{y :: ys \mid y \in \alpha_V(x), ys \in \alpha_{seq}(xs)\} & \text{if } s = x :: xs \\ \{\epsilon\} & \text{otherwise} \end{cases}$$

$$\alpha_T(S; F; instr^*) = \begin{cases} \alpha_F(F.\mathsf{fid}, F'.\mathsf{pc}, F'.\mathsf{stor}, F, F'.\mathsf{args}\ F'.\mathsf{index}, instr^*) \cup \alpha_T\big(S; F'; instr^{*\prime}\big) & \text{if } instr^* = E[\mathbf{frame}_n\{F'\}\ instr^{*\prime}\ \mathbf{end}] \\ \alpha_F(F.\mathsf{fid}, \mathbf{pc}, S, F, \epsilon, instr^*) & \text{if } instr^* = E[\mathbf{cmd_{pc}}] \\ \alpha_{trap}(F.\mathsf{fid}, F.\mathsf{stor}, F) & \text{if } instr^* = E[\mathbf{trap}] \end{cases}$$

$$\alpha_F\begin{pmatrix}\mathbf{fid}, \mathbf{pc}, S,\\ F, p, instr^*\end{pmatrix} = \begin{array}{l}\big\{MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \mid st \in \alpha_{seq}(\eta_S(instr^*) +\!\!+ p), gt \in \alpha_{seq}(\eta_G(S, F)), lt \in \alpha_{seq}(F.\mathsf{locals}),\\ \quad at_0 \in \alpha_{seq}(F.\mathsf{args}), gt_0 \in \alpha_{seq}(\eta_G(F.\mathsf{stor}, F)), 0 \le i \le \eta_{MS}(S, F), mem = \alpha_M(i, S, F), mem_0 = \alpha_M(i, F.\mathsf{stor}, F)\big\}\\ \cup\ \alpha_t(S, F)\end{array}$$

$$\alpha_{trap}(\mathbf{fid}, S, F) = \big\{Trap_{\mathbf{fid}}(at_0, gt_0, mem_0) \mid at_0 \in \alpha_{seq}(F.\mathsf{args}), gt_0 \in \alpha_{seq}(\eta_G(S, F)), 0 \le i \le \eta_{MS}(S, F), mem_0 = \eta_M(i, S, F)\big\}$$

$$\alpha_t(S, F) = \big\{Table(i, e[i], |e|) \mid S.\mathsf{tables}[F.\mathsf{moduleinst.tableaddrs}[0]].\mathsf{elem} = e, 0 \le i < |e|\big\}$$

$$\eta_S(instr^*) = \begin{cases} \mathsf{t.const}\ x :: \eta_S\big(instr^{*\prime}\big) & \text{if } instr^* = \mathsf{t.const}\ x :: instr^{*\prime} \\ \eta_S\big(instr^{*\prime}\big) & \text{if } instr^* = \mathbf{label}_n\{\} :: instr^{*\prime} \\ \epsilon & \text{otherwise} \end{cases}$$

$$\eta_G(S, F) = \eta_G(S, F) = \eta'_G(S, F, 0)$$

$$\eta'_G(S, F, i) = \begin{cases} x = S.\mathsf{globals}[F.\mathsf{module.globaladdrs}[i]].\mathsf{value} :: \eta'_G(S, F, i+1) & \text{if } i < |F.\mathsf{module.globaladdrs}[i]| \\ \epsilon & \text{otherwise} \end{cases}$$

$$\eta_{MS}(S, F) = \begin{cases} 2^{32} & \text{if } \eta_M(S, F).\mathsf{max} = \epsilon \\ \eta_M(S, F).\mathsf{max} \cdot 2^{16} & \text{otherwise} \end{cases}$$

$$\eta_M(S, F) = S.\mathsf{mems}[F.\mathsf{moduleinst.memaddrs}[0]]$$

$$\alpha_M(i, S, F) = \begin{cases} Mem\big(i, d[i], |d|/2^{16}\big) & \text{if } i < |d| \\ Mem\big(i, 0, |d|/2^{16}\big) & \text{otherwise} \end{cases} \quad \text{where } d = \eta_M(S, F).\mathsf{data}$$

$$\alpha_C(S) = \bigcup_{\substack{0 \le \mathbf{fid} < |S.\mathsf{funcs}|,\ \mathbf{cmd_{pc}} \in annotate(\mathbf{block}\ f.\mathsf{type}\ f.\mathsf{code}\ \mathbf{end}) \\ f = S.\mathsf{funcs}[\mathbf{fid}]}} \bigcup \begin{cases} (\!|\mathbf{if}|\!)^{\mathbf{fid}}_{\mathbf{pc}} \cup (\!|\mathbf{block}\ |\!)^{\mathbf{fid}}_{\mathbf{pc}+1} & \text{if } \mathbf{cmd} = \mathbf{if} \\ (\!|\mathbf{block}\ |\!)^{\mathbf{fid}}_{\mathbf{pc}} \cup (\!|\mathbf{end}|\!)^{\mathbf{fid}}_{\mathbf{pc}-1} & \text{if } \mathbf{cmd} = \mathbf{else} \\ (\!|\mathbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}} & \text{otherwise} \end{cases}$$

$$\alpha(S; F; instr^*) = \alpha_T(S; F; instr^*) \cup \alpha_C(S)$$

Fig. 4: The abstraction function for configurations.

❶ $$MState_{\mathbf{fid},0}(st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge \big(\textsc{tm} \in \mathsf{SI.sm}(\mathbf{fid})\big)\ ?\ \big(0 \le v' \le 255\big) : \big(v = v'\big) \wedge$$
$$\bigwedge_{0 \le \mathbf{i} < \mathsf{SI.gs}()} \Big((\mathbf{lb}, \mathbf{ub}) = \mathsf{SI.bnd}_G(\mathbf{fid}, \mathbf{i})\Big)\ ?\ \Big(\mathbf{lb} \overset{?}{\le} rgt[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) : \Big(rgt[\mathbf{i}] = gt[\mathbf{i}]\Big) \wedge$$
$$\bigwedge_{\substack{0 \le \mathbf{i} < \mathsf{SI.rs}(\mathbf{fid}) \\ (\mathbf{lb}, \mathbf{ub}) = \mathsf{SI.bnd}_R(\mathbf{fid}, \mathbf{i})}} \Big(\mathbf{lb} \overset{?}{\le} rt[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) \wedge \big(\textsc{gm} \in \mathsf{SI.sm}(\mathbf{fid})\big)\ ?\ \big(size \le size' \le \mathbf{max}\big) : \big(size = size'\big) \implies$$
$$Return_{\mathbf{fid}}\big(rt, rgt, Mem\big(i, v', size'\big), at_0, gt_0, mem_0\big)$$

❷ $$MState_{\mathbf{fid},0}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge \textsc{tt} \in \mathsf{SI.sm}(\mathbf{fid}) \implies Table(idx, te, tsz)$$

❸ $$MState_{\mathbf{fid},0}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge \textsc{af} \in \mathsf{SI.sm}(\mathbf{fid}) \implies FunctionsAdded()$$

Fig. 5: Abstractions for imported functions.

abstract semantics, whereas the latter take two steps at once and, thus, do have a semantic import. We prove, however, that the two semantics are equivalent.

Definition 1 (Fusing Configuration)**.** We call a configuration $c = S; F; instr^*$ *fusing*, formally $f(c)$, if $instr^*$ can be factored into an evaluation context of either of the following forms by applying the structural rules: $E[(\mathsf{i32.const}\ c)\ \mathbf{br\_if}\ k]$ (for $c \neq 0$), $E[\mathbf{br\_table}]$, $E[\mathbf{local.tee}\ x]$, $E[\mathbf{call}]$, or $E[\mathbf{call\_indirect}]$.

Theorem 1 (Equivalence)**.** *Given a function $\beta(\cdot)$ which removes all annotations, it holds that*

$$\forall c_1, c_2 \Big(c_1 \twoheadrightarrow c_2 \iff \big(\neg f(c_1) \wedge \beta(c_1) \hookrightarrow \beta(c_2)\big) \vee \big(f(c_1) \wedge \exists c' (\beta(c_1) \hookrightarrow c' \hookrightarrow \beta(c_2))\big)\Big)$$

b) Soundness of Abstract Semantics: To prove the abstract semantics sound, we need to introduce an order on abstract configurations, which characterizes their precision. In WAPPLER, abstract configurations are sets of logical formulas comprising facts (predicate applications), which describe configurations, and Horn clauses, which describe the behavior of the module. Our refinement relation $\geq$, where $\Delta_1 \geq \Delta_2$ means that Δ_1 abstracts all concrete states that Δ_2 abstracts (and possibly more), can be defined as set inclusion:

$$\Delta_1 \geq \Delta_2 \iff \Delta_1 \supseteq \Delta_2$$

To express that a set of formulas Δ_2 is logically derivable from Δ_1 by applying a standard first-order calculus (e.g. SMT-solving or resolution), we write $\Delta_1 \vdash \Delta_2$. With these definitions, we can define a soundness theorem:

Theorem 2 (Soundness of Reachability)**.** *For all concrete configurations c_1 and c_2, where c_2 can be derived from c_1 by applying the transition function arbitrarily many times, it holds that if Δ_1 abstracts c_1, we can logically derive an abstract configuration Δ_2 from Δ_1 that abstracts c_2.*

$$\forall c_1, c_2, \Delta_1 \Big(c_1 \overset{*}{\twoheadrightarrow} c_2 \wedge \Delta_1 \geq \alpha(c_1) \implies \exists \Delta_2 \, (\Delta_1 \vdash \Delta_2 \wedge \Delta_2 \geq \alpha(c_2)) \Big)$$

We prove this result in appendix B (cf. Corollary 3).

Corollary 1 (Soundness of Reachability (original))**.**

$$\forall c_1, c_2, \Delta_1 \Big(\beta(c_1) \overset{*}{\hookrightarrow} c' \hookrightarrow \beta(c_2) \wedge \neg f(c') \wedge \Delta_1 \geq \alpha(c_1) \implies \exists \Delta_2 \, (\Delta_1 \vdash \Delta_2 \wedge \Delta_2 \geq \alpha(c_2)) \Big)$$

Proof. By Theorem 2 and the transitive closure of Theorem 1. More details are available in appendix B. □

To illustrate this concept, consider the following configuration, which may appear during the execution of the code in Fig. 3.

$$c_1 = S; F_0; \mathbf{frame}_1\{F\}\, \mathbf{label}_0\{\epsilon\}(\text{i32.}\mathbf{const}\ \textit{1031})\,(\mathbf{local.get}_{16}\ \textit{2})\ \text{i32.}\mathbf{store8}_{17}\ \mathbf{end}_{18}\ \mathbf{end}_{19}$$

Assuming that $x = 4294967271$, $F.\mathsf{fid} = 0$, $F.\mathsf{locals} = F.\mathsf{args} = [x, 0, 5]$, and that the module contains no global variables, Δ_1 with $\Delta_1 \geq \alpha(c)$ has to contain at least one fact $\mathit{MState}_{0,16}([1031], [], [x, 0, 5], \mathit{mem}, [x, 0, 5], [], \mathit{mem}_0)$ for every pair of memory cells $(\mathit{mem}, \mathit{mem}_0)$, where mem is a memory cell from the current memory, and mem_0 is a memory cell at the time the function was called (see section III-C3a for details on the memory) because $\alpha(c_1)$ contains such facts. An abstract configuration that additionally contained facts of the form $\mathit{MState}_{0,16}([1032], [], [x+1, 0, 5], \mathit{mem}, [x+1, 0, 5], [], \mathit{mem}_0)$ (e.g., from a call with different arguments) would be more abstract than Δ_1, but, as long as it contained all formulas from $\alpha(c_1)$, would still be a valid abstraction. Additionally, Δ_1 has to contain a Horn clause $\mathit{MState}_{0,16}(\mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0) \implies \mathit{MState}_{0,17}(\mathit{lt}[2] :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0)$ (an instantiation of the rule from section III-C3), since the code of the module contains an instance of **local.get** *2* in function 0, at program counter 16. The next concrete configuration c_2 looks like

$$c_2 = S; F_0; \mathbf{frame}_1\{F\}\, \mathbf{label}_0\{\epsilon\}\,(\text{i32.}\mathbf{const}\ \textit{1031})\,(\text{i32.}\mathbf{const}\ \textit{5})\ \text{i32.}\mathbf{store8}_{17}\ \mathbf{end}_{18}\ \mathbf{end}_{19}$$

which means that any abstraction $\Delta_2 \geq \alpha(c_2)$ has to contain facts of the form $\mathit{MState}_{0,17}([5, 1031], [], [x, 0, 5], \mathit{mem}, [x, 0, 5], [], \mathit{mem}_0)$. Since Δ_1 contains facts of the form $\mathit{MState}_{0,16}$ and a Horn clause that allows logically deriving $\mathit{MState}_{0,17}$-facts from $\mathit{MState}_{0,16}$-facts, we can logically derive Δ_2 from Δ_1, or, more formally, we have $\Delta_1 \vdash \Delta_2$.

Given that the memory cell 1031 from Fig. 3 contains part of `"TRUSTED"`, c_2 violates *No-Sensitive-Overwrite* from section III-B2. The soundness theorem now guarantees that if there is a concrete (non-intermediate) configuration violating any reachability property, there is also an abstract configuration violating an abstraction of that property. The abstractions for *No-Sensitive-Overwrite* and the other properties from section III-B are presented in section IV-B.

Intuitively, the proof of soundness ensures that such logical derivations are possible for all modules. For that, one must first establish that the structural rules are abstracted correctly, then prove each possible instruction case-by-case.

IV. IMPLEMENTATION AND EVALUATION

A. Implementation

WAPPLER was implemented using the HORST framework [25] which provides a domain specific language for specifying Horn-clause-based abstractions. The specification of the abstract semantics itself is around 1900 lines of HORST code that is structured in the same way as the specification. Additionally, there is JAVA™ code to parse Wasm modules and calculate static information like program counters and stack layouts. Since this HORST did not yet support tuple types (which we use excessively for stacks, locals, globals, etc.), we extended its type system to support tuples of compile-time-known lengths that may depend on other compile-time-known constants (such as the parameters of predicates, e.g. program counters and function ids). These changes will be included in the next release of HORST.

B. Reachability Properties

In the following, we present several use cases for reachability properties. All experiments were conducted on a virtual machine with 64 CPU cores (AMD EPYC 7713, limited to 2GHz) and 128 GiB RAM.

1) Verifying Interactions with Host Functions: Consider the function from Fig. 1. It is meant to calculate the absolute value of a function f that is provided by the embedder via the imported table. We cannot make any assumptions on the behavior of f but can check that the result of the

composed function (to which we assign the id 0) is never negative by querying if the following formula is unsatisfiable. Since the table is imported, our abstract execution starts with the *FunctionsAdded* fact already in the abstract configuration (which allows the abstraction of **call_indirect** (line 6) to return arbitrary values (see rule 46 in the appendix for details).

$$\mathit{Return}_0(\mathit{rt}, \mathit{gt}, \mathit{rmem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0) \wedge \mathit{rt}[0] < 0$$

Perhaps surprisingly, the formula is satisfiable, as our function has a subtle bug. When instructing z3 to output a model for the formula, we learn that if the f returns -2^{31}, the multiplication in line 15 will overflow since 2^{31} is not representable as a signed 32-bit integer. The generation of the model (respectively proving a fixed version secure) takes z3 under a second.

2) Verifying General Reachability Properties: To analyze `update_game_state` from Fig. 3, we use our overapproximation of imported functions to check all possible inputs. For that, we extend the C file with the following definitions:

```
1 extern int nondetint(void);
2 extern char nondetchar(void);
3 ...
4 void run_test() {
5   update_game_state(nondetint(),nondetint(),
      nondetchar());
6 }
```

Since the `nondetint` and `nondetchar` functions are imported, they can return any possible value, meaning that if there are values that lead to a property violation, they will be found. The *No*-i32.**add**-*Overflow* property from section III-B1 can be overapproximated by verifying the unsatisfiability of the formula below for every occurrence of i32.**add** in the module (with $\mathbf{fid}$ and $\mathbf{pc}$ instantiated to appropriate values).

$$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: y :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0)\\ &\quad \wedge (x < 0) = (y < 0) \wedge (x + y < 0) \neq (y < 0)\end{aligned}$$

Similarly, we can overapproximate *Non-Sensitive-Overwrite* from section III-B2 as follows (assuming suitable instantiations for $\mathbf{fid}$, $\mathbf{pc}$, $\mathbf{low}$, $\mathbf{high}$, and $\mathbf{offset}$)

$$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(c :: i :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0)\\ &\wedge \mathbf{low} \leq \mathbf{offset} + i + N/8 \wedge \mathbf{offset} + i \leq \mathbf{high})\end{aligned}$$

For both of these properties, z3 can prove presence (or absence in a fixed version) in under one second.

For general assertion checking, it suffices to show that the following formula is unsatisfiable (assuming $\mathbf{fid}$ is the id of `reach_error`):

$$\mathit{MState}_{\mathbf{fid},0}(\mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0)$$

Proving that the assertions presented in III-B3 hold needs, however, additional information. Without any annotations, the analysis assumes that calls to `nondetint` and `nondetchar` can write arbitrary values to the linear memory. Since the assertions check the contents of `trusted` *after* these functions have been called, the analysis assumes that `trusted` contains arbitrary values, thereby violating the assertion. `nondetint` and `nondetchar` are not meant to touch the store, therefore we set $\mathsf{SI.sm}(\mathbf{fid}) = \{\}$ (where $\mathbf{fid}$ are the respective functions ids). With these assumptions in place, we can show an assertion violation in the original version in less than a second and prove the fixed version secure in 55 seconds.

We suggest that the increase in the time needed to verify this property comes down to two reasons: Firstly, there are eight assertions to check in this program, which leads to a more complicated control flow to `reach_error`, compared to the other two properties where we had to check only two i32.**add**, respectively one i32.**store**. Secondly, and more relevant, verifying this property requires us to reason about the memory *contents*, which we do handle soundly but without taking advantage of any assumptions (such as having only aligned accesses) or analyses of pointer relationships [8].

C. WebAssembly Specification Test Suite

To corroborate the correctness of our implementation, we analyzed the relevant part of the WebAssembly specification with our tool. The test suite (at the last commit that only tests Wasm 1.0 features [37]) consists of 74 files that specify the intended behavior of a WebAssembly implementation. The tests we are interested in specify invocations of functions and their respective return values (respectively that they trap), whereby the execution of one invocation might influence the next one (i.e., the state is shared between them). 59 of the aforementioned files contain such tests. We extract the Wasm modules (523) and associated test cases (15776) and exclude 22 modules (115 test cases) as they rely on features we currently do not support (such as the interplay between multiple modules). For the remaining 15661 tests, we check for soundness (is the specified result derivable) and precision (is the specified result the only one that is reachable). We group the tests into groups that might depend on each other by a simple heuristic, generate the SMT files, and analyze them with z3. Within a 10s timeout, z3 can prove that the correct result is reachable for 15349 (98%) tests. The majority (196 of 312) of the tests that time out come from three modules that describe complicated control flow and memory interactions (`align`, `endian`, and `left-to-right`). The test cases in these modules are wrongly classified as interdependent by our heuristic. By treating the tests from the aforementioned modules as independent of each other and increasing the time out to 30s, z3 can solve 15514 (99%) of the soundness queries. Concerning precision, we first observe that the majority of test cases describe the intricacies of floating point calculations, which we do not handle precisely. If we exclude all tests from files that explicitly concern floats or contain floats in their arguments or return values, we are left with 2851 test cases. Within a 10s timeout, we can prove 2502 (88%) of them precise. Of the 349 (12%) failing tests 264 (9%) are due to timing out and 85 (3%) are due to imprecisions of the analysis (and therefore overapproximated). We identified two sources of imprecision: imported functions and functions that return different values with the same input parameters (such as functions that grow the memory and return its new size). While

the former imprecisions could be alleviated by specifying the semantics of the imported functions, the latter could be alleviated by changing the test setup to integrate values such as the initial memory size into the test specification.

D. Scope and Limitations

The aim of this work is to provide automated, sound, and yet precise, static analysis of WebAssembly, which is why we focused on the principal viability of using reachability properties for security analysis and corroborating the implementation's soundness by analyzing the official test suite. The tool proves effective on our benchmarks, but several improvements are possible. First, analyzing large modules with complex control flows and many accesses to the unstructured linear memory might make the analysis very expensive or even lead to non-termination. Hence, an interesting direction for future research is to devise preprocessing techniques, possibly leveraging compiler-provided information whenever the source code is available, to gather higher-level information on memory access patterns before performing a fully-fledged analysis.

Furthermore, we leave a precise modeling of floating point numbers as a future work since their implementation in HORST would not pose any conceptual challenge but require a non-negligible engineering effort without contributing to clarify the core ideas introduced in this work.

V. Related Work

While there are either sound or automated static analysis techniques for WebAssembly, to the best of our knowledge, we present the first approach that is both sound *and* automated.

The possibly first static analysis tool for WebAssembly discussed in academic literature was presented by Lehmann et al. [11]. The presented tool was used to gather information on memory usage patterns and control flows but not to assess security properties in the narrower sense.

Since one of the early widespread uses for WebAssembly was the illicit mining of cryptocurrencies in web browsers (so-called *cryptojacking*) [16], there have been multiple tools that aim to classify binaries as cryptojackers. These include MinerRay [23], MineSweeper [10], and MINOS [18]. None of these come with a soundness guarantee, since the target property does not lend itself to be formalized.

The second group of automated static analysis tools for WebAssembly aims to provide more general analyses, although none of them claims to be sound. Wasmati [3] generates so-called code property graphs (CPGs), that combine information about control flow, data flow, and syntax in a single structure. These graphs are then queried for patterns that indicate different types of vulnerabilities (such as use-after-free or buffer overflows). The aim of Wasmati, however, is scanning for vulnerabilities efficiently not proving their absence. This is also evident in its evaluation which mentions some false negatives. Wasp [12] is an analysis based on concolic execution, a variant of symbolic execution that integrates concrete executions. As such, it can generate inputs that trigger certain program behaviors (thereby providing witnesses for reachability properties) but can not exclude problematic program behaviors. Wassail [27] is a tool that computes function summaries on control flow graphs in a compositional manner. These function summaries track which input values (arguments and globals) may influence output values (return values or again globals) via taint tracking. Wassail does not track memory flows through the linear memory as a deliberate choice, thereby trading soundness for precision. While Wassail can, due to its compositional nature, analyze far bigger datasets than WAPPLER, its summaries are less expressive than WAPPLER's general reachability properties.

A different group of publications made an effort to prove properties of WebAssembly modules or WebAssembly itself in mechanized (but not automated) proof assistants, starting with the mechanization of WebAssembly in [31] that was later extended in [33]. [32] introduces Wasm Logic, a sound program logic with which the authors were able to verify a B-Tree implementation. [22] focuses on the interaction between modules and presents a mechanized higher-order separation logic that they use to reason about a stack-module that makes use use of imported functions.

In the realm of mitigations for memory unsafety, several approaches have been proposed. MSWasm [5], [14] is a memory-safe extension to WebAssembly that can make use of higher-level information to protect against memory safety violations. [28] surveys several methods with which a compiler could compile a memory-safe language to Wasm. One discussed approach makes use of existing Wasm features, such as using different modules, the other targets MSWasm. Additionally, there exists a proposal for supporting multiple memories [35], which would allow for implementing syntactically verifiable read-only memories.

Regarding the modeling of external functions, we are unaware of any tool that provides such a mechanism for WebAssembly. For LLVM, there is SeaHorn [7], which, by default, does not model all possible behaviors of external functions but offers special functions to approximate the behavior of external functions on the C/C++ level. [19] JNI [21], the primary technology used to implement native functions in JAVA™, provides a standardized interface on both the JAVA™ and the C/C++ level. This interface allows static analyses to reason about the interaction between inside and outside of the Java Virtual Machine. For WebAssembly, currently no such standard exists. There is, however, the component model proposal [36], which aims to "the definition of portable, virtualizable, statically-analyzable, capability-safe, language-agnostic interfaces", which could be used to enable such analyses in the future.

VI. Conclusion and Future Work

We presented WAPPLER, the first sound and automated static analysis technique for WebAssembly, and the formalization of several general and Wasm-specific security properties. To facilitate a more efficient analysis, we first annotated the original semantics with program counters and function identifiers and then abstracted this annotated semantics using

Horn clauses. We then formulated several security properties and evaluated them on our examples. Lastly, we corroborated the soundness of our implementation by analyzing all 15k+ applicable test cases from the official test suite.

In terms of future work, we identify multiple possible research directions: To improve performance, one could try to make the analysis more efficient by using 32-bit values where possible. This would require substantial additions to HORST's type system or a less elegant specification using separate stacks for different types. Regarding feature completeness, precise handling of floating point numbers (which is contingent on their support in the underlying solvers), support for analyzing multiple modules, or implementing the features in Wasm 2.0 are possible extensions.

A more theoretical endeavor would be extending our analysis to more expressive classes than reachability properties, such as k-safety properties. To make (sound) analyses scale, the composability of proof results is essential. One possible strategy for analyses like ours could be the automated extraction of function invariants.

Lastly, while it is already possible for specialist users to extract the inputs that lead to property violations from the models that the SMT solvers generate, automatic reconstruction of attack traces would increase the usefulness of our analysis for a larger audience.

Acknowledgements. We thank our anonymous reviewers for their valuable feedback. This work was partially supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the SpyCode SFB project F8510-N; by the Austrian Research Promotion Agency (FFG) through the COMET K1 SBA; by the Vienna Science and Technology Fund (WWTF) through the project ForSmart; and by the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT).

REFERENCES

[1] Ethereum webassembly (ewasm). https://ewasm.readthedocs.io/en/mkdocs/, Jan 23, 2020. [Online; accessed 19. April 2023].

[2] Eos virtual machine: A high-performance blockchain webassembly interpreter. https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter/, June 20, 2019. [Online; accessed 19. April 2023].

[3] T. Brito, P. Lopes, N. Santos, and J. F. Santos. Wasmati: An efficient static vulnerability scanner for webassembly. *Computers & Security*, 118:102745, 2022.

[4] Bytecode Alliance. wasmtime. https://github.com/bytecodealliance/wasmtime.

[5] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019.

[6] B. C. Gain. When webassembly replaces docker. https://thenewstack.io/when-webassembly-replaces-docker/, Jun 7, 2022. [Online; accessed 20. April 2023].

[7] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361. Springer, 2015.

[8] A. Gurfinkel and J. A. Navas. Abstract interpretation of LLVM with a region-based memory model. In R. Bloem, R. Dimitrova, C. Fan, and N. Sharygina, editors, *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 122–144. Springer, 2021.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the Web up to Speed with WebAssembly. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[10] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1714–1730, 2018.

[11] D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security*. USENIX Association, 2020.

[12] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão. Concolic execution for webassembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[13] P. Mendki. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pages 161–166. IEEE, 2020.

[14] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[15] D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 361–382. Springer, 2016.

[16] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 23–42. Springer, 2019.

[17] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 699–716, 2020.

[18] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac. Minos: A lightweight real-time cryptojacking detection system. In *NDSS*, 2021.

[19] J. Navas. Understand example involving pointers. https://github.com/seahorn/seahorn/issues/193#issuecomment-476969011.

[20] M. Nieke, L. Almstedt, and R. Kapitza. Edgedancer: Secure mobile webassembly services on the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, pages 13–18, 2021.

[21] Oracle. Jni apis and developer guides. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[22] X. Rao, A. L. Georges, M. Legoupil, C. Watt, J. Pichon-Pharabod, P. Gardner, and L. Birkedal. Iris-Wasm: Robust and modular verification of WebAssembly programs. In *Conference on Programming Language Design and Implementation (PLDI) (accepted)*, 2023.

[23] A. Romano, Y. Zheng, and W. Wang. Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1129–1140, 2020.

[24] M. Scherer, J. F. Blaabjerg, A. Sjösten, M. Solitro, and M. Maffei. WAPPLER: Sound reachability analysis for webassembly (technical report). https://secpriv.wien/wappler.

[25] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Conference on Computer and Communications Security (CCS)*. ACM, 2020.

[26] A. Shekhirin. Awesome webassembly plugins. https://github.com/shekhirin/awesome-webassembly-plugins, Oct 2, 2019. [Online; accessed 20. April 2023].

[27] Q. Stiévenart and C. De Roover. Compositional information flow analysis for webassembly programs. In *2020 IEEE 20th International Working*

Conference on Source Code Analysis and Manipulation (SCAM), pages 13–24. IEEE, 2020.
[28] M. Vassena and M. Patrignani. Memory safety preservation for webassembly. In *47th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2020.
[29] W3C. WebAssembly Core Specification, version 1.0. https://www.w3.org/TR/wasm-core-1/.
[30] E. Wallace. An update on plugin security. https://www.figma.com/blog/an-update-on-plugin-security/, Oct 2, 2019. [Online; accessed 20. April 2023].
[31] C. Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.
[32] C. Watt, P. Maksimović, N. R. Krishnaswami, and P. Gardner. A program logic for first-order encapsulated webassembly. In *33rd European Conference on Object-Oriented Programming*, 2019.
[33] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner. Two mechanisations of webassembly 1.0. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*, pages 61–79. Springer, 2021.
[34] C. Watt, M. Treia, P. Lammich, and F. Märkl. WasmRef-Isabelle: a verified monadic interpreter and industrial fuzzing oracle for webassembly. In *Conference on Programming Language Design and Implementation (PLDI) (accepted)*, 2023.
[35] WebAssembly Community Group. Multiple per-module memories for Wasm. https://github.com/WebAssembly/multi-memory.
[36] WebAssembly Community Group. Repository for design and specification of the component model. https://github.com/WebAssembly/component-model.
[37] WebAssembly Community Group. spec/test/core at f2b62c3067ac7e9e367296378621ccbd4fee79c1 · WebAssembly/spec. https://github.com/WebAssembly/spec/tree/f2b62c3067ac7e9e367296378621ccbd4fee79c1/test/core.
[38] WebAssembly Community Group. WebAssembly Core Specification, live document. https://webassembly.github.io/spec/core/.
[39] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *USENIX Security Symposium (accepted)*, 2023.

APPENDIX

A. Additional Details on Section III

1) Predicates and Types:

a) Optional Bit Vectors: In some situations, we want to allow a special value ϵ to denote an undefined value. We denote the set $\mathbb{B}_{64} \cup \{\epsilon\}$ by $\mathbb{B}_{64}^{?}$.

b) Description of $\mathsf{SI}.*$: We assume a globally available family of functions $\mathsf{SI}.*$ that provides this information: the existence of such a family follows from the soundness of WebAssembly type system. $\mathsf{SI.ss}(\mathbf{fid}, \mathbf{pc})$ returns the size of the value stack in a function $\mathbf{fid}$ at program counter $\mathbf{pc}$. $\mathsf{SI.gs}()$ returns the number of globals in the analyzed module. $\mathsf{SI.ls}(\mathbf{fid})$ returns the number of locals in the function $\mathbf{fid}$, while $\mathsf{SI.as}(\mathbf{fid})$ and $\mathsf{SI.rs}(\mathbf{fid})$ return its number of arguments and return values, respectively.

c) Description of Predicate Signature: $Return_{\mathbf{fid}}$ models that the function $\mathbf{fid}$ returns the values in the first argument with the modified globals and memory cell in the second and third argument. The last three values describe the arguments, globals, and memory when the function was called. Similarly, the $Trap_{\mathbf{fid}}$-predicate abstracts encountering an evaluation context $E[\mathsf{trap}]$ during the execution of a function $\mathbf{fid}$ where three arguments again describe the inputs with which the function was called.

Table encodes an entry in the function table: the first value is the index, the second the id of the function at this index (possibly undefined), and the last one the size of the table.

If *FunctionsAdded* is derivable, we are in a state where functions that are neither defined in nor imported by the analyzed module are callable. This is the case if the table is imported (and filled by the embedder) or if host functions add unknown functions to the table. The only case where *FunctionsAdded* is used is when handling **call_indirect** (see ❹❻, ❹❼).

2) Abstraction: $\alpha(\cdot)$ translates a concrete configuration to a set of predicate applications describing the current program state (generated by $\alpha_T(\cdot)$) and a set of Horn clauses describing all module behaviors (generated by $\alpha_C(\cdot)$). $\alpha_T(\cdot)$'s definition follows the structural rules: If we find an activation in the evaluation context, we do two things: Firstly, we reconstruct the state as it was when calling the function (putting the function's arguments and, in case of **call_indirect** also the function index, back on the stack) and abstract the state with $\alpha_F(\cdot)$; Secondly, we recurse into $\alpha_T(\cdot)$ with the function-local data and the instructions encapsulated by the activation.

If the current instruction sequence can be decomposed into $E[\mathsf{cmd}_{\mathbf{pc}}]$, that is, if we are in the currently executing function, we call $\alpha_F(\cdot)$, without reconstructing a state.

If a **trap** appears in an evaluation context, $\alpha_{trap}(\cdot)$ abstracts the configuration to a set of $Trap_{\mathbf{fid}}$-facts. These facts carry the inputs from when the function was called to increase precision.

$\alpha_F(\cdot)$ takes the function id and program counter of the abstracted configuration, in addition to a store S and frame F, a possible prefix of constant values p to put on the value stack, and the current instruction sequence $instr^*$. The constant stack values, globals, and locals, as well as the original arguments and globals, are extracted from the concrete configurations by a family of functions $\eta(\cdot)$ and abstracted by $\alpha_{seq}(\cdot)$. A call to $\alpha_t(\cdot)$ abstracts the table.

All values $\mathsf{t.const}\ v$ that appear in concrete configurations are abstracted by $\alpha_V(\cdot)$ to sets of 64-bit vectors. Singleton sets represent integer values; floating point values are not precisely tracked and are represented as sets containing all possible bit vectors of the correct size. 32-bit values are extended to 64-bit values by applying $zeropad(\cdot)$. $\alpha_{seq}(\cdot)$ lifts $\alpha_V(\cdot)$ to sequences.

Additionally, we instantiate one *MState* for each memory cell i, where i ranges from 0 to the maximum byte size of the memory. The current memory cell *mem* is extracted from the current store; the initial memory cell mem_0 from the copy of the store tracked in the frame object. This way of handling the memory is inspired by [15] and has the advantage of making the bulk memory operations introduced in Wasm 2.0 easier to implement in the future. $\alpha_t(\cdot)$ instantiates a *Table*-fact for every entry in the currents module instance's table.

$\alpha_C(S)$ abstracts the behavior of all functions in the store S[11]. For this, it iterates over all functions in S, assigning its index in funcs as function id. It annotates the code

[11]This function would look different when handling multiple modules.

$$annotate(\mathit{instr}^*) = annotate'(0, \mathit{instr}^*)$$

$$annotate'(pc, \mathit{instr}^*) = \begin{cases} \mathbf{if}_{pc} :: annotate'\big(pc+2, \mathit{instr}^{*\prime}\big) & \text{if } \mathit{instr}^* = \mathbf{if} :: \mathit{instr}^{*\prime} \\ \mathbf{else}_{pc+1} :: annotate'\big(pc+2, \mathit{instr}^{*\prime}\big) & \text{if } \mathit{instr}^* = \mathbf{else} :: \mathit{instr}^{*\prime} \\ \mathbf{cmd}_{pc} :: annotate'\big(pc+1, \mathit{instr}^{*\prime}\big) & \text{if } \mathit{instr}^* = \mathbf{cmd} :: \mathit{instr}^{*\prime} \wedge \mathbf{cmd} \neq \mathbf{if} \wedge \mathbf{cmd} \neq \mathbf{else} \\ \epsilon & \text{otherwise} \end{cases}$$

Fig. 6: Definition of the annotation function.

$$\begin{aligned} \mathit{MState}_{\mathbf{fid}\times\mathbf{pc}} &: \mathbb{B}_{64}{}^{\mathsf{Sl.ss}(\mathbf{fid},\mathbf{pc})} \times \mathbb{B}_{64}{}^{\mathsf{Sl.gs}()} \times \mathbb{B}_{64}{}^{\mathsf{Sl.ls}(\mathbf{fid})} \times \mathit{Memory} \times \mathbb{B}_{64}{}^{\mathsf{Sl.as}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{Sl.gs}()} \times \mathit{Memory} \\ \mathit{Return}_{\mathbf{fid}} &: \mathbb{B}_{64}{}^{\mathsf{Sl.rs}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{Sl.gs}()} \times \mathit{Memory} \times \mathbb{B}_{64}{}^{\mathsf{Sl.as}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{Sl.gs}()} \times \mathit{Memory} \\ \mathit{Trap}_{\mathbf{fid}} &: \mathbb{B}_{64}{}^{\mathsf{Sl.as}(\mathbf{fid})} \times \mathbb{B}_{64}{}^{\mathsf{Sl.gs}()} \times \mathit{Memory} \\ \mathit{Table} &: \mathbb{B}_{64} \times \mathbb{B}_{64}^{?} \times \mathbb{B}_{64} \\ \mathit{FunctionsAdded} &: () \end{aligned}$$

Fig. 7: Predicate signature of the abstract analysis.

with $annotate(\cdot)$ and abstracts the command $\mathbf{cmd}_{\mathbf{pc}}$ with a function $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$. In the case of an **if** $\cdot$ **else** $\cdot$ **end**-construct, we introduce "virtual" blocks to handle their semantics more uniformly. $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ is given in Fig. 5, respectively Figs. 8-14.

❹ describes encountering a constant $\mathbf{c}$ in a functions code. There is no corresponding reduction rule to this abstract rule since the concrete semantics handles these instructions implicitly. For handling binary operations (❻-❽), we assume two families of binary functions $\cdot \circledast_{\mathbf{op}} \cdot : \mathbb{B}_{64} \times \mathbb{B}_{64} \mapsto \mathcal{P}(\mathbb{B}_{64})$ and $\cdot \widetilde{\circledast_{\mathbf{op}}} \cdot : \mathbb{B}_{64} \times \mathbb{B}_{64} \mapsto \mathcal{P}\big(\mathbb{B}_{64}^{?}\big)$ which return at least all results that the corresponding operation in the concrete semantics can yield (in practice, operations on integer type return singleton sets here, while floating point operation return all possible values). Then, we just take the two topmost from the stack and add the result to the stack (❻,❼). If the result can be ϵ, we imply $\mathit{Trap}_{\mathbf{fid}}$ (❼).

For a more complicated rule, consider the rule abstracting **call_indirect** (❹❶-❹❺). For each instance of **call_indirect** we instantiate this rule for each possibly called function $\mathbf{cid}$ (where $\mathbf{cid}$ ranges over all functions of the correct type that are declared in the module). As an additional premise, we require that a *Table*-fact, which encodes that $\mathbf{cid}$ is stored at position x (where x is the topmost stack value), is derivable. ❹❶: To call a function, we take the correct amount of values from the stack[12], reverse it, and imply $\mathit{MState}_{\mathbf{cid},0}(\ \dots\ )$, the first position when executing $\mathbf{cid}$. The initial stack is empty, the globals *gt* and the memory cell *mem* are just copied, and the locals are set to the arguments *at* padded with zeroes. The last three arguments are initialized to the same values as the "working copies". ❹❷: Returning from a function looks similar: we construct *at* (the arguments) the same way as in ❹❶ and use it to match on a $\mathit{Return}_{\mathbf{fid}}$-fact. The returned values *rt* are then propagated to the next program counter in the calling function. ❹❸: Works similar to ❹❷ but propagates traps instead of values. ❹❻ and ❹❼ model the case that the embedder added a function that is neither imported nor defined by the model. It allows arbitrary return values, globals, memory contents, and traps to be derived. ❹❹ and ❹❺ encode the trap case for **call_indirect**. ❹❹: If the value in the table at index x does not correspond to one of the possible functions, we trap. ❹❺: Likewise, we trap if there is a table of smaller size than x.

[12]We use the slicing syntax for tuples like in programming languages like `python` or the original WebAssembly specification.

B. Proof of Soundness

For a regular execution (i.e. one that has not yet trapped), the next configurations is determined by the mutually recursive structural rules SR1 and SR2, and a set of non-structural rules which describe the semantics of different instructions. SR1 removes labels and values that are not important for the next rule that is applied from the instruction sequence, while SR2 changes the frame data in the recursive application of the step function. After the last application of SR1, a non-structural rule is applied.

Definition 2 (Non-trapping Configuration)**.** We call a configuration whose instruction sequence does not contain **trap** *non-trapping configuration*.

Definition 3 (Currently Executing Activation)**.** Within instruction sequence (or within the instruction sequence of a concrete configuration) we call a (well-nested) subsequence $\mathbf{frame}_n\{F\}\ \mathit{instr}^*\ \mathbf{end}$ the *currently executing activation*, if instr^* does not contain any activations.

Definition 4 (Currently Executing Instruction)**.** Non-structural rules can only be applied in the currently executing activation. These left-hand sides of theses rules are of either of these forms

- Regular instructions: $\mathit{val}^*\mathbf{cmd}_{\mathbf{pc}}$
- Branching instructions: $\mathbf{label}_n\{\mathit{instr}^*\}B^k[\mathit{val}^*\mathbf{cmd}_{\mathbf{pc}}]\mathbf{end}$
- Exiting a block: $\mathbf{label}_n\{\mathit{instr}^*\}\mathit{val}^*\mathbf{end}_{\mathbf{pc}}$

❹ $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(\mathbf{v} :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i32.**const** , . . .
❺ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge res \in unOp_{\mathbf{op}}(x) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i64.**clz**, f32.**abs**, . . .
❻ $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge res \in y \circledast_{\mathbf{op}} x \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i32.**add**, . . .
❼ $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge res \in y \widetilde{\circledast_{\mathbf{op}}} x \wedge res \in \mathbb{B}_{64} \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$ ❽ $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge res \in y \widetilde{\circledast_{\mathbf{op}}} x \wedge \neg(res \in \mathbb{B}_{64}) \implies$ $Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	i64.**mod**, . . .
❾ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(cvtOp_{\mathbf{op}}(x) :: st, gt, lt, mem, at_0, gt_0, mem_0)$	Conversions i32.**wrap_i64**, . . .
❿ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(cvtOp_{\mathbf{op}}(x) :: st, gt, lt, mem, at_0, gt_0, mem_0)$	Trapping Conversions
⓫ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	i32.**trunc_f64_u**, . . .

Fig. 8: $(\!|\textbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a numeric instruction **cmd** taken from the right column .

⓬ $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(lt[\mathbf{idx}] :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**local.get** $\mathbf{idx}$
⓭ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt[\mathbf{idx} \leftarrow x], mem, at_0, gt_0, mem_0)$	**local.set** $\mathbf{idx}$
⓮ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(x :: st, gt, lt[\mathbf{idx} \leftarrow x], mem, at_0, gt_0, mem_0)$	**local.tee** $\mathbf{idx}$
⓯ $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(gt[\mathbf{idx}] :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**global.get** $\mathbf{idx}$
⓰ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt[\mathbf{idx} \leftarrow x], lt, mem, at_0, gt_0, mem_0)$	**global.set** $\mathbf{idx}$

Fig. 9: $(\!|\textbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a variable instruction **cmd** taken from the right column .

⓱ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**drop**
⓲ $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: z :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x = 0 \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(y :: st, gt, lt, mem, at_0, gt_0, mem_0)$ ⓳ $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: z :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x \neq 0 \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(z :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**select**

Fig. 10: $(\!|\textbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a parametric instruction **cmd** taken from the right column .

- Explicit returning from a function: $\textbf{frame}_n\{F\}B^k[\textbf{return}_{\mathbf{pc}}]\textbf{end}$
- Returning at the end of a function: $\textbf{frame}_n\{F\}val^*\textbf{end}_{\mathbf{pc}}$

We call the instruction whose program counter annotation is

20 $x + \mathbf{offset} + N \div 8 - 1 < size \cdot 2^{16} \wedge MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge$ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(x + \mathbf{offset} + 0, vs[0], size), at_0, gt_0, mem_{0_0}) \wedge \ldots \wedge$ $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(x + \mathbf{offset} + N, vs[N], size), at_0, gt_0, mem_{0_0}) \wedge u = combine(vs) \wedge$ $w = extends\_sx_{N,\lvert t\rvert}(u) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(w :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0)$	t.**loadN_sx**
21 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge x + \mathbf{offset} + N \div 8 - 1 \geq size \cdot 2^{16} \implies$ $Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	
22 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge y + \mathbf{offset} + N \div 8 - 1 < size \cdot 2^{16} \wedge$ $d = i - y + \mathbf{offset} \wedge u = wrap_{\lvert t\rvert,N}(x) \wedge w = u \gg d \cdot 8 \;\&\; 255 \wedge v' = (d < N \div 8)\,?\,(w) : (v) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, Mem(i, v', size), at_0, gt_0, mem_0)$	t.**storeN_sx**
23 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge y + \mathbf{offset} + N \div 8 - 1 \geq size \cdot 2^{16} \implies$ $Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	
24 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(size :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0)$	**memory.size**
25 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge$ $ret = (growOK(x, size, \mathbf{max}))\,?\,(size) : (-1) \wedge size' = grow(x, size, \mathbf{max}) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(ret :: st, gt, lt, Mem(i, v, size'), at_0, gt_0, mem_0)$	**memory.grow**

Fig. 11: $(\!|\mathbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a memory instruction **cmd** taken from the right column .

26 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	**unreachable**
27 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**nop**
28 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**block**
29 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**loop**
30 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{next}}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**end**

Fig. 12: $(\!|\mathbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a control instruction **cmd** taken from the right column (part 1).

shown in the above listing the *currently executing instruction* and formally write c.cmd to designate the currently executing instruction of a configuration c. We identify the rules for exiting a block and returning at the end of a function with the program counter of the corresponding **end**. Only non-trapping configurations have a currently executing instruction.

Definition 5 (Local Instructions)**.** Using the categorization from above, we call regular instructions excluding **call** x and **call_indirect**, branching instructions and exiting a block *local instructions*.

Definition 6 (Constant Prefix)**.** Given a sequence of annotated instructions $instr^*$ that does not contain any $\mathbf{frame}_n\{F\} \ldots \mathbf{end}$ instructions and an annotated instruction $\mathbf{cmd}_{\mathbf{pc}}$ that appears at index i within $instr^*$, we define the *constant prefix* as follows:

$$cp(instr^*, \mathbf{cmd}_{\mathbf{pc}}) = cp'(instr^*, i-1) \text{ where } instr^*[i] = \mathbf{cmd}_{\mathbf{pc}}$$

$$cp'(instr^*, i) = \begin{cases} cp'(instr^*, i-1) :: instr^*[i] & \text{if } instr^*[i] = \mathbf{t.const}_{\mathbf{pc}'}\ c \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 7 (Start of Constant Prefix)**.** Given a sequence of annotated instructions $instr^*$ that does not contain any $\mathbf{frame}_n\{F\} \ldots \mathbf{end}$ instructions and an annotated instruction $\mathbf{cmd}_{\mathbf{pc}}$ that appears at index i within $instr^*$, we define the *start of the constant prefix* as follows:

$$cp_{pc}(instr^*, \mathbf{cmd}_{\mathbf{pc}}) = \begin{cases} \mathbf{pc}' & \text{if } cp(instr^*, \mathbf{cmd}_{\mathbf{pc}}) = \mathbf{t.const}_{\mathbf{pc}'}\ c :: x \\ \mathbf{pc} & \text{if } cp(instr^*, \mathbf{cmd}_{\mathbf{pc}}) = \epsilon \end{cases}$$

Proof of Theorem 1. Theorem 1 states that for any step between two annotated configurations c_1 and c_2 one of two things may happen: either c_1 is not fusing and after removing

31 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{br}}(unwind_{\mathbf{n}}(st), gt, lt, mem, at_0, gt_0, mem_0)$	**br**
32 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x \neq 0 \implies$ $MState_{\mathbf{fid},\mathbf{br}}(unwind_{\mathbf{n}}(st), gt, lt, mem, at_0, gt_0, mem_0)$ **33** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x = 0 \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**br_if**
34 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x = \mathbf{idx} \implies$ $MState_{\mathbf{fid},\mathbf{br}}(unwind_{\mathbf{n}}(st), gt, lt, mem, at_0, gt_0, mem_0)$	**br_table**
35 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x \geq \mathbf{sz} - 1 \implies$ $MState_{\mathbf{fid},\mathbf{br}}(unwind_{\mathbf{n}}(st), gt, lt, mem, at_0, gt_0, mem_0)$	**br_table**
36 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x \neq 0 \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$ **37** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge x = 0 \implies MState_{\mathbf{fid},\mathbf{else}}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**if**

Fig. 13: $(\!|\textbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a control instruction **cmd** taken from the right column (part 2).

38 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies$ $MState_{\mathbf{cid},0}([\,], gt, at +\!\!+ [0; \mathsf{SI.ls}(\mathbf{cid}) - \mathsf{SI.as}(\mathbf{cid})], mem, at, gt, mem)$ **39** $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \wedge$ $Return_{\mathbf{cid}}(rt, rgt, rmem, at, gt, mem) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(rt +\!\!+ st[\mathsf{SI.as}(\mathbf{cid}) :], rgt, lt, rmem, at_0, gt_0, mem_0)$ **40** $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \wedge Trap_{\mathbf{cid}}(at, gt, mem) \implies$ $Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	**call** x
41 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge Table(x, \mathbf{cid}, ts) \wedge reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies$ $MState_{\mathbf{cid},0}([\,], gt, at +\!\!+ [0; \mathsf{SI.ls}(\mathbf{cid}) - \mathsf{SI.as}(\mathbf{cid})], mem, at, gt, mem)$ **42** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge Table(x, \mathbf{cid}, ts) \wedge Return_{\mathbf{cid}}(rt, rgt, rmem, at, gt, mem) \wedge$ $reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies MState_{\mathbf{fid},\mathbf{pc}+1}(rt +\!\!+ st[\mathsf{SI.as}(\mathbf{cid}) :], rgt, lt, rmem, at_0, gt_0, mem_0)$ **43** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge Table(x, \mathbf{cid}, ts) \wedge Trap_{\mathbf{cid}}(at, gt, mem) \wedge$ $reverse(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	**call_indirect**
44 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge Table(x, mte, ts) \wedge (mte = \epsilon \vee mte = te) \wedge$ $\bigwedge_{(\mathbf{idx},\mathbf{cid}) \in \mathsf{possibleCallTargets}(\mathbf{fid},\mathbf{pc})} te \neq \mathbf{cid} \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$ **45** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge Table(ti, mte, ts) \wedge x \geq ts \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	**call_indirect**
46 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, Mem(i, v, size), at_0, gt_0, mem_0) \wedge 0 \leq v' \leq 255 \wedge size \leq size' \leq \mathbf{max} \wedge$ $FunctionsAdded() \implies MState_{\mathbf{fid},\mathbf{pc}+1}(rt +\!\!+ st[\mathbf{as} :], rgt, lt, Mem(i, v', size'), at_0, gt_0, mem_0)$ **47** $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \wedge FunctionsAdded() \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	**call_indirect**
48 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies Return_{\mathbf{fid}}(st[: \mathsf{SI.rs}(\mathbf{fid})], gt, mem, at_0, gt_0, mem_0)$	function exit

Fig. 14: $(\!|\textbf{cmd}|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ for a control instruction **cmd** taken from the right column (part 3). **48** is instantiated for $\mathbf{pc}$ = program counter of the **end** of the activation (i.e. the last pc in a function).

$$\beta(S; F; \mathit{instr}^*) = S; \beta_F(F); \beta_{i^*}(\mathit{instr}^*)$$

$$\beta_F\left(\begin{Bmatrix}\mathsf{module}\ f, \mathsf{locals}\ l, \mathsf{args}\ a,\\ \mathsf{stor}\ S, \mathsf{pc}\ pc, \mathsf{fid}\ \mathit{fid}, \mathsf{index}\ i\end{Bmatrix}\right) = \{\mathsf{module}\ f, \mathsf{locals}\ l\}$$

$$\beta_{i^*}(\mathit{instr}^*) = \begin{cases} \mathbf{frame}_n\{\beta_F(F)\}\ \beta_{i^*}\left(\mathit{instr}^{*\prime}\right)\ \mathbf{end} & \text{if } \mathit{instr}^* = \mathbf{frame}_n\{F\}\ \mathit{instr}^{*\prime}\ \mathbf{end}_{\mathbf{pc}} \\ \mathbf{label}_n\{\beta_{i^*}\left(\mathit{instr}^{*\prime\prime}\right)\}\ \beta_{i^*}\left(\mathit{instr}^{*\prime}\right)\ \mathbf{end} & \text{if } \mathit{instr}^* = \mathbf{label}_n\{\mathit{instr}^{*\prime\prime}\}\ \mathit{instr}^{*\prime}\ \mathbf{end}_{\mathbf{pc}} \\ \mathbf{cmd} +\!\!+ \beta_{i^*}\left(\mathit{instr}^{*\prime}\right) & \text{if } \mathit{instr}^* = \mathbf{cmd}_{\mathbf{pc}} +\!\!+ \mathit{instr}^{*\prime} \\ \epsilon & \text{if } \mathit{instr}^* = \epsilon \end{cases}$$

Fig. 15: Definition of annotation removal.

the annotations form c_1 we can take a step in the original semantics to end up in a state that is equal to c_2 after removing annotations; or c_1 is fusing and after removing the annotations form c_1 we can take two steps in the original semantics to end up in a state that is equal to c_2 after removing annotations. Formally:

$$\forall c_1, c_2 \Big(c_1 \twoheadhookleftarrow c_2 \iff \big(\neg f(c_1) \land \beta(c_1) \hookrightarrow \beta(c_2)\big) \lor \big(f(c_1) \land \exists c' (\beta(c_1) \hookrightarrow c' \hookrightarrow \beta(c_2))\big)\Big)$$

where $f(\cdot)$ is defined in Definition 1 and $\beta(\cdot)$.

In the first case, the proof is trivial, as no transition rule for non-fusing configuration uses the annotation to influence the non-annotated parts of the configuration. In the second case, the original semantics can rewrite instructions on the stack to other instructions. In the lemmas for **br_if** (Lemma 34), **br_table** (Lemma 36), **local.tee** (Lemma 21), **call** (Lemma 37), and **call_indirect** (Lemma 38) the transition rules in $\twoheadhookleftarrow$ are given. By comparing these rules with the transition rules in $\hookrightarrow$ (see [29]), we observe that they have exactly the effect of applying $\hookrightarrow$ twice. □

Lemma 1. *Taking a step in $\hookrightarrow$ in a fusing configuration leads to a non-fusing configuration.*

$$\forall c_1, c_2 \, (c_1 \hookrightarrow c_2 \land f(c_1) \implies \neg f(c_2))$$

Proof. The currently executing instruction after taking a step in a fusing configuration is either **br**, **local.get**, or **invoke**, none of which are considered fusing. □

Lemma 2 (Iterated Equivalence)**.** *For every arbitrary-length execution in $\hookrightarrow$, whose second-to-last configuration is non-fusing, there exists an equivalent execution in $\twoheadhookleftarrow$.*

$$\forall c_1, c_2, c' \Big(\beta(c_1) \overset{*}{\hookrightarrow} c' \hookrightarrow \beta(c_2) \land \neg f\left(c'\right) \implies c_1 \overset{*}{\twoheadhookleftarrow} c_2\Big)$$

Proof. By induction on n in $\beta(c_1) \overset{n}{\hookrightarrow} c'$.

$n = 0$. By Theorem 1.

$n = 1$. If $f(c_1)$ by Theorem 1. If $f(c')$ trivially. If $\neg f(c')$ by Theorem 1.

$n \to n + 1$. We can view an execution like this (i.e., the execution from the hypothesis with one additional step): $\beta(c_1) \overset{n}{\hookrightarrow} \beta(c') \hookrightarrow \beta(c_2) \hookrightarrow \beta(c'')$. If $f(\beta(c_2))$ the lemma holds trivially. If $\neg f(\beta(c_2))$ we do a case distinction on $f(\beta(c'))$. If $f(\beta(c'))$, we know either $n = 0$, or, by Lemma 1, that the preceding configuration of c' was not fusing. In any case, this allows us to apply the induction hypothesis, i.e., we have $c_1 \overset{*}{\twoheadhookleftarrow} c'$. Since c' is fusing we can apply Theorem 1, to show that $c_1 \overset{*}{\twoheadhookleftarrow} c''$. If $\neg f(\beta(c'))$, we can apply the induction hypothesis directly which, together with Theorem 1 concludes the proof. □

Proof of Corollary 1. Corollary 1 states

$$\forall c_1, c_2, \Delta_1 \Big(\beta(c_1) \overset{*}{\hookrightarrow} c' \hookrightarrow \beta(c_2) \land \neg f\left(c'\right) \land \Delta_1 \geq \alpha(c_1) \implies \exists \Delta_2 \, (\Delta_1 \vdash \Delta_2 \land \Delta_2 \geq \alpha(c_2))\Big)$$

This is proven via application of Lemma 1 followed by Theorem 2. □

Lemma 3 (Maintenance of program counter annotations)**.** *Every instruction that is executed, has a program counter annotation (where exiting blocks and returning at the end of functions are identified with the program counter of the corresponding* **end***).*

Proof. The initial instruction sequence is annotated. Therefore, we only have to check every instruction that is the result of a applying a transition rule maintains these annotations. We proceed by case distinction. There is a finite number of transition rules in $\hookrightarrow$ that put executable (i.e. non-**const**, non-**label**, non-**frame**) instructions in the instruction sequence. In $\twoheadhookleftarrow$ these are handled as follows:

- **call** k, **call_indirect** (**invoke** does not appear in the definition of $\twoheadhookleftarrow$): these functions call $annotate(\cdot)$ on the instruction sequence.
- **if**, **block**, **loop**: the **end** instruction keeps the program counter annotation (and it is the one, which identifies the rule)
- **br** l: this puts instr^* of $\mathbf{label}_n\{\mathit{instr}^*\}$ on the stack, but instr^* is annotated
- **br_if** l, **br_table** l^*: in the $\hookrightarrow$, these rewrite to **br** l, in $\twoheadhookleftarrow$ they handle the semantics of **br** l also (where the same argument as above apply).
- **local.tee** x: in $\hookrightarrow$ this rewrites to **local.set** x. We introduce a rule that takes to steps at once, thereby eliminating this intermediate step.

□

Corollary 2. *All instructions to the right of the currently executing instruction have a program counter annotation.*

Lemma 4 (Changes to frame objects)**.** *After a step $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, the only frame object, that might be changed is the one of the currently executing activation.*

Proof. SR3 and SR4 may change not any frame objects. SR2 may change the frame object within an activation, if the frame object has been changed in a sub-derivation but may not change the frame object of its configuration. SR2 can change the frame object within an activation only, if the first rule applied in the sub-derivation is SR1. SR1 can change the frame object of its configuration, if the sub-derivation can change the frame object of its configuration. Since we already observed that SR2 can never change the frame object of its configuration, the rule that is applied in SR1 has to be a non-structural-rule if it changes the frame object. Non-structural rules can only be executed in the current activation and (by exhaustive inspection) can be shown not to modify the frame objects of any activation. □

The remainder of the proof will proceed as follows: First, we prove that if we have a correct abstraction for all non-structural rules, our abstraction is sound. Then we prove that we have a correct abstraction for all non-structural rules.

Lemma 5 (Immutable frame fields)**.** *The fields* pc*,* stor*,* args*, and* index *of frame objects are not modified by any rule.*

Proof. By exhaustive inspection. □

Lemma 6 (Growth of call stack)**.** *Given two configurations c_1 and c_2 with $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, where n_1 is the number of activations in c_1 and n_2 is the number of activations in c_2: we have $|n_1 - n_2| \leq 1$.*

Proof. By exhaustive inspection we see that each rule either puts a single activation on the frame (**call** x, **call_indirect**), removes a single activation from the stack **return**, **frame**$_n\{F\}$ *instr** **end**, SR4), or does not change the number of activations (all other). □

Lemma 7 (Abstraction of call stacks)**.** *Given two configurations c_1 and c_2 with $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, where n_1 is the number of activations in c_1 and n_2 is the number of activations in c_2: when abstracting c_1 and c_2 with $\alpha_T(\cdot)$ the first $min(n_1, n_2) - 1$ calls to $\alpha_F(\cdot)$ have the same arguments.*

Proof. By Lemma 6 we have either $n_1 = n_2$, $n_1 + 1 = n_2$ or $n_1 = n_2 + 1$. We proceed by case distinction.

$n_1 = n_2$*:* The executing instruction in c_1 may neither remove the currently executing activation from the instruction sequence nor put a new activation in the instruction sequence, therefore the number of activations in the instruction sequence is the same. It may, however, change frame objects (but, by Lemma 4, only that of the currently executing activation), and the store. This means that no activation but the currently executing one may change. We consider the all cases of $\alpha_T(\cdot)$. In the first case we see that instruction sequences that are not in the currently executing activation only make use of a) frame object that are not the frame object of the currently executing activation b) data that (by Lemma 5), that cannot change between c_1 and c_2. In the second case, we allow changes, as it is the last invocation of $\alpha_F(\cdot)$ In the third cases there is no call to $\alpha_F(\cdot)$, where there is one when calling $\alpha_T(c_1)$, which we also allow.

$n_1 + 1 = n_2$*:* In this case, the currently executing instruction in c_1 is **call** c or **call_indirect**. Furthermore, $min(n_1, n_2) - 1 = n_1 - 1$. This means, that all calls to $\alpha_F(\cdot)$ but the last one in $\alpha_T(c_1)$ and the last two in $\alpha_F(c_2)$ have to have the same arguments. By the rules for **call** c or **call_indirect**, the instruction sequence in c_1 changes in that *val****cmd** is rewritten to **frame**$_n\{F\}$ *instr** **end**. This means that in $\alpha_T(c_2)$ only the last two applications of $\alpha_F(\cdot)$ can change arguments, which we allow.

$n_1 = n_2 + 1$*:* In this case, the currently executing instruction in c_1 is **return** or **frame**$_n\{F\}$ *val** **end**. Furthermore, $min(n_1, n_2) - 1 = n_2 - 1$. This means, that all calls to $\alpha_F(\cdot)$ but the last one in $\alpha_T(c_2)$ and the last two in $\alpha_F(c_1)$ have to have the same arguments. By the rules for **return** or **frame**$_n\{F\}$ *val** **end**, the instruction sequence in c_2 changes in that **frame**$_n\{F\}$ *instr** **end** is rewritten to *val**. This means that in $\alpha_T(c_2)$ has one application of $\alpha_F(\cdot)$ less than $\alpha_T(c_1)$ and that the last application of $\alpha_F(\cdot)$ may have different arguments (which we allow). □

Lemma 8 (Abstraction of structural rules (trapping configuration))**.** *For any configuration c_1 and trapping configuration c_2, with $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$ we have that if $\Delta_1 \geq \alpha(c_1)$, we have that $\exists \Delta_2\, (\Delta_1 \vdash \Delta_2 \wedge \Delta_2 \geq \alpha(c_2))$.*

Proof. We proceed by case distinction. The three kinds of rules that could be applied to step from c_1 to c_2 are

- any trapping non-structural rule
- the structural rule SR3
- the structural rule SR4

non-structural rule: In this case (by Lemma 7) $\alpha(c_1)$ contains the same facts as $\alpha(c_2)$, with the exception of the last recursive call to $\alpha_T(\cdot)$ which produces a set of *Trap*$_{\mathtt{fid}}$-facts. That these facts can be generated is proven in the respective lemmas for the different trapping instructions (see Lemma 12, Lemma 13, Lemma 16, Lemma 24, Lemma 25, Lemma 29, and Lemma 38).

SR3: SR3 rewrites $E[$**trap**$]$ to **trap** within the currently executing configuration. Since both of these cases are handled by the third case in $\alpha_T(\cdot)$ and $\alpha_{trap}(\cdot)$ does not take the instruction sequence into account, we have $\alpha(c_1) = \alpha(c_2)$, which with $\Delta_1 \geq \alpha(c_1)$ implies that a Δ_2 with $\Delta_2 \geq \alpha(c_2)$ is trivially derivable by setting $\Delta_1 = \Delta_2$.

SR4: SR4 handles propagating traps down the call stack. Instruction sequence of c_1 differs from c_2 in that the currently executing activation **frame**$_n\{F\}$ **trap end** is been replaced with **trap**. This last trap appears within the currently execution activation of c_2 like this **frame**$_m\{F'\}$ **trap end**. By Lemma 7, c_1 and c_2 agree on the call stack below the currently executing activation. This means that c_2 differs from c_2 only by containing facts generated by $\alpha_{trap}(F'.\mathsf{fid}, F'.\mathsf{stor}, F')$ where c_1 contains those generated by $\alpha_{trap}(F.\mathsf{fid}, F.\mathsf{stor}, F)$. Facts of this form are propagated by **40** and **43**. □

Lemma 9 (Abstraction of structural rules)**.** *For any non-trapping configurations c_1 and c_2 with $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$ and $\Delta_1 \geq \alpha(c_1)$, where the last application of SR1 was of the form $S;F;E[\mathit{instr}^*] \hookrightarrow\!\!\!\!\!\rightarrow S';F';E[\mathit{instr}^{*\prime}]$: we have $\exists \Delta_2\,(\Delta_1 \vdash \Delta_1 \wedge \Delta_2 \geq \alpha(c_2))$ if we have $\alpha_T(S;F;E[\mathit{instr}^*]), \alpha_C(S) \vdash \alpha_T\big(S';F';E[\mathit{instr}^{*\prime}]\big)$*

Proof. Since c_2 is non-trapping the result of the last applied non-structural rule cannot have been **trap**. Let $\mathbf{cmd}_{\mathbf{pc}_1}$ be the currently executing command of c_1. There are three classes of instructions that $\mathbf{cmd}_{\mathbf{pc}_1}$ might fall into:

- local instructions
- **call** x or **call_indirect**
- **return** or $\mathbf{frame}_n\{F\}\ \mathit{val}^*\ \mathbf{end}$

We proceed by case distinction on these classes.

local instructions: In this case, by Lemma 7, $\alpha(c_1)$ and $\alpha(c_2)$ agree on everything but the last invocation of $\alpha_T\big(S';F';E[\mathit{instr}^{*\prime}]\big)$. This we can derive from $\alpha_T(S;F;E[\mathit{instr}^*])$ (which is a subset of Δ_1 by the assumption.

call *x or* **call_indirect***:* By Lemmas 7, 37 and 38.

return *or* $\mathbf{frame}_n\{F\}\ \mathit{val}^*\ \mathbf{end}$*:* By Lemmas 7 and 40. □

Lemma 10 (Soundness of single-step reachability)**.** *For all concrete configurations c_1 and c_2 and abstract configuration Δ_1 with $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$ and $\Delta_1 \geq \alpha(c_1)$ there exists an abstract configuration Δ_2 that can be derived from Δ_1.*

$$\forall c_1, c_2, \Delta_1 \big(c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \wedge \Delta_1 \geq \alpha(c_1) \implies \exists \Delta_2\,(\Delta_1 \vdash \Delta_2 \wedge \Delta_2 \geq \alpha(c_2))\big)$$

Proof. If c_1 is trapping, by Lemma 8. If c_2 is non-trapping by Lemma 9 and Lemmas 12 to 41. □

Corollary 3. *For all concrete configurations c_1 and c_2 and abstract configuration Δ_1 with $c_1 \overset{*}{\hookrightarrow\!\!\!\!\!\rightarrow} c_2$ and $\Delta_1 \geq \alpha(c_1)$ there exists an abstract configuration Δ_2 that can be derived from Δ_1.*

$$\forall c_1, c_2, \Delta_1 \big(c_1 \overset{*}{\hookrightarrow\!\!\!\!\!\rightarrow} c_2 \wedge \Delta_1 \geq \alpha(c_1) \implies \exists \Delta_2\,(\Delta_1 \vdash \Delta_2 \wedge \Delta_2 \geq \alpha(c_2))\big)$$

Proof. By applying Lemma 10 inductively. □

Lemma 11 (Abstraction of local commands)**.** *Given two configurations $c_1 = S;F;\mathit{instr}^*{}_1 = S;F;E[\mathit{instr}^*]$ and $c_2 = S';F';\mathit{instr}^*{}_2 = S';F';E[\mathit{instr}^{*\prime}]$ where the last applied non-structural rule was $\mathit{instr}^* \hookrightarrow\!\!\!\!\!\rightarrow \mathit{instr}^{*\prime}$, the currently executing instruction of c_1 is $\mathbf{cmd}_{\mathbf{pc_1}}$, the currently executing instruction of c_2 is $\mathbf{cmd}_{\mathbf{pc_2}}$, $\mathbf{cmd}_{\mathbf{pc_1}}$ is a local instruction, $\mathbf{fid} = F.\mathsf{fid}$, and $\mathbf{pc}' = cp_{pc}(\mathit{instr}^*{}_2, \mathbf{cmd}_{\mathbf{pc_2}})$.*

Then we have

$$\begin{aligned}\forall \Delta_1 \Big(\Delta_1 \geq \alpha(c_1) \wedge \exists \Delta' \big(\Delta_1 \vdash \Delta' \wedge \\ \forall x, c, gt, lt, mem, at_0, gt_0, mem_0 (c \in \alpha_{seq}(cp(\mathit{instr}^*{}_2, \mathbf{cmd}_{\mathbf{pc_2}})) \wedge \\ \mathit{MState}_{\mathbf{fid},\mathbf{pc_2}}(c +\!\!+ x, gt, lt, mem, at_0, gt_0, mem_0) \in \alpha(c_2) \\ \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}'}(x, gt, lt, mem, at_0, gt_0, mem_0) \in \Delta')\big) \\ \implies \exists \Delta_2 (\Delta_2 \geq \alpha(c_2) \wedge \Delta_1 \vdash \Delta_2)\Big)\end{aligned}$$

Proof. No local instruction can modify $S.\mathsf{funcs}$, so $\alpha_C(S) = \alpha_C(S')$. Similarly, no local instruction modifies the $S.\mathsf{tables}$ or $F.\mathsf{moduleinst.tableaddrs}$ fields, so $\alpha_t(S,F) = \alpha_t(S',F')$. Furthermore, since $\mathit{instr}_1 = E[\mathit{instr}^*]$, $\mathit{instr}_2 = E[\mathit{instr}^*]$, and $\mathbf{cmd}_{\mathbf{pc_1}}$ is a local instruction, we know that neither $\mathit{instr}^*{}_1$ nor $\mathit{instr}^*{}_2$ contain a frame. Additionally, we know that neither c_1 nor c_2 are trapping. This means that $\alpha_T(c_1) = \alpha_F(F.\mathsf{fid}, \mathbf{pc_1}, S, F, \epsilon, \mathit{instr}_1)$ and $\alpha_T(c_2) = \alpha_F(F'.\mathsf{fid}, \mathbf{pc_2}, S', F', \epsilon, \mathit{instr}_2)$.

The constant prefix of $\mathbf{cmd}_{\mathbf{pc_2}}$ starts at $\mathbf{pc}'$ and ends at $\mathbf{pc_2}$. Due to rule ❹, we can derive $\mathit{MState}_{\mathbf{fid},\mathbf{pc_2}}(\ \dots\ )$-facts from an abstract configuration containing $\mathit{MState}_{\mathbf{fid},\mathbf{pc}'}(\ \dots\ )$-facts. The only argument that changes in these derivations is the first one encoding the stack. The value $\mathbf{v}$ in ❹ takes exactly the values of $\alpha_V(v)$ for every $\mathsf{t.const}\ v$ in $cp(\mathit{instr}^*{}_2, \mathbf{cmd}_{\mathbf{pc_2}})$, which means that from Δ' can derive all facts that have the values for c that are required to be in $\alpha(c_2)$. Due to

$$\begin{aligned}\forall x, c, gt, lt, mem, at_0, gt_0, mem_0 (c \in \alpha_{seq}(cp(\mathit{instr}^*{}_2, \mathbf{cmd}_{\mathbf{pc_2}})) \wedge \\ \mathit{MState}_{\mathbf{fid},\mathbf{pc_2}}(c +\!\!+ x, gt, lt, mem, at_0, gt_0, mem_0) \in \alpha(c_2) \\ \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}'}(x, gt, lt, mem, at_0, gt_0, mem_0) \in \Delta')\big)\end{aligned}$$

we know that we have the stack values x as well as all value for all other arguments in Δ' that are required to be in $\alpha(c_2)$.

This means that if we can derive Δ' from Δ_1, we can derive Δ_2 from Δ_1. □

Lemma 12 (Single-Step Soundness for $\mathsf{t.unop}$)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(\mathsf{t.const}\ n)\,\mathsf{t.unop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \big(c_1 = (S;F;E[(\mathsf{t.const}\ n)\,\mathsf{t.unop}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\mathsf{t.unop}_{\mathbf{pc}}$ are

$$\frac{c \in \mathit{unop}_t(c_1)}{(\mathsf{t.const}\ c_1)\ \mathsf{t.unop}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow (\mathsf{t.const}\ c)}$$

$$\frac{\mathit{unop}_t(c_1) = \emptyset}{(\mathsf{t.const}\ c_1)\ \mathsf{t.unop}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{trap}}$$

This means that (in the non-trapping case) that c_1 and c_2 look as follows

$$c_1 = (S;F;E[(\mathsf{t.const}\ c_1)\ \mathsf{t.unop}_{\mathbf{pc}_1}])$$
$$c_2 = (S;F;E[(\mathsf{t.const}\ c)\ \mathit{val}^*\ \mathbf{cmd}_{\mathbf{pc}_2}])$$

where val^* is the constant prefix of $\mathbf{cmd}_{\mathbf{pc}_2}$.[13]

Let $\mathbf{fid}$ be $F.\mathsf{fid}$. Then we have, by definition, $(\!|\mathsf{t.unop}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by ❺. We first observe that even though the formulation allows for trapping unary operations, WebAssembly 1.0 does not contain such operations. For every possible $\mathit{unop}_t(\cdot)$, we introduce a function $\mathit{unOp}_{\mathbf{op}}(\ \cdot\ )$ that fulfills the following property:

$$\forall n_1\ n.(n \in \mathit{unop}_t(n_1) \implies n \in \mathit{unOp}_{\mathbf{op}}(n_1))$$

[13]Here, and in the following lemmas, we will by slight abuse of notation subsume the case that $\mathbf{cmd}_{\mathbf{pc}_2}$ is $\mathbf{end}_{\mathbf{pc}_2}$ in the notation $E[(\mathsf{t.const}\ c)\ \mathit{val}^*\ \mathbf{cmd}_{\mathbf{pc}_2}]$, even if the correct factorization would look like $E[\mathbf{label}_n\{\mathit{instr}^*\}\mathit{val}^{*\prime}(\mathsf{t.const}\ c)\ \mathit{val}^*\ \mathbf{end}_{\mathbf{pc}_2}]$.

Rule ❺ imples a new *MState*-fact with n_1 removed from the stack and n pushed to the stack. All other components of the fact remain unchanged, as S and F do not change. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd}_{\mathbf{pc}_2}$'s constant prefix. We conclude the proof by applying Lemma 11.

□

Lemma 13 (Single-Step Soundness for t.**binop**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.binop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\,\big(c_1 = (S; F; E[(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.binop}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\mathsf{t.binop}_{\mathbf{pc}}$ are

$$\frac{n \in binop_t(n_1, n_2)}{(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.binop}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow (\mathsf{t.const}\ n)}$$

$$\frac{binop_t(n_1, n_2) = \emptyset}{(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.binop}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \mathsf{trap}}$$

This means that (in the non-trapping case) that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.binop}_{\mathbf{pc}_1}])$$
$$c_2 = (S; F; E[(\mathsf{t.const}\ n)\ val^*\ \mathbf{cmd}_{\mathbf{pc}_2}])$$

where val^* is the constant prefix of $\mathbf{cmd}_{\mathbf{pc}_2}$.Let $\mathbf{fid}$ be F.fid. Then we have, by definition, $(\!|\mathsf{t.binop}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by ❻ (respectively ❼ and ❽). These rules assume the existence of a family of functions $\cdot \circledast_{\mathbf{op}} \cdot : \mathbb{B}_{64} \times \mathbb{B}_{64} \mapsto \mathcal{P}(\mathbb{B}^?_{64})$. For every possible $binop_t(\cdot)$ from the concrete semantics the corresponding $\circledast_{\mathbf{op}}$ is assumed to fulfill

$$\forall n_1\ n_2\ n.(n \in binop_t(n_1, n_2) \implies n \in n_1 \circledast_{\mathbf{op}} n_2) \wedge (\emptyset = binop_t(n_1, n_2) \implies \epsilon \in n_1 \circledast_{\mathbf{op}} n_2)$$

Rules ❻ and ❼ imply a new *MState*-fact with n_1 and n_2 removed from the stack and n pushed to the stack (if n is a value) (while ❽ can be used to derive a fitting *Trap*-fact). All other components of the fact remain unchanged, as S and F do not change. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd}_{\mathbf{pc}_2}$'s constant prefix. We conclude the proof by applying Lemma 11. □

Lemma 14 (Single-Step Soundness for t.**testop**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t.const}\ n_1)\,\mathsf{t.testop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\,\big(c_1 = (S; F; E[(\mathsf{t.const}\ n_1)\,\mathsf{t.testop}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. This instruction is handled by ❺ and thus proven similarly to Lemma 12. □

Lemma 15 (Single-Step Soundness for t.**relop**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.relop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\,\big(c_1 = (S; F; E[(\mathsf{t.const}\ n_1)\,(\mathsf{t.const}\ n_2)\,\mathsf{t.relop}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. This instruction is handled by ❻ and thus proven similarly to Lemma 13. □

Lemma 16 (Single-Step Soundness for $\mathsf{t_2}$.**cvtOp_t1_sx**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t_2.const}\ n_1)\,\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\,\big(c_1 = (S; F; E[(\mathsf{t_2.const}\ n_1)\,\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}}$ are

$$\frac{n \in cvtop^{sx}_{t_1,t_2}(n_1)}{(\mathsf{t.const}\ n_1)\,\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow (\mathsf{t.const}\ n)}$$

$$\frac{cvtop^{sx}_{t_1,t_2}(n_1) = \emptyset}{(\mathsf{t.const}\ n_1)\,\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \mathsf{trap}}$$

This means that (in the non-trapping case) that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{t.const}\ n_1)\,\mathsf{t_2.cvtOp\_t_1\_sx}_{\mathbf{pc}_1}])$$
$$c_2 = (S; F; E[(\mathsf{t.const}\ n)\ val^*\ \mathbf{cmd}_{\mathbf{pc}_2}])$$

where val^* is the constant prefix of $\mathbf{cmd}_{\mathbf{pc}_2}$.Let $\mathbf{fid}$ be F.fid.

Then we have, by definition, $(\!|\mathsf{t_2.cvtOp\_t_1\_sx}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by ❾ (respectively ❿ and ⓫). These rules assume the existence of a family of functions $cvtOp_{\mathbf{op}}(\cdot) : \mathbb{B}_{64} \mapsto \mathcal{P}(\mathbb{B}_{64})$. For every possible $cvtop^{sx}_{t_1,t_2}(\cdot)$ from the concrete semantics the corresponding $cvtOp_{\mathbf{op}}(\cdot)$ is assumed to fulfill

$$\forall n_1\ n.(n \in cvtop^{sx}_{t_1,t_2}(n_1) \implies n \in cvtOp_{\mathbf{op}}(n_1))$$

Rules ❾ and ❿ imply a new *MState*-fact with n_1 and n_2 removed from the stack and n pushed to the stack (if n is a value). All other components of the fact remain unchanged, as S and F do not change. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd}_{\mathbf{pc}_2}$'s constant prefix. If the concrete operation is one that can have an undefined result (return $\emptyset$), we additionally allow to unconditionally derive a *Trap*-fact via ⓫.[14] We conclude the proof by applying Lemma 11.

□

Lemma 17 (Single-Step Soundness for **drop**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[val\ \mathbf{drop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

[14] Since all possibly trapping conversions are from floating point types and WAPPLER does not handle these precisely, an unconditional trap in these cases does not decrease precision.

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathit{val}\ \mathbf{drop_{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{drop_{pc}}$ is

$$\frac{}{\mathit{val}\ \mathbf{drop_{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \epsilon}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathit{val}\ \mathbf{drop_{pc_1}}])$$
$$c_2 = (S; F; E[\mathit{val}^*\ \mathbf{cmd_{pc_2}}])$$

where val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$. Let $\mathbf{fid}$ be $F.\mathsf{fid}$.

Then we have, by definition, $(\!|\mathbf{drop}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by **17**. This rule implies a new fact with one element remove from the top of the stack. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd_{pc_2}}$'s constant prefix. We conclude the proof by applying Lemma 11. □

Lemma 18 (Single-Step Soundness for **select**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ n)\ \mathbf{select_{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ n)\ \mathbf{select_{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{select_{pc}}$ is

$$\frac{n \neq 0}{\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ n)\ \mathbf{select_{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \mathit{val}_1}$$

$$\frac{n = 0}{\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ n)\ \mathbf{select_{pc}} \hookrightarrow\!\!\!\!\!\rightarrow \mathit{val}_2}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ n)\ \mathbf{select_{pc_1}}])$$
$$c_2 = (S; F; E[\mathit{val}'\ \mathit{val}^*\ \mathbf{cmd_{pc_2}}])$$

where val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$ and val' is either val_1 or val_2 depending on n. Let $\mathbf{fid}$ be $F.\mathsf{fid}$. Then we have, by definition, $(\!|\mathbf{select}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by **18**, **19**. Both rules remove two element from the stack and push back one of the removed values (as required). The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd_{pc_2}}$'s constant prefix. We conclude the proof by applying Lemma 11. □

Lemma 19 (Single-Step Soundness for **local.get** x)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathbf{local.get_{pc}}\ x)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\mathbf{local.get_{pc}}\ x)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{local.get_{pc}}\ x$ is

$$\frac{F.\mathsf{locals}[x] = \mathit{val}}{F; (\mathbf{local.get_{pc}}\ x) \hookrightarrow\!\!\!\!\!\rightarrow F; \mathit{val}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathbf{local.get_{pc_1}}\ x])$$
$$c_2 = (S; F; E[\mathit{val}'\ \mathit{val}^*\ \mathbf{cmd_{pc_2}}])$$

where val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$ and val' is $F.\mathsf{locals}[x]$. Let $\mathbf{fid}$ be $F.\mathsf{fid}$.

Then we have, by definition, $(\!|\mathbf{local.get}\ x|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by **12**. This rule implies a new fact with $\mathit{lt}[x]$ on the top of the stack. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd_{pc_2}}$'s constant prefix. By definition of $\alpha_F(\cdot)$, there exists a lt in the premises, such that $\mathit{lt}[x] = F.\mathsf{locals}[x]$. We conclude the proof by applying Lemma 11. □

Lemma 20 (Single-Step Soundness for **local.set** x)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\mathit{val}\ (\mathbf{local.set_{pc}}\ x)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathit{val}\ (\mathbf{local.set_{pc}}\ x)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{local.set_{pc}}\ x$ is

$$\frac{F' = F \text{ with } \mathsf{locals}[x] = \mathit{val}}{F; \mathit{val}\ (\mathbf{local.set_{pc}}\ x) \hookrightarrow\!\!\!\!\!\rightarrow F'; \epsilon}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathit{val}\ (\mathbf{local.set_{pc_1}}\ x])$$
$$c_2 = (S; F'; E[\mathit{val}^*\ \mathbf{cmd_{pc_2}}])$$

where val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$ and F' is F with $\mathsf{locals}[x] = \mathit{val}$. Let $\mathbf{fid}$ be $F.\mathsf{fid}$.

Then we have, by definition, $(\!|\mathbf{local.set}\ x|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by **13**. This rule implies a new fact with lt updated to val at x and val removed from the stack. The program counter of this new fact is set to $\mathbf{pc}_1 + 1$, which is the start of $\mathbf{cmd_{pc_2}}$'s constant prefix. By definition, $\alpha_F(c_1)$ contains facts with x at the top of the stack in the premises, such that $x = \mathit{val}$. At the same time, $\alpha_F(c_2)$ contains facts such that the tuple encoding the local variables is updated to val at x. Exactly such a tuple is propagated by **13**. We conclude the proof by applying Lemma 11. □

Lemma 21 (Single-Step Soundness for **local.tee** x)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathbf{local.tee_{pc}}\ x)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\mathbf{local.tee_{pc}}\ x)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{local.tee_{pc}}\ x$ is

$$\frac{F' = F \text{ with } \mathsf{locals}[x] = \mathit{val}}{S; F; \mathit{val}\ (\mathbf{local.tee}\ x) \hookrightarrow\!\!\!\!\!\rightarrow S; F'; \mathit{val}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[val' \ (\textbf{local.tee}_{\textbf{pc}_1} \ x])$$
$$c_2 = (S; F'; E[val' val^* \ \textbf{cmd}_{\textbf{pc}_2}])$$

where val^* is the constant prefix of $\textbf{cmd}_{\textbf{pc}_2}$ and F' is F with $\mathsf{locals}[x] = val$. Let $\textbf{fid}$ be $F.\mathsf{fid}$.

Then we have, by definition, $(\!|\textbf{local.tee}\ x|\!)^{\textbf{fid}}_{\textbf{pc}_1} \subseteq \alpha_C(S)$, which is given by ⓮. The proof proceeds similar to Lemma 19 and Lemma 20. □

Lemma 22 (Single-Step Soundness for **global.get** x)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\textbf{global.get}_{\textbf{pc}}\ x)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\textbf{global.get}_{\textbf{pc}}\ x)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The abstract transition rule is given by ⓯, the proof procedes similar to Lemma 19. □

Lemma 23 (Single-Step Soundness for **global.set** x)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\textbf{global.set}_{\textbf{pc}}\ x)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\textbf{global.set}_{\textbf{pc}}\ x)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The abstract transition rule is given by ⓰, the proof procedes similar to Lemma 20. □

Lemma 24 (Single-Step Soundness for t.**loadN_sx**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{i32}.\textbf{const}\ i)\ (\mathsf{t}.\textbf{loadN\_sx}_{\textbf{pc}}\ memarg)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\mathsf{i32}.\textbf{const}\ i)\ (\mathsf{t}.\textbf{loadN\_sx}_{\textbf{pc}}\ memarg)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rules for t.$\textbf{loadN\_sx}_{\textbf{pc}}$ are

$$\frac{ea = i + memarg.\mathsf{offset} \quad ea + |t|/8 \le |S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}| \quad bytes_t(n) = S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}[ea : |t|/8]}{S; F; (\mathsf{i32}.\textbf{const}\ i)\ (\mathsf{t}.\textbf{load}\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; (\mathsf{t}.\textbf{const}\ n)}$$

$$\frac{ea = i + memarg.\mathsf{offset} \quad ea + N/8 \le |S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}| \quad bytes_t(n) = S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}[ea : N/8]}{S; F; (\mathsf{i32}.\textbf{const}\ i)\ (\mathsf{t}.\textbf{loadN\_sx}\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; (\mathsf{t}.\textbf{const}\ extend\_sx_{N,|t|}(n))}$$

$$\frac{\text{otherwise}}{S; F; (\mathsf{i32}.\textbf{const}\ i)\ (\mathsf{t}.\textbf{loadN\_sx}\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \textbf{trap}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{t}.\textbf{loadN\_sx}_{\textbf{pc}}\ memarg)\ (\textbf{local.tee}_{\textbf{pc}_1}\ x])$$
$$c_2 = (S; F'; E[(\mathsf{t}.\textbf{const}\ n) val^*\ \textbf{cmd}_{\textbf{pc}_2}])$$

where n are the N bytes starting from ea in $S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}$, interpreted as signed or unsigned depending on sx, and val^* is the constant prefix of $\textbf{cmd}_{\textbf{pc}_2}$. We will only proof the second case as (in our formulation) it is subsuming the first. Let $\textbf{fid}$ be $F.\mathsf{fid}$. Then we have, by definition, $(\!|\mathsf{t}.\textbf{loadN\_sx}\ memarg|\!)^{\textbf{fid}}_{\textbf{pc}_1} \subseteq \alpha_C(S)$, which is given by ⓴ and ㉑. ㉑ implies a fitting *Trap*-fact in case the memory access is out-of-bounds. ⓴ has $N + 1$ *MState*-predicates as premises, one for every byte loaded and an additional one with an arbitrary memory cell. The contents of the memory are loaded into an N-tuple *vs*, which are then concatenated and stored in a variable u (in case of t.**load**, N is set to $|t|$). The result of extending u according to sx and N is stored in w, which is then put on the stack. By definition, $\alpha_F(c_1)$ returns an *MState*-fact for every integer from 0 to the maximum memory size of the module. This means there also exists an *MState*-fact in the premise for every memory cell, especially for those from $S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}[ea : N/8]$. After application of ⓴ we have for every fact in $\alpha_F(c_1)$ one where t.**const** i has been replaced by the loaded value. We conclude the proof by applying Lemma 11. □

Lemma 25 (Single-Step Soundness for t.**storeN**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{storeN}_{\textbf{pc}}\ memarg)]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{storeN}_{\textbf{pc}}\ memarg)]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rules for t.$\textbf{storeN}_{\textbf{pc}}$ are

$$\frac{ea = i + memarg.\mathsf{offset} \quad a = F.\mathsf{moddule.memaddrs}[0] \quad ea + |t|/8 \le |S.\mathsf{mems}[a].\mathsf{data}| \quad S' = S \text{ with } S.\mathsf{mems}[a].\mathsf{data}[ea : |t|/8] = bytes_t(n)}{S; F; (\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{store}[\mathbf{pc}]\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \epsilon}$$

$$\frac{ea = i + memarg.\mathsf{offset} \quad a = F.\mathsf{moddule.memaddrs}[0] \quad ea + N/8 \le |S.\mathsf{mems}[a].\mathsf{data}| \quad S' = S \text{ with } S.\mathsf{mems}[a].\mathsf{data}[ea : N/8] = bytes_{iN}\left(wrap_{|t|,N}(n)\right)}{S; F; (\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{storeN}_{\textbf{pc}}\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \epsilon}$$

$$\frac{\text{otherwise}}{S; F; (\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{storeN}_{\textbf{pc}}\ memarg) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \textbf{trap}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{i32}.\textbf{const}\ i)(\mathsf{t}.\textbf{const}\ n)(\mathsf{t}.\textbf{storeN}_{\textbf{pc}}\ memarg)])]$$
$$c_2 = (S'; F; E[val^*\ \textbf{cmd}_{\textbf{pc}_2}])$$

where S' is a copy of S where the $N/8$ bytes from ea in $S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}$ have been updated to the least significant N bytes of n, and val^* is the constant prefix of $\textbf{cmd}_{\textbf{pc}_2}$. We will only proof the second case as (in our formulation) it is subsuming the first. Let $\textbf{fid}$ be

F.fid. Then we have, by definition, $(\!|\mathsf{t}.\textbf{storeN}\ \mathit{memarg}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by 22 and 23. 23 implies a fitting *Trap*-fact in case the memory access is out-of-bounds. 22 first calculates if the effective address *ea* is larger than the current size. It then calculates a value d which describes, the relation between i, the index of the current memory cell. If d is greater than or equal to 0 and less than $N/8$, this means that the contents of the memory cell at i have to be updated. The new value is calculated wrapping the stored value, then shifting the resulting value by d, and then masking it. By definition, $\alpha_F(c_1)$ returns an *MState*-fact for every integer from 0 to the maximum memory size of the module. This means there also exists an *MState*-fact in the premise for every memory cell, especially for those from $S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}[\mathit{ea} : N/8]$. After application of 22 we have for every fact in $\alpha_F(c_1)$ one where there the memory cell value has been updated to the correct value (or left the same, if the respective memory cell is not touched). We conclude the proof by applying Lemma 11. □

Lemma 26 (Single-Step Soundness for **memory.size**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{memory.size}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\textbf{memory.size}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\textbf{memory.size}_{\mathbf{pc}}$ are

$$\frac{|S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}| = \mathit{sz} \cdot 64 \cdot 1024}{S; F; \textbf{memory.size} \hookrightarrow\!\!\!\!\!\rightarrow S; F; (\mathsf{i32.const}\ \mathit{sz})}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\textbf{memory.size}]$$
$$c_2 = (S; F; E[(\mathsf{i32.const}\ \mathit{sz})\ \mathit{val}^*\ \textbf{cmd}_{\mathbf{pc}_2}])$$

where sz is $|S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}|$ divided by $64 \cdot 1024 (= 2^{16})$, and val^* is the constant prefix of $\textbf{cmd}_{\mathbf{pc}_2}$. Let $\mathbf{fid}$ be F.fid. Then we have, by definition, $(\!|\textbf{memory.size}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by 24. By definition, $\alpha_F(c_1)$ contains a *MState*-fact for every memory position and the tuple describing the memory position holds $|S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}|/2^{16}$ in its third position. 24 pushes this value to the top of the stack. We conclude the proof by applying Lemma 11.

□

Lemma 27 (Single-Step Soundness for **memory.grow**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{i32.const}\ n)\ \textbf{memory.grow}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[(\mathsf{i32.const}\ n)\ \textbf{memory.grow}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\textbf{memory.grow}_{\mathbf{pc}}$ are

$$\frac{F.\mathsf{module.memaddrs}[0] = a \qquad \mathit{sz} = |S.\mathsf{mems}[a].\mathsf{data}|/2^{16} \qquad S' = S \text{ with } \mathsf{mems}[a] = \mathit{growmem}(S.\mathsf{mem}[a], n)}{S; F; (\mathsf{i32.const}\ n)\ \textbf{memory.grow} \hookrightarrow\!\!\!\!\!\rightarrow S'; F; (\mathsf{i32.const}\ \mathit{sz})}$$

$$S; F; (\mathsf{i32.const}\ n)\ \textbf{memory.grow} \hookrightarrow\!\!\!\!\!\rightarrow S; F; (\mathsf{i32.const}\ -1)$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{i32.const}\ n)\ \textbf{memory.grow}_{\mathbf{pc}_1}])$$
$$c_2 = (S; F; E[(\mathsf{i32.const}\ \mathit{sz})\ \mathit{val}^*\ \textbf{cmd}_{\mathbf{pc}_2}])$$

where sz is $|S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}|$ divided by $64 \cdot 1024 (= 2^{16})$ (or -1), S' is S where the $S.\mathsf{mem}[a]$ has been replaced with a copy that is extended by $n \cdot 2^{16}$ 0-bytes (or not), and val^* is the constant prefix of $\textbf{cmd}_{\mathbf{pc}_2}$. If a request for growing the memory fails (i.e., the memory is not changed and -1 is put on the stack) is dependent on the embedder, but if a growing the memory is would increase its size beyond the limit given in the module declaration (henceforth $\mathbf{max}$, if there is no limit given $\mathbf{max} = 2^16$), it has to fail. Let $\mathbf{fid}$ be F.fid. Then we have, by definition, $(\!|\textbf{memory.grow}|\!)^{\mathbf{fid}}_{\mathbf{pc}_1} \subseteq \alpha_C(S)$, which is given by 25. By definition, $\alpha_F(c_1)$ contains a *MState*-fact for every possible memory position from 0 to $\mathbf{max}$, where each has $|S.\mathsf{mems}[F.\mathsf{moddule.memaddrs}[0]].\mathsf{data}|$ in its third component. 25 has two possible results: either the allocation fails, the memory cell is not changed and -1 is pushed to the stack, or the allocation succeeds, the old size is pushed and the size in every memory cell is increased. As the newly allocated memory cells in were already available in $\alpha_F(c_1)$ (with a value of 0), we can derive a new abstract configuration that contains the memory cell with the correct memory size. We conclude the proof by applying Lemma 11. □

Lemma 28 (Single-Step Soundness for **nop**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{nop}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\textbf{nop}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. This instruction is handled by 27 and proven by direct application of Lemma 11. □

Lemma 29 (Single-Step Soundness for **unreachable**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{unreachable}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\textbf{unreachable}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. This instruction unconditionally rewrites to **trap** and is handled by 26 (which implies a fitting *Trap*-fact). □

Lemma 30 (Single-Step Soundness for **block**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form*

$S; F; E[\mathbf{block_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}]$ *and* $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$*, then we have* $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$*:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathbf{block_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{block_{pc}}\ [t^n]$ is

$$\mathbf{block_{pc_1}}\ [t^n]\ instr^*\ \mathbf{end_{pc_{end}}} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_n\{\}\ instr^*\ \mathbf{end_{pc_{end}}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathbf{block_{pc_1}}\ [t^n]\ instr^*\ \mathbf{end_{pc_{end}}}]$$
$$c_2 = (S; F; E[\mathbf{label}_n\{\}\ val^*\ \mathbf{cmd_{pc_2}}\ instr^{*\prime}\ \mathbf{end_{pc_{end}}}])$$

where $\mathbf{cmd_{pc_2}}$ and $\mathbf{end_{pc_{end}}}$ might coincide and val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$. This instruction is handled by 28 and, since only the program counter changes, proven by direct application of Lemma 11. □

Lemma 31 (Single-Step Soundness for **loop**)**.** *For any two concrete configurations* c_1 *and* c_2*: If* c_1 *is of the form* $S; F; E[\mathbf{loop_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}]$ *and* $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$*, then we have* $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$*:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathbf{loop_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{loop_{pc}}\ [t^n]$ is

$$\mathbf{loop_{pc_{loop}}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_0\{\mathbf{loop_{pc_{loop}}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}\}\ instr^*\ \mathbf{end_{pc_{end}}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathbf{loop_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}]$$
$$c_2 = (S; F; E[\mathbf{label}_n\{\mathbf{loop_{pc_1}}\ [t^?]\ instr^*\ \mathbf{end_{pc_{end}}}\}\ val^*\ \mathbf{cmd_{pc_2}}\ instr^{*\prime}\ \mathbf{end_{pc_{end}}}])$$

where $\mathbf{cmd_{pc_2}}$ and $\mathbf{end_{pc_{end}}}$ might coincide and val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$. $\alpha_F(c_1)$ returns exactly the same set as if **loop** was **block**, therefore, the proof proceeds same as Lemma 30. □

Lemma 32 (Single-Step Soundness for **if** · **else** · **end**)**.** *For any two concrete configurations* c_1 *and* c_2*: If* c_1 *is of the form* $S; F; E[\mathbf{if_{pc_{if}}}\ [t^?]\ instr^*{}_1\ \mathbf{else_{pc_{else}}}\ \mathbf{end_{pc_{end}}}]$ *and* $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$*, then we have* $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$*:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathbf{if_{pc_{if}}}\ [t^?]\ instr^*{}_1\ \mathbf{else_{pc_{else}}}\ \mathbf{end_{pc_{end}}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rules for $\mathbf{if_{pc_{if}}}\ [t^?]\ instr^*{}_1\ \mathbf{else_{pc_{else}}}\ \mathbf{end_{pc_{end}}}$ are

$$\frac{c \neq 0}{(\mathsf{i32.const}\ c)\ \mathbf{if_{pc_{if}}}\ t^n\ instr_1^*\ \mathbf{else_{pc_{else}}}\ instr_2^*\ \mathbf{end_{pc_{end}}} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_n\{\epsilon\}\ instr_1^*\ \mathbf{end_{pc_{else}-1}}}$$

$$\frac{c = 0}{(\mathsf{i32.const}\ c)\ \mathbf{if_{pc_{if}}}\ t^n\ instr_1^*\ \mathbf{else_{pc_{else}}}\ instr_2^*\ \mathbf{end_{pc_{end}}} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_n\{\epsilon\}\ instr_2^*\ \mathbf{end_{pc_{end}}}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[(\mathsf{i32.const}\ c)\ \mathbf{if_{pc_{if}}}\ t^n\ instr_1^*\ \mathbf{else_{pc_{else}}}\ instr_2^*\ \mathbf{end_{pc_{end}}}]$$
$$c_2 = (S; F; E[\mathbf{label}_n\{\}\ val^*\ \mathbf{cmd_{pc_2}}\ instr^{*\prime}\ \mathbf{end_{pc_{end}}'}])$$

where $\mathbf{cmd_{pc_2}}$ and $\mathbf{end_{pc_{end}}'}$ might coincide, val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$, and $\mathbf{pc_{end}}'$ is either $\mathbf{pc_{else}}$ or $\mathbf{pc_{end}}$. We first observe that $annotate(\cdot)$ introduces "holes" in the numbering of instructions when encountering **if** or **else**, that is: the first instruction after **if** $\mathbf{pc}$ and $\mathbf{else_{pc}}$ is not $\mathbf{pc}+1$ but $\mathbf{pc}+2$. Additionally, it leaves a "hole" in front of **else**. $\alpha_C(c_1)$ uses these program counters to introduce additional abstractions for **block** and **end** (which will be handled by 28 and 30). **if** · **else** · **end** itself is handled by 36 and 37. Which just set the program counter according to the top of stack. The rest of the proof proceeds as Lemma 30. □

Lemma 33 (Single-Step Soundness for **br**)**.** *For any two concrete configurations* c_1 *and* c_2*: If* c_1 *is of the form* $S; F; E[\mathbf{br_{pc}}\ l]$ *and* $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$*, then we have* $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$*:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[\mathbf{br_{pc}}\ l]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rule for $\mathbf{br}\ l$ is

$$\mathbf{label}_n\{instr^*\}\ B^l\ [val^n\ (\mathbf{br}\ l)]\ \mathbf{end} \hookrightarrow\!\!\!\!\!\rightarrow val^n\ instr^*$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathbf{label}_n\{instr^*\}\ B^l\ [val^n\ (\mathbf{br_{pc_1}}\ l)]\ \mathbf{end_{pc_{end}}}])$$
$$c_2 = (S; F; E[val^n\ val^*\ \mathbf{cmd_{pc_2}}])$$

where val^* is the constant prefix of $\mathbf{cmd_{pc_2}}$. We observe two possible scenarios: either $instr^*$ is ϵ or it is $\mathbf{loop_{pc_{loop}}}\ [t^?]\ instr^{*\prime}\ \mathbf{end_{pc_{end}}}$. In the first case, $\mathbf{cmd_{pc_2}}$ is the first instruction after $\mathbf{end_{pc_{end}}}$, in the second case it is $\mathbf{loop_{pc_{loop}}}$. This instruction is handled by 31. 31 assumes the existence of a constant $\mathbf{br}$, which is either $\mathbf{pc_{end}}$ or $\mathbf{pc_{loop}}$ (it can be statically determined to which label **br** l jumps and if the label is the result of **loop**). $unwind(st)$ is a function that does the necessary stack transformations, i.e., it keeps the top n elements and drops the elements that are not taken out of the label. In case $\mathbf{cmd_{pc_2}}$ was **loop**, this concludes the proof, otherwise we observe that 30 sets the program counter to the start of the constant prefix of $\mathbf{cmd_{pc_2}}$ and apply Lemma 11. □

Lemma 34 (Single-Step Soundness for **br_if**)**.** *For any two concrete configurations* c_1 *and* c_2*: If* c_1 *is of the form* $S; F; E[(\mathsf{i32.const}\ n)\ \mathbf{br\_if_{pc}}\ l]$ *and* $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$*, then we have* $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$*:*

$$\forall c_1, c_2 \left(c_1 = (S; F; E[(\mathsf{i32.const}\ n)\ \mathbf{br\_if_{pc}}\ l]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. The transition rules for **br_if** l are

$$\frac{c \neq 0}{\textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n\ (\textsf{i32.const}\ c)(\textsf{br\_if}_{\mathbf{pc}}\ l)]\ \textsf{end}_{\mathbf{pc_{end}}} \hookrightarrow\!\!\!\!\rightarrow \textit{val}^n\ \textit{instr}^*}$$

$$\frac{c = 0}{\textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n\ (\textsf{i32.const}\ c)(\textsf{br\_if}_{\mathbf{pc}}\ l)]\ \textsf{end}_{\mathbf{pc_{end}}} \hookrightarrow\!\!\!\!\rightarrow \textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n]\ \textsf{end}_{\mathbf{pc_{end}}}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n\ (\textsf{br}_{\mathbf{pc_1}}\ l)]\ \textsf{end}_{\mathbf{pc_{end}}}])$$
$$c_2 = (S; F; E[\textit{val}^n\ \textit{val}^*\ \textsf{cmd}_{\mathbf{pc_2}}])$$

where $\textit{val}^*$ is the constant prefix of $\textsf{cmd}_{\mathbf{pc_2}}$. We observe two possible scenarios: either $c = 0$ or $c \neq 0$. In the first case, which is handled by **33**, $\textsf{cmd}_{\mathbf{pc_2}}$ is in the same label as $\textsf{br}_{\mathbf{pc_1}}\ l$ and $\mathbf{pc_1} + 1$ is the start of the constant prefix of $\textsf{cmd}_{\mathbf{pc_2}}$. This allows us to apply Lemma 11. In the second case, which is handled by **32**, we proceed like in Lemma 33. □

Lemma 35 (Single-Step Soundness for **br_table**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\textsf{i32.const}\ i)\ \textsf{br\_table}_{\mathbf{pc}}\ l^*\ l_N]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[(\textsf{i32.const}\ i)\ \textsf{br\_table}_{\mathbf{pc}}\ l^*\ l_N]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for $\textsf{br\_table}\ {l_i}^*\ l_N$ are

$$\frac{i < |{l_i}^*| \qquad {l_i}^*[i] = l}{\textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n\ (\textsf{i32.const}\ i)(\textsf{br\_table}_{\mathbf{pc}}\ {l_i}^*\ l_N)]\ \textsf{end}_{\mathbf{pc_{end}}} \hookrightarrow\!\!\!\!\rightarrow \textit{val}^n\ \textit{instr}^*}$$

$$\frac{i \geq |{l_i}^*| \qquad l_N = l}{\textsf{label}_n\{\textit{instr}^*\}\ B^l\ [\textit{val}^n\ (\textsf{i32.const}\ i)(\textsf{br\_table}_{\mathbf{pc}}\ {l_i}^*\ l_N)]\ \textsf{end}_{\mathbf{pc_{end}}} \hookrightarrow\!\!\!\!\rightarrow \textit{val}^n\ \textit{instr}^*}$$

This instruction is handled by **34** and **35**. **34** is instantiated for every element of ${l_i}^*$, where $\mathbf{idx}$ takes value from 0 to $|{l_i}^*|$ and $\mathbf{br}$ takes the appropriate program counters as with **br**. The proof proceed analogous to Lemma 33. □

Lemma 36 (Single-Step Soundness for **end**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textsf{label}_n\{\textit{instr}^*\}\ \textit{val}^m\ \textsf{end}_{\mathbf{pc_{end}}}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\textsf{label}_n\{\textit{instr}^*\}\ \textit{val}^m\ \textsf{end}_{\mathbf{pc_{end}}}]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rule for **end** is

$$\textsf{label}_n\{\textit{instr}^*\}\ \textit{val}^m\ \textsf{end}_{\mathbf{pc}} \hookrightarrow\!\!\!\!\rightarrow \textit{val}^m$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\textsf{label}_n\{\textit{instr}^*\}\ \textit{val}^m\ \textsf{end}_{\mathbf{pc_1}}])$$
$$c_2 = (S; F; E[\textit{val}^m\ \textit{val}^*\ \textsf{cmd}_{\mathbf{pc_2}}])$$

where $\textit{val}^*$ is the constant prefix of $\textsf{cmd}_{\mathbf{pc_2}}$. **end** is handled by **30** whose only effect is setting the program counter to $\mathbf{next}$. There are two scenarios for which **30** is instantiated. Either the $\textsf{end}_{\mathbf{pc_1}}$ appears in the code or it introduced when $\alpha_C(\cdot)$ encounters an $\textsf{if}_{\mathbf{pc_{if}}}\ \cdot\ \textsf{else}_{\mathbf{pc_{else}}}\ \cdot\ \textsf{end}_{\mathbf{pc_{end}}}$. In the first case, $\mathbf{next}$ is set to $\mathbf{pc_1} + 1$, which is the start of $\textsf{cmd}_{\mathbf{pc_2}}$ constant prefix. In the second case, $\mathbf{next}$ is set to $\mathbf{pc_{end}} + 1$. The instance of **30** that is introduced in the second case is only encountered if the "then"-branch is abstractly executed. After executing this branch, the next executed comes somwhere after $\textsf{end}_{\mathbf{pc_{end}}}$ (as discussed in Lemma 32). Both cases allow us to apply Lemma 11. □

Lemma 37 (Single-Step Soundness for $\textsf{call}\ x$)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textit{val}^n\ \textsf{call}\ x]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\textit{val}^n\ \textsf{call}\ x]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We assume the function called is a Wasm-function from the same module. All other functions are treated as host functions (see Lemma 41).

The transition rule for $\textsf{call}\ x$ is

$$\frac{\begin{array}{c} F.\textsf{module.funcaddrs}[x] = a \\ S.\textsf{funcs}[a] = f \qquad f.\textsf{type} = [t_1^n] \mapsto [t_2^m] \\ m \leq 1 \qquad f.\textsf{code} = \{\textsf{type}\ x, \textsf{locals}\ t^k, \textsf{body}\ \textit{instr}^*\ \textsf{end}\} \\ F' = \{\textsf{module}\ f.\textsf{module}, \textsf{locals}\ \textit{val}^n(\textsf{t.const}\ 0)^k, \\ \textsf{args}\ \textit{val}^n, \textsf{stor}\ S, \textsf{pc}\ pc, \textsf{fid}\ a, \textsf{index}\ \epsilon\} \\ \textit{instr}^{*\prime} = \textit{annotate}(\textsf{block}\ t_2^m\ \textit{instr}^*\ \textsf{end end}) \end{array}}{S; F; \textit{val}^n\ (\textsf{call}_{pc}\ x) \hookrightarrow\!\!\!\!\rightarrow S; F; \textsf{frame}_m\{F'\}\ \textit{instr}^{*\prime}}$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\textit{val}^n\ \textsf{call}_{\mathbf{pc_1}}\ x])$$
$$c_2 = (S; F; E[\textsf{frame}_m\{F'\}\ \textsf{block}_{\mathbf{pc_2}}\ t_2^m\ \textit{instr}^*\ \textsf{end}_{\mathbf{pc_{end}}}\ \textsf{end}_{\mathbf{pc_{end}}+1}])$$

Where $\mathbf{pc_2} = 0$. This instruction is handled by **38**. Since the ID of the called function ($F.\textsf{module.funcaddrs}[x]$) is statically known we can reference it in **38** as $\mathbf{cid}$. By Lemma 7, all but the last two invocations of $\alpha_F(\cdot)$ in $\alpha_T(c_1)$ and $\alpha_T(c_2)$ are different. In particular, $\alpha_T(c_2)$ has one more call to $\alpha_F(\cdot)$ (for the added frame). **38** does not imply any new facts abstracting the activation of c_1, so all the facts returned by $\alpha_T(c_2)$ for this activation must stay the same. If the second to last call to $\alpha_F(\cdot)$ in $\alpha_T(c_2)$ returns the same facts as the last call to $\alpha_F(\cdot)$ does in $\alpha_T(c_1)$ this is achieved. The last call to $\alpha_F(\cdot)$ in $\alpha_T(c_1)$ looks like $\alpha_F\big(F.\textsf{fid}, \mathbf{pc_1}, S, F, \epsilon, {\textit{instr}^*}_1\ {\textit{val}^*}^n\ \textsf{call}_{\mathbf{pc_2}}\ x\ \textit{instr}_2\big)$ while the second to last call in $\alpha_T(c_2)$ looks like $\alpha_F(F.\textsf{fid}, F'.\textsf{pc}, F'.\textsf{stor}, F, F'.\textsf{args} \quad F'.\textsf{index}, {\textit{instr}^*}_1\ \textsf{frame}_m\{F'\}\ {\textit{instr}^*}_3\ \textsf{end}\ {\textit{instr}^*}_2)$. From the transition rule, we see that $F'.\textsf{pc}$ is set to $\mathbf{pc_1}$ and $F'.\textsf{stor}$ is set to S. This means, the two invocations only differ in their last two arguments,

which are concretely ϵ and $\mathit{instr}^*{}_1\ \mathit{val}^{*n}\ \mathbf{call}_{\mathbf{pc_2}}\ x\ \mathit{instr}_2$ (henceforth i_1) in $\alpha_T(c_1)$ and val^n and $\mathit{instr}^*{}_1\ \mathbf{frame}_m\{F'\}\ \mathit{instr}^*{}_3\ \mathbf{end}\ \mathit{instr}^*{}_2$ (henceforth i_2). These two parameters are only used in the abstraction of *st*, where for $\alpha_T(c_1)$ we have $\eta_S(i_1) +\!\!+ \epsilon = \eta_S(i_1) = \eta_S(\mathit{instr}^*{}_1) +\!\!+ \mathit{val}^n$ and for $\alpha_T(c_2)$ we have $\eta_S(i_2) +\!\!+ F.\mathsf{args} = \eta_S(\mathit{instr}^*{}_1) +\!\!+ \mathit{val}^n$. This concludes this part of the proof. It remains to be proven that the facts returned by the last invocation of $\alpha_F(\cdot)$ in $\alpha_T(c_2)$ are derivable via ❸❽. This invocation is called with the following parameters $\alpha_F(F'.\mathsf{fid}, \mathbf{pc_2}, S, F', \epsilon, \mathbf{block}_{\mathbf{pc_2}}\ t_2^m\ \mathit{instr}^*\ \mathbf{end}_{\mathbf{pc_{end}}})$.
As established above, $\mathbf{pc_2} = 0$ and $\mathbf{cid} = F.\mathsf{module.funcaddrs}[x] = F'.\mathsf{fid}$. We observe that $\eta_S(\mathbf{block}_{\mathbf{pc_2}}\ t_2^m\ \mathit{instr}^*\ \mathbf{end}_{\mathbf{pc_{end}}})$ returns [], which is what is derivable via ❸❽. As can be deduced from the transition rule $F'.\mathsf{stor} = S$. Additionally, since we only analyze a single module in WAPPLER, $F.\mathsf{module} = F'.\mathsf{module}$. This means, that the memory and globals of the last call to $\alpha_F(\cdot)$ in $\alpha_T(c_1)$ have to be the same as in $\alpha_T(c_2)$. This is achieved by ❸❽ copying these values. Similarly, $S = F'.\mathsf{stor}$, which means setting the initial globals/memory to copies of *mem* and *gt* is sound. Lastly, the $n = \mathsf{Sl.as}(\mathbf{cid})$ values from the stack are taken, reversed and stored in the initial arguments and the locals field (here we introduce the required padding with zeroes). □

Lemma 38 (Single-Step Soundness for **call_indirect**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\mathit{val}^n\ (\mathsf{i32.const}\ i)\ \mathbf{call\_indirect}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\mathit{val}^n\ (\mathsf{i32.const}\ i)\ \mathbf{call\_indirect}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rules for **call_indirect** x are

$$\frac{\begin{array}{c} S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[0]].\mathsf{elem}[i] = a \\ F.\mathsf{module.types}[x] = f.\mathsf{type} \\ S.\mathsf{funcs}[a] = f \qquad f.\mathsf{type} = [t_1^n] \mapsto [t_2^m] \\ m \leq 1 \qquad f.\mathsf{code} = \{\mathsf{type}\ x, \mathsf{locals}\ t^k, \mathsf{body}\ \mathit{instr}^*\ \mathbf{end}\} \\ F' = \{\mathsf{module}\ f.\mathsf{module}, \mathsf{locals}\ \mathit{val}^n(\mathsf{t.const}\ 0)^k, \\ \mathsf{args}\ \mathit{val}^n, \mathsf{stor}\ S, \mathsf{pc}\ \mathbf{pc}, \mathsf{fid}\ a, \mathsf{index}\ (\mathsf{i32.const}\ i)\} \\ \mathit{instr}^{*\prime} = \mathit{annotate}(\mathbf{block}\ t_2^m\ \mathit{instr}^*\ \mathbf{end}\ \mathbf{end}) \end{array}}{S; F; \mathit{val}^n\ (\mathsf{i32.const}\ i)(\mathbf{call\_indirect}_{\mathbf{pc}}\ x) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \mathbf{frame}_m\{F'\}\ \mathit{instr}^{*\prime}}$$

$$\frac{|S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[0]].\mathsf{elem}| \leq i}{S; F; \mathit{val}^n\ (\mathsf{i32.const}\ i)(\mathbf{call\_indirect}\ x) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \mathbf{trap}}$$

$$\frac{S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[0]].\mathsf{elem}[i] = \epsilon}{S; F; \mathit{val}^n\ (\mathsf{i32.const}\ i)(\mathbf{call\_indirect}\ x) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \mathbf{trap}}$$

$$\frac{\begin{array}{c} S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[0]].\mathsf{elem}[i] = a \\ F.\mathsf{module.types}[x] \neq S.\mathsf{funcs}[a].\mathsf{type} \end{array}}{S; F; \mathit{val}^n\ (\mathsf{i32.const}\ i)(\mathbf{call\_indirect}\ x) \hookrightarrow\!\!\!\!\!\rightarrow S; F; \mathbf{trap}}$$

The main difference between **call** and **call_indirect** is that **call_indirect** read the function to be executed from the modules' table, while it is statically known for **call**. In case the index (the topmost stack element) is greater than the table's size **trap** is propagates (this is handled by ❹❺). Likewise, **trap** is propagated when the table at the index is not initialized or the type of the function stored in the function is not the required type. This case is handled by ❹❹. If $\mathbf{cid}$ is not one of the possible call targets (i.e. functions of the correct type in the module) or ϵ, *Trap* is implied. The list of possible call targets is only known if no functions have been added to the table since the module was initialized. In WebAssembly 1.0 only host function can add functions to the store and modify the table. Such a function can always cause a **trap** which is handled by ❹❼.

In the non-trapping case, the abstraction is handled similar to Lemma 37, with three changes: ❹❶ is generated for every possibly called function, identified by $\mathbf{cid}$. If $\mathbf{cid}$ is in the Table at index i (as generated by $\alpha_t(c_1)$, an *MState*-fact in function $\mathbf{cid}$ at program counter 0 is implied. If functions have been added by a host function (i.e., *FunctionsAdded* was derived), ❹❻ implies an *MState* with an arbitrary store. Lastly, in the second to last invocation of $\alpha_F(c_2)$, we have to also reconstruct $\mathsf{i32.const}\ i$ which was taken from the stack by **call_indirect**. As can be seen in the transition rule, $F.\mathsf{index}$ is set to $\mathsf{i32.const}\ i$ which is then prepended to the fifth argument of $\alpha_F(\cdot)$. The rest of the proof proceeds as Lemma 37. □

Lemma 39 (Single-Step Soundness for **return**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\mathbf{return}_{\mathbf{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\mathbf{return}_{\mathbf{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. The transition rule for **call_indirect** x is

$$\mathbf{frame}_n\{F\}B^k\ [\mathit{val}^n\ \mathbf{return}_{\mathbf{pc}}]\ \mathbf{end} \hookrightarrow\!\!\!\!\!\rightarrow \mathit{val}^n$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathbf{frame}_n\{F'\}\ B^k\ [\mathit{val}^n\ \mathbf{return}_{\mathbf{pc_1}})]\ \mathbf{end}_{\mathbf{pc_{end}}}])$$
$$c_2 = (S; F; E[\mathit{val}^n\ \mathit{val}^*\ \mathbf{cmd}_{\mathbf{pc_2}}])$$

This behavior is handled by ❹❽, ❸❾, and ❹❷. We first observe that every activation (especially $\mathbf{frame}_n\{F'\}\ \ldots\ \mathbf{end}_{\mathbf{pc_{end}}}$) is put on the stack by an instance of $\mathbf{call}_{\mathbf{pc'}}$ or $\mathbf{call\_indirect}_{\mathbf{pc'}}$ (summarized as $\mathbf{cmd}_{\mathbf{pc'}}$ in the following). When this instruction is replaced, the store and the arguments at this time are copied into F', the frame object of the new activation. These values are immutable and used to abstract the initial globals/memory/arguments in the new activation (in the last invocation of $\alpha_F(\cdot)$ in $\alpha_T(c_1)$) and to restore the state at the in second topmost activation (in the second to last invocation of $\alpha_F(\cdot)$ in $\alpha_T(c_1)$). When ❹❽ implies a new *Return*-fact the initial values are copied in the last three arguments. At the same time, we are guaranteed to have *MState*-facts as they were at the time $\mathbf{cmd}_{\mathbf{pc'}}$ was

executed in the premises (see Lemma 37 and Lemma 38), which are combined with with the *Return*-fact implied by ❹❽ in ❸❾, and ❹❷. ❸❾, and ❹❷ imply *MState*-facts that put val^n on the stack and set the program counter to $\mathbf{pc}' + 1$ which is the start of the constant prefix of $\mathbf{cmd_{pc_2}}$. An application of Lemma 11 concludes the proof. □

Lemma 40 (Single-Step Soundness for returning from a function)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\mathbf{frame}_n\{F'\}\ val^n\ \mathbf{end_{pc}}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha_T(c_1), \alpha_C(S) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[\mathbf{frame}_n\{F'\}\ val^n\ \mathbf{end_{pc}}]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

The transition rule for returning implicitly from a function is

$$\mathbf{frame}_n\{F\}\ val^n\ \mathbf{end} \hookrightarrow\!\!\!\!\rightarrow val^n$$

This means that c_1 and c_2 look as follows

$$c_1 = (S; F; E[\mathbf{frame}_n\{F'\}\ val^n\ \mathbf{end_{pc_1}}])$$
$$c_2 = (S; F; E[val^n\ val^*\ \mathbf{cmd_{pc_2}}])$$

We observe, that ❹❽ is instantiated for every function at the program counter after the last program counter. The rest of the proof proceeds as in Lemma 39.

Lemma 41 (Single-Step Soundness for calling host functions)**.** *The effects of host functions are soundly overapproximated by* ❶-❸.

Proof. The transition rules for calling host functions in $\hookrightarrow$ are as follows

$$\frac{S.\mathsf{funcs}[a] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \quad (S'; result) \in hf(S; val^n)}{S; val^n\ (\mathbf{invoke}\ a) \hookrightarrow S'; result}$$

$$\frac{S.\mathsf{funcs}[a] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \quad \bot \in hf(S; val^n)}{S; val^n\ (\mathbf{invoke}\ a) \hookrightarrow S; val^n\ (\mathbf{invoke}\ a)}$$

In terms of reachability, the second case (i.e., not progressing on diverging host code) is trivial. The second case notably does not modify the frame object of the configuration only rewrites val^n to *return* on the stack. The standard furthermore specifies how S' may differ S: functions might be added but not removed or changed; globals may be added (which cannot be observed in already instantiated modules) or modified, if they are marked mutable; memory instances might be added (which cannot be observed from already instantiated modules) and existing memory instances might be grown or modified; and table instances might be added (which cannot observed from already instantiated modules) and existing table instances might be grown or modified. We assume the existence of the functions $\mathsf{SI.bnd}_G$, $\mathsf{SI.bnd}_R$, and $\mathsf{SI.sm}$ that, given a store S and a module instance m, fulfill the following axioms:

$$\forall a\ hf\ l\ h\ val^n\ result$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge (l, h) = \mathsf{SI.bnd}_R(a, i) \wedge (\_, result) \in hf(S; val^n) \implies l \overset{?}{\leq} result[i] \overset{?}{<} h$$

$$\forall a\ hf\ l\ h\ val^n\ S'$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge (l, h) = \mathsf{SI.bnd}_G(a, i) \wedge (S', \_) \in hf(S; val^n) \implies l \overset{?}{\leq} S'.\mathsf{globals}[m.\mathsf{globaladdrs}[i]] \overset{?}{<} h$$

$$\forall a\ hf\ l\ h\ val^n\ S'$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge \text{TM} \notin \mathsf{SI.sm}(a, i) \wedge (S', \_) \in hf(S; val^n) \implies S.\mathsf{mems}[m.\mathsf{memaddrs}[0]] = S'.\mathsf{mems}[m.\mathsf{memaddrs}[0]]$$

$$\forall a\ hf\ l\ h\ val^n\ S'$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge \text{GM} \notin \mathsf{SI.sm}(a, i) \wedge (S', \_) \in hf(S; val^n) \implies |S.\mathsf{mems}[m.\mathsf{memaddrs}[0]]| = |S'.\mathsf{mems}[m.\mathsf{memaddrs}[0]]|$$

$$\forall a\ hf\ l\ h\ val^n\ S'$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge \text{TT} \notin \mathsf{SI.sm}(a, i) \wedge (S', \_) \in hf(S; val^n) \implies S.\mathsf{tables}[m.\mathsf{tableaddrs}[0]]| = |S'.\mathsf{tables}[m.\mathsf{tableaddrs}[0]]$$

$$\forall a\ hf\ l\ h\ val^n\ S'$$
$$S.\mathsf{funcs}[m.\mathsf{funcaddrs}[a]] = \{\mathsf{type}\ t_1^n \to t_2^m, \mathsf{hostcode}\ hf\} \wedge \text{AF} \notin \mathsf{SI.sm}(a, i) \wedge (S', \_) \in hf(S; val^n) \implies \nexists f\ .\ f \in S'.\mathsf{tables} \wedge f \notin S.\mathsf{tables}$$

For every imported function, we instantiate ❶ which implies *Return*-facts which are used by Lemma 37 and Lemma 38. The returned values, globals, and memory cells conform to the specification. In case the table can be modified, ❷ can derive arbitrary *Table*-facts. In case the functions are added to the store (and the table), ❸ can derive *FunctionsAdded*. If a function is added to the store (and the table), all of the subsequent invocations of **call_indirect** could modify the store/frame without any guarantees (see ❹❻ in Lemma 38. □