WAPPLER: Sound Reachability Analysis for WebAssembly (Technical Report, Draft)

Anonymous Authors

***Abstract*—WebAssembly (Wasm) is an increasingly common low-level language to provide 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 is of utmost importance, which motivates the development of sound and automated static analysis techniques. While Wasm provides a formal semantics that helps the development of such techniques, one must take care since the specification devices of the semantics aim at elegance, not efficiency of implementation.**

We present WAPPLER, to our knowledge, the first sound and automated analysis technique for WebAssembly. To make implementing the analysis with SMT solvers viable, we first introduce annotations to the semantics, which we then use in our abstraction. To increase precision, we introduce a mechanism to specify assumptions on environment behavior. Lastly, we provide the formalization of several general and Wasm-specific security properties to demonstrate the expressiveness of our analysis.

I. INTRODUCTION

WebAssembly [24] (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 some isolation guarantees [7].

These isolation guarantees were designed with its original use case, performant code in web browsers, in mind but turned out to be useful in smart contracts [1], [2], plugin systems [21], [25], isolation of untrusted components [15], lightweight containers [5] and edge computing [11], [17].

Since all of these use cases are security relevant, being able to verify security properties in WebAssembly modules is crucial.

So far there have been works on verifying the semantics of WebAssembly itself [26], [28], mechanized (but non-automated) verification of program properties [18], [27], automated (but unsound) bug-finding [3], [10], [22] and identification of malicious WebAssembly code [8], [16], [19], but, to our knowledge no sound *and* automated verification techniques.

This is a challenging research task for various reasons:

While WebAssembly laudably [24], [32] comes with a formally specified semantics, this semantics is at its core described as a series of rewriting rules on an evaluation context. As recently described in other work [29], a direct implementation of this approach comes with several performance issues, chief amongst them the need to rebuild the evaluation context in each step. Additionally, directly translating the sequence of symbols that is rewritten would necessitate the use of nested data structures which are less performant to use in SMT solvers than flat structures.

WebAssembly is meant to be embedded in larger systems that provide application-specific functionalities. The interactions with these embedders are subject to rules according to the WebAssembly specification but overapproximating them soundly, respectively explicating assumptions about them, requires care.

Lastly, WebAssembly is a low-level language. While being more structured than other low-level languages, it still provides less information for analysis than higher-level languages. 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 [9] has identified that WebAssembly lacks certain mitigations for memory safety violations that modern compilers can provide for native code.

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.

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 for placing assumptions on embedder behavior to enhance precision (while soundly overapproximating all cases not covered by the assumptions)

Our contributions can be summarized as follows:

- WAPPLER, the first sound and automated static analysis technique for WebAssembly;
- A formal soundness proof for WAPPLER against the original WebAssembly semantics;
- A formalization of several reachability properties to rule out certain general and Wasm-specific 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.

[1]WebAssembly Proven Program Logic Encoding Reachability

```
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 1))
19 )
```

Fig. 1: Example of a WebAssembly module

II. Preliminaries

This section provides an overview of Wasm, focusing in particular on Wasm 1.0 [24]. We refer to the official live documentation [32] and the initial publication [7] 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. The functions consist of a sequence of instructions that are not first-class, meaning they cannot be passed as parameters to other functions. 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).

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). Since these are the only types, complex data such as arrays, objects, and pointers from a higher-level source language must be translated to these primitive types when compiled. 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 based on a stack machine and instructions in Wasm pop values from and push results to an implicit operand stack. There are six different kinds of instructions in Wasm: *numeric* instructions that provide basic operations (e.g., addition) over numeric values, *parametric* instructions that can operate on operands of any value (e.g., throwing away a single operand), *variable* instructions that allow access to local and global variables, *memory* instructions that manipulate the linear memory, *control* instructions that affect the control flow (e.g., nop and branching), and *administrative* instructions. In the function in Figure 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 (when interpreted signed) than the topmost element (and pushes either i32.**const** *0* or tConsti321 as a result), 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: $\textbf{label}_n\{\epsilon\} \cdot \textbf{end}$ is used to model local control flow and is described in greater detail below. $\textbf{frame}_n\{F\}\cdot\textbf{end}$, which is called an *activation*, describes a function on the call stack (which WebAssembly supports as a concept).

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 description of a program state. The configurations can be described 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 of the current function F 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 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 can be seen in the definition below, an evaluation context is an instruction sequence that is preceded by a sequence of already evaluated values and $\textbf{label}_n\{\mathit{instr}^*\}$ instructions.

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

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 $\textbf{frame}_n\{F'\}$ 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 $S; F;$ **trap** as a terminal configuration).

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

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

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

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

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 $t.op$ that takes the top n values from the stack and applies a function op to them.[2]

$$\frac{c \in op(c_1, \ldots, c_n)}{(\mathsf{t}_1.\textbf{const}\ c_1) \ldots (\mathsf{t}_n.\textbf{const}\ c_n)\ t.op \hookrightarrow (\mathsf{t}.\textbf{const}\ c)}$$

Similarly, the following rule models the unsuccessful execution.

$$\frac{c \notin op(c_1, \ldots, c_n)}{(\mathsf{t}_1.\textbf{const}\ c_1) \ldots (\mathsf{t}_n.\textbf{const}\ c_n)\ t.op \hookrightarrow (\textbf{trap})}$$

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

4) Structured Control-Flow: The instructions of a function in Wasm are organized into blocks, 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 $\textbf{label}_n\{\mathit{instr}^*\}\ \cdot\ \textbf{end}$.

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

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

These labels are then used to define the so-called *block context* as follows:

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

The block context is then used to define the behavior of control flow instructions such as **br** and **return**. As shown below, when such a label becomes to 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.

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

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

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.[3] 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 values 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[4] 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 the **memory.grow** instruction (unless it will exceed an optionally-specified maximum size), and the current size can be queried using the **memory.size** instruction. 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.

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

[3]See sections 4.2.3 and 4.2.5 of [24] for the full definition.

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

III. Analysis

Our analysis is based on a Horn-clause-based abstraction in the style of [6]. 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 predicate applications (also called facts) 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, the logical entailment $\vdash$ 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: in subsection III-A, we will present our lightweight annotations to the semantics, in subsection III-B, we will review some reachability properties that are desirable for Wasm modules, and in subsection III-C, we will present parts of our abstract semantics.

A. Annotated Semantics

The original semantics of WebAssembly, with its progress based on rewriting a sequence of instructions, is not well-suited for being analyzed with SMT solvers. Consider, for example, the following configuration, which may appear during the evaluation of the code in Figure 1.

$$S; F_0; \textbf{frame}_1\{F\}\ \textbf{label}_1\{\epsilon\}\ \textbf{label}_1\{\epsilon\}$$
$$(\text{i32}.\textbf{const}\ \mathit{42})\ (\text{i32}.\textbf{const}\ \mathit{1})\ (\textbf{br\_if}\ \mathit{0})\ (\text{i32}.\textbf{const}\ -\mathit{1})\ \text{i32}.\textbf{mul}$$
$$\textbf{end end end}$$

In order to derive the next configuration, we need first to apply the structural rule SR2. This allows us to start a sub-derivation with the thread consisting of F and the instruction sequence within the activation. In the sub-derivation, we have to apply SR1to determine the next configuration, and in order to do that, we have to decompose the instruction sequence into an evaluation context $E[\mathit{instr}^*]$ such that instr^* can be rewritten to $\mathit{instr}^{*\prime}$. The instruction sequence can be decomposed into different evaluation contexts e.g. $E[\text{i32}.\textbf{const}\ \mathit{42}]$ or $E[(\text{i32}.\textbf{const}\ \mathit{1})]$, but the only one where a step is possible is $E[(\text{i32}.\textbf{const}\ \mathit{1})\ (\textbf{br\_if}\ \mathit{0})]$ which can be rewritten to $E[\textbf{br}\ \mathit{0}]$. The next evaluation context on which a transition may happen, however, is not $E[\textbf{br}\ \mathit{0}]$ but $E[\textbf{label}_1\{\epsilon\}\ (\text{i32}.\textbf{const}\ \mathit{42})\ (\textbf{br}\ \mathit{0})\ (\text{i32}.\textbf{const}\ -\mathit{1})\ \text{i32}.\textbf{mul}\ \textbf{end}]$ which can be rewritten to $E[\text{i32}.\textbf{const}\ \mathit{42}]$.

The evaluation context and the structural rules are primarily specification devices, but naively relying on it to structure an analysis leads to inefficiencies. Similar observations have recently been made in the domain of Wasm interpreters, where e.g. the official reference implementation struggles with deeply nested function calls or control flow structures [29].

Instead of traversing the activations and different evaluation contexts in each step to find out which rewrite to apply next, we will instead identify the different points in the execution with a program counter and a function id. There are two obstacles original formulation presents to this endeavor: 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. If an instruction carries a program counter $\mathtt{pc}$ (and it is relevant in the given context), we write $\textbf{cmd}_{\mathtt{pc}}$ instead of **cmd**. 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 $E[\mathit{val}^*\ \textbf{cmd}]$ 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 $\textbf{label}_n\{\mathit{instr}^*\}$ 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 $\mathtt{t}.\textbf{const}$ as results of numeric instructions) do not carry a program counter (with one exception in 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.

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

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

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

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

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

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):

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

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 instructions the stored instructions before putting them in the configuration. The $\mathit{annotate}(\cdot)$-function just assigns an increasing counter to each instruction, with the exception that in case of **if** and **else**, we leave a "gap" to be able to uniquely identify the start and end of the instructions in the then-branch (see below). 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.

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

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

$$\frac{c \neq 0}{(\mathsf{i32.}\mathbf{const}\ c)\ \mathbf{if}\ t^n\ \mathit{instr}_1^*\ \mathbf{else}_{pc}\ \mathit{instr}_2^*\ \mathbf{end} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_n\{\epsilon\}\mathit{instr}_1^*\mathbf{end}_{pc-1}}$$

$$\frac{c = 0}{(\mathsf{i32.}\mathbf{const}\ c)\ \mathbf{if}\ t^n\ \mathit{instr}_1^*\ \mathbf{else}\ \mathit{instr}_2^*, \mathbf{end}_{pc} \hookrightarrow\!\!\!\!\!\rightarrow \mathbf{label}_n\{\epsilon\}\mathit{instr}_2^*\mathbf{end}_{pc}}$$

B. Reachability Properties for WebAssembly

We will now examine a simple example to illustrate to what degree reachability analysis can be helpful to analyze Wasm-modules. The program in Fig. 3 models a simple game that is played on an 8x8 grid (such as chess or mine sweeper). The game state is stored in a `char` array of appropriate size, `game_state`; `update_game_state` updates the cell at the appropriate coordinates to a provided value and afterward calls `eval` on a constant string value `trusted`. `eval` and `trusted` are stand-ins for a privileged function and a piece of data that is assumed to be trusted and immutable. A concrete instance in a browser would be a function that calls out to the JavaScript environment and evaluates a piece of code that is stored in WebAssembly (where it is important to preserve the integrity of `trusted` 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 and linkers have means to assure the integrity of `trusted`, which could, for example, be put on a read-only page in the RAM, thereby raising an exception when written to.

As observed by Lehmann et al. [9], WebAssembly does not provide such mitigations: all data that is stored in the linear memory is potentially writeable. The vulnerability in Fig. 3 stems from insufficient restriction on `x` and `y`: since these values are treated as signed, they could be negative and then be used in the index of `game_data`. In the compiled version of the function (shown in Fig. 4), 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 [9], 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[5].

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. 4) to be able to write to it. A security property that we will call *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 is executed, whose operands have the same

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

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

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

Fig. 2: Definition of the annotation function.

```c
static const char* trusted = "TRUSTED";
char game_state[64] = ...;
extern void eval(const char*);

void update_game_state(int x, int y, char c) {
  if (x < 8 && y < 8) {
    game_state[y * 8 + x] = c;
  }
  eval(TRUSTED);
}
```

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

```
 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. 4: Example of a vulnerable function (Wasm).

sign, the 32-bit-sum of its operands has the same sign as the operands.

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

Similar properties can be provided for other potentially overflowing instructions such as i32.**sub**, i64.**mul** etc.

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. 4) to a memory region (line 22 in Fig. 4) 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. 4, for instance, addresses 1024 to 1032 never should be written to. Given this information as an interval of memory addresses $[\mathit{low}, \mathit{high}]$, 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*[6] we have that that $\mathit{ea} + N/8$ is smaller than *low* and *ea* is greater than *high*.

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

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:

```c
void assert(int b) {
  if(!b) reach_error();
}
```

Given these two functions, we can check for the absence of assertion violations by ensuring that `reach_error` never is called.

In our example program, we could, for example, assert that `trusted` indeed contains `"TRUSTED"` before `eval` is called like this:

```c
assert(trusted[0] == 'T');
assert(trusted[1] == 'R');
...
assert(trusted[6] == 'D');
assert(trusted[7] == '\0');
```

Assuming `reach_error` has the function id **fid**$_{re}$, we can (given the program contain `assert` as defined above) as follows:

[6]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*.

A module starting its execution in a configuration c_1 does not violate any assertion, if for any configuration $(S; F; \mathit{instr}^*)$ that can be derived (including sub-derivations), we have that that $F.\mathsf{fid} \neq \mathbf{fid}_{re}$.

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

An important caveat for source-level specification of properties is that an overeager compiler might optimize them away even when the problematic behavior can occur during the execution [33][7].

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.[8] The restriction on a single analyzed module instance could be easily lifted by adding a parameter identifying the instance to the *MState*- and *Table*-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. 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 $\mathbb{B}_{64}^?$. *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.

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

The signature of predicates over which our analysis is defined is shown in Fig. 5. Given a valid WebAssembly module, we can precompute useful information for the analysis. 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. Given this information, we can describe the $\mathit{MState}_{\mathbf{fid} \times \mathbf{pc}}$-predicate, which (in dependence of $\mathbf{fid}$ and $\mathbf{pc}$) has the following arguments: a stack of values, two tuples holding the globals and locals, and a memory cell. Additionally, it holds the values 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. $\mathit{Return}_{\mathbf{fid}}$ 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 $\mathit{Trap}_{\mathbf{fid}}$-predicate abstracts encountering an evaluation context $E[\mathbf{trap}]$ 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 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 ❽, ❾).

3) Abstraction: The abstraction function $\alpha(\cdot)$ is described in Fig. 6. $\alpha(\cdot)$ 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 $E[\mathbf{cmd}_{\mathbf{pc}}]$, 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 $\mathit{Trap}_{\mathbf{fid}}$-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 $\mathtt{t}.\mathbf{const}\ v$ 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 $\mathit{zeropad}(\cdot)$. $\alpha_{seq}(\cdot)$ lifts $\alpha_V(\cdot)$ to sequences.

[7]The presented assertions are indeed compiled away by `emcc`. By removing the `static` keyword from `trusted`'s declaration, the compiler can be convinced not to apply this particular optimization which does not change the program behavior when executed with WebAssembly.

[8]Instantiation and invocation are described in full detail in [24] in sections 4.5.4 and 4.5.5. We additionally assume that the `start` function has already been executed.

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

Fig. 5: Predicate signature of the abstract analysis.

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 [13] 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[9]. For this, it iterates over all functions in S, assigning its index in funcs as function id. It the annotates the code with $\mathit{annotate}(\cdot)$ and abstracts the command $\mathbf{cmd}_{\mathbf{pc}}$ with a function $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$. In the case of an **if** · **else** · **end**-construct, we introduce "virtual" blocks to handle their semantics more uniformly. For space reasons, we do not present $(\!|\cdot|\!)^{\mathbf{fid}}_{\mathbf{pc}}$ in the main body of the paper but restrict ourselves to select examples. A selection of Horn clauses that abstract the behavior of function $\mathbf{fid}$ at $\mathbf{pc}$ is given in Fig. 7. ❶ 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. The for handling binary operations (❷-❹), 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 (❷,❸). If the result can be ϵ, we imply $\mathit{Trap}_{\mathbf{fid}}$ (❸).

For a more complicated rule, consider the rule abstracting **call_indirect** (❺-⓫). 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. ❺: To call a function, we take the correct amount of values from the stack[10], 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". ❻: Returning from a function looks similar: we construct *at* the same way as in ❺ 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. ❼: Works similar to ❻ but propagates traps instead of values. ❽ and ❾ 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. ❿ and ⓫ encode the trap case for **call_indirect**. ❿: If the value in the table at index x does not correspond to one of the possible functions, we trap. ⓫: Likewise, we trap if there is a table of smaller size than x.

A complete list of abstract semantics rules is available in the appendix.

4) Overapproximation of Host Functions: The WebAssembly standard gives a few guarantees on host behaviour:[11] They may trap or return, but if they return, they have to return values of the appropriate types. They may not alter the data stored on the call stack but may alter the store. When altering the store, a host function may write arbitrary values to memories and tables and may grow (but not shrink) them. Furthermore, it may not modify immutable globals, and it may not remove or alter functions. It may, however, add new functions.

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 control flows within the module. Every execution of the **call_indirect** instruction looks up the function that is about 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, as we will see, possible, but doing so for every call to an imported function would be detrimental to our analysis' precision. To mitigate this problem, we allow to specify restrictions on the behavior of host functions.

We assume the existence of the (partial) functions for the

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

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

[11]A full description can be found in section 4.4.7.3 of [24].

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

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

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

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

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

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

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

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

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

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

$$\alpha_M(i, S, F) = Mem\big(i, d[i], |d|/2^{16}\big) \quad \text{where } d = \eta_M(S, F).\mathsf{data}$$

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

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

Fig. 6: The abstraction function for configurations.

analyzed module:

$$\begin{aligned} \mathsf{SI.bnd}_G &: \mathbb{N} \times \mathbb{N} \mapsto (\mathbb{B}^?_{64} \times \mathbb{B}^?_{64}) \cup \{\circledcirc\} \\ \mathsf{SI.bnd}_R &: \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}^?_{64} \times \mathbb{B}^?_{64} \\ \mathsf{SI.sm} &: \mathbb{N} \mapsto \mathcal{P}(\{\text{TM}, \text{GM}, \text{TT}, \text{AF}\}) \end{aligned}$$

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

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. $\text{TM} \notin \mathsf{SI.sm}(\mathbf{fid})$ means that the function **fid** may not touch the memory).

In Fig. 8, the abstractions for imported functions **fid** are presented. In ⓬, 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 $\overset{?}{\le}$ 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 value for memory contents and size is dependent on the set returned by SI.sm. Rule ⓭, allows to derive arbitrary *Table* facts, while ⓮ implies the *FunctionsAdded* fact, which is handled explicitly in ❽.

5) *Soundness Statement:* Our refinement relation $\ge$, where $\Delta_1 \ge \Delta_2$ means that Δ_1 abstracts all concrete states that Δ_2 abstracts (and possibly more) can simply be defined as set

❶ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\mathbf{v} :: st, gt, lt, mem, at_0, gt_0, mem_0)$$

i32.**const** $\mathbf{v}, \ldots$

❷ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \circledast_{\mathbf{op}} x \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$$

i32.**add**, f32.**sub**, ...

❸ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \widetilde{\circledast_{\mathbf{op}}} x \land res \in \mathbb{B}_{64} \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$$

❹ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \widetilde{\circledast_{\mathbf{op}}} x \land \neg(res \in \mathbb{B}_{64}) \implies \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

i64.**mod**, i32.**div**, ...

❺ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{Table}(x, \mathbf{cid}, ts) \land \mathit{reverse}(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies \mathit{MState}_{\mathbf{cid},0}([\,], gt, at +\!\!+ [0; \mathsf{SI.ls}(\mathbf{cid}) - \mathsf{SI.as}(\mathbf{cid})], mem, at, gt, mem)$$

❻ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{Table}(x, \mathbf{cid}, ts) \land \mathit{Return}_{\mathbf{cid}}(rt, rgt, rmem, at, gt, mem) \land \mathit{reverse}(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(rt +\!\!+ st[\mathsf{SI.as}(\mathbf{cid}) :], rgt, lt, rmem, at_0, gt_0, mem_0)$$

❼ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{Table}(x, \mathbf{cid}, ts) \land \mathit{Trap}_{\mathbf{cid}}(at, gt, mem) \land \mathit{reverse}(st[: \mathsf{SI.as}(\mathbf{cid})]) = at \implies \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

❽ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, \mathit{Mem}(i, v, size), at_0, gt_0, mem_0) \land 0 \leq v' \leq 255 \land size \leq size' \leq \mathbf{max} \land \mathit{FunctionsAdded}() \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}\big(rt +\!\!+ st[\mathbf{as} :], rgt, lt, \mathit{Mem}\big(i, v', size'\big), at_0, gt_0, mem_0\big)$$

❾ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{FunctionsAdded}() \implies \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

❿ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{Table}(x, mte, ts) \land (mte = \epsilon \lor mte = te) \land \bigwedge_{(\mathbf{idx},\mathbf{cid}) \in \mathsf{possibleCallTargets}(\mathbf{fid},\mathbf{pc})} te \neq \mathbf{cid} \implies \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

⓫ $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \land \mathit{Table}(ti, mte, ts) \land x \geq ts \implies \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

call_indirect t

Fig. 7: Abstractions for some instructions.

⓬ $$\mathit{MState}_{\mathbf{fid},0}(st, gt, lt, \mathit{Mem}(i, v, size), at_0, gt_0, mem_0) \land \big(\textsc{tm} \in \mathsf{SI.sm}(\mathbf{fid})\big) \mathbin{?} \big(0 \leq v' \leq 255\big) : \big(v = v'\big) \land \bigwedge_{0 \leq \mathbf{i} < \mathsf{SI.gs}()} \Big((\mathbf{lb}, \mathbf{ub}) = \mathsf{SI.bnd}_G(\mathbf{fid}, \mathbf{i})\Big) \mathbin{?} \Big(\mathbf{lb} \overset{?}{\leq} rgt[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) : \Big(rgt[\mathbf{i}] = gt[\mathbf{i}]\Big) \land \bigwedge_{\substack{0 \leq \mathbf{i} < \mathsf{SI.gs}() \\ (\mathbf{lb},\mathbf{ub}) = \mathsf{SI.bnd}_R(\mathbf{fid},\mathbf{i})}} \Big(\mathbf{lb} \overset{?}{\leq} rt[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) \land \big(\textsc{gm} \in \mathsf{SI.sm}(\mathbf{fid})\big) \mathbin{?} \big(size \leq size' \leq \mathbf{max}\big) : \big(size = size'\big) \implies \mathit{Return}_{\mathbf{fid}}\big(rt, rgt, \mathit{Mem}\big(i, v', size'\big), at_0, gt_0, mem_0\big)$$

⓭ $$\mathit{MState}_{\mathbf{fid},0}(st, gt, lt, mem, at_0, gt_0, mem_0) \land \textsc{tt} \in \mathsf{SI.sm}(\mathbf{fid}) \implies \mathit{Table}(idx, te, tsz)$$

⓮ $$\mathit{MState}_{\mathbf{fid},0}(st, gt, lt, mem, at_0, gt_0, mem_0) \land \textsc{af} \in \mathsf{SI.sm}(\mathbf{fid}) \implies \mathit{FunctionsAdded}()$$

Fig. 8: Abstractions for imported functions.

inclusion:

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

All these definitions finally allow us to formulate a soundness theorem:

Theorem 1 (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.*

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

We prove this result in the appendix A, in Corollary 2.

IV. Implementation and Evaluation

A. Implementation

WAPPLER was implemented using the HORST framework [20] 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 WebAssembly modules and calculate static information like program counters and stack layouts. Since this framework did not yet support tuple types (which we use excessively for stacks, locals, globals etc.), we extended HORST's 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). We made the HoRSt developers aware of these changes, and they will be included in the next release.

Our tool is available under: https://anoncodedrop.github.io/.

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 Figure 1. It is meant to calculate the absolute value of a function f that is provided by the embedder via the imported table. As we cannot make any assumptions on the behavior of f, the execution starts with the *FunctionsAdded* fact already in the abstract configuration which allows the **call_indirect** instruction in line 6 to return arbitrary values (see ❽). We can, however, 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.

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

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: When analyzing `update_game_state` from Figure 4, we can use our overapproximation of imported functions to check all possible inputs. To do that, we extend the C file with the following definitions:

```
extern int nondetint(void);
extern char nondetchar(void);
...
void run_test() {
  update_game_state(nondetint(),nondetint(),
    nondetchar());
}
```

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 defined in III-B1 can be overapproximated by verifying the unsatisfiability of the following formula for every occurrence of i32.**add** in the module (where **fid** and **pc** are instantiated to the appropriate values).

$$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: y :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0) \\ \land (x < 0) = (y < 0) \land (x + y < 0) \neq (y < 0)$$

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

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

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 **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 **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.

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 [31]) 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.

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 at the same time.

The possibly first static analysis tool for WebAssembly discussed in academic literature was presented in [9]. 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*) [14], there have been multiple tools that aim to classify binaries as cryptojackers. These include MinerRay [19], MineSweeper [8], and MINOS [16]. None of these come with a soundness guarantee, which is also not the focus here 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), graphs 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 vulnerabilities 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 [10] 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 [22] 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 [26] that was later extended in [28]. [27] introduces Wasm Logic, a sound program logic with which the authors were able to verify a B-Tree implementation. [18] focuses on the interaction between modules and presents a mechanized higher-order separation logic that they use to reason about a stack-module that make use use of imported functions.

In the realm of mitigations for memory unsafety, several approaches have been proposed. MSWasm [4], [12] is a memory-safe extension to WebAssembly that can make use of higher-level information to protect against memory safety violations. [23] 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 [30], which would allow for implementing syntactically verifiable read-only memories.

VI. Conclusion and Future Work

We presented Wappler, to our knowledge, 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.

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] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] M. Vassena and M. Patrignani. Memory safety preservation for webassembly. In *47th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2020.

[24] W3C. WebAssembly Core Specification, version 1.0. https://www.w3.org/TR/wasm-core-1/.

[25] 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].

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] WebAssembly Community Group. Multiple per-module memories for Wasm. https://github.com/WebAssembly/multi-memory.

[31] WebAssembly Community Group. spec/test/core at f2b62c3067ac7e9e367296378621ccbd4fee79c1 · WebAssembly/spec. https://github.com/WebAssembly/spec/tree/f2b62c3067ac7e9e367296378621ccbd4fee79c1/test/core.

[32] WebAssembly Community Group. WebAssembly Core Specification, live document. https://webassembly.github.io/spec/core/.

[33] 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. 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 A.1 (Non-trapping Configuration)**.** We call a configuration whose instruction sequence does not contain trap *non-trapping configuration*.

Definition A.2 (Currently Executing Activation)**.** Within instruction sequence (or within the instruction sequence of a concrete configuration) we call a (well-nested) subsequence $\mathsf{frame}_n\{F\}$ *instr** end the *currently executing activation*, if *instr** does not contain any activations.

Definition A.3 (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: $val^* \mathsf{cmd}_{\mathbf{pc}}$
- Branching instructions: $\mathsf{label}_{instr^*}\{B\}^k[val^* \mathsf{cmd}_{\mathbf{pc}}]\mathsf{end}$
- Exiting a block: $\mathsf{label}_n\{instr^*\}val^* \mathsf{end}_{\mathbf{pc}}$

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

We call the instruction whose program counter annotation is shown in the above listing the *currently executing instruction*. 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 A.4 (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*.

Lemma 1 (Determinism of $\hookrightarrow\!\!\!\!\rightarrow$)**.** *For every instruction sequence, there is at most one rule to apply.*

Proof. By exhaustive inspection of all rules. □

Lemma 2 (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 $\hookrightarrow\!\!\!\!\rightarrow$ these are handled as follows:

- **call** k, **call_indirect** (**invoke** does not appear in the definition of $\hookrightarrow\!\!\!\!\rightarrow$): 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 $\textsf{label}_n\{\mathit{instr}^*\}$ on the stack, but *instr** is annotated
- **br_if** l, **br_table** l^*: in the $\hookrightarrow$, these rewrite to **br** l, in $\hookrightarrow\!\!\!\!\rightarrow$ 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 1. *All instructions to the right of the currently executing instruction have a program counter annotation.*

Lemma 3 (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 4 (Immutable frame fields)**.** *The fields* pc*,* stor*,* args*, and* index *of frame objects are not modified by any rule.*

Proof. By exhaustive inspection. □

Lemma 5 (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**, $\textsf{frame}_n\{F\}\,\mathit{instr}^*\,\textsf{end}$, SR4), or does not change the number of activations (all other). □

Lemma 6 (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 5 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 3, 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 4), 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 function 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 function 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 7 (Development of the currently executing instruction's program counter)**.** *Given a concrete step $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$ where the currently executing instruction in c_1 is* $\mathbf{cmd}_{\mathbf{pc}_1}$ *and* $\mathbf{cmd}_{\mathbf{pc}_2}$ *in c_2, and* $\mathbf{cmd}_{\mathbf{pc}_1}$ *is a local instruction: If there is is a rule $MState_{\mathbf{fid},\mathbf{pc}_1}(\ \dots\ ) \implies MState_{\mathbf{fid},\mathbf{pc}'}(\ \dots\ )$, then either $\mathbf{pc}' = \mathbf{pc}_2$, or there is a sequence of annotated constant instructions before* $\mathbf{cmd}_{\mathbf{pc}_2}$*, i.e.* $\mathsf{t}.\mathbf{const}_{\mathbf{pc}'}, \mathsf{t}.\mathbf{const}_{\mathbf{pc}'+1}, \dots, \mathsf{t}.\mathbf{const}_{\mathbf{pc}_2-1}\mathbf{cmd}_{\mathbf{pc}_2}$.

Proof. Let *instr*$^* \hookrightarrow\!\!\!\!\rightarrow$ *instr*$^{*\prime}$ be the non-structural rule that has been applied. We proceed by case distinction on the result of $\mathbf{cmd}_{\mathbf{pc}_1}$. By exhaustive inspection, the result might either

- a possibly empty sequence of values *val**
- a label **label**$_n\{$*instr*$^{*\prime}\}$ *instr** **end**
- *val** **loop** *instr** **end**.

val**:* by Corollary 1 we have that all instructions to the right of $\mathbf{cmd}_{\mathbf{pc}_1}$ have a program counter annotation. Since $\mathbf{cmd}_{\mathbf{pc}_1}$ only puts only values in the instruction sequence, $\mathbf{cmd}_{\mathbf{pc}_2}$ has to be to be the first non-constant instruction right of it. By the definition of $annotate(\cdot)$ we have that all constant instruction right of *val** are annotated with increasing program counters.

label$_n\{$*instr*$^{*\prime}\}$ *instr** **end***:* In this case, $\mathbf{cmd}_{\mathbf{pc}_2}$ is either in *instr** or it is the **end** of the result (if *instr** only consists of constants). In any case the possible prefix of constants will be correctly annotated by the same reasoning as above.

val* **loop** *instr** **end***:* This can only be the case if $\mathbf{cmd}_{\mathbf{pc}_1}$ was a branching instruction and the continuation non-empty (i.e. it contained a loop). In this case we have by always have $\mathbf{pc}' = \mathbf{pc}_2$. □

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 $\exists \Delta_2\,(\Delta_1 \vdash \Delta_2 \land \Delta_2 \geq \alpha(c_2))$.*

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 6) $\alpha(c_1)$ 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 $Trap_{\mathbf{fid}}$-facts. That these facts can be generated is proven in the respective lemmas for the different trapping instructions.

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 6, 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 35 and 38. □

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 $S; F; E[instr^*] \hookrightarrow\!\!\!\!\rightarrow c_2 = S'; F'; E[instr^{*\prime}]$: we have $\exists \Delta_2\,(\Delta_1 \vdash \Delta_1 \land \Delta_2 \geq \alpha(c_2))$ if we have $\alpha_T(S; F; E[instr^*]) \vdash \alpha_T\big(S'; F'; E[instr^{*\prime}]\big)$*

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 **frame**$_n\{F\}$ *val** **end**

We proceed by case distinction on these classes.

local instructions: In this case, by Lemma 6, $\alpha(c_1)$ and $\alpha(c_2)$ agree on everything but the last invocation of $\alpha_T\big(S'; F'; E[instr^{*\prime}]\big)$. This we can derive from $\alpha_T(S; F; E[instr^*])$ (which is a subset of Δ_1 by the assumption.

call *x or* **call_indirect***:* By Lemmas 6, 34 and 35.

return *or* **frame**$_n\{F\}$ *val** **end***:* By Lemmas 6 and 36. □

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.*

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

Proof. If c_1 is trapping, by Lemma 8. If c_2 is non-trapping by Lemma 9 and Lemmas 13 to 36. □

Corollary 2. *For all concrete configurations c_1 and c_2 and abstract configuration Δ_1 with $c_1 \overset{*}{\hookrightarrow} c_2$ and $\Delta_1 \geq \alpha(c_1)$ there exists an abstract configuration Δ_2 that can be derived from Δ_1.*

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

Proof. By applying Lemma 10 inductively. □

Lemma 11. *When an instruction causes a trap, the execution stops at the following configuration:*

$$c_2 = (S; F; E[\textsf{trap}])$$

The rule for deriving a trap from a generic instruction has the following form: We need to show that $\alpha(c_2) \leq \Delta_2$, using the following abstraction function:

$$\alpha_{trap}(F.\textsf{fid}, F.\textsf{stor}, F) = \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0)$$

The fact obtained through function α_{trap} complies with the Rule 9, thus the abstraction is sound.

Lemma 12. *Assume that the execution is inside a function: after the execution of the last instruction, the program must return to the point immediately after the function call and resume the execution.*

The return from a function is modelled by a rule of this form: Figure 10 The MState fact in the premise represents the state of the execution before the function call was executed.

The Return fact takes the following arguments:

- *rt, the state of the stack at the end of the function;*
- *rgt, the value of the global variables at the end of the function;*
- *$rmem$, the state of the linear memory at the end of the function;*
- *at_0, gt_0, mem_0, defined as usual.*

The ellipsis (...) in the premise represents other possible predicates that can occur in the rule.

The consequence of the rule expresses the fact that, after returning from a function, the execution resumes at the instruction immediately after the function call (found at $pc + 1$), with the following arguments:

- *the stack becomes $rt + st[\textsf{Sl.as}(\textbf{cid}) :]$, namely a concatenation between the return values computed inside the function, and the top part part of the stack (without the arguments with which the function was called).*
- *the globals are set to rgt, i.e., the value of the globals after the execution of the function;*
- *the locals remain unmodified, because they are out of the scope of the function;*
- *the memory is set to $rmem$, which means that any modification of the memory happened during the execution of the function is made effective after the return.*
- *all the other arguments (at_0, gt_0, mem_0) remain unmodified.*

When the execution returns from a function, the program counter is set to the instruction found immediately after the function call. Therefore, the final configuration is $c_2 = (S; F; E[\textsf{cmd}_{\mathbf{pc}+1}])$, that gets abstracted as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textsf{cmd}_{\mathbf{pc}+1}]) \\ &= \alpha_F(F.\textit{fid}, pc, S, F, \epsilon, instr^*) \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textsf{cmd}_{\mathbf{pc}+1}])), \alpha_{seq}(\eta_G(S, F)), \\ &\qquad \alpha_{seq}(F.locals), \alpha_M(i; S; F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(rt + st[\textsf{Sl.as}(\textbf{cid}) :], \\ &\qquad rgt, lt, rmem, at_0, gt_0, mem_0)\}\end{aligned}$$

The function $\alpha_{seq}(\eta_S(E[\textsf{cmd}_{\mathbf{pc}+1}]))$ returns the state of the stack after the execution of the function. According to the specification, described in [24], when the function returns, the computed values are put on the stack:

$$\textsf{frame}_n\{F\}val^n\ \textsf{end} \hookrightarrow val^n$$

The arguments used to call function have been consumed, therefore $\alpha_{seq}(\eta_S(E[\textsf{cmd}_{\mathbf{pc}+1}]))$ is going to return what expected.

The functions $\alpha_{seq}(\eta_G(S, F))$ and $\alpha_M(i; S; F)$ return respectively rgt and $rmem$, as they abstract the current values of the globals and the memory, that have possibly been modified by the function.

Since the abstraction of c_2 matches the consequence of 10, the condition $\alpha(c_2) \leq \Delta_2$ is satisfied, and therefore the return from a function is abstracted soundly.

Proof. We prove the soundness of constant instruction. The concrete configurations are defined as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textsf{t.const}\ n]) \\ c_2 &= (S; F; E[\textsf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

where $\textsf{cmd}_{\mathbf{pc}+1}$ represent whatever command is found immediately after pc. We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textsf{t.const}\ n])), \alpha_{seq}(\eta_G(S, F)), \\ &\qquad \alpha_{seq}(F.locals), \alpha_M(i; S; F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for numeric instructions is where $\mathbf{v}$ is an abstraction of $\textsf{t.const}\ n$.

The resulting abstract configuration differs from the previous one because of the new value added on top of the stack. The set of *MState* facts derived from the rule constitute Δ_2.

To finish the proof, we need to show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$, i.e., $\alpha(c_2)$ is at least as precise as Δ_2. Considering $c_2 = (S; F; E[\textsf{cmd}_{\mathbf{pc}+1}])$, we have that:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textsf{cmd}_{\mathbf{pc}+1}]) \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(val_1 :: st, gt, lt, mem, at_0, gt_0, mem_0) \\ &\qquad ... \\ &\qquad \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(val_m :: st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

where $\{val_1, ..., val_m\} = \alpha_V(\textsf{t.const}\ n)$.

$$\ldots \quad \Longrightarrow \mathit{Trap}_{\mathbf{fid}}(at_0, gt_0, mem_0) = \Delta_2$$

Fig. 9: Generic rule to derive a trap

$$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge \ldots \wedge\\ &\quad \mathit{Return}_{cid}(rt, rgt, rmem, at, gt, mem)\\ &\quad\Longrightarrow \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(rt +\!\!\!+\ st[\mathsf{Sl.as}(\mathbf{cid}):], rgt, lt,\\ &rmem, at_0, gt_0, mem_0)\}\end{aligned}$$

Fig. 10: Rule for constant instructions

$$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\\ &\quad\Longrightarrow \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\mathbf{v} :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \in \Delta_2\end{aligned}$$

Fig. 11: Rule for constant instructions

We can deduce that $\alpha(S; F; E[\mathsf{cmd}_{\mathbf{pc}+1}])$ produces exactly Δ_2, because after the execution of $\mathsf{t.const}\ n$, the following happens:

- the locals, the globals, and the memory remain unmodified from the previous configuration;
- a new value is found on top of the stack. The abstraction of the value produces a set of values, each of which appears in a different *MState* fact. For each of these facts, Rule 11 holds.
- the program counter increases according to the function *ann*, defined in Figure 2. The instruction $\mathsf{t.const}\ n$ falls into the second case of *ann′*, therefore gets assigned to program counter pc. The instruction following $\mathsf{t.const}\ n$ is annotated by recursively calling *ann′* on $pc + 1$ and the remaining sequence $\mathit{instr}^{*'}$: this means that, independently of what instruction follows $\mathsf{t.const}\ n$, it will be assigned to program counter $pc + 1$. This proves that the *MState* facts composing Δ_2 have program counter $pc+1$.

This proves the soundness of the abstraction for constant instructions. □

Lemma 13 (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 $\alpha(c_1) \vdash \alpha(c_2)$:*

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

Proof. Let us define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[(\mathsf{t.const}\ n)\, \mathsf{t.unop}_{\mathbf{pc}}])\\ c_2 &= (S; F; E[\mathsf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{t.const}\ n)\, \mathsf{t.unop}])),\\ &\alpha_{seq}(\eta_G(S, F)), \alpha_{seq}(F.locals), \alpha_M(i; S; F),\\ &at_0, gt_0, mem_0)\}\\ = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for numeric instructions is ❷.

In case of a unary operation, the resulting abstract configuration differs from the previous one for two reasons:

1) value x is consumed and a new value $unop(x)$ is pushed on the stack to replace it,
2) the program counter is increased by one.

To finish the proof, we need to show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$. Considering $c_2 = (S; F; E[\mathsf{cmd}_{\mathbf{pc}+1}])$, we have that:

$$\alpha(c_2) = \alpha(S; F; E[\mathsf{cmd}_{\mathbf{pc}+1}])$$

where $\mathsf{cmd}_{\mathbf{pc}+1}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. We can deduce that $\alpha(S; F; E[\mathsf{cmd}_{\mathbf{pc}+1}])$ produces exactly Δ_2:

- the locals, the globals, and the memory remain unmodified from the previous configuration;
- after the execution of $(\mathsf{t.const}\ n)\, \mathsf{t.unop}$, a new value is found on top of the stack, substituting the old one. The abstraction of the value produces a set of values, each of which appears in a different *MState* fact. For each of these facts, ❷ holds.
- the program counter increases according to the function *annotate()*, defined in Figure 2. The instruction $\mathsf{t.unop}$ falls into the second case of *ann′*, therefore gets assigned to program counter pc. The instruction following $\mathsf{t.unop}$ is annotated by recursively calling *ann′* on $pc + 1$ and the remaining sequence $\mathit{instr}^{*'}$: this means that, independently of what instruction follows $\mathsf{t.unop}$, it will be

assigned to program counter $pc+1$. This proves that the *MState* facts composing Δ_2 have program counter $pc+1$. □

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

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

Proof. We define the concrete configurations as follows:

$$c_1 = (S; F; E[(\mathsf{t.}\mathbf{const}\ n)(\mathsf{t.}\mathbf{const}\ m)\,\mathsf{t.}\mathbf{binop}]) \\ c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\alpha(c_1) = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\\ \alpha_{seq}(\eta_S(E[(\mathsf{t.}\mathbf{const}\ n)(\mathsf{t.}\mathbf{const}\ m)\,\mathsf{t.}\mathbf{binop}])), \\ \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ at_0, gt_0, memo_0)\} \\ = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(n :: m :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \\ \leq \Delta_1$$

The program counter is in front of t.**binop**, thus the constants have already been processed and put on the stack.

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for numeric instructions is ❸. This rule is applied to every *MState* fact in the set. In case of a binary operation, the resulting abstract configuration differs from the previous one for two reasons:

1) values n and m are consumed and a new value $binop(x,y)$ is pushed on the stack to replace it,
2) the program counter is increased by one.

To finish the proof, we need to show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$.Considering $c_2 = (S; F; E[instr^*])$, we have that:

$$\alpha(c_2) = \alpha(S; F; E[instr^*])$$

where $\mathbf{cmd_{pc+1}}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. We can deduce that $\alpha(S; F; E[instr^*])$ produces exactly Δ_2:

- the locals, the globals, and the memory remain unmodified from the previous configuration;
- after the execution of $(\mathsf{t.}\mathbf{const}\ n)(\mathsf{t.}\mathbf{const}\ m)\,\mathsf{t.}\mathbf{binop}$, the stack size is decreased by one and a new value is found on top. The abstraction of the operands produces a set of values, each of which appears in a different *MState* fact. For each of these facts, Rule ❸ holds.
- the program counter increases according to the function *annotate()*, defined in Figure 2. The instruction t.**binop** falls into the second case of *ann′*, therefore gets assigned to program counter pc. The instruction following t.**binop** is annotated by recursively calling *ann′* on $pc+1$ and the remaining sequence $instr^{*\prime}$: this means that, independently of what instruction follows t.**binop**, it will be assigned to program counter $pc+1$. This proves that the *MState* facts composing Δ_2 have program counter $pc+1$.

In light of these observations, we can compute $\alpha(c_2)$ as follows:

$$\alpha(c_2) = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(res_1 :: st, gt, lt, mem, \\ at_0, gt_0, memo_0)\} \\ \dots \\ = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(res_p :: st, gt, lt, mem, \\ at_0, gt_0, memo_0)\}$$

where $\{res_1, ..., res_p\} = x$ binop symbol y

Some binary instructions can cause a a trap. For example, t.**div** and t.**mod** are undefined for certain values, as the following cases show:

$$(\mathsf{t.}\mathbf{const}\ n)(\mathsf{t.}\mathbf{const}\ 0)\,\mathsf{t.}\mathbf{div} \\ (\mathsf{t.}\mathbf{const}\ n)(\mathsf{t.}\mathbf{const}\ 0)\,\mathsf{t.}\mathbf{mod}$$

Just like unary operations, also binary operations $\circledast_{\mathbf{binop}}$ have a defined domain:

$$\circledast_{\mathbf{binop}} : X \subseteq \mathbb{B}_{64} \times Y \subseteq \mathbb{B}_{64} \to \mathcal{P}(\mathbb{B}_{64})$$

That is, $\circledast_{\mathbf{binop}}$ takes in input two $\mathbb{B}_{64}$ vectors and returns a set of $\mathbb{B}_{64}$ values. If the result falls outside the output set, a trap occurs.

The rule that models a trapping binary operation are ❹ and ❺. By Lemma 11, the abstraction is sound. □

Lemma 15 (Single-Step Soundness for $\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}$). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t_1.}\mathbf{const}\ n)\,\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S; F; E[(\mathsf{t_1.}\mathbf{const}\ n)\,\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}]) \wedge \\ c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S; F; E[(\mathsf{t_1.}\mathbf{const}\ n)\,\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}]) \\ c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\alpha(c_1) = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \\ = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{t_1.}\mathbf{const}\ n)\,\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}])), \\ \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ at_0, gt_0, memo_0)\} \leq \Delta_1$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for a $\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}$ instruction is ❻. For a $\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}$ instruction, the resulting abstract configuration differs from the previous one for the following reasons:

1) the value n is popped from the stack;
2) value $\mathsf{t_2.}\mathbf{cvtOp\_t_1\_sx}(n)$ is pushed on the stack;
3) the program counter is increased by one.

To finish the proof, we need to show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$.Considering $c_2 = (S; F; E[instr^*])$, we have that:

$$\alpha(c_2) = \alpha(S; F; E[instr^*])$$

where $\textbf{cmd}_{\mathbf{pc}+1}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. We can deduce that $\alpha(S;F;E[instr^*])$ produces exactly Δ_2:

- the locals, the globals, and the memory remain unmodified from the previous configuration;
- after the execution of $t_2.\textbf{cvtOp\_t}_1\textbf{\_sx}$, the value found on top of the stack is popped;
- the program counter increases according to the function *annotate()*, defined in Figure 2. The instruction $t_2.\textbf{cvtOp\_t}_1\textbf{\_sx}$ falls into the second case of ann', therefore gets assigned to program counter pc. The instruction following $t_2.\textbf{cvtOp\_t}_1\textbf{\_sx}$ is annotated by recursively calling ann' on $pc{+}1$ and the remaining sequence $instr^{*'}$: this means that, independently of what instruction follows $t_2.\textbf{cvtOp\_t}_1\textbf{\_sx}$, it will be assigned to program counter $pc+1$. This proves that the *MState* facts composing Δ_2 have program counter $pc+1$.

Based on these consideration, the computation of $\alpha(c_2)$ will be:

$$\alpha(c_2) = \{MState_{\mathbf{fid},\mathbf{pc}+1}(cvtOp_{\mathbf{op}}(n) :: st, gt, lt, mem, \\ at_0, gt_0, mem_0)\}$$

which complies exactly with the consequence of ❻.

Some conversions can cause a trap, like the trunc operation. These situations are handled by rules ❼ and ❽. By Lemma 11, the abstraction is sound. □

Lemma 16 (Single-Step Soundness for **drop**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(t.\textbf{const}\ n)\ \textbf{drop}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[(t.\textbf{const}\ n)\ \textbf{drop}]) \land \\ c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S;F;E[(t.\textbf{const}\ n)\ \textbf{drop}]) \\ c_2 = (S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$$

We obtain the abstraction of c_1 by applying the abstraction functions to the concrete configuration:

$$\alpha(c_1) = \{MState_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[(t.\textbf{const}\ n)\ \textbf{drop}])), \\ \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ at_0, gt_0, mem_0)\} \\ = \{MState_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for a **drop** instruction is For a **drop** instruction, the resulting abstract configuration differs from the previous one for two reasons:

1) values n is popped from the stack,
2) the program counter is increased by one.

To finish the proof, we need to show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$.Considering $c_2 = (S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$, we have that:

$$\alpha(c_2) = \alpha(S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$$

where $\textbf{cmd}_{\mathbf{pc}+1}$ represents any set of instructions following the execution of the unary operation. We can deduce that $\alpha(S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$ produces exactly Δ_2:

- the locals, the globals, and the memory remain unmodified from the previous configuration;
- after the execution of **drop**, the value found on top of the stack is popped;
- the program counter increases according to the function *ann*, defined in Figure 2. The instruction **drop** falls into the second case of ann', therefore gets assigned to program counter pc. The instruction following **drop** is annotated by recursively calling ann' on $pc{+}1$ and the remaining sequence $instr^{*'}$: this means that, independently of what instruction follows **drop**, it will be assigned to program counter $pc{+}1$. This proves that the *MState* facts composing Δ_2 have program counter $pc+1$.

□

Based on these observations, the computation of $\alpha(c_2)$ will return the following set:

$$\alpha(c_2) = \{MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)\}$$

which complies exactly with the consequence of ❻. Therefore, the abstraction is sound.

Lemma 17 (Single-Step Soundness for **select**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(t.\textbf{const}\ n)(t.\textbf{const}\ m)(t.\textbf{const}\ p)\ \textbf{select}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[(t.\textbf{const}\ n)(t.\textbf{const}\ m)(t.\textbf{const}\ p)\ \textbf{select}]) \land \\ c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S;F;E[(t.\textbf{const}\ n)(t.\textbf{const}\ m)(t.\textbf{const}\ p)\ \textbf{select}]) \\ c_2 = (S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\alpha(c_1) = \{MState_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(\\ E[(t.\textbf{const}\ n)(t.\textbf{const}\ m)(t.\textbf{const}\ p)\textbf{select}])), \\ \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, \\ gt_0, mem_0)\} \\ = \{MState_{\mathbf{fid},\mathbf{pc}}(n :: m :: p :: st, gt, lt, mem, at_0, gt_0, \\ mem_0)\} \leq \Delta_1$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. There are two rules for a **select** instruction, ❿ and ⓫. Let us consider Rule ❿: since $n = 0$, the second parameter, m, is selected and kept on the stack.

Now we show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \geq \alpha(c_2)$.Considering $c_2 = (S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$, we have that:

$$\alpha(c_2) = \alpha(S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$$

where $\textbf{cmd}_{\mathbf{pc}+1}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. We can deduce that $\alpha(S;F;E[\textbf{cmd}_{\mathbf{pc}+1}])$ produces exactly Δ_2:

$$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0)\\ &\quad\Longrightarrow \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)\end{aligned}$$

Fig. 12: Rule for **drop**

1) the locals, the globals, and the memory remain unmodified from the previous configuration;
2) after the execution of **select**, the previous values are popped from the stack, and m is pushed again.
3) the program counter increases according to the function *ann*, defined in Figure 2. The instruction **select** falls into the second case of *ann′*, therefore gets assigned to program counter pc. The instruction following **select** is annotated by recursively calling *ann′* on $pc + 1$ and the remaining sequence $instr^{*'}$: this means that, independently of what instruction follows select, it will be assigned to program counter $pc + 1$. This proves that the *MState* facts composing Δ_2 have program counter $pc + 1$.

Based on these observations, the computation of $\alpha(c_2)$ will return the following set:

$$\alpha(c_2) = \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(y :: st, gt, lt, mem, at_0, gt_0, mem_0)\}$$

which complies exactly with the consequence of 10. Therefore, the abstraction is sound.

The second case in analogous, except for point 2., as both the top two values are popped. □

Lemma 18 (Single-Step Soundness for **local.get idx**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{local.get idx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}\forall c_1, c_2\, \big(&c_1 = (S; F; E[\textbf{local.get idx}]) \land\\ &c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textbf{local.get}\, idx])\\ c_2 &= (S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{local.get}\, idx])),\\ &\alpha_{seq}(\eta_G(S, F)), \alpha_{seq}(F.locals), \alpha_M(i; S; F),\\ &at_0, gt_0, mem_0)\}\\ = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \le \Delta_1\end{aligned}$$

The rule to derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$, for a **local.get** instruction is 12. Now we compute the abstraction of c_2: suppose that $lt[\textbf{idx}] = val$, and that val gets abstracted to $\alpha_V(val) = \{val_1, ..., val_n\}$; then $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\mathbf{pc}+1}])), \alpha_{seq}(\eta_G(S, F)),\\ &\qquad \alpha_{seq}(F.locals), (\eta_G(S, F)), \alpha_{seq}(F.locals),\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(val_1 :: st, gt, lt, mem, at_0, gt_0, mem_0),\\ &\qquad ...\\ &\qquad \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(val_n :: st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

Since every value val_i in the set of *MState* facts represents an abstraction of $lt[\textbf{idx}]$, and the set of *MState* facts matches the consequence of 12, we can confirm that the abstraction is sound. □

Lemma 19 (Single-Step Soundness for **local.set idx**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.set idx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}\forall c_1, c_2\, \big(&c_1 = (S; F; E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.set idx}]) \land\\ &c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.set idx}])\\ c_2 &= (S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.set idx}])),\\ &\alpha_{seq}(F.locals), \alpha_{seq}(\eta_G(S, F)), \alpha_M(i; S; F), at_0,\\ &gt_0, mem_0)\}\\ = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \le \Delta_1\end{aligned}$$

The rule to derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$, for a **local.set** instruction is 13. Now we compute the abstraction for c_2: the value n to be assigned to $lt[\textbf{idx}]$ gets abstracted with $\alpha_V(n) = \{val_1, ..., val_m\}$. Therefore, $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\mathbf{pc}+1}])), \alpha_{seq}(F.locals),\\ &\qquad \alpha_{seq}(\eta_G(S, F)), \alpha_M(i; S; F), at_0, gt_0, mem_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt[\textbf{idx} \leftarrow val_1], mem,\\ &\qquad at_0, gt_0, mem_0),\\ &\qquad ...\\ &\quad \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt[\textbf{idx} \leftarrow val_m], mem,\\ &\qquad at_0, gt_0, mem_0)\}\end{aligned}$$

Based on these observations, the computation of $\alpha(c_2)$ will return the following set:

$$\begin{aligned}\alpha(c_2) = \{&\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt[\textbf{idx} \leftarrow val_i], mem, at_0, gt_0,\\ &mem_0)\}\end{aligned}$$

where $val_i \in \alpha_V(n)$. Since the set of *MState* facts complies with the consequence of 13, we can say that the abstraction is sound. □

Lemma 20 (Single-Step Soundness for **local.tee idx**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.tee idx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}\forall c_1, c_2\, \big(&c_1 = (S; F; E[(\mathsf{t}.\textbf{const}\ n)\,\textbf{local.tee idx}]) \land\\ &c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S; F; E[(\mathsf{t.const}\ n)\ \mathbf{local.tee\ idx}])$$
$$c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\mathbf{fid,pc}}(n :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{t.const}\ n)\ \mathbf{local.tee\ idx}])), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ &\quad at_0, gt_0, mem_0)\} \le \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The rule for **local.tee** is 14.

Now we abstract c_2: suppose that $lt[\mathbf{idx}] = val$, and that val gets abstracted to $\alpha_V(val) = \{val_1, ..., val_m\}$; then $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\mathbf{cmd_{pc+1}}]) \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[\mathbf{cmd_{pc+1}}])), \alpha_{seq}(F.locals), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc+1}}(n :: st, gt, lt[\mathbf{idx} \leftarrow val_1], mem, \\ &\quad at_0, gt_0, mem_0), \\ &\quad ... \\ &\quad \mathit{MState}_{\mathbf{fid,pc+1}}(n :: st, gt, lt[\mathbf{idx} \leftarrow val_m], mem, \\ &\quad at_0, gt_0, mem_0)\}\end{aligned}$$

Since every value val_i is an abstraction of n, we can say that $\alpha(c_2)$ complies with Rule 14, thus the overall abstraction is sound. $\square$

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

$$\forall c_1, c_2\, \big(c_1 = (S; F; E[\mathbf{global.get\ idx}]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S; F; E[\mathbf{global.get\ idx}])$$
$$c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\mathbf{fid,pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[\mathbf{global.get\ idx}])), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ &\quad at_0, gt_0, mem_0)\} \le \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The rule for **global.get** is 15. Now we show that by abstracting c_2 we obtain a set of facts $\alpha(c_2)$ that satisfies $\Delta_2 \ge \alpha(c_2)$. Assume that $gt[\mathbf{idx}] = val$, and that val gets abstracted to $\alpha_G(\eta_G(S,F)) = \{val_1, ..., val_m\}$; then $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\mathbf{cmd_{pc+1}}]) \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[\mathbf{cmd_{pc+1}}])), \alpha_{seq}(F.locals), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc+1}}(val_1 :: st, gt, lt, mem, at_0, gt_0, mem_0), \\ &\quad ... \\ &\quad \mathit{MState}_{\mathbf{fid,pc+1}}(val_m :: st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

where $\mathbf{cmd_{pc+1}}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. Now we abstract c_2: suppose that $lt[\mathbf{idx}] = val$, and that val gets abstracted to $\alpha_V(val) = \{val_1, ..., val_m\}$; then $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\mathbf{cmd_{pc+1}}]) \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[\mathbf{cmd_{pc+1}}])), \alpha_{seq}(F.locals), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc+1}}(val_1 :: st, gt, lt, mem, at_0, gt_0, mem_0), \\ &\quad ... \\ &\quad \mathit{MState}_{\mathbf{fid,pc+1}}(val_m :: st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

Since every value val_i is an abstraction of n, we can say that $\alpha(c_2)$ complies with Rule 15, thus the overall abstraction is sound. $\square$

Lemma 22 (Single-Step Soundness for **global.set idx**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{t.const}\ n)\ \mathbf{global.set\ idx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

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

Proof. We define the concrete configurations as follows:

$$c_1 = (S; F; E[(\mathsf{t.const}\ n)\ \mathbf{global.set\ idx}])$$
$$c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{t.const}\ n)\ \mathbf{global.set\ idx}])), \\ &\quad \alpha_{seq}(F.locals), \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, \\ &\quad mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(n :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \le \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The rule for **global.set** is 16. Now we compute the abstraction of c_2 to show that we obtain a set of facts $\alpha(c_2)$ satisfying $\Delta_2 \ge \alpha(c_2)$. Consider $c_2 = (S; F; E[\mathbf{cmd_{pc+1}}])$, and assume that the value n gets abstracted to $\{val_1, ..., val_m\} = \alpha_V(\mathsf{t.const}\ n)$. The abstraction $\alpha(c_2)$ will then be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\mathbf{cmd_{pc+1}}]) \\ &= \{\mathit{MState}_{\mathbf{fid,pc}}(\alpha_{seq}(\eta_S(E[\mathbf{cmd_{pc+1}}])), \alpha_{seq}(F.locals), \\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid,pc+1}}(st, gt[\mathbf{idx} \leftarrow val_1], lt, mem, at_0, gt_0, \\ &\quad mem_0) \\ &\quad ... \\ &\quad \mathit{MState}_{\mathbf{fid,pc+1}}(st, gt[\mathbf{idx} \leftarrow val_m], lt, mem, at_0, gt_0, \\ &\quad mem_0)\} \le \Delta_1\end{aligned}$$

where $\mathbf{cmd_{pc+1}}$ represents any set of (correctly factorized) instructions following the execution of the unary operation. Now we abstract c_2: suppose that $lt[\mathbf{idx}] = val$, and that val

gets abstracted to $\alpha_V(val) = \{val_1, ..., val_m\}$; then $\alpha(c_2)$ will be computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textbf{cmd}_{\textbf{pc}+1}])\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\textbf{pc}+1}])), \alpha_{seq}(F.locals),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(n :: st, gt, lt[\textbf{idx} \leftarrow val_1], mem,\\ &\quad at_0, gt_0, memo_0),\\ &\quad \dots\\ &\quad \mathit{MState}_{\textbf{fid},\textbf{pc}+1}(n :: st, gt, lt[\textbf{idx} \leftarrow val_m], mem,\\ &\quad at_0, gt_0, memo_0)\}\end{aligned}$$

Since every value val_i is an abstraction of n, we can say that $\alpha(c_2)$ complies with Rule 16, thus the overall abstraction is sound. □

Lemma 23 (Single-Step Soundness for t.**loadN_sx**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(\text{t.}\textbf{const}\ n)\ \text{t.}\textbf{loadN\_sx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[(\text{t.}\textbf{const}\ n)\ \text{t.}\textbf{loadN\_sx}]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S;F;E[(\text{i32.}\textbf{const}\ n)\ \text{t.}\textbf{loadN\_sx}\ \mathit{memarg}])\\ c_2 &= (S;F;E[\textbf{cmd}_{\textbf{pc}+1}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(\alpha_{seq}(\eta_S(E[(\text{i32.}\textbf{const}\ n)\ \text{t.}\textbf{loadN\_sx}])),\\ &\quad \alpha_{seq}(F.locals), \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0,\\ &\quad memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \leq \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The rule for t.**loadN_sx** is 17. The premise of the rule expresses the constraints the instruction needs to comply with in order to execute successfully.

After the execution of the instruction, we should find of the stack w, which is an abstraction of the value found in the memory. To see whether the abstraction complies with Rule 17, we compute $\alpha(c_2)$ as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textbf{cmd}_{\textbf{pc}+1}])\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\textbf{pc}+1}])), \alpha_{seq}(F.locals),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(w :: st, gt, lt, mem, at_0, gt_0, memo_0)\}\end{aligned}$$

where $\textbf{cmd}_{\textbf{pc}+1}$ represents any set of (correctly factorized) instructions following the execution of the unary operation.

Since $\alpha(c_2)$ matches exactly the consequence of 17 ($\alpha(c_2) = \Delta_2$), we can say that the abstraction is sound.

In case of a loading instruction, a trap can occur if the memory bounds are not complied with: the rule for a *Trap* caused by a load instruction is 18. The premise of the rule is that the memory bound exceed the maximum size, therefore causing an immediate interruption of the execution. By Lemma 11, the abstraction is sound. □

Lemma 24 (Single-Step Soundness for t.**storeN_sx**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(\text{i32.}\textbf{const}\ a)(\text{t.}\textbf{const}\ n)\ \text{t.}\textbf{storeN\_sx}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[(\text{i32.}\textbf{const}\ a)(\text{t.}\textbf{const}\ n)\ \text{t.}\textbf{storeN\_sx}]) \wedge c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S;F;E[(\text{i32.}\textbf{const}\ a)(\text{t.}\textbf{const}\ n)(\text{t.}\textbf{storeN\_sx}\ \mathit{memarg})])\\ c_2 &= (S;F;E[\textbf{cmd}_{\textbf{pc}+1}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(n :: a :: st, gt, lt, mem, at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(\alpha_{seq}(\eta_S(\\ &\quad E[(\text{i32.}\textbf{const}\ a)(\text{t.}\textbf{const}\ n)(\text{t.}\textbf{storeN\_sx}\ \mathit{memarg})])),\\ &\quad \alpha_{seq}(F.locals), \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F),\\ &\quad at_0, gt_0, memo_0)\}\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The rule for t.**store** is 19. The premise of the rule expresses the constraints the instruction needs to comply with in order to execute successfully.

After the execution of the instruction, both the address a and the constant n are popped from the stack, and the value nv (which results from an appropriate processing of n) is stored in the linear memory. To see whether the abstraction complies with Rule 19, we compute $\alpha(c_2)$ as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textbf{cmd}_{\textbf{pc}+1}])\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\textbf{pc}+1}])), \alpha_{seq}(F.locals),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_M(i;S;F), at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(st, gt, lt, \mathit{Mem}(i, nv, size), at_0, gt_0,\\ &\quad memo_0)\}\end{aligned}$$

where $\textbf{cmd}_{\textbf{pc}+1}$ represents any set of (correctly factorized) instructions following the execution of the unary operation.

Since $\alpha(c_2)$ matches exactly the consequence of 19 ($\alpha(c_2) = \Delta_2$), we can say that the abstraction is sound.

A generic t.**storeN_sx** instruction can cause a trap, if the memory bounds are not complied with. The rule that models this scenario is 18. The premise of the rule is that the memory bound exceed the maximum size, therefore causing an immediate interruption of the execution. By Lemma 11, the abstraction is sound. □

Lemma 25 (Single-Step Soundness for **unreachable**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[\textbf{unreachable}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

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

Proof. Let's prove that using abstraction function α_S is sound. Let's define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S;F;E[\textbf{unreachable}])\\ c_2 &= (S;F;E[\textbf{trap}])\end{aligned}$$

Configuration c_1 gets abstracted as follows:

$$\begin{aligned}\alpha(c_1) &= \mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{unreachable}])),\\ &\quad \alpha_{seq}(\alpha_V(S.globals)), \alpha_{seq}(\alpha_V(F.vals)),\\ &\quad \alpha_M(i; S; F), at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

The rule for the instruction **unreachable** is 21. In order to prove the soundness, we show that the abstraction obtain by applying α to c_2 complies with the rule, but it is sufficient to invoke Lemma 11 to finish the proof. □

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

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

Proof. Let's prove that using abstraction function α_S is sound. Let's define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textbf{nop}])\\ c_2 &= (S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

The first part of the proof concerns obtaining Δ_1 as a sound abstraction for c_1 (i.e., $\Delta_1 \geq \alpha(c_1)$): we achieve this by applying the abstraction functions for local variables, global variables, memory, and stack:

$$\begin{aligned}\alpha(c_1) &= \mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(\textbf{nop})), \alpha_{seq}(\alpha_V(S.globals)),\\ &\quad \alpha_{seq}(\alpha_V(F.vals)), \alpha_M(i; S; F), at_0, gt_0, mem_0)\}\\ &\leq \Delta_1\end{aligned}$$

We now derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for **nop** is 22. The resulting abstract configuration is identical to the previous one, except for the program counter, which is increased by one. This models the fact that the execution of a **nop** instruction doesn't affect any part of the program or the memory, and afterwards the execution proceeds sequentially to the next instruction.

To finish the proof, we need to show that $\Delta_2 \geq \alpha(c_2)$, that is, we show that the abstraction derived from Δ_1 by means of Rule 22 is a sound abstraction for $\alpha(c_2)$, obtained by applying the abstraction functions to c_2.

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

Since the **nop** instruction does not affect any variable or memory cell, the new set of *MState* predicates are going to look exactly like the old one, except for the program counter, which is increased according to the annotation function.

Since $\alpha(c_2)$ is equal to the consequence of 22, we can say that our abstraction is sound. □

Lemma 27 (Single-Step Soundness for **block**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{block}]$ and $c_1 \hookrightarrow\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}\forall c_1, c_2\, \big(c_1 = (S; F; E[\textbf{block}]) \wedge\\ c_1 \hookrightarrow\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textbf{block}\, \mathit{instr}^*\, \textbf{end}])\\ c_2 &= (S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\end{aligned}$$

where *instr** represents an arbitrary sequence of instructions. The first instruction of *instr** has program counter $pc + 1$:

$$\mathit{instr}^* = \textbf{cmd}_{\mathbf{pc}+1}; \mathit{instr}^*_{cont}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S; F; E[\textbf{block}\, \mathit{instr}^*\, \textbf{end}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{block}\, \mathit{instr}^*\, \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S, F)), \alpha_{seq}(F.locals), \alpha_M(i; S; F),\\ &\quad at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

The execution of **block** does not modify the variables, the memory, or the stack. For this reason, the set of *MState* facts will look exactly like the previous one, except for the program counter, that must be increased by 1.

Now we need to show that we can derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$: the rule for a **block-end**-statement is 23.

After **block**, the execution flows to the first instruction of $\textbf{cmd}_{\mathbf{pc}+1}$; therefore, the abstraction of c_2 will be:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textbf{cmd}_{\mathbf{pc}+1}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\mathbf{pc}+1}])), \alpha_{seq}(\eta_G(S, F)),\\ &\quad \alpha_{seq}(F.locals), \alpha_M(i; S; F), at_0, gt_0, mem_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

where *instr** denotes an arbitrary sequence of instructions.

The premise of the rule matches $\alpha(c_1)$, and the consequence Δ_2 matches $\alpha(c_2)$. This proves the soundness of our abstraction, because we can derive $\alpha(c_2)$ from $\alpha(c_1)$. □

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

$$\begin{aligned}\forall c_1, c_2\, \big(c_1 = (S; F; E[\textbf{loop}\, \mathit{instr}^*\, \textbf{end}]) \wedge\\ c_1 \hookrightarrow\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textbf{loop}\ t^n\ \mathit{instr}^*\ \textbf{end}])\\ c_2 &= (S; F; E[\textbf{label}_0\{\textbf{loop}\ t^n\ \mathit{instr}^*\ \textbf{end}\}\ \mathit{instr}^*\ \textbf{end}])\end{aligned}$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S; F; E[\textbf{loop}\ t^n\ \mathit{instr}^*\ \textbf{end}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{loop}\ t^n\ \mathit{instr}^*\ \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S, F)), \alpha_{seq}(F.locals), \alpha_M(i; S; F),\\ &\quad at_0, gt_0, mem_0)\} \leq \Delta_1\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

The rule that models the semantics of the loop is 24.

We now prove that our abstraction of c_2 complies with the consequence of the rule:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textbf{label}_0\{\textbf{loop}\ t^n\ \textit{instr}^*\ \textbf{end}\}\ \textit{instr}^*\ \textbf{end}])\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\alpha_{seq}(\\ &\quad \eta_S(E[\textbf{label}_0\{\textbf{loop}\ t^n\ \textit{instr}^*\ \textbf{end}\}\ \textit{instr}^*\ \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F),\\ &\quad at_0, gt_0, mem_0)\}\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

The **loop** instruction, serving only control flow purposes, does not modify the variables, the memory, or the stack, but has the only effect of carrying the execution inside the body of the loop. Moreover, it is worth mentioning that label instructions do not have a program counter, as they are administrative instructions.

Therefore, in the final configuration, the following things happen:

- The program counter is increased by 1, pointing at the first instruction of *instr**;
- All the arguments of *MState* remain unchanged.

This agrees with what is stated by Rule 24, therefore the abstraction is sound. □

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

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[\textbf{end}]) \land \\ c_1 \hookrightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S;F;E[\textbf{end}])$$
$$c_2 = (S;F;E[\textbf{cmd}_{\mathbf{next}}])$$

where $next$ represents the value of the program counter after executing **end**.

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S;F;E[\textbf{end}])\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{end}])), \alpha_{seq}(\eta_G(S,F)),\\ &\quad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

Let us say that $\alpha(c_1) = \Delta_1$, where Δ_1 represents a generic abstraction that is at most as precise as $\alpha(c_1)$; the rule to derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$ is 25. The execution of **end** does not affect any argument of the *MState*: its only purpose is to mark the end of a structure such as **block** or **loop** and redirect the program counter to the right location. In particular, the new value of the pc, namely $next$, is computed as shown in Fig. 13: let $\textbf{end}_{pc'}$ be the instruction that closes the block environment of the "then" branch, and let $\textbf{end}_{pc}$ be the instruction closing the **if-else-end** statement; then the new program counter is computed as shown in Equation 13 .

The first case of the function refers to the following scenario:

```
(i32.const n)
(if
    (then instr*_then )
    (else instr*_else )
end_pc)
cmd_pc+1

```

Listing 1: Example

Suppose $n \neq 0$: in this case, the branch then is taken and the sequence $\textit{instr}^*_{\text{then}}$ is executed. At the moment of the execution, these instructions are enclosed by a block, as shown below:

$$\textbf{label}_n\{\epsilon\}\ \textit{instr}^*_{\text{then}}\ \textbf{end}_{pc'}$$

If, after the execution of $\textbf{end}_{\mathbf{pc'}}$, we incremented the program counter by 1, we would end up in the **else** branch: this behaviour does not reflect the semantics of an **if-else** construct.

In example 1, instead, the function *Next* states that, after executing $\textbf{end}_{\mathbf{pc'}}$, the program counter gets assigned to $\mathbf{pc}+1$, i.e. to the value of the program counter of the instruction found immediately after the **if-else** construct. In all the other cases in which we find an **end** instruction, we simply increment the program counter by 1.

In light of these considerations, $\alpha(c_2)$ is computed as shown below:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textbf{cmd}_{\mathbf{next}}])\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{next}}(\alpha_{seq}(\eta_S(E[\textbf{cmd}_{\mathbf{next}}])), \alpha_{seq}(\eta_G(S,F)),\\ &\quad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, mem_0)\}\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{next}}(st, gt, lt, mem, at_0, gt_0, mem_0)\}\end{aligned}$$

The configuration obtained matches the consequence of Rule 26:

- the arguments of the *MState* fact remain unchanged, as the execution of the jump does not modify the stack, the globals, the locals, or the memory;
- the program counter is computed with the function *Next*.

Based on this, we can say that our abstraction is sound. □

Lemma 30 (Single-Step Soundness for **br** ℓ). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[\textbf{br}\ \ell]$ and $c_1 \hookrightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2\ \big(c_1 = (S;F;E[\textbf{br}\ \ell]) \land \\ c_1 \hookrightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)$$

Proof. We define the concrete configurations as follows:

$$c_1 = (S;F;E[\textbf{br}\,\ell])$$
$$c_2 = (S;F;E[\textbf{cmd}_{\mathbf{br}}])$$

where $\mathbf{br}$ represents the value of the program counter after the execution of the jump. We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S;F;E[\textbf{br}\,\ell])\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{br}\,\ell])), \alpha_{seq}(\eta_G(S,F)),\\ &\quad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, mem_0)\}\\ &= \{\textit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

$$Next(\textbf{end}_{pc'}) = \begin{cases} pc+1 & \text{if } \textbf{end}_{pc'} \text{ closes a } \mathsf{then} \text{ branch in conditional statement} \\ & \text{and } pc \text{ is the program counter at the end of } \mathsf{if\text{-}else\text{-}end} \\ pc'+1 & \text{otherwise} \end{cases}$$

Fig. 13: Definition of *Next*.

Let us say that $\alpha(c_1) = \Delta_1$, where Δ_1 represents a generic abstraction that is at most as precise as $\alpha(c_1)$; the rule to derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$ is 26. The function $unwind_n$ is used to put the right amount of arguments on top of the stack after a branch. Suppose we have the following situation:

$$val^m \ \textbf{label}_n val^k \ \textbf{br} \ \ell \ \textbf{end}$$

When the jump is executed, the function $unwind_n$ puts n values on top of the stack. If $k > n$, only the first n values are considered, because the arity of the label is n.

After the execution of **br** ℓ, the program performs a jump: this means that the execution does not proceed sequentially (thus pc will not be simply increased by 1), but instead the program counter gets assigned to the first instruction found after the ℓ-th block enclosing **br** ℓ. This address is be computed with the function *Dest* (Appendix C). Let $br = Dest(\widetilde{B}^0[\textbf{br}\,\ell])$; then $\alpha(c_2)$ is computed as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\mathsf{cmd}_{\mathbf{br}}]) \\ &= \{MState_{\mathbf{fid},\mathbf{br}}(\alpha_{seq}(\eta_S(E[\mathsf{cmd}_{\mathbf{br}}])), \alpha_{seq}(\eta_G(S,F)), \\ &\qquad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, memo_0)\} \\ &= \{MState_{\mathbf{fid},\mathbf{br}}(st, gt, lt, mem, at_0, gt_0, memo_0)\}\end{aligned}$$

The configuration obtained matches the consequence of Rule 26:

- the arguments of the *MState* fact remain unchanged, as the execution of the jump does not modify the stack, the globals, the locals, or the memory;
- the program counter is computed with *Dest*, that we proved to compute the address of the jump correctly.

In light of these considerations, we can confirm that our abstraction is sound. □

Lemma 31 (Single-Step Soundness for **br_if**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(\mathsf{t}.\textbf{const}\ \ n)\,\textbf{br\_if}\ \ell]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S;F;E[(\mathsf{t}.\textbf{const}\ \ n)\,\textbf{br\_if}\ \ell]) \wedge \right. \\ \left. c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. We define the initial configuration as follows:

$$c_1 = (S;F;E[(\mathsf{t}.\textbf{const}\ \ n)\,\textbf{br\_if}\ \ell])$$

The final configuration depends on the evaluation of the branching condition, and will thus be defined case by case. First, we obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S;F;E[(\mathsf{i32}.\textbf{const}\ \ n)\,\textbf{br\_if}\ \ell]) \\ &= \{MState_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \\ &= \{MState_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[(\mathsf{i32}.\textbf{const}\ \ n)\,\textbf{br\_if}\ \ell])), \\ &\qquad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), \\ &\qquad at_0, gt_0, memo_0)\} \leq \Delta_1\end{aligned}$$

To execute **br_if** ℓ, the value of the constant n on top of the stack must be evaluated: if $n \neq 0$, the jump is performed, otherwise the execution carries on to the instruction found at $pc+1$. The rules that express this behaviour are 27 and 28.

The program counter in the consequence of Rule 27 is computed with the same function *Dest* that we used in the proof for **br** ℓ, in Lemma 30.

a) **br_if** ℓ *in case of true condition:* If $n \neq 0$, **br_if** behaves as a *br* instruction. As a matter of fact, the semantics of **br_if** is defined in [24] as follows:

$$\begin{aligned}&(\mathsf{i32}.\textbf{const}\ \ c)(\textbf{br\_if}\,\ell) \hookrightarrow (\textbf{br}\,\ell) && \text{if } c \neq 0 \\ &(\mathsf{i32}.\textbf{const}\ \ c)(\textbf{br\_if}\,\ell) \hookrightarrow \epsilon && \text{if } c = 0\end{aligned}$$

The soundness of **br** has already been proven in A, therefore the soundness of **br_if** in this scenario comes as a consequence.

b) **br_if** ℓ *in case of false condition:* Now, let us consider the second scenario, in which the condition is not satisfied: in this case, since the condition for the jump is not satisfied, the execution continues sequentially, thus the final configuration will be:

$$c_2 = (S;F;E[\mathsf{cmd}_{\mathbf{pc+1}}])$$

We must show that the abstraction of c_2 complies with the consequence of Rule 28:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\mathsf{cmd}_{\mathbf{pc+1}}]) \\ &= \{MState_{\mathbf{fid},\mathbf{pc}+1}(\alpha_{seq}(\eta_S(E[\mathsf{cmd}_{\mathbf{pc+1}}])), \alpha_{seq}(\eta_G(S,F)), \\ &\qquad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, memo_0)\} \\ &= \{MState_{fid,pc+1}(st, gt, lt, mem, at_0, gt_0, memo_0\}\end{aligned}$$

Constant n is popped from the stack and the program counter is increased by one. The rest of the arguments in *MState* remain untouched, so the result obtained through the abstraction is compliant to the consequence of Rule 28. □

Lemma 32 (Single-Step Soundness for **br_table**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[(\mathsf{i32}.\textbf{const}\ \ idx)\,\textbf{br\_table}\ \ell^*\ell_N]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\forall c_1, c_2 \left(c_1 = (S;F;E[(\mathsf{i32}.\textbf{const}\ \ idx)\,\textbf{br\_table}\ \ell^*\ell_N]) \wedge \right. \\ \left. c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\right)$$

Proof. We define the initial configuration as follows:

$$c_1 = (S; F; E[(\text{i32}.\textbf{const}\ idx)\ \textbf{br\_table}\ \ell^* \ell_N$$

The final configuration depends on to the evaluation of the branching condition, and will thus be defined case by case.

The instruction **br_table** takes two inputs:

- ℓ^* is a vector of unsigned i32, indexed through i, where each entry contains a jump destination;
- ℓ_N is the default jump destination, in case the provided index i is too large with respect to the size of the vector.

First, we obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S; F; E[\text{i32}.\textbf{const}\ i\,dx; \textbf{br\_table}\,\textbf{vec}\ \textbf{idx}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\\ &\quad \eta_S(E[\text{i32}.\textbf{const}\ i\,dx; \textbf{br\_table}\,\textbf{vec}\ \ell_{default}])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F),\\ &\quad at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \leq \Delta_1\end{aligned}$$

The rules that express this behaviour are 29 and 30.

c) The first immediate is a valid index: Let us analyze the first case: in this scenario, the semantics described in [24] is:

$$(\text{i32}.\textbf{const}\ i\,)(\textbf{br\_table}\,\ell^*\ \ell_N) \hookrightarrow (\textbf{br}\,\ell_i) \qquad \text{if } \ell^*[i] = \ell_i$$

This means that, after the evaluation of the index, **br_table** behaves just like **br**. Since we already proved the soundness of **br** in A, the proof for this case is complete.

d) The first immediate is not valid index: The second case is similar: the semantics described in [24] is:

$$(\text{i32}.\textbf{const}\ i\,)(\textbf{br\_table}\,\ell^*\ \ell_N) \hookrightarrow (\textbf{br}\,\ell_N) \qquad \text{if } i \geq |\ell^*|$$

Just like the previous scenario, the instruction rewrites to a simple **br**, for which we have already proved the soundness. Therefore, the proof is complete. □

Lemma 33 (Single-Step Soundness for **if**). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{if}\ instr^*_{then}\ \textbf{then}\ instr^*_{else}\ \textbf{end}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}&\forall c_1, c_2\ \big(c_1 = (S; F; E[\textbf{if}\ instr^*_{then}\ \textbf{then}\ instr^*_{else}\ \textbf{end}]) \wedge\\ &\qquad c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configuration as follows:

$$c_1 = (S; F; E[(\text{i32}.\textbf{const}\ n)\ \textbf{if}\ instr^*_{\text{then}}\ \textbf{then}\ instr^*_{\text{else}}\ \textbf{end}])$$

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(\\ &\quad E[(\text{i32}.\textbf{const}\ n)\ \textbf{if}\ instr^*_{\text{then}}\ \textbf{then}\ instr^*_{\text{else}}\ \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0,\\ &\quad memo_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(n :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \leq \Delta_1\end{aligned}$$

After the evaluation of the condition, c_2 can take up two different forms:

$$\begin{aligned}c_{2,\text{then}} &= (S; F; E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{then}}\ \textbf{end}])\\ c_{2,\text{else}} &= (S; F; E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{else}}\ \textbf{end}])\end{aligned}$$

As you can see, the sequence of instructions get enclosed by a label with empty continuation. This additional instructions are not written manually by the programmer, but are generated automatically by WebAssembly before the execution. The semantics of a **if-else**-statement is defined in [24] as:

$$\begin{aligned}&(\text{i32}.\textbf{const}\ c)\ \textbf{if}\ t^n\ instr^*_{\text{then}}\ \textbf{else}\ instr^*_{\text{else}}\ \textbf{end}\\ &\qquad \hookrightarrow \textbf{label}_n\{\epsilon\}\ instr^*_{\text{then}}\ \textbf{end} \qquad (\text{if } c \neq 0)\\ &(\text{i32}.\textbf{const}\ c)\ \textbf{if}\ t^n\ instr^*_{\text{then}}\ \textbf{else}\ instr^*_{\text{else}}\ \textbf{end}\\ &\qquad \hookrightarrow \textbf{label}_n\{\epsilon\}\ instr^*_{\text{else}}\ \textbf{end} \qquad (\text{if } c = 0)\end{aligned}$$

e) The condition is true: Let us consider the first case, i.e., the case in which the condition is satisfied: we compute the abstraction of $c_{2,\text{then}}$, namely $\alpha(S; F; E[\text{label}\{\epsilon\}instr^*_{\text{then}}\ \textbf{end}])$, and we obtain the following:

$$\begin{aligned}\alpha(c_{2,\text{then}}) &= \alpha(S; F; E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{then}}\ \textbf{end}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{then}}\ \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0,\\ &\quad gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, memo_0)\}\end{aligned}$$

With respect to $\alpha(c_1)$, the difference is that the constant n has been popped from the stack, while all the other arguments remain unmodified.

Since the set of *MState* facts obtained with $\alpha(c_{2,\text{then}})$ complies with the consequence of Rule 31, we can say that the abstraction is sound.

f) The condition is not satisfied: We now examine the case in which the condition is not satified, i.e., $n = 0$. In this scenario, the final configuration will be $c_{2,\text{else}} = (S; F; E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{else}}\ \textbf{end}])$. The abstraction $\alpha(c_{2,else})$ is computed as follows:

$$\begin{aligned}\alpha(c_{2,else}) &= \alpha(S; F; E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{else}}\ \textbf{end}])\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{else}}(st, gt, lt, mem, at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{else}}(\alpha_{seq}(\eta_S(E[\textbf{label}_n\{\epsilon\}\ instr^*_{\text{else}}\ \textbf{end}])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0,\\ &\quad gt_0, memo_0)\}\\ &= \{\mathit{MState}(st, gt, lt, mem, at_0, gt_0, memo_0)\}\end{aligned}$$

where $else$, used with fid as a subscript of *MState*, represents the program counter of the instruction else: this models the fact that, is the condition is not satisfies, the flow of the program jumps to the else branch.

We can see that the abstraction obtained by computing $\alpha(c_{2,\text{else}})$ is coherent with the Rule 32. This concludes the proof.

□

Lemma 34 (Single-Step Soundness for **call** k). *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[\textbf{call}\ k]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}&\forall c_1, c_2\ \big(c_1 = (S; F; E[\textbf{call}\ k]) \wedge\\ &\qquad c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[\textbf{call}\ k])\\ c_2 &= (S; F; E[\textbf{frame}_m\{F'\}\ \textbf{block}\ instr^*\ \textbf{end}\ \textbf{end}])\end{aligned}$$

where the sequence *instr** represents the body of the function and m is the arity of the function, i.e., the number of outputs it produces.

We obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S; F; E[\textbf{call } k]) \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{call } k])), \alpha_{seq}(\eta_G(S, F)), \\ &\qquad \alpha_{seq}(F.locals), \alpha_M(i; S; F), at_0, gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

After the function call, the execution flows to the **block** instruction that encloses the body of the function. Furthermore, the following changes occur:

1) The function identifier, denoted as **fid** in the subscript of *MState*, changes to **cid** ("called function identifier"). The **cid** can be statically determined through the function table;
2) The program counter becomes 0: since its value is relative to the current function, $pc = 0$ means that we are ready to execute the first instruction of the function's body;
3) A new frame is introduced;
4) The stack is emptied;
5) The local variables set to the value of the function arguments, padded with zeroes.

Rule 33 models this behaviour, showing us how to derive Δ_2 from Δ_1, i.e., $\Delta_1 \vdash \Delta_2$. The predicate $\mathit{reverseEq}_{\mathsf{SI.as}(\mathbf{cid})}(st[: \mathsf{SI.as}(\mathbf{cid})], at)$ checks that the top part of the stack, where the arguments of the function are stored, is equal to the reversed array of arguments. This control is necessary to check that the arguments are correctly stored on the stack.

1) Entering a function: The instructions found in the evaluation context of c_2 fall in the first case of $\alpha_T(S; F; \mathit{instr}^*)$, as shown in Figure 6, therefore the abstraction of c_2 is obtained as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S; F; E[\textbf{frame}_m\{F'\}\,\textbf{block}\,\mathit{instr}^*\,\textbf{end}\,\textbf{end}]) \\ &= \alpha_F(F.fid, F'.pc, F'.\mathsf{stor}, F, F'.args +\!\!\!+ F'.index, \mathit{instr}^*) \\ &\quad \cup \alpha_T(S; F'; \mathit{instr}^*)\end{aligned}$$

Note that $F'.index$ is present only in case of an indirect call, because it represents the index in the table where we find the function address. Therefore, in our case the function reduces to:

$$\alpha_F(F.\mathsf{fid}, F'.\mathsf{pc}, F'.\mathsf{stor}, F, F'.\mathsf{args}, \mathit{instr}^*) \cup \alpha_T(S; F'; \mathit{instr}^*)$$

This translates to the following:

$$\begin{aligned}&\alpha_F(F.\mathsf{fid}, F'.\mathsf{pc}, F'.\mathsf{stor}, F, F'.\mathsf{args}, \mathit{instr}^*)) = \\ &\quad \{\mathit{MState}_{\mathbf{fid},0}(\alpha_{seq}(\eta_S(\mathit{instr}^*) +\!\!\!+ F'.\mathsf{args}), \alpha_{seq}(\eta_G(S, F)), \\ &\qquad \alpha_{seq}(F.locals), \eta_M(i, S, F), \alpha_{seq}(F.\mathsf{args}), \\ &\qquad \alpha_{seq}(\eta_G(F.\mathsf{stor}, F)), \eta_M(i, F.\mathsf{stor}, F))\}\end{aligned}$$

Before computing the abstraction $\alpha_T(S; F'; \mathit{instr}^*)$, we should consider that, according to [24], when a function is called the frame is:

$$F = \{\mathsf{module}\ f.\mathsf{module},\ \mathsf{locals}\ val^n\ val_0^*\}$$

This means that, as mentioned at the beginning of the proof at point 5), the local variables set to the value of the function arguments, padded with zeroes. Therefore, the abstraction function will return the following result:

$$\begin{aligned}&\alpha_T(S;F'; \mathit{instr}^*) = \alpha_F(F'.\mathsf{fid}, pc, S, F', \epsilon, \mathit{instr}^*) = \\ &\quad \{\mathit{MState}_{cid,0}([\,], \alpha_{seq}(\eta_G(S, F')), \\ &\qquad at +\!\!\!+ [0; \mathsf{SI.ls}(\mathbf{cid}) - \mathsf{SI.as}(\mathbf{cid})], \eta_M(i, S, F'), \\ &\qquad \alpha_{seq}(F'.\mathsf{args}), \alpha_{seq}(\eta_G(F'.\mathsf{stor}, F')), \eta_M(i, F.\mathsf{stor}, F))\}\end{aligned}$$

This set of *MState* facts matches the consequence of Rule 33, which means that they are derivable from Δ_1: this proves that our abstraction is sound.

2) Returning from a function: Let us now consider the situation where the execution returns from a function. The rule that governs this behaviour is 34: its structure is compatible with the generic return rule shown in Lemma 12, therefore its soundness is proved by the Lemma itself.

3) Trapping call: The execution of a function can cause a trap. To model this scenario, we propose 35. The subscript of the *Trap* fact in the premise states that the trap occurred inside the function. In order to prove the soundness of our abstraction, it is sufficient to invoke Lemma 11. □

Lemma 35 (Single-Step Soundness for **call_indirect**)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S; F; E[(\mathsf{i32}.\textbf{const}\ x)\ \textbf{call\_indirect}\ k]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}&\forall c_1, c_2\, \big(c_1 = (S; F; E[(\mathsf{i32}.\textbf{const}\ x)\ \textbf{call\_indirect}\ k]) \land \\ &\qquad c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2)\big)\end{aligned}$$

Proof. The instruction **call_indirect** requires many rules to be modelled correctly. In this section, we go through each one of them in the respective paragraphs and analyze the different scenarios.

4) Entering a function: The first proof concern the call a function through a call_indirect instruction. We define the concrete configurations as follows:

$$\begin{aligned}c_1 &= (S; F; E[(\mathsf{i32}.\textbf{const}\ x)\ \textbf{call\_indirect}\ k) \\ c_2 &= (S; F; E[\textbf{frame}_m\{F'\}\,\textbf{block}\,\mathit{instr}^*\,\textbf{end}\,\textbf{end}])\end{aligned}$$

where the sequence *instr** represents the body of the function and m is the arity of the function, i.e., the number of output values it produces.

The first part of the proof is similar to the one made for call instruction: to obtain the abstraction of c_1, denoted by $\alpha(c_1)$, by applying the abstraction functions to the concrete configuration:

$$\begin{aligned}\alpha(c_1) &= \alpha(S; F; E[(\mathsf{i32}.\textbf{const}\ x)\ \textbf{call\_indirect}\ k]) \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\alpha_{seq}(\eta_S(E[\textbf{call\_indirect}\ k])), \\ &\qquad \alpha_{seq}(\eta_G(S, F)), \alpha_{seq}(F.locals), \alpha_M(i; S; F), at_0, \\ &\qquad gt_0, mem_0)\} \\ &= \{\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0)\} \leq \Delta_1\end{aligned}$$

The rule for **call_indirect**, 36, is similar to the one for normal function calls. The only difference with respect to Rule 33 is the fact *Table*$(x, \mathbf{cid}, ts)$ in the premise, that encodes an entry in the function table. used to check whether the index provided is valid. From the semantics described in [24], we notice that both call and call_indirect reduce to an invoke instruction:

$$F;(\textsf{call } x) \hookrightarrow F;(\textsf{invoke } a) \qquad (\text{if } F.\textsf{module.funcaddrs}[x]=a)$$

$$\begin{aligned}S;F;(\textsf{i32.const } i)(\textsf{call\_indirect } x \hookrightarrow S;F;(\textsf{invoke } a)\\ (\text{if } S.\textsf{tables}[F.\textsf{module.tableaddrs}[0]].\textsf{elem}[i]=a\\ \wedge S.\textsf{funcs}[a]=f\\ \wedge F.\textsf{module.types}[x]=f.\textsf{type})\end{aligned}$$

Since both **call** and **call_indirect** reduce to the same instruction, we can leverage on the same proof used in A1 to prove the soundness of the abstraction.

a) Returning from a function: In this paragraph, we analyze the return from a function: the rule that governs this behaviour is 37. The *Return* fact in the premise is derived through Rule 43, discussed in Section A.

Rule 37's structure is compatible with the generic return rule shown in Lemma 12, therefore its soundness is proved by the lemma itself.

b) Trapping functions: There are four rules that model the occurrence of a trap at some point of the function execution: 38, 39, 40, and 42.

Rule 38 models the generation of a trap caused by the execution of an instruction inside the function.

Rule 39 represents the situation in which a trap is caused by an invalid function identifier: the predicate *Table*(ti, mte, ts) with $(mte=\epsilon \vee mte=te)$ and (last predicate) expresses the fact that the identifier mte is either empty, or is not among the identifier in the table, therefore does not correspond top any known function.

Rule 40 models the scenario in which the index is greater than the size of the table itself, and thus does not correspond to any entry.

Rule 42, finally, expresses a trap occurring during the execution of a function added externally.

The soundness of our abstraction is proved by Lemma 11 about traps.

5) Function added externally: We now analyze what happens when a function is added externally: this scenario is explained in Section III-C. Consider the following configurations:

$$\begin{aligned}c_1 &= (S;F;E[(\textsf{t.const } x)\ \textsf{call } i])\\ c_2 &= (S;F;E[\textsf{cmd}_{\textbf{pc}+1}])\end{aligned}$$

where x represents an index in the function table, and i is the identifier of the external function added to the table.

The abstraction of c_1 is computed as usual:

$$\begin{aligned}\alpha(c_1) &= \alpha(S;F;E[(\textsf{t.const } x)\ \textsf{call } i])\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(\alpha_{seq}(\eta_S(E[(\textsf{t.const } x)\ \textsf{call } i])),\\ &\quad \alpha_{seq}(\eta_G(S,F)), \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0,\\ &\quad gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, memo_0)\} \leq \Delta_1\end{aligned}$$

The rule that governs the behaviour of the program is 41: this rule models the possible modifications to the memory due to the execution of an external function. The premise states that:

- the value stored v' is stored at index i (which must not be greater than 255, since each index stores a byte of information);
- the new size of the memory, $size'$, must be comprised between the old size, $size$, and the maximum size, **max**. This is due to the fact that externally added functions cannot decrease the memory size.

The set of *MState* facts obtained from $\alpha(c_1)$ matches the one shown in the premise of 41. To finish the proof, we must check that also $\alpha(c_2)$ matches the consequence, which states that, after executing the function, the stack, the globals, and the memory have possibly been modified. In particular, the stack will contain the values returned by the function, to which we concatenate the stack without the arguments with which the function was called (the notation $st[as:]$ takes the elements from the stack from as (excluded) on).

[24] describes the semantics of returning from a host function as follows:

$$\begin{aligned}S;val^n\ (\textsf{invoke } a) \hookrightarrow S';result\\ (\text{if } S.\textsf{funcs}[a]=\{\textsf{type } [t_1^n]\rightarrow[t_2^m], \textsf{hostcode } hf\}\\ \wedge (S';result) \in hf(S;val^n))\end{aligned}$$

That is, if the function with address a takes in input n arguments, and produces in output m values, then when we execute invoke a with n values on the stack, we end up in a configuration where the store has possibly been modified and the results have been put on top of the stack.

In light of these observation, we compute $\alpha(c_2)$ as follows:

$$\begin{aligned}\alpha(c_2) &= \alpha(S;F;E[\textsf{cmd}_{\textbf{pc}+1}])\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(\alpha_{seq}(\eta_S(E[\textsf{cmd}_{\textbf{pc}+1}])), \alpha_{seq}(\eta_G(S,F)),\\ &\quad \alpha_{seq}(F.locals), \alpha_M(i;S;F), at_0, gt_0, memo_0)\}\\ &= \{\mathit{MState}_{\textbf{fid},\textbf{pc}+1}(result +\!\!+\, st[as:], rgt, lt,\\ &\quad Mem(i,v',size'), at_0, gt_0, memo_0)\}\end{aligned}$$

The sequence of inputs val^n have been popped on the stack, and $result$ has been pushed to replace them: this is correctly expressed with $result +\!\!+\, st[as:]$, which matches $rt +\!\!+\, st[as:]$ in 41. As mentioned above, some external functions have the ability to modify the store and the memory, generating a new store S' from S: this is reflected in the arguments rgt and $Mem(i,v',size')$ in the *MState* fact.

Overall, the result obtained matches Rule 41, therefore we can confirm that the abstraction is sound. □

Lemma 36 (Single-Step Soundness for Function Exit)**.** *For any two concrete configurations c_1 and c_2: If c_1 is of the form $S;F;E[\textsf{frame}_n\{F\}\,B^k[val^n\ \textsf{return}]\ \textsf{end}]$ or $S;F;E[\textsf{frame}_n\{F\}\ val^n\ \textsf{end}]$ and $c_1 \hookrightarrow\!\!\!\!\rightarrow c_2$, then we have $\alpha(c_1) \vdash \alpha(c_2)$:*

$$\begin{aligned}\forall c_1, c_2\ ((c_1 = (S;F;E[\textsf{frame}_n\{F\}\,B^k[val^n\ \textsf{return}]\ \textsf{end}]) \vee\\ c_1 = (S;F;E[\textsf{frame}_n\{F\}\ val^n\ \textsf{end}])) \wedge\\ c_1 \hookrightarrow\!\!\!\!\rightarrow c_2 \implies \alpha(c_1) \vdash \alpha(c_2))\end{aligned}$$

Proof. In this section, we prove the soundness of the rule for exiting from a function. There are two situations that trigger this event, which we report below:

$$\textbf{frame}_n\{F\}\, B^k[val^n\ \textbf{return}]\ \textbf{end} \hookrightarrow val^n$$
$$\textbf{frame}_n\{F\}\, val^n\ \textbf{end} \hookrightarrow val^n$$

Equation (9), described at 4.4.5.9 in [24], refers to the case in which we encounter a return instruction within a block context, which causes the function to return immediately. On the other hand, Equation (10), defined at 4.4.7.2, describes the return from a function that happens when we execute the last instruction of the body.

In both cases, a number of values equal to the arity of the function (n in this case) are put on top of the stack.

Rule 43 allows us to derive a *Return* fact, namely a fact that keeps track of the values that will be returned after the function ends and pushed onto the stack.

This fact is used in the premises of Rules A2 and A4a, that model what happens when the execution returns from a function. □

B. Abstractions for imported functions

Rule 44 for imported functions allows us to derive a *Return* fact. Given the state of the program as described by the *MState* fact, the rule considers two cases:

- if the imported function can modify the memory (i.e., it has semantic marker TM), then the value v' is any value between 0 and 255;
- otherwise, v' must be the same as the previous value v, because the memory cannot be modified by the function.

Then, for every global variable i we try to compute a lower bound **lb** and an upper bound **up** with the function $\mathsf{Sl.bdn}_G$(fid,i): if the function can modify the global variables, then we succeed in the computation and the value returned for each global $rgt[i]$ must be comprised between these bounds. Otherwise, the function cannot modify the globals, thus the return value for each global must be the same as the old value, $gt[i]$.

Then we check whether the return values on the stack are also comprised between the bounds. If the function has semantic marker GM(i.e., the function can grow the memory), then the new size must be greater or equal than the previous size, without exceeding the maximum limit. Otherwise, the size should remain the same.

If all of these premises are satisfied, then we can derive a *Return* fact, stating that the stack becomes rst, the globals are updated to rgt and the memory at index i now contains the value v'.

a) Imported functions with semantic marker TT: Rule 14 deals with imported functions with semantic marker TT, that have the ability to change the function table, without adding functions: This rule allows to derive arbitrary table facts.

b) Imported functions with semantic marker AF: There are three rules that allow to derive predicates used in the rules for imported functions. Here we give a short description of each one of them.

c) Derive a Return fact: Rule 15 deals with imported functions with semantic marker AF, that can add new, unknown functions to the store. This marker also implies that the table is changed, as adding functions without changing the table would be unobservable. This rule allows to derive a *FunctionAdded()* fact, expressing that new functions have been added to the table.

C. Computing the jump after a branching instruction

Let us define a *static block context* as follows:

$$\begin{aligned}
\widetilde{B}^0[\textbf{br}_{pc}\ \ell] = {} & instr_1^*\ \mathsf{block}\ \ldots\ \textbf{br}_{pc}\ \ell\ldots\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{loop}\ \ldots\ \textbf{br}_{pc}\ \ell\ldots\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{if}\ \ldots\ \textbf{br}_{pc}\ \ell\ldots\ \mathsf{else}\ instr_{\text{else}}^*\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{if}\ instr_{\text{then}}^*\ \mathsf{else}\ \ldots\ \textbf{br}_{pc}\ \ell\ldots\ \textbf{end}\, instr_2^*\\
\widetilde{B}^{\ell+1}[\textbf{br}_{pc}\ \ell+1] = {} & instr_1^*\ \mathsf{block}\ \widetilde{B}^{\ell}[\textbf{br}_{pc}\ \ell+1]\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{loop}\ \widetilde{B}^{\ell}[\textbf{br}_{pc}\ \ell+1]\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{if}\ \widetilde{B}^{\ell}[\textbf{br}_{pc}\ \ell+1]\ \mathsf{else}\ instr^*\ \textbf{end}\, instr_2^*\\
& \mid\ instr_1^*\ \mathsf{if}\ instr^*\ \mathsf{else}\ \widetilde{B}^{\ell}[\textbf{br}_{pc}\ \ell+1]\ \textbf{end}\, instr_2^*
\end{aligned}$$

Suppose we can factorize the block context of a program as follows:

$$\begin{aligned}
&\widetilde{B}^{\ell+1}[\,\textbf{br}_{pc}\ \ell+1] =\\
&\qquad instr_{\ell+1,1}^*\ \mathsf{block}_{pc_1}\, \widetilde{B}^{\ell}[\widetilde{B}^{\ell-1}[\ldots\widetilde{B}^{0}[\,\textbf{br}_{pc}\ \ell]\ \ldots]]\ \ \textbf{end}_{pc_2} instr_{\ell+1,2}^*
\end{aligned}$$

The notation $instr_{\ell+1,1}^*$ and $instr_{\ell+1,2}^*$ represent the sequences of instruction that compose block $\ell+1$.

Given an instruction **br** ℓ in a block of arbitrary depth, we can compute the destination of the jump with function *dest*: its first argument is the break instruction for which we are trying to compute the destination, while the second argument represents the depth of the block context that we predict is going to be the target of the jump.

$$Dest(\widetilde{B}^0[\textbf{br}_{pc}\ \ell]) = dest(\widetilde{B}^0[\textbf{br}_{pc}\ \ell_2])$$

$$dest(\widetilde{B}^{\ell_1}[\textbf{br}_{pc}\ \ell_2]) = \begin{cases} pc(\textbf{end},\ell_1) & \text{if } \ell_1=\ell_2\\ & \wedge(\widetilde{B}^{\ell_1} = instr_1^*\ \mathsf{block}\ \ldots\ \textbf{end}\, instr\\ & \vee\ \widetilde{B}^{\ell_1} = instr_1^*\ \mathsf{if}\ \ldots\ \mathsf{else}\ \ldots\ \textbf{end}\, in\\ pc(\mathsf{loop},\ell_1) & \text{if } \ell_1=\ell_2 \wedge \widetilde{B}^{\ell_1} = instr_1^*\ \mathsf{loop}\ \ldots\ \mathsf{e}\\ dest(\widetilde{B}^{\ell_1+1}[\textbf{br}\ \ell_2]) & \text{if } \ell_1\neq\ell_2 \end{cases}$$

The function *Dest* is called with $\widetilde{B}^0$, because we start our computation from the block that directly encloses the branching instruction. The idea is that, if the static block context $\widetilde{B}^{\ell_1}$ and the branch target ℓ_2 are the same, then we reached the right context, and we can jump at the **end** instruction of that block. If $\ell_1 \neq \ell_2$, than we need to “jump out” to the outer block to get closer to the target.

The function $pc(\textbf{end},\ell)$ returns the program counter of the **end** instruction of block context $\widetilde{B}^{\ell}$.

Suppose we have the following piece of code:

```
(block                    ;;  ℓ = 2
    (i32.const 2)
    (i32.const 3)
    (block                ;;  ℓ = 1
        (i32.const 4)
        (i32.const 5)
        (i32.add)
```

$$\mathit{MState}_{fid,0}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge \mathrm{TT} \in \mathsf{SI.sm}(fid) \\ \Longrightarrow \mathit{Table}(idx, te, tsz)$$

Fig. 14: Rule for with semantic marker TT

$$\mathit{MState}_{fid,0}(st, gt, lt, mem, at_0, gt_0, mem_0) \wedge \mathrm{AF} \in \mathsf{SI.sm}(fid) \\ \Longrightarrow \mathit{FunctionsAdded}()$$

Fig. 15: Rule for added functions with semantic marker AF

```
      (block              ;; ℓ = 0
          (i32.const 8)
          (i32.neg)
          (br 2)
      end)                ;; destination if ℓ1 = 0 (no
    recursive call)
   end)                   ;; destination at the first
    recursive call
 end)                     ;; destination the second (and
    final) recursive call
```

We show that function *dest* is able to compute the jump target correctly. We must evaluate *Dest*($\widetilde{B}^0$[$\mathbf{br}_{pc}\ell$]): ... to complete We prove the correctness of the function through induction.

Proof. Let us prove that the base cases are correct. Suppose that $\widetilde{B}^{\ell_1}[\mathbf{br}_{pc}\ \ell_2] = \widetilde{B}^{\ell_1}[\mathbf{br}_{pc}\ 0]$: this means that the target of the jump is the block containing **br** 0 (i.e. there are no other **block** or loop statements before encountering **br** 0). Then we distinguish two cases:

1) If the enclosing block is **block**, then we jump at the end of the current block. The target of the jump is thus correctly computed by $pc(\mathbf{end}, \ell_1)$, as it expresses the pc of the **end** instruction of the current block.
2) If the enclosing block is loop, then we jump at the beginning of the loop. The target of the jump is thus correctly computed by $pc(\mathsf{loop}, \ell_1)$, as it expresses the pc of the beginning of the loop.

Let us prove the inductive step: ... □

SI is a data structure containing the size of the various parameters of *MState*.

SI.as(cid) returns the size of the arguments of function call cid, while SI.ls(cid) returns the size of the local variables of cid.

reverseEq(x,y) is a predicate that evaluates to true only if $x = rev(y)$ or $y = rev(x)$ (where $rev(x)$ means array x reversed).

reverseEq$_{\text{SI.as(cid)}}(st[:\ \text{SI.as(cid)}], at)$ returns true if the top part of the stack (where the arguments of the function are stored) taken in reverse is equal to the arguments at

Supp. we have

$$val^m\ \mathsf{label}_n val^k\ \mathbf{br}\ \ell\ \mathbf{end}$$

The function *unwind*$_n$ puts n values on top of the stack. If $k > n$, then only the first n values are considered (the number of values put on the stack must be compatible with arity of the label).

1 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(\mathbf{v} :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i32.**const** ,f32.**const** , . . .
2 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(unOp_{\mathbf{op}}(x) :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i64.**clz**, f32.**abs**, . . .
3 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \circledast_{\mathbf{op}} x \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$	i32.**add**, f32.**sub**, . . .
4 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \widetilde{\circledast_{\mathbf{op}}} x \land res \in \mathbb{B}_{64} \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(res :: st, gt, lt, mem, at_0, gt_0, mem_0)$ 5 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: st, gt, lt, mem, at_0, gt_0, mem_0) \land res \in y \widetilde{\circledast_{\mathbf{op}}} x \land \neg(res \in \mathbb{B}_{64}) \implies$ $Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	i64.**mod**, i32.**div**, . . .
6 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(cvtOp_{\mathbf{op}}(x) :: st, gt, lt, mem, at_0, gt_0, mem_0)$	Conversions i32.**wrap_i64**, . . .
7 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(cvtOp_{\mathbf{op}}(x) :: st, gt, lt, mem, at_0, gt_0, mem_0)$ 8 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies Trap_{\mathbf{fid}}(at_0, gt_0, mem_0)$	Trapping Conversions i32.**trunc_f64_u**, . . .

Fig. 16: Abstractions for numeric instructions

9 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt, mem, at_0, gt_0, mem_0)$	**drop**
10 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: z :: st, gt, lt, mem, at_0, gt_0, mem_0) \land x = 0 \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(y :: st, gt, lt, mem, at_0, gt_0, mem_0)$ 11 $MState_{\mathbf{fid},\mathbf{pc}}(x :: y :: z :: st, gt, lt, mem, at_0, gt_0, mem_0) \land x \neq 0 \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(z :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**select**

Fig. 17: Abstractions for parametric instructions

12 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(lt[\mathbf{idx}] :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**local.get** idx
13 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt, lt[\mathbf{idx} \leftarrow x], mem, at_0, gt_0, mem_0)$	**local.set** idx
14 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(x :: st, gt, lt[\mathbf{idx} \leftarrow x], mem, at_0, gt_0, mem_0)$	**local.tee** idx
15 $MState_{\mathbf{fid},\mathbf{pc}}(st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(gt[\mathbf{idx}] :: st, gt, lt, mem, at_0, gt_0, mem_0)$	**global.get** idx
16 $MState_{\mathbf{fid},\mathbf{pc}}(x :: st, gt, lt, mem, at_0, gt_0, mem_0) \implies$ $MState_{\mathbf{fid},\mathbf{pc}+1}(st, gt[\mathbf{idx} \leftarrow x], lt, mem, at_0, gt_0, mem_0)$	**global.set** idx

Fig. 18: Abstractions for variable instructions

17 $x + \mathbf{offset} + N \div 8 - 1 < \mathit{size} \cdot 2^{16} \wedge \mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v, \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge$
$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(x + \mathbf{offset} + 0, \mathit{vs}[0], \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_{0_0} \,) \wedge \ldots \wedge$
$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(x + \mathbf{offset} + N, \mathit{vs}[N], \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_{0_0} \,) \wedge u = \mathit{combine}(\, \mathit{vs} \,) \wedge$
$w = \mathit{extends\_sx}_{N,|t|}(u) \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, w :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v, \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

18 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v, \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x + \mathbf{offset} + N \div 8 - 1 \geq \mathit{size} \cdot 2^{16} \implies$
$\mathit{Trap}_{\mathbf{fid}}(\, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

t.**loadN_sx**

19 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: y :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v, \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge y + \mathbf{offset} + N \div 8 - 1 < \mathit{size} \cdot 2^{16} \wedge$
$d = i - y + \mathbf{offset} \wedge u = \mathit{wrap}_{|t|,N}(\, x \,) \wedge w = u \gg d \cdot 8 \;\&\; 255 \wedge v' = (\, d < N \div 8 \,) \;?\; (\, w \,) : (\, v \,)$
$\implies \mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v', \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

20 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: y :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{Mem}(i, v, \mathit{size}), \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge y + \mathbf{offset} + N \div 8 - 1 \geq \mathit{size} \cdot 2^{16}$
$\implies \mathit{Trap}_{\mathbf{fid}}(\, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

t.**storeN_sx**

Fig. 19: Abstractions for memory instructions

21 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies \mathit{Trap}_{\mathbf{fid}}(\, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$ **unreachable**

22 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$ **nop**

23 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$ **block**

24 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$ **loop**

25 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies \mathit{MState}_{\mathbf{fid},\mathbf{next}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$ **end**

Fig. 20: Abstractions for some control instructions

26 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{br}}(\, \mathit{unwind}_{\mathbf{n}}(\, \mathit{st} \,), \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

br

27 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x \neq 0 \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{br}}(\, \mathit{unwind}_{\mathbf{n}}(\, \mathit{st} \,), \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

28 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x = 0 \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

br_if

29 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x = \mathbf{idx} \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{br}}(\, \mathit{unwind}_{\mathbf{n}}(\, \mathit{st} \,), \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

br_table

30 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x \geq \mathbf{sz} - 1 \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{br}}(\, \mathit{unwind}_{\mathbf{n}}(\, \mathit{st} \,), \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

br_table

31 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x \neq 0 \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{pc+1}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

32 $\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\, x :: \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,) \wedge x = 0 \implies$
$\mathit{MState}_{\mathbf{fid},\mathbf{else}}(\, \mathit{st}, \mathit{gt}, \mathit{lt}, \mathit{mem}, \mathit{at}_0, \mathit{gt}_0, \mathit{mem}_0 \,)$

if

Fig. 21: Abstractions for some control instructions

33 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{reverseEq}_{\mathsf{SI.as}(\mathbf{cid})}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})],\mathit{at}\,) \implies \\ &\mathit{MState}_{\mathbf{cid},0}(\,[\,]\,,\mathit{gt},\mathit{at} +\!\!+ [0;\mathsf{SI.ls}(\mathbf{cid})-\mathsf{SI.as}(\mathbf{cid})]\,,\mathit{mem},\mathit{at},\mathit{gt},\mathit{mem}\,)\end{aligned}$$

34 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{reverseEq}_{\mathsf{SI.as}(\mathbf{cid})}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})],\mathit{at}\,) \land \\ &\mathit{Return}_{\mathbf{cid}}(\,\mathit{rt},\mathit{rgt},\mathit{rmem},\mathit{at},\mathit{gt},\mathit{mem}\,) \implies \\ &\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\,\mathit{rt} +\!\!+ \mathit{st}[\mathsf{SI.as}(\mathbf{cid})\ :],\mathit{rgt},\mathit{lt},\mathit{rmem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

35 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{reverseEq}_{\mathsf{SI.as}(\mathbf{cid})}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})],\mathit{at}\,) \land \\ &\mathit{Trap}_{\mathbf{cid}}(\,\mathit{at},\mathit{gt},\mathit{mem}\,) \implies \mathit{Trap}_{\mathbf{fid}}(\,\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

call k

36 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{Table}(\,x,\mathbf{cid},\mathit{ts}\,) \land \mathit{reverse}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})]\,) = \mathit{at} \\ &\implies \mathit{MState}_{\mathbf{cid},0}(\,[\,]\,,\mathit{gt},\mathit{at} +\!\!+ [0;\mathsf{SI.ls}(\mathbf{cid})-\mathsf{SI.as}(\mathbf{cid})]\,,\mathit{mem},\mathit{at},\mathit{gt},\mathit{mem}\,)\end{aligned}$$

37 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{Table}(\,x,\mathbf{cid},\mathit{ts}\,) \land \\ &\mathit{Return}_{\mathbf{cid}}(\,\mathit{rt},\mathit{rgt},\mathit{rmem},\mathit{at},\mathit{gt},\mathit{mem}\,) \land \mathit{reverse}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})]\,) = \mathit{at} \implies \\ &\mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\,\mathit{rt} +\!\!+ \mathit{st}[\mathsf{SI.as}(\mathbf{cid})\ :],\mathit{rgt},\mathit{lt},\mathit{rmem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

38 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{Table}(\,x,\mathbf{cid},\mathit{ts}\,) \land \mathit{Trap}_{\mathbf{cid}}(\,\mathit{at},\mathit{gt},\mathit{mem}\,) \land \\ &\mathit{reverse}(\,\mathit{st}[:\ \mathsf{SI.as}(\mathbf{cid})]\,) = \mathit{at} \implies \mathit{Trap}_{\mathbf{fid}}(\,\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

call_indirect

39 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{Table}(\,x,\mathit{mte},\mathit{ts}\,) \land (\mathit{mte} = \epsilon \lor \mathit{mte} = \mathit{te}) \land \\ &\bigwedge_{(\mathbf{idx},\mathbf{cid})\in\mathsf{possibleCallTargets}(\mathbf{fid},\mathbf{pc})} \mathit{te} \neq \mathbf{cid} \implies \mathit{Trap}_{\mathbf{fid}}(\,\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

40 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{Table}(\,\mathit{ti},\mathit{mte},\mathit{ts}\,) \land x \geq \mathit{ts} \implies \\ &\mathit{Trap}_{\mathbf{fid}}(\,\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

call_indirect

41 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{Mem}(i,v,\mathit{size}),\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land 0 \leq v' \leq 255 \land \mathit{size} \leq \mathit{size}' \leq \mathbf{max} \land \\ &\mathit{FunctionsAdded}() \implies \mathit{MState}_{\mathbf{fid},\mathbf{pc}+1}(\,\mathit{rt} +\!\!+ \mathit{st}[\mathbf{as}\ :],\mathit{rgt},\mathit{lt},\mathit{Mem}(i,v',\mathit{size}'),\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)\end{aligned}$$

42 $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,x :: \mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \mathit{FunctionsAdded}() \implies \mathit{Trap}_{\mathbf{fid}}(\,\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)$$

call_indirect

43 $$\mathit{MState}_{\mathbf{fid},\mathbf{pc}}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \implies \mathit{Return}_{\mathbf{fid}}(\,\mathit{st}[:\ \mathsf{SI.rs}(\mathbf{fid})],\mathit{gt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,)$$

function exit

Fig. 22: Abstractions for some control instructions

44 $$\begin{aligned}&\mathit{MState}_{\mathbf{fid},0}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{Mem}(i,v,\mathit{size}),\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land (\,\textsc{tm} \in \mathsf{SI.sm}(\mathbf{fid})\,)\ ?\ (\,0 \leq v' \leq 255\,) : (\,v = v'\,) \land \\ &\bigwedge_{0\leq\mathbf{i}<\mathsf{SI.gs}()} \Big(\,(\mathbf{lb},\mathbf{ub}) = \mathsf{SI.bnd}_G(\mathbf{fid},\mathbf{i})\Big)\ ?\ \Big(\,\mathbf{lb} \overset{?}{\leq} \mathit{rgt}[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) : \Big(\,\mathit{rgt}[\mathbf{i}] = \mathit{gt}[\mathbf{i}]\Big) \land \\ &\bigwedge_{\substack{0\leq\mathbf{i}<\mathsf{SI.gs}()\\(\mathbf{lb},\mathbf{ub})=\mathsf{SI.bnd}_R(\mathbf{fid},\mathbf{i})}} \Big(\mathbf{lb} \overset{?}{\leq} \mathit{rt}[\mathbf{i}] \overset{?}{<} \mathbf{ub}\Big) \land (\,\textsc{gm} \in \mathsf{SI.sm}(\mathbf{fid})\,)\ ?\ (\,\mathit{size} \leq \mathit{size}' \leq \mathbf{max}\,) : (\,\mathit{size} = \mathit{size}'\,) \implies \\ &\mathit{Return}_{\mathbf{fid}}\big(\,\mathit{rt},\mathit{rgt},\mathit{Mem}\big(i,v',\mathit{size}'\big),\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,\big)\end{aligned}$$

45 $$\mathit{MState}_{\mathbf{fid},0}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \textsc{tt} \in \mathsf{SI.sm}(\mathbf{fid}) \implies \mathit{Table}(\,\mathit{idx},\mathit{te},\mathit{tsz}\,)$$

46 $$\mathit{MState}_{\mathbf{fid},0}(\,\mathit{st},\mathit{gt},\mathit{lt},\mathit{mem},\mathit{at}_0,\mathit{gt}_0,\mathit{mem}_0\,) \land \textsc{af} \in \mathsf{SI.sm}(\mathbf{fid}) \implies \mathit{FunctionsAdded}()$$

Fig. 23: Abstractions for imported functions.