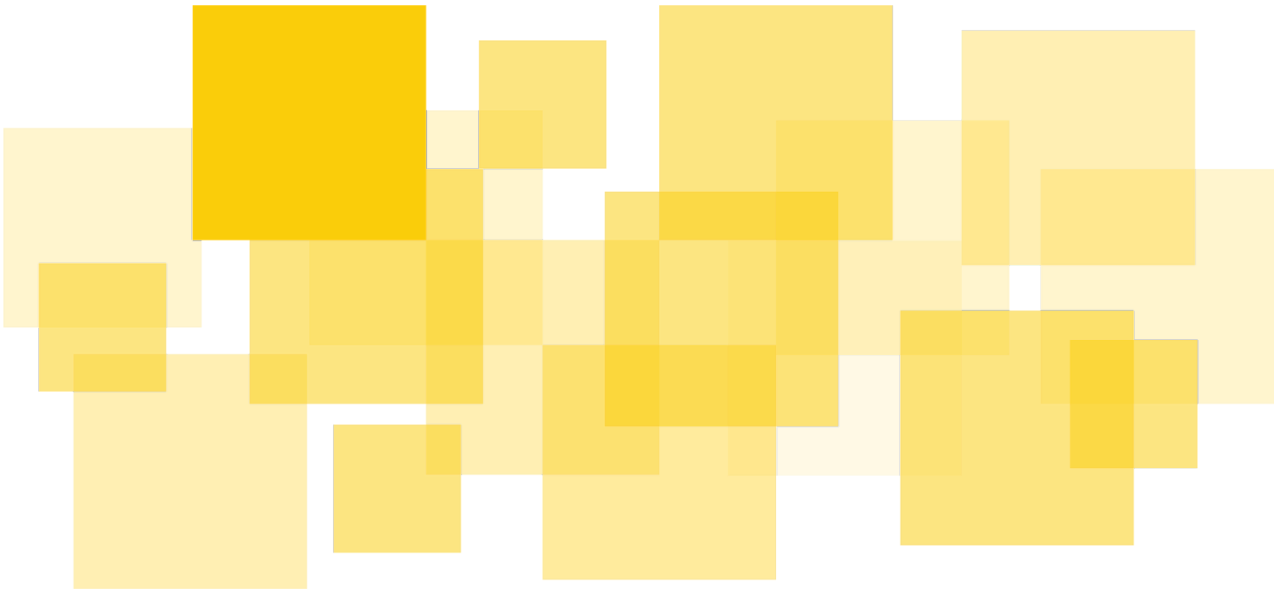
Security Audit Report

Wasmi - WebAssembly (Wasm) Interpreter Stellar

Delivered: November 27, 2024



Prepared for Stellar Network by


runtime
verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Blockchain technology is still a nascent software arena, and any related implementation and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Executive Summary

Stellar Network engaged Runtime Verification Inc. to conduct a security audit of the Wasmi interpreter, which Wasmi Labs is custodian of. The objective was to review the logic and implementation of critical components of the interpreter and identify any issues that could cause erroneous or undefined behavior that may lead to exploitation or malicious interaction with the Stellar network.

The audit was conducted over the course of 8 calendar weeks (August 21, 2024, through October 16, 2024) and focused primarily on analyzing the executor and translator crates of the interpreter, as well as the abstract relationship between Wasm and Wasmi. Given the large volume and high complexity of code comprising the interpreter, a unique approach was taken to the audit that would result in highest guarantees possible for the allocated time frame. The audit would have two surfaces of analysis, a best effort code review approaching components in order of priority, and dedicated fuzzing using a variety of fuzzers and configurations.

The Wasmi codebase is in excellent shape: Code is generally well-organized, adheres to Rust best practices and contains informative doc comments in various places, as well as explanations for particular invariants which may be unobvious.

The audit led to identifying issues of potential severity for the protocol's health, which have been identified as follows:

- Errors in exceeding bounds: CompiledFuncEntity can take more than `u32::MAX` entries, Inconsistencies in `RegisterAlloc` Bounds, Large inputs to raw pointer functions may cause undefined behaviour depending on target
- An error in fuel computation: Underestimated Fuel Consumption of Table and Memory Instructions
- Differences between Wasmi's execution and other WebAssembly interpreters:
  - Output mismatch between Wasmi and Wasmtime #1
  - Output mismatch between Wasmi and Wasmtime #2
  - Output mismatch between Wasmi and Wasmtime #3
- A crash due to a bug in instruction optimizations: Abort on `realloc()` due to faulty `br_table` optimization

- A failed assertion due to a translation optimization: Translator debug assertion fired due to `ref.is_null` constant propagation
- A segmentation fault due to a missed case in a visitor: Segmentation fault due to `visit_input_regs` bug
- A hang due to faulty logic in an instructions execution: Executor hang on unreleased version due to `copy_span` bug
- A crash due to a miscalculated branch table offset: Executor panic on unreleased version due to `br_table_many` bug

In addition, several informative findings, contributions, and general recommendations also have been made, including:

- Improvements in Best Practices and Notable Particularities
- Results of `cargo-audit`, an automatic dependency analysis
- Documenting a shortfall of a helper function: `visit_input_registers` does not visit all registers of a `RegisterSpan`
- A potentially unsound optimization in translation: Missing `copy` in `select` translation
- Suggestions involving validation post translation: Suggested Executor Assertions, Translation Output Validation
- Open Source Contributions

Additionally, the document contains a high-level description of Wasmi's design:

- An overview of the Wasmi translation and execution process: Platform Logic and Features Description
- Appendices to complement the description:
  - A reference of the Wasmi instruction set: Appendix: Wasmi Instruction Set Overview
  - Diagrams that visualize the translation process: Appendix: Translation Sequence Diagrams
  - Diagrams that visualize the data structures used for translation and execution: Appendix: `Engine` Class Diagrams, Appendix: `FuncTranslator` Class Diagrams
  - A detailed description of the translation process: Appendix: Algorithmic Description of the Translator

At the time of writing, all crashes and output differences have been fixed both in Wasmi's `main` line as well as in a branch of `v0.36.*` versions. The potential errors in exceeding bounds and

the fuel miscalculation have been acknowledged but not addressed. Informative findings and suggestions have been acknowledged and partially addressed in Wasmi's `main` line.

Goal

Given the large volume and high complexity of code comprising the interpreter, Runtime Verification Inc. and Stellar Network agreed on an approach to the audit that would maximize the coverage and quality of analysis performed in the allocated time for the audit. Unfortunately total analysis of such a large and complicated code base would be impossible to achieve in the allocated 8 weeks. Furthermore, the on-going work on the interpreter means there is potential that updates could occur in the future, leaving the value of the analysis locked to a particular version eventually only used for legacy versions of Stellar. Therefore, the approach that would be taken was one of both dynamic analysis with fuzzing and simultaneous code review, with an additional eye towards possible future developments of both Stellar and Wasmi. To elaborate on these branches:

1. Dynamic Analysis focused on fuzzing the translator and executor with structured bytecode, utilizing a variety of fuzzing tools both standard and custom to effectively detect possible crashes, identify mismatches with other Wasm implementations, and reveal possible DoS vectors.
2. Code Review for the duration of the audit would prioritize recently added logic to the target of the audit (v0.36.0) as there is likely to be more chance of finding errors in newer code. Furthermore, code that had high amounts of `unsafe` usage would be prioritized, followed by code that has high complexity. This meant that focus would be first directed to the executor, then the translator module, with any remaining code being reviewed should time allow;

The audit focuses on identifying issues in the interpreter's logic and implementation that could potentially create erroneous or undefined behavior and therefore render Stellar network vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve the safety, efficiency, or readability of the implementation.

Scope

The scope of the audit is limited to the code contained in a public Github repository provided by the client (wasmi-labs/wasmi). The version that is the target of the audit is tagged release v0.36.0 which has commit hash `02621ad7a7f769dc97524075a693cc96e2049cb5`.

Within the repository are multiple crates and files, some of which are highlighted as in the scope of the audit. The repository and relevant crates and files are described below:

- `crates/`
  - `cli/` : entrypoint for fuzzing and code inspection
  - `collections/` : helper data structures
  - `core/` : foundational data and error types for Wasmi execution and translation
  - `wasmi/`
    - `engine/`
      - `bytecode` : Wasmi instruction set
      - `executor` : Wasmi execution implementation
      - `translator/` : translation from Wasm to Wasmi
- `fuzz/` : fuzz testing harness

The comments provided in the code, a general description of the project, including samples of tests used for interacting with the platform, and online documentation provided by the client were used as reference material.

The audit's focus is the translation of Wasm to Wasmi bytecode and the execution of Wasmi bytecode, in the `engine` directory of the `wasmi` crate. It is limited in scope to the artifacts listed above.

Commits addressing the findings presented in this report (with versions `0.36.1-5`) have also been analyzed to ensure the resolution of potential issues.

Methodology

As mentioned in section Goal, this audit will have two parallel streams of analysis running: code review, and dynamic analysis through fuzzing. The methodology for each will be described separately.
Findings will be classified according to the Runtime Verification Audit Methodology.

Code Review
It should be restated that manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, however we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the intention and design logic of the code, seeking to understand the intention of the Wasmi interpreter design choices, to evaluate if the current implementation is susceptible to security-critical design flaws and to ensure the absence of loopholes in the design logic. To this end, we first carefully analysed the specification of Wasm and understood it's relation to Wasmi, seeking to understand the proposed features of Wasmi and the differences and similarities between the two (details of which are located in Platform Logic and Features Description). We also created design documents and artefacts that we communicated with the client to ensure that our mental model of the design and implementation is accurate.

Second we began review of the Wasmi v0.36.0 source code, focusing on the engine and translator crates that perform the translation from Wasm to Wasmi before execution. The review aimed to ensure that the intended features of Wasmi are indeed implemented error free, and that unintended extra behaviors which may be exploitable are not implemented. Wasmi translation performs optimization of Wasm bytecode, such as constant propagation, or op-code fusion. As part of our analysis we documented the translation from Wasm to Wasmi and produced the first specification of Wasmi external to the source code.

Another priority that concerns the Stellar Network is that v0.36.0 introduces many instances of `unsafe` code in the upgraded executor crate. Usage of `unsafe` may present a higher risk of error, since the compiler will relax its guarantees of safety in order to provide the extra features unavailable in `safe` code. The canonical way to determine safety for `unsafe` usage is to understand the invariants that must be upheld by the usage (often listed in the data structures documentation), and to provide a safety comment that details how the calling / contextual code

is upholding that invariant. As part of our analysis we inspected each usage of `unsafe` and endeavour to ensure the invariants are upheld and that the safety comment communicates that accurately.

Fuzzing

To augment our code review, we additionally ran a fuzzing campaign - feeding a barrage of random inputs to the program-under-test as a means to dynamically identify vulnerabilities which might be overlooked by manual review alone. This campaign primarily focused on Wasmi v0.36.0, but eventually switched to the v0.36.x-dev branch to avoid re-encountering the same issues after they were identified and patched. At times, we discovered "shallow" bugs or crashes which prevented the fuzzer from reaching deeper code paths, and we switched to fuzzing the current Wasmi main in the interim until these issues were patched.

We began our fuzzing campaign by identifying testable properties that are expected to hold across all inputs. With Wasmi, later stages of the translation and execution pipeline make implicit assumptions about the correctness of earlier stages (see Translation Output Validation). The resulting interconnectedness makes it difficult to isolate smaller sub-components and their associated invariants, particularly given the scope of the audit relative to the size of the code base. Moreover, fuzzing is a randomized process which benefits from a large number of iterations, so given finite time and compute power, there's a trade-off between the total number of tested properties versus the amount of resources dedicated to testing each one individually.

For all these reasons, we decided the most effective approach was to focus our efforts and resources on a narrowly selected set of end-to-end invariants of the entire interpretation pipeline. This allows these properties to be very thoroughly tested, and the end-to-end nature ensures all components are still covered while increasing the likelihood of identifying vulnerabilities caused exactly by the aforementioned interconnectedness.

To accomplish this end-to-end testing, we generated random Wasm modules, then translated and executed each exported function with randomly generated arguments. We verified memory safety and crash-freedom by running this process with sanitizer instrumentation and debug assertions enabled, and verified functional correctness through differential fuzzing which tests conformance against another established Wasm implementation (wasmtime). Further details are given in Fuzzing Discussion and Findings.

We also gave equal care to how these random inputs are generated, ensuring that our properties are tested against a diversity of inputs adequately covering edge cases. At a high-

level, most fuzzers proceed in the same way: generating random inputs, running them and gathering feedback, then using this feedback to inform the generation of new inputs. However, the particulars of this process varies greatly between tools - using different metrics to decide which inputs are interesting, different mutation strategies to produce new inputs, etc. - making it important to consider multiple options.

Among prominent fuzzers, standard benchmarking such as FuzzBench consistently shows two top contenders for general-purpose fuzzing effectiveness: AFL++ and honggfuzz. We did initial exploratory runs with both of these top choices, using afl.rs and honggfuzz-rs to integrate with the Rust code. For our particular targets, we found that both achieved similar levels of effectiveness, but honggfuzz scaled better across cores out-of-the-box without the need to optimally configure advanced aspects of AFL++. Additionally, honggfuzz can take advantage of more information sources directly from the hardware, allowing faster iteration time by avoiding instrumentation when desired, as well as providing a base for the custom execution-time based fuzzing we describe later.

Honggfuzz is structure-unaware, meaning that it generates inputs which are simply raw byte sequences, and the test code itself is responsibility for converting that `&[u8]` seed into more structured data. Care must be taken to ensure that this interacts well with the employed mutation strategies - that small mutations of the input seed produce small changes in the structured data. For generating Wasm modules, this functionality is already offered by the wasm-smith crate, with configuration to enable or disable various Wasm features. To focus on vulnerabilities relevant to Soroban, we extended wasm-smith's configuration with an option to disable the generation of floating-point instructions and types, and this was later merged into wasm-smith v0.218.0.

With a basic harness set up, our strategy was then to continually refine these harnesses by repeating the following steps:

1. Start the fuzzer on the given harness, leaving it running continuously in the background.
2. While that long-running harness makes progress, repeatedly
    1. Inspect any code coverage and performance information.
    2. Based on the collected information, tweak the configuration or test code.
    3. Do a shorter fuzzing run to explore if the changes were beneficial, deciding whether they should be kept or reverted.

3. Once the long running harness fails to find any new interesting inputs over a large span of time, or if a significantly better configuration is found, switch the long-running harness to the new configuration.

This process ensures that our compute resources are fully utilized for the duration of the audit, enabling us to collectively perform a very large number of iterations, while still doing the exploratory work needed to achieve a high-level of effectiveness.

Platform Logic and Features Description

Wasmi is an efficient and lightweight WebAssembly interpreter with a focus on constrained and embedded systems. It relies on translating the stack-based Wasm bytecode to Wasmi's internal, register-based instruction set. Conceptually, the Wasmi processing pipeline thus consists of the following steps:

1. **Compilation**. Parsing the Wasm bytecode, validating the parsed module, and translating the parsed module into Wasmi's internal representation.
2. **Execution**. Interpreting a function compiled to Wasmi.

Compilation

During compilation, a Wasm module, represented as bytecode, is parsed, validated and translated into Wasmi's internal data structure for representing programs, called a code map (see `CodeMap` in Appendix: `Engine` Class Diagrams) that the execution phase interprets. Parsing and validation relies on the external `wasmparser-nostd` crate, a fork of `wasmparser`.

Module compilation has the following steps:

1. Processing the module header.
2. Processing functions.
3. Processing the data section.

Module header and data section processing include parsing and validating the input payload by payload, then storing the processed data. In addition to validation as defined by the Wasm specification, Wasmi also enforces structural limits defined in its configuration, e.g. on the maximal number of global variables (see `Config` in Appendix: `Engine` Class Diagrams).

This is always done eagerly, i.e. when compiling a given module. However, Wasmi provides different compilation modes that enable processing functions lazily:

1. **Eager**. All functions are parsed, validated and translated eagerly.
2. **Lazy translation**. All functions are parsed and validated eagerly, but translated lazily on first use.
3. **Lazy**. All functions are parsed, validated and translated lazily on first use.

Additionally, function validation may be skipped altogether in all compilation modes. The module parsing, validation and translation process is shown in detail (for the eager setting) in Appendix: Translation Sequence Diagrams.

During compilation, the code map is populated with `FuncEntity` instances. In eager mode, functions are translated into `CompiledFuncEntity` instances. Such an object stores all data, necessary to execute the given function, in particular, a sequence of Wasmi instructions. In lazy and lazy translation modes, a function is represented as an `UncompiledFuncEntity` that stores the byte code and the module header that enable translating (and optionally, validating) the function on-demand.

Translation Algorithm

Translation transforms a stack-based Wasm program into a register-based Wasmi program.

The function translator is implemented as a visitor over `wasmparser`'s instruction AST (i.e. implements the `VisitOperator` trait) that is invoked when an instruction is parsed. Hence translation is tightly coupled with the parsing process (as is validation). Throughout the translation process, several compiler optimizations are applied, e.g. constant propagation, peephole optimization, dead code elimination, and opcode fusion. After linearly processing each instruction, a finalization step is performed that resolves labels and consolidates the register space.

For a per-instruction description of the translation process, see Appendix: Algorithmic Description of the Translator.

Data structures used in the translation process are depicted in Appendix: `FuncTranslator` Class Diagrams.

Example

We're going to demonstrate the translation process on a simple example. Consider the following Wasm function:

```
func (param i32) (result i32)
  local.get 0
  if (result i32)
    i32.const 1
  else
    i32.const 2
  end
  return
end
```

When translating this function, as an initialization step, a new control frame for the function body is pushed onto the control stack (see `ControlFrame` and `ControlStack` in Appendix: `FuncTranslator` Class Diagrams) that tracks information about the current control structure. For example, a new symbolic label, say, `L0`, is created and stored in the control frame to represent the jump location at the end of the function. Since the concrete address is not yet known, it will be resolved later when visiting the corresponding `end` instruction (and in some cases during the finalization step).

Next `local.get 0` is processed, and a new tagged provider (wrapping the register storing the value for local 0, say, `R0`) is pushed onto the value stack (see `TaggedProvider` and `ValueStack` in Appendix: `FuncTranslator` Class Diagrams).

Next, `if i32` is processed, and a new control frame is pushed which tracks the `if` with, among other things, labels for the end of the `if-else` block and the start of the `else` branch (say, `L1` and `L2`, respectively). Since the if produces an `i32` value, a new register, `R1`, is allocated to hold the result of the control structure. At the same time, the provider is consumed from the stack, and the first (albeit incomplete) instruction is generated using the instruction encoder (see `InstrEncoder` in Appendix: `FuncTranslator` Class Diagrams):

```
branch_i32_eq_imm R0 0 L2
```

encoding that, if register `R0` is `0`, then jump to the (yet unknown) destination `L2` (the `else` branch), otherwise continue with the next instruction (the `then` branch).

Next, `i32.const 1` is processed, which pushes the constant value `1` onto the value stack.

Next, `else` is processed, which marks (1) the end of the `then` branch (2) the start of the `else` branch. Accordingly, the value stack is popped and pushed, and label `L2` is pinned. Also, `1` is popped from the value stack, and two new instructions are generated:

```
copy_imm32 R1 1
branch L1
```

which writes the `if`-result into `R1`, then jumps after the `if-else` block.

Next, `i32.const 2` is processed, which pushes the constant value `2` onto the value stack.

Next, `end` is processed, which marks the end of the `if-else` block. Accordingly, the control stack is popped, and `L1` is pinned. Also, `2` is popped from the value stack and `R1` is pushed onto it to produce the result of the `if-else` block, and a new instruction is generated:

```
copy_imm32 R1 2
```

which writes the `else`-result into `R1`.

Next, `return` is processed, which simply pops the value stack and generates

```
return_reg R1
```

Finally, `end` is processed, which marks the end of the function body, and the control stack is popped.

The following finalization step then resolves labels and jump offsets in instructions, thus the final Wasmi program is

```
branch_i32_eq_imm R0 0 3
copy_imm32 R1 1
branch 2
copy_imm32 R1 2
return_reg R1
```

Execution

Wasmi execution is relatively straightforward. For each function call, a call frame is pushed onto the call stack, and function local constants, function arguments, and function locals are allocated and initialized on the value stack, the execution-time equivalent of the class with the same name used for translation (see `EngineStacks` in Appendix: `Engine` Class Diagrams). The instructions stored in the code map are then interpreted, mutating the call and value stacks.

Continuing the example above, let's assume that the translated function has been called with an `i32` argument `arg`.

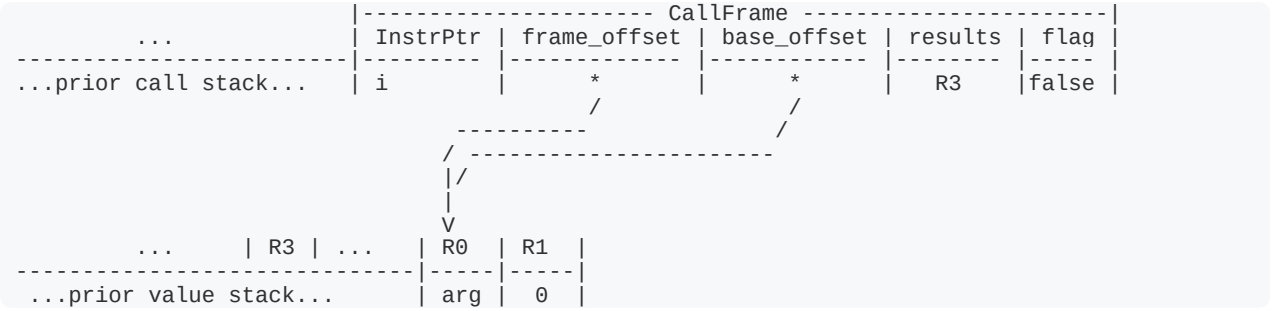
- The function body instructions and their addresses are

```
Address  Wasmi-Instruction
-------  ------------------------
i        branch_i32_eq_imm R0 0 3
i+1      copy_imm32 R1 1
i+2      branch 2
i+3      copy_imm32 R1 2
i+4      return_reg R1
```

- A `CallFrame` has been added to the *call stack*. It indicates where in the *caller's* register slots the single `i32` result should be written (`R3` for the sake of an example), and contains the instruction pointer of the code to execute (starting at address `i`), as well as offsets into the *value stack*;
- A "value frame" of size 2 has been added to the *value stack*. It contains the single `i32` argument `arg` in register slot `R0`, and an additional register slot `R1` to use in the function

body (initialised with `0`).

- Both *frame offset* and *base offset* pointers in the `CallFrame` point to the register slot `R0`. If the function was using constants, they would be allocated before this slot (accessed with negative register numbers), and the *frame pointer* would point to the first constant.



Execution:

- Execution starts by reading the instruction at address `i`, which is `branch_i32_eq_imm`. This compound (fused) instruction compares the `arg` in register slot `R0` to the immediate value `0` and performs a branch if the result is `true`.
- If the two values are equal (i.e., result `1` for `true`), the instruction pointer will be incremented by `3`, and `i+3` is the next instruction.
  - This instruction `copy_imm32` will write `2` into the register slot `R1`. The value stack is then `...| arg | 2 |`
  - `i+4` is the next instruction

- If the values are not equal (i.e., result `0` for `false`), the branch is not taken, `i+1` is the next instruction.
  - `copy_imm32` will write `1` into the register slot `R1`. The value stack after this is `...| arg | 1 |`
  - The next instruction `branch` increments the instruction pointer by `2`, and `i+4` is the next instruction.

- The `return_reg` in `i+4` performs a function return with the value from `R1`
  - the value in `R1` (either `1` or `2`) is written to the target destination `R3` in the prior value frame,
  - the value stack size is reduced to remove `R0` and `R1` of the finished call,

- and the call frame is popped from the call stack.

In order for execution to be as performant as possible, the executor intensively relies on `unsafe` functions. Safety of these calls mostly relies on correctness of the translation See Translation Output Validation for further details.

Code Review Discussion and Findings

This section communicates results, data, and findings of the code review performed over the duration of the audit. The code review was performed in accordance with section Methodology where analysis of the translator and executor were performed; as well as specific targeted inspection of each `unsafe` call.

The Wasmi codebase is generally well-organized, adheres to Rust best practices and contains informative doc comments in various places, as well as explanations for particular invariants which may be unobvious.

Findings presented in this section are issues that can cause the interpreter to fail, malfunction, and/or be exploited, and should be properly addressed. Informative findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need external support or deviate from best practices.

Translator

We performed a best-effort code review of the Translator (under `crates/wasmi/src/engine/translator`). Results are summarized in the following table.

File	Code Review Findings
control_frame.rs	Informative findings
control_stack.rs	No issues found
driver.rs	Informative findings
error.rs	No issues found
instr_encoder.rs	Informative findings
labels.rs	No issues found
mod.rs	Informative findings, Missing `copy` in `select` translation
relink_result.rs	Informative findings
stack/consts.rs	Informative findings
stack/locals.rs	No issues found
stack/mod.rs	Informative findings
stack/provider.rs	Informative findings
stack/register_alloc.rs	Inconsistencies in `RegisterAlloc` Bounds
typed_value.rs	No issues found
utils.rs	No issues found
visit_register.rs	`visit_input_registers` does not visit all registers of a `RegisterSpan`
visit.rs	Informative findings

Informative findings mentioned in this table include small inconsistencies, typos, code quality suggestions or other trivial issues in code or documentation. These have been reported directly to the Wasmi development team as soon as detected, and are not detailed further in this report.

Commits fixing informative findings:

- https://github.com/wasmi-labs/wasmi/commit/c0f79e1f8ad38847023b72fea5a738ddca13167b
- https://github.com/wasmi-labs/wasmi/commit/39414e86b75e25747d84204417c99ba2c970f373
- https://github.com/wasmi-labs/wasmi/commit/efb0329b320278f46653c561ca0a2c1f47750a78

Executor

We established an overview of the Wasmi instructions in comparison to corresponding Wasm instructions. Then we inspected the implementation of each family of instructions, with a focus on potential problems caused by their use of `unsafe` Rust code.

`unsafe` Rust Code

Throughout the wasmi code base there are various usages of `unsafe`. Each of these usages expands the capabilities of Rust, and so weakens the safety guarantees typically enforced by the Rust compiler. The conventional way to ensure that a usage of `unsafe` Rust is safe is to check the safety invariants that the particular usage requires, and to write a `Safety` comment that details how the calling code / context upholds the invariants. As part of the audit we inspected many usages of `unsafe` Rust, focusing on the unsafe Rust usage in the `engine` and particularly `executor` (sub-)modules. Each inspection had a best effort to ensure that the `Safety` invariants are satisfied, and so no undefined behaviour could be expected to occur from these locations.

A table detailing the analysis can be found in the Appendix: Appendix: `unsafe` Rust Checklist

[C1] CompiledFuncEntity can take more than `u32::MAX` entries

Severity: Low | Recommended Action: Fix Code | Addressed by client

Description

Inside code_map.rs inside the engine module, the struct `CompiledFuncEntity` has a function `new` with a comment that indicates that the function should panic if called with `instrs` length greater than `u32::MAX`. However this is not enforced in the code, and testing verified it is possible to exceed the bound without triggering a panic.

Recommendations

Likely the best course of action is to add an assertion that triggers the panic. However if it is intended for this bound to be able to be exceeded then the comment should be removed. Furthermore, there should be an upper bound enforced that is consistent with Wasm.

Status

This issue is fixed on `main` by PR#1207

[C2] Inconsistencies in `RegisterAlloc` Bounds

Severity: Low | Recommended Action: Fix Code | Not addressed by client

Description

According to code inspection and discussions with the Wasmi development team, the dynamic and preservation space of registers satisfy the following invariants of `RegisterAlloc`:

```
dynamic(r):  min_dynamic  <= r <  max_dynamic
preserve(r): min_preserve <  r <= max_preserve
```

Relationship between `max_dynamic` and `min_preserve`

In order for the two register spaces to be distinct, `max_dynamic <= min_preserve` needs to hold. However, `RegisterAlloc` methods do not agree whether equality is allowed or not.

For example, `push_dynamic()` admits equality (it performs the check before applying the increment):

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/stack/register_alloc.rs

Line 248 to 257 in 02621ad

```
248     pub fn push_dynamic(&mut self) -> Result<Register, Error> {
249         self.assert_alloc_phase();
250         if self.next_dynamic == self.min_preserve {
251             return Err(Error::from(TranslationError::AllocatedTooManyRegisters));
252         }
253         let reg = Register::from_i16(self.next_dynamic);
254         self.next_dynamic += 1;
255         self.max_dynamic = max(self.max_dynamic, self.next_dynamic);
256         Ok(reg)
257     }
```

whereas `push_dynamic_n` does not (it performs the check after applying the increment):

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/stack/register_alloc.rs

Line 268 to 283 in 02621ad

```rust
    pub fn push_dynamic_n(&mut self, n: usize) -> Result<RegisterSpan, Error> {
        fn next_dynamic_n(this: &mut RegisterAlloc, n: usize) -> Option<RegisterSpan> {
            let n = i16::try_from(n).ok()?;
            let next_dynamic = this.next_dynamic.checked_add(n)?;
            if next_dynamic >= this.min_preserve {
                return None;
            }
            let register = RegisterSpan::new(Register::from_i16(this.next_dynamic));
            this.next_dynamic += n;
            this.max_dynamic = max(this.max_dynamic, this.next_dynamic);
            Some(register)
        }
        self.assert_alloc_phase();
        next_dynamic_n(self, n)
            .ok_or_else(|| Error::from(TranslationError::AllocatedTooManyRegisters))
    }
```

Uninhabitable register index

Even if `max_dynamic = min_preserve` is allowed, for `r = max_dynamic = min_preserved`, neither `dynamic(r)` nor `preserve(r)`.

This can lead to spurious `TranslationError::AllocatedTooManyRegisters` errors:

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/stack/register_alloc.rs

Line 250 to 252 in 02621ad

```rust
        if self.next_dynamic == self.min_preserve {
            return Err(Error::from(TranslationError::AllocatedTooManyRegisters));
        }
```

Moreover, `register_space()` returns `RegisterSpace::Dynamic` for such a register:

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/stack/register_alloc.rs

Line 165 to 176 in 02621ad

```rust
    pub fn register_space(&self, register: Register) -> RegisterSpace {
        if register.is_const() {
            return RegisterSpace::Const;
        }
        if self.is_local(register) {
            return RegisterSpace::Local;
        }
        if self.is_preserved(register) {
            return RegisterSpace::Preserve;
        }
        RegisterSpace::Dynamic
    }
```

Recommendation

Adjust the definition of the bounds so that the whole register space can be utilized. Document these invariants. Make sure checks for bounds are consistent.

Status

Acknowledged by client.

[C3] Underestimated Fuel Consumption of Table and Memory Instructions

Severity: Low | Recommended Action: Fix Code | Not addressed by client

Description

`FuelCosts::cost_per` uses a truncating `u64` division to compute fuel amounts.

wasmi-labs/wasmi/crates/wasmi/src/engine/config.rs

Line 132 to 135 in 02621ad

```
132     /// Returns the fuel consumption of the amount of items with costs per items.
133     fn costs_per(len_items: u64, items_per_fuel: NonZeroU64) -> u64 {
134         len_items / items_per_fuel
135     }
```

This may result in `fuel_for_bytes` and `fuel_for_copies` returning `0` (with the default cost of 8 registers (copies) per fuel and 64 bytes per fuel).

`fuel_for_copies` is used in several places. During the translation when generating `Copy` and `Return` instructions, a `base` fuel is added to account for the truncation

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/instr_encoder.rs

Line 468 to 473 in 02621ad

```
468                 // Note: The fuel for copies might result in 0 charges if there aren't
469                 //       enough copies to account for at least 1 fuel. Therefore we need
470                 //       to also bump by `FuelCosts::base` to charge at least 1 fuel.
471                 self.bump_fuel_consumption(fuel_info, FuelCosts::base)?;
472                 self.bump_fuel_consumption(fuel_info, |costs| {
473                     costs.fuel_for_copies(rest.len() as u64 + 3)
```

but several other call sites do not make this adjustment: the code to finalize function translation and for table-related operations don't adjust fuel.

`fuel_for_bytes` is only used in `memory` instructions. These instructions consume their fuel dynamically, without adjustment to account for the truncation. See `MemoryEntity::grow` for an example:

wasmi-labs/wasmi/crates/wasmi/src/memory/mod.rs

Line 294 to 307 in 02621ad

```
294          if let Some(fuel) = fuel {
295              let additional_bytes = additional.to_bytes().unwrap_or(usize::MAX) as u64;
296              if fuel
297                  .consume_fuel_if(|costs| costs.fuel_for_bytes(additional_bytes))
298                  .is_err()
299              {
300                  return notify_limiter(limiter,
EntityGrowError::TrapCode(TrapCode::OutOfFuel));
301              }
302          }
303          // At this point all checks passed to grow the linear memory:
304          //
305          // 1. The resource limiter validated the memory consumption.
306          // 2. The growth is within bounds.
307          // 3. There is enough fuel for the operation.
```

Recommendation

The most appropriate fix would be to modify the `costs_per` function such that it will round *up* instead of truncate the computed fuel value.

Status

Acknowledged by client

[C4] Large inputs to raw pointer functions may cause undefined behaviour depending on target

Severity: High | Recommended Action: Fix Code | Not addressed by client

Description

There are 4 calls to `*mut` or `*const` functions that have a common Safety comment. These calls are:

- engine/bytecode/instr_ptr.rs::InstructionPtr::offset calling <const* T>::offset with `count: isize`;
- engine/executor/stack.rs::FrameRegister::register_offset calling <mut* T>::offset with `count: isize`;
- engine/bytecode/instr_ptr.rs::InstructionPtr::add calling <const* T>::add with `count: usize`
- engine/executor/stack/values.rs::BaseValueStackOffset::stack_ptr_at calling <mut* T>::add with `count: usize`;

These calls in particular are common in that they both take an unbounded argument on the type that may violate the Safety comment of ():

> /// * The offset in bytes, `count * size_of::<T>()`, computed on mathematical integers (without
> /// "wrapping around"), must fit in an `isize`.

$$let\ \texttt{TYPE} = \texttt{InstructionPtr}|\texttt{UntypedVal}$$

$$\texttt{sizeof::<TYPE>} \times count \leq \texttt{isize::MAX}$$

$$count \leq \lfloor \frac{\texttt{isize::MAX}}{\texttt{sizeof::<TYPE>}} \rfloor$$

$$count \leq \lfloor \frac{2^{n-1}-1}{8} \rfloor$$

$$count \leq \lfloor \frac{2^{n-1}-1}{2^3} \rfloor$$

Calculating the maximum for `count` for each word size gives:

Case n = 8:

$$count \leq \lfloor \frac{2^{7}-1}{2^3} \rfloor$$

$$count \leq 15$$

Case n = 16:

$$count \leq \lfloor \frac{2^{15}-1}{2^3} \rfloor$$

$$count \leq 4095$$

Case n = 32:

$$count \leq \lfloor \frac{2^{31}-1}{2^3} \rfloor$$

$$count \leq 268435455$$

Case n = 64:

$$count \leq \lfloor \frac{2^{63}-1}{2^3} \rfloor$$

$$count \leq 1152921504606846975$$

Case n = 128:

$$count \leq \lfloor \frac{2^{127}-1}{2^3} \rfloor$$

$$count \leq 21267647932558653966460912964485513215$$

n-bit	`count` max	`count` max as integer	u`n` ::MAX	i`n` ::MAX
n	$\frac{\texttt{i<n>::MAX}}{2^3}$	...	$2^n - 1$	$2^{n-1} - 1$
8	$\frac{\texttt{i8::MAX}}{2^3}$	15	255	127
16	$\frac{\texttt{i16::MAX}}{2^3}$	4095	65535	32767
32	$\frac{\texttt{i32::MAX}}{2^3}$	268435455	4294967295	2147483647
64	$\frac{\texttt{i64::MAX}}{2^3}$	1152921504606846975	18446744073709551615	9223372036854775807
128	$\frac{\texttt{i128::MAX}}{2^3}$	21267647932558653966460912964485513215	340282366920938463463374607431768211455	170141183460469231731687303715884105727

It is possible to overflow for all functions, as the input is unbounded `isize` for `InstructionPtr::offset` and `FrameRegister::register_offset`, and the input is

unbounded `usize` for `InstructionPtr::add` and `ValueStack::stack_ptr_at`. With exception that FrameRegisters::register_offset is restricts `offset` to an `i16` no matter the architecture.

Recommendations

There should be a guard to ensure that the maximum value is not exceeded. If this is a performance issue it should be on `extra-checks` option.

Status

Acknowledged by client. Furthermore, there is active development to add postconditions and extra checks into Wasmi in this PR.

[CI1] Best Practices and Notable Particularities

Severity: Informative | Addressed by client

Description

Here are some notes on the protocol particularities, comments, and suggestions to improve the code or the design logic of the protocol in a best-practice sense. They do not in themselves present issues to the audited protocol but are advised to either be aware of or to be followed when possible, and may explain minor unexpected behaviors on the deployed project.

1. Panic comment on for engine/bytecode/utils.rs::from_source_to_dst does not actually occur in code.
2. Magic numbers in engine/code_map.rs::UncompiledFuncEntity::compile.
3. EngineFuncSpan relies on an invariant that `start <= end`, however it is possible to create an instance of this struct that violates that.
4. engine/executor/instrs.rs::get_entity! macro matches on a function with parameter `store: &StoreInner`, however this parameter is unused.
5. The way the `FunctionBody` gets deconstructed in `parse_buffered_code`, and then reconstructed in `FuncTranslationDriver::new` appears inconsistent: While "eager" translation uses an `offset` which is obtained from the original `FunctionBody`, the "lazy" translation uses an offset of `0`. The `offset` appears redundant.
6. The `Sync` instance for `Arena<Idx,T>` requires `T` to be `Send` (should be `Sync`). Given the instances used with `Arena`, this is without consequence at the moment.
7. Currently Wasmi is largely maintained by 1 person which presents a single point of failure. Should there be a pressing issue or update required of Wasmi and that person is unavailable, then it may be difficultly for other developers to address the issue.
8. In engine/executor/instrs/return_.rs there are functions of the form `execute_return*` which execute instructions of the form `Instructions::Return*`. Many of the comments for these functions incorrectly label which instruction is being executed. Here is one example.

Recommendations

For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Remove the comment, the client confirmed that the function should not panic if it would return `BranchOffset(0)`
2. Change these numbers to constants, or add them to the config.
3. Add a function to `EngineFuncSpan::new(start: EngineFunc, end: EngineFunc)` that constructs an `EngineFuncSpan` but with a guard that enforces `start <= end`. All constructions of EngineFuncSpan must happen through this function.
4. Remove the unused parameter
5. Investigate the de-facto value of the `offset` and remove it if redundant, possibly use original `FunctionBody` for translation.
6. Change `T: Send` to `T: Sync`
7. Increase the knowledge base of Wasmi to more people. A good start could be adding code review for PRs to Stellar / Soroban, so that they are now becoming increasingly familiar with the code as it evolves.
8. Correct the comments to the mislabeled instructions.

Status

1. Removed on `main` branch (`v.0.37.0` and above)
2. Fixed on `main` branch PR#1239
3. Fixed on `main` branch PR#1239
4. Changed on `main` branch (`v0.37.2` and above)
5. Fixed on `main` branch. PR#1241
6. Fixed on `main` branch PR#1239
7. Acknowledged. Furthermore, there are active efforts to increase the knowledge base of Wasmi to more people, and code review for Stellar / Soroban on PRs is considered.
8. Fixed on `main` branch PR#1239

[CI2] Results of `cargo-audit`

Severity: Informative | Addressed by client

Description

Rustsec tool `cargo-audit` was run and returned two warnings (RUSTSEC-2024-0375, RUSTSEC-2021-0145) related to one crate atty which is unmaintaned. Recommended fix is to use std::io::IsTerminal for Rust version `^1.70.0`. However this crate is a downstream dependency of wasi-cap-std-sync.

Recommendations

Change dependencies in such a way that avoids using the `atty` crate.

Status

This issue was already addressed by https://github.com/wasmi-labs/wasmi/pull/1140 for `main` and `v0.37.0+`. No fix is implemented for `0.36.x`, however since this is a `wasi` dependency it should not affect Stellar / Soroban.

[CI3] `visit_input_registers` does not visit all registers of a `RegisterSpan`

Severity: Informative | Recommended Action: Document Prominently | Not addressed by client

Description

The `visit_input_registers` trait provides a way to visit, and potentially modify, the input register references contained in instructions. However, the implementation is incomplete because in case of a `RegisterSpan`, only the first register of the span is visited and modified. It is not possible in general to implement a complete solution for `RegisterSpan` because the *length* of a `RegisterSpan` (i.e., how many and which registers are in fact addressed by it) is not known from the data type alone, but determined by the context in which the `RegisterSpan` is used.

For the use case of *defragmenting* the preservation space, the `visit_input_registers` implementation is sufficient because the entire register span will be moved when moving its first register.

The second call site of `visit_input_registers`, within `encode_local_set`, tries to determine whether a given register was used by an instruction. This will not detect usage when the given register is addressed as part of a `RegisterSpan`.

Recommendation

For future development on Wasmi, it should be documented that `visit_input_registers` is incomplete to avoid introducing bugs because of an assumption of completeness.

Status

Reported to the client, implications discussed. Because the given register is in the *preservation* space, it is believed that no cases exist where it could be part of a register span and therefore no fix is required.

[CI4] Missing `copy` in `select` translation

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

Description

The translation step for `select` relies on constant propagation to achieve an efficient encoding for the instruction.

For example, if the condition is a constant, then the selected value can just be pushed back onto the value stack. However, if the value is a dynamic or preservation register, in order to avoid overwriting the value later, a `copy` instruction is emitted (see Output mismatch between Wasmi and Wasmtime #2):

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/mod.rs

Line 2209 to 2220 in 8a1c6d8

```
2209                            // Case: constant propagating a dynamic or preserved register
might overwrite it in
2210                            //       future instruction translation steps and thus we may
require a copy instruction
2211                            //       to prevent this from happening.
2212                            let result = self.alloc.stack.push_dynamic()?;
2213                            let fuel_info = self.fuel_info();
2214                            self.alloc.instr_encoder.encode_copy(
2215                                &mut self.alloc.stack,
2216                                result,
2217                                selected,
2218                                fuel_info,
2219                            )?;
2220                            return Ok(());
```

If the two values represent the same register, a similar optimization applies. In this case however, dynamic and preservation registers are not special cased, and no `copy` instruction is emitted:

wasmi-labs/wasmi/crates/wasmi/src/engine/translator/mod.rs

Line 2229 to 2235 in 8a1c6d8

```
2229                    if lhs == rhs {
2230                        // # Optimization
2231                        //
2232                        // Both `lhs` and `rhs` are equal registers
2233                        // and thus will always yield the same value.
2234                        self.alloc.stack.push_register(lhs)?;
2235                        return Ok(());
```

Recommendation

It is not evident that this is a safe optimization. If it is, consider documenting it with comments. If it is not, the `copy` instruction should be emitted. Adding an assertion to enforce the desired behaviour is also recommended.

Status

Acknowledged by client.

Suggested Executor Assertions

Severity: Informative | Not addressed by client

The execution of some of the Wasmi instructions relies on assumptions on the translator's instruction output.
In some of these cases, assertions could be inserted into the executor code to double-check that the instructions adhere to the expected invariants (if desired, such assertions can be implemented with `debug_assert` to avoid performance penalties in release builds).

Copying between registers, assuming no overlap

The `CopySpanNonOverlapping` and `CopyManyNonOverlapping` are intended to be used when it can be guaranteed that while copying the values, no register is read from that has already been written to before (assuming a forward traversal of the registers).
Assuming no such overlap between the source and target register sets, values are copied directly without a temporary buffer.

- For `CopySpanNonOverlapping`, the function `has_overlapping_copies(_, _)` can be used to ensure at runtime that this is respected by the generated code.
- For `CopyManyNonOverlapping`, each of the source registers would have to be checked individually, by keeping a record of registers that have already been written to (in the respective code, the span starting at `results.head()` up to the current `result`. A similar check is implemented in the translator).

These can also be implemented as *static checks* (see Translation Output Validation) because the registers involved in the `Copy` instructions are known after translation.

`BranchTable` instruction assumes a valid sequence of `Branch` or `Return` instructions follows

The implementation of `BranchTable` increments the instruction pointer by the `index` value, assuming that it will point to a `Branch` or `Return` instruction afterwards. This could be checked by confirming the type of the instruction that `ip` points to after the adjustment. It can also be a *static* check (see Translation Output Validation) because the maximal length is known statically.

Check `InstructionPtr` (and `FrameRegister`) range to (dynamically) protect against overflow

The `offset` and `add` methods to adjust the `InstructionPtr` require the caller to ensure that the resulting instruction pointer always remains within the bounds of the current function. This could be checked within those methods by storing the valid range (in the `InstructionPtr` itself or in function call frames). This will however have a performance penalty because of the high frequency of instruction pointer changes. Therefore a check of the translated code before execution, as described in Translation Output Validation, is preferable, especially because all call sites of `InstructionPtr::add` and `InstructionPtr::offset` have statically-known or bounded arguments.

Similarly, `FrameRegister` could store the range to ensure that `register_offset` function does not overflow or underflow.

Status

These assertions are acknowledged. However their inclusion may occur in one of two different configurations, `extra-check`, or `translation post-conditions`. These configurations aim to create a modular approach to including higher security guarantees. `extra-checks` will be optional on production code to include stronger guarantees that only incur an insignificant or minor overhead. `translation post-conditions` are exclusively for `debug` compilation as the checks will incur a high performance cost. PR#1233 has active development for `translation post-conditions`.

Translation Output Validation

Severity: Informative | Not addressed by client

Many instruction implementations in Wasmi rely on assumed properties of the instruction sequence produced by the Wasm-to-Wasmi translator, which in turn relies on the Wasm bytecode being validated.
In order to ensure that the Wasmi bytecode does not crash during execution, some properties of the Wasmi bytecode could be double-checked. These properties can be either local to compiled (translated) functions, or refer to global properties that concern an entire *module* (including all its imports).

Instruction Sequence Validations

As soon as a Wasm function is *compiled* to Wasmi, its `CompiledFuncEntity` can be validated.

wasmi-labs/wasmi/crates/wasmi/src/engine/code_map.rs

Line 710 to 722 in 02621ad

```
710  pub struct CompiledFuncEntity {
711      /// The sequence of [`Instruction`] of the [`CompiledFuncEntity`].
712      instrs: Pin<Box<[Instruction]>>,
713      /// The constant values local to the [`EngineFunc`].
714      consts: Pin<Box<[UntypedVal]>>,
715      /// The number of registers used by the [`EngineFunc`] in total.
716      ///
717      /// # Note
718      ///
719      /// This includes registers to store the function local constant values,
720      /// function parameters, function locals and dynamically used registers.
721      len_registers: u16,
722  }
```

The sequence of `instrs` must have the following properties:

1. Some instructions require certain "instruction parameters" (encoded in the same `Instruction` type) to follow immediately. In turn, these instruction parameters (e.g., `Const32`, `TableIdx`, `Register`) never occur by themselves without preceeding context. Details are described in `wasmi::bytecode::Instruction` and in Appendix: Wasmi Instruction Set Overview.
    - This can be checked easily by a forward pass through the instruction sequence.

2. All instructions in the sequence should refer only to registers in the range allocated by `alloc_call_frame`, of length `len_registers`. However, registers are accessed using an offset of `consts.len()` from the (value) frame offset, i.e., accesses to constants inside a function use a *negative* register index (see `ValueStack`, `StackOffsets` and the `sp: FrameRegisters` field in `Engine`).
    - A forward pass through the instruction sequence can verify this for all instructions and instruction parameters that carry `Register` arguments.
    - However, instructions with `RegisterSpan` arguments will sometimes require consideration of the instruction context (e.g., `CopyMany`), as the *length* of the `RegisterSpan` is not part of this data structure.
    - For these instructions, the maximum register that is accessed typically depends on how many `RegisterList`, `Register`, `Register2`, or `Register3` instructions follow. This requires reading ahead in the instruction sequence.

3. All branch targets must stay within the function's instruction sequence.
   The Wasm instruction set uses the concept of *structured control instructions* and implicit *labels* to specify branch targets for branch instructions. Wasmi translation resolves these labels to integer `offsets` from the current instruction pointer. The Wasm `If` instruction is also compiled to a (fused) Wasmi branch instruction. The generated `offset`s in branch instructions are not allowed to move the instruction pointer outside the range of the current function's instructions.
    - This can be implemented by a forward pass through the instructions considering the relative position of the instruction in the `instrs` array.
    - For each `offset` found in an instruction, `offset + position` must be between `[0..instrs.len()]`.
    - *Branch tables* in Wasmi are built from sequences of branch instructions which do not require special treatment. The `BranchTable` instruction itself performs a forward jump of dynamic but *bounded* size and can be checked using the size bound.

Checking References to Wasm Store Objects Contained in Instructions

Besides these local properties, the instruction sequence contains references to objects from the Wasm store: tables, memories, data segments, globals, and other functions. In order to ensure

that no instruction crashes, it must be ensured that this store indeed contains the referenced entity for all relevant instructions:

- **Table indexes** within `CallIndirectParams`, `TableSize`, and of course `TableIdx`, must refer to existing tables;
- **Data segment indexes** within `DataSegmentIdx` and `DataDrop`, must refer to existing data segments;
- **Element segment indexes** within `ElementSegmentIdx` and `ElemDrop`, must refer to existing data segments;
- **Function indexes** within `RefFunc`, must refer to existing (compiled or uncompiled) functions.

These properties can be checked using the function's respective `ModuleHeader`.

wasmi-labs/wasmi/crates/wasmi/src/module/mod.rs

Line 68 to 86 in 02621ad

```rust
pub struct ModuleHeader {
    inner: Arc<ModuleHeaderInner>,
}

#[derive(Debug)]
struct ModuleHeaderInner {
    engine: EngineWeak,
    func_types: Arc<[DedupFuncType]>,
    imports: ModuleImports,
    funcs: Box<[DedupFuncType]>,
    tables: Box<[TableType]>,
    memories: Box<[MemoryType]>,
    globals: Box<[GlobalType]>,
    globals_init: Box<[ConstExpr]>,
    exports: Map<Box<str>, ExternIdx>,
    start: Option<FuncIdx>,
    engine_funcs: EngineFuncSpan,
    element_segments: Box<[ElementSegment]>,
}
```

All indexed entities are already present in the module header, although they are *populated* during module instantiation. The Wasm validation already checks all entity indexes before translation to Wasmi, so checking the existence of respective entities in Wasmi data structures duplicates these checks.

Checking that function calls have the correct arity (arg. count)

`Call*` instructions (and `ReturnCall*` variants for tail call optimization) provide function arguments in subsequent `Register*` instructions.

For each function call, it should be checked that the function is called with the correct amount of arguments. The amount of arguments (arity) cannot easily be determined from the `CompiledFuncEntity`. Besides the argument count, the `len_registers` includes both a (known) amount of constants (`consts`) and an amount of registers for internal use.
The actual arity is bounded by `len_registers - consts.len()`. Checking that no more than `len_registers - consts.len()` arguments are provided ensures that the call to `copy_call_params` (`call.rs: 241`) cannot overflow the allocated value stack frame.
More precise arity information could be obtained from the `ModuleHeader`, which contains the types of all functions in the module.

Status

Planned, Add debug post-conditions for Wasmi translation created.

Fuzzing Discussion and Findings

This section communicates specific results and findings for the fuzzing runs performed over the duration of the audit. The fuzzing was performed in accordance with description given in section Methodology - see there for a more in-depth explanation motivating our approach and goals for the fuzzing campaign. The end of this section contains detailed information about all identified issues.

Time and Compute Resources

Because fuzzing is a randomized, feedback-driven process which gives better results the more iterations are run, with modern fuzzing tools also able to scale near linearly with the number of available cores, additional compute power and time significantly increases the likelihood that a vulnerability is identified. Towards that end, we allocated two powerful 16-core / 32-thread machines for the fuzzing campaign:

- An AMD Ryzen Threadripper 1950X with 64GB of RAM
- An AMD Ryzen 9 7950X with 128 GB RAM

After an initial 1.5 week period spent familiarizing with the codebase, we ran fuzzers on these machines near continuously for the remaining audit duration, yielding approximately 13 machine-weeks of total fuzzing time. The actual total duration is likely higher, as we also performed numerous short exploratory runs on other machines while these longer-running fuzzers were still executing.

As described in Methodology, our approach focused on end-to-end testing of the entire interpretation pipeline for crash-freedom and conformance with other Wasm implementations, homing in on thoroughly testing a small number of targets rather than distributing compute resources too thinly. The table below provides the approximate machine-time spent on each target, with descriptions of each target provided in the following sections.

Target	Machine-Days	Version	Bugs
Translation	14	v0.36.0	1
Execution	26	v0.36.x	2
Execution	8	v0.37.x	2
Differential	30	v0.36.x	3
Performance	13	v0.36.x	0

Harnesses

For the standard, coverage-guided fuzzing which seeks to find interesting inputs by exploring as many control-flow edges as possible, we built off of Wasmi's existing harnesses.

As an initial test case to evaluate our machine configuration and choice of fuzzing toolings, we ran the Wasmi translator harness unmodified on both machines for approximately 1 week. This translator harness simply generates random Wasm modules and calls the translator on them, with no specific testing assertions other than the invariant that no crashes occur. While this did reveal one bug, we decided not to dedicate further time to this harness after this period because the other harnesses already adequately cover these same code paths, while also hitting the more `unsafe`-dense parts of codebase with the executor where vulnerabilities are likely to occur.

We then focused primarily on the two other harnesses, both of which generate a random Wasm module using wasm-smith then call each exported function:

- An executor harness, which makes no explicit test assertions, but which we compile with sanitizer instrumentation and debug assertions enabled to identify crashes, failed assertions, and memory safety violations.
- A differential fuzzing harness, which checks results against wasmtime as well as a prior stack-based version of Wasmi.

We iteratively refined these harnesses as described in Methodology, identifying the best configurations for both the harness and the fuzzing tool to achieve maximal coverage. This process is guided by intuition stemming from knowledge of the fuzzer internals, along with empirical testing using numerous shorter fuzzing runs to evaluate possible changes, noting factors such as the rate of coverage increase, the breadth of coverage (i.e. line and function-based coverage), the depth of coverage (i.e. branch and path-based coverage), and the time for each test iteration. Because of the exploratory nature and abundant number of shortly-tested variations, in lieu of specific quantitative data, we provide a qualitative account highlighting the most effective modifications found. These modifications have since been upstreamed into Wasmi.

The wasm-smith configuration controls which Wasm features are available when generating inputs from a given `&[u8]` seed, with parameters to set the minimum or maximum numbers of various syntactic elements. In the existing harnesses, reasonable defaults were set for this

configuration with all supported features enabled. One of the most effective changes we made was to instead switch to a swarm-testing approach, where the wasm-smith configuration is itself dynamically produced from the input seed rather than fixed across all inputs. With a fixed configuration, where all possible features are enabled, the numerous features compete against each other for limited space within a single generated input, leading to each individual feature only being exercised shallowly. With swarm testing, however, only a smaller subset of features is enabled at any given time. This allows each of those features to be more thoroughly represented within a single generated input, while the random selection of the enabled subset still ensures a breadth of feature coverage across multiple inputs.

In the original executor harness, each randomly generated function is always called with a value of `1` for every argument, with the client's hope being that mutation of the function body itself would provide sufficient coverage despite these statically fixed inputs. In practice though, we found that randomizing the inputs improved the overall coverage, both in terms of it's depth and the number of iterations required to discover deeper control-flow paths. Crucially, the arguments are generated using bytes from areas of the seed unconsumed by wasm-smith's generation process.

At a high level, the improvements here make sense because randomizing the arguments provides more opportunities for small mutations in the seed to produce correspondingly small changes in the interpreter control-flow, improving the fuzzer's ability to iteratively work towards producing edge-case inputs needed to hit deeper control flow paths. More explicitly, control-flow paths are often guarded by comparisons requiring a particular edge-case value - checksums, maximums, minimums, zeros, NaNs, etc. Honggfuzz attempts to find such values by recording the Hamming distance between arguments of each `cmp` assembly instruction, preferentially exploring inputs which shorten this distance for any particular `cmp` and thus are more likely to change the comparison result and any dependent control-flow. Although wasm-smith makes an effort to ensure small changes in the input seed produce small changes in the generated Wasm module, there are many cases where this fails in practice - altering a single decision point in the generation logic, e.g. the type of an argument, can radically alter the generated module. Speculatively, this makes honggfuzz's `cmp` feedback mechanism less effective, as the entire path leading to a particular `cmp` changes with each mutation if it is even reached at all, rather than allowing some mutations which keeping that path relatively fixed and only alter the data under comparison. By also randomizing the function arguments with the remaining un-consumed parts of the seed, some mutations end up fixing the generated function body and

only altering the generated argument value, still preserving the bulk of the interpreter control-flow while moving closer to the desired `cmp` result, making it easier to achieve greater coverage depth.

The original executor harness also relied on wasm-smith's `ensure_termination` functionality to prevent the generation of infinite loops, which we replaced with Wasmi's built-in fuel metering. This of course improved coverage for the fuel metering code, as well as iteration time due to a more efficient implementation, but interestingly, it also seemed to decrease the number of iterations needed to achieve a given level of coverage breadth. Speculatively, this is because `ensure_termination` only inserts fuel metering at function headers and loop bodies, so for a fixed fuel cost, more iterations are spent exploring seeds near a local-maxima wherein generating additional instructions in the un-metered, straight line sections of code marginally increases coverage or total instruction count (which is also considered as part of honggfuzz's metric) without actually building towards more interesting edge-cases. With Wasmi's built-in fuel metering, which instead applies a cost for each instruction, extending the straight line sections of code comes at the cost of reducing complexity elsewhere, making it less likely that fuzzers considered these inputs interesting

For the differential fuzzing harness, the original harness had nondeterministic failures where one of compared Wasm interpreters ran out of stack space or register allocations due to differing resource limits as permitted by the Wasm spec. This render the results uninteresting, as it's difficult to differentiate actual issues versus this sort of false positive. For our fuzzing campaign, we then modified the differential target to pass on these nondeterministic cases instead of panicking, isolating any panics to the cases where all of the executors finish successfully but have different outputs. We also modified the wasm-smith configuration for this target to generate Wasm modules that look like the kind of modules that will be accepted by the Soroban client, e.g. disabling multi-value support, floating points, threads, etc.

We additionally attempted to develop a performance-focused fuzzer, seeking to maximize execution-time-per-unit-fuel and identify possible DoS vectors. This included both a fairly simple modification to honggfuzz, swapping the order of consideration for instruction count versus coverage updates when selecting previously-run inputs as a base for new inputs, as well as a more involved custom fork of honggfuzz to develop an approach based on PerfFuzz. The latter PerfFuzz-inspired fuzzer is still under development, and unfortunately some bugs in the fuzzer implementation were not identified until after the runs performed for this audit, which likely invalidates the efficacy of those attempts. No new vulnerabilities were revealed by either of

these performance-based fuzzing efforts, although it must be emphasized that the experimental nature of this approach means that a lack of findings doesn't provide any guarantees regarding the absence of relevant issues.

Identified Issues

In the rest of this section, we present all the issues identified by our fuzzing runs. In total, we found 8 violations of expected behavior:

- 3 silently incorrect outputs
- 2 segfaults or memory corruption issues
- 1 hang
- 1 panic due to hitting code marked unreachable
- 1 failed debug assertion

For each issue, we report the versions of Wasmi that are affected, a brief description of the actual vs expected behavior, a test case which triggers the bug, as well as links to any fixes made by the client and the relevant releases. At the time of writing, all issues identified have been patched in the most recent `v0.36.x` and `v0.37.x` release.

As with other findings in this report, we include a severity level ranging from low to high. One subtlety worth highlighting is that a few of the identified issues require the `reference-types` or `multi-value` Wasm proposals, which are disabled by present-day Soroban. For these issues, if they are isolated and seem to fundamentally require the unsupported proposals, we assign low severity regardless of other considerations. However, if they instead seem to be indicative of a more general design risk, with a likelihood of there being yet-undiscovered analogous vulnerabilities which do not require the unsupported proposals, we assign a severity as if the issue applied to the Wasm MVP.

Note, however, that such issues marked low severity due to requiring `reference-types` or `multi-value` were still important to address. Soroban may decide to extend their supported Wasm features in the future as these proposals become more established, especially given that they were recently enabled by default for the Rust compiler's code generation.

[F1] Abort on `realloc()` due to faulty `br_table` optimization

Severity: High | Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🆗, v0.36.2 🆗, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗

A Wasm program causes Wasmi to abort during a `realloc()`.

Program

`realloc.wasm`:

```
(module
  (func (;0;)
    call 1
    drop
    drop
    drop
    drop
    drop
    drop
    drop
    drop
    drop
    call 0)
  (func (;1;) (result i64 i64 i64 i64 i64 i64 i64 i64 i64)
    (local i32)
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    i64.const 0
    local.get 0
    br_table 0 0)
  (export "" (func 0)))
```

Behavior

```
$ wasmi_cli realloc.wasm
realloc(): invalid next size
Aborted
```

Status

Fixed for 0.36 versions with a646d27 included in release v0.36.1. Does not occur in 0.37 versions due to changes in the `br_table` encoding which inadvertently addressed the issue.

Severity

This issue is marked as high severity because:

- It is a critical memory error where the heap has been corrupted resulting in an out-of-bounds write.
- Although this particular issue is only reproducible with the `multi-value` proposal, which is disabled by present-day Soroban, it is indicative of a more generally risky design that is unrelated to `multi-value`.

Explicitly, one of the contributing factors for this issue is that `RegSpan` does not record its own length, requiring it to be maintained correctly elsewhere lest a memory error occurs. It's hard to guarantee no other similar vulnerabilities are present, and another issue related to this design was already discovered in `visit_input_registers` does not visit all registers of a `RegisterSpan`.

[F2] Translator debug assertion fired due to `ref.is_null` constant propagation

Severity: Low | Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🆗, v0.36.2 🆗, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗
Unreleased: b48685f 🐞

A Wasm program causes the type-checking debug assertion in `TypedVal::i64_eq` to fire.

Program

`assert.wat`:

```
(module
  (func (result i32)
    ref.null func
    ref.is_null
  )
)
```

Behavior

```
$ wasmi_cli --invoke '' --compilation-mode lazy assert.wat
thread 'main' panicked at wasmi-0.36.0/src/engine/translator/typed_value.rs:180:5:
assertion failed: matches!(self.ty(), < i64 as Typed > :: TY)
```

Status

Fixed for 0.36 versions with f09d121 included in release v0.36.1. Fixed for 0.37 versions with 78788f9 prior to the release of v0.37.0.

Severity

This issue is marked as low severity because:

- It only occurs with debug assertions enabled.
- Although the fired assertion does indicate an overlooked case in the design, the actual runtime behavior would still be correct if the assertion were simply removed.

Namely, the root cause is that `ref.is_null` delegates to the translation logic for `i64.eqz`, taking advantage of the fact that Wasmi serializes `null` as a 64-bit `0`. However, during constant propagation, this calls `TypedVal::i64_eq` which contains a strict type checking assertion requiring both sides to be a `ValType::I64` before comparing them as `Untyped`. The assertion fails because a `FuncRef` is not literally a `ValType::I64`, even though the actual comparison as `Untyped` would still be semantically correct.

[F3] Segmentation fault due to `visit_input_regs` bug

Severity: Low | Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🐞, v0.36.2 🐞, v0.36.3 🐞, v0.36.4 🐞, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗

A Wasm program causes a segmentation fault during execution of `table.get 0`.

Program

`segv.wat`:

```
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    (local i32)
    local.get 0
    i32.const 0
    local.set 0
    table.get 0
    drop
  )
  (table (;0;) 1 2 funcref)
  (export "" (func 0))
)
```

Behavior

```
$ wasmi_cli segv.wat
zsh: segmentation fault (core dumped)  wasmi_cli bad.wat
```

Status

The root cause is that `visit_input_regs` fails to visit the input register for `Instruction:TableGet::index`, so the preserved register input to `Instruction::TableGet (index)` is not defraged at the end of translation.

Fixed for 0.36 versions with 82c9388 included in release v0.36.5. Does not occur in 0.37 versions due to a refactoring which allows these visitors to be generated automatically, exactly to avoid this sort of bug.

Severity

This issue is marked as low severity because:

- It only occurs with the `reference-types` proposal, which is disabled by present-day Soroban.
- Although the issue is a critical memory error, it is an isolated typo with no indication of broader risk to the MVP or present-day Soroban.

[F4] Output mismatch between Wasmi and Wasmtime #1

Severity: Low | Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🆗, v0.36.2 🆗, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗
Unreleased: 1e5a4ba 🐞

A Wasm program behaves differently on the Wasmi register-based executor when compared to both the Wasmi stack-based executor and wasmtime.

Program

`diff.wat` :

```
(module
  (func (export "") (param i32) (result i32 i32 i32 i32)
    local.get 0
    local.get 0
    block (param i32 i32)
      local.tee 0
      block (param i32 i32)
        local.get 0
        local.get 0
        br 2 ;; returns
      end
    end
    unreachable
  )
)
```

Behavior

Wasmi and Wasmtime report different outputs.

```
$ wasmi_cli diff.wat 1
[0, 1, 1, 1]
$ wasmtime diff.wat 1
1
1
1
1
```

Status

Fixed for 0.36 versions with 415a919 included in release v0.36.1. Fixed for 0.37 versions with e9c6acf prior to the release of v0.37.0.

Severity

This issue is marked as low severity because:

- It only occurs with the multi-value proposal, which is disabled by present-day Soroban.
- It is an isolated issue with no indication of broader risk for to the MVP or present-day Soroban.

[F5] Output mismatch between Wasmi and Wasmtime #2

Severity: High Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🐞, v0.36.2 🐞, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🐞, v0.37.1 🆗, v0.37.2 🆗

A Wasm program behaves differently on the Wasmi register-based executor when compared to both the Wasmi stack-based executor and wasmtime.

Program

`diff.wat` :

```
(module
  (func (export "test") (param i32) (result i32)
    (i32.popcnt (local.get 0))        ;; case: true  (i32.const  0)
    (i32.clz (i32.eqz (local.get 0))) ;; case: false (i32.const 31)
    (i32.const 0)                     ;; condition   (i32.const  0)
    (select)                          ;; case: true  (i32.const 31)
    (i32.const 0)                     ;; case: false (i32.const  0)
    (i32.eqz (local.get 0))           ;; condition   (i32.const  1)
    (select)
  )
)
```

Behavior

```
$ wasmtime diff.wat
warning: using `--invoke` with a function that returns values is experimental and may break
in the future
31
$ wasmi_cli diff.wat
1
```

Status

Fixed for 0.36 versions with 15a3802 included in release v0.36.3. Fixed for 0.37 versions with 8ed9469 included in release v0.37.1.

Severity

This issue is marked as high severity because it affects the Wasm MVP as supported by Soroban and can silently result in arbitrarily incorrect results.

[F6] Output mismatch between Wasmi and Wasmtime #3

Severity: High | Addressed by client

Context

Version 0.36: v0.36.0 🐞, v0.36.1 🐞, v0.36.2 🐞, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🐞, v0.37.1 🐞, v0.37.2 🆗

A Wasm program behaves differently on the Wasmi register-based executor when compared to both the Wasmi stack-based executor and wasmtime.

Program

`diff.wat`:

```
(module
    (func (export "") (param i32) (result i32)
        (local.set 0 (i32.const 0))
        (local.get 0)
        (loop $continue
            (if (i32.eqz (local.get 0))
                (then
                    (local.set 0 (i32.const 1))
                    (br $continue)
                )
            )
        )
    )
)
```

Behavior

```
$ wasmtime diff.wat 1
0
$ wasmi_cli diff.wat 1
1
```

Status

Fixed for 0.36 versions with 5859e15 included in release v0.36.4. Fixed for 0.37 versions with ddc8e5e included in release v0.37.2.

Severity

This issue is marked as high severity because it affects the Wasm MVP as supported by Soroban and can silently result in arbitrarily incorrect results.

[F7] Executor hang on unreleased version due to `copy_span` bug

Severity: Low | Addressed by client

Context

Version 0.36: v0.36.0 🆗, v0.36.1 🆗, v0.36.2 🆗, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗
Unreleased: 17a4242 🐞

A Wasm program causes the executor to incorrectly hang, independently of fuel metering.

Program

`hang.wat` :

```
(module ;; hangs on main branch
  (func (export "") (result i32 i32 i32)
    (local i32 i32 i32)
    i32.const 0
    (block (result i32 i32 i32) ;; label = @1
      local.get 0
      local.get 1
      local.get 2
      (block
        ;; The next two instructions together cause an integer-overflow trap.
        f64.const 0x1.b1ddf4040cd22p+901
        i32.trunc_f64_u
        drop
      )
    )
    drop
  )
)
```

Behavior

Wasmi v0.36.0 correctly reports an integer overflow

```
$ wasmi_cli --invoke '' --compilation-mode lazy --fuel 10000 hang.wat
Error: failed during execution of : integer overflow
```

but the same invocation hangs when tested on commit 17a4242.

Status

Does not occur in 0.36 versions. Fixed for 0.37 versions with 1e5a4ba prior to the release of v0.37.0.

Note that this bug only occurred mid-development and was never published in a released version of Wasmi.

Severity

This issue is marked as low severity because:

- It only occurs with the multi-value proposal, which is disabled by present-day Soroban.
- It is an isolated issue with no indication of broader risk for to the MVP or present-day Soroban.

[F8] Executor panic on unreleased version due to `br_table_many` bug

Severity: Low | Addressed by client

Context

Version 0.36: v0.36.0 🆗, v0.36.1 🆗, v0.36.2 🆗, v0.36.3 🆗, v0.36.4 🆗, v0.36.5 🆗
Version 0.37: v0.37.0 🆗, v0.37.1 🆗, v0.37.2 🆗
Unreleased: a101c50 🐞

A Wasm program causes the executor to panic.

Program

`crash.wat` :

```
(module
  (func (result i32 i32 i32 i32)
    i32.const 1
    i32.const 0
    i32.const 1
    i32.const 0
  )
  (func (export "") (result i32 i32 i32 i32)
    (block (result i32 i32 i32 i32) ;; label = @1
      i32.const 0
      call 0
      br_table 0 1 1
    )
  )
)
```

Behavior

Wasmi v0.36.0 correctly runs the example

```
$ wasmi_cli --invoke '' --compilation-mode lazy crash.wat
[0, 1, 0, 1, 0]
```

but the same invocation panics with commit a101c50

```
$ wasmi_cli --invoke '' --compilation-mode lazy crash.wat
thread 'main' panicked at crates/wasmi/src/engine/executor/instrs/return_.rs:252:27:
internal error: entered unreachable code: unexpected `Instruction` found while executing
`Instruction::ReturnMany`: BranchTableTarget { results: RegSpan(Reg(6)), offset:
BranchOffset(3) }
```

Status

Does not occur in 0.36 versions. Fixed for 0.37 versions with cc02394 prior to the release of v0.37.0.

Note that this bug only occurred mid-development and was never published in a released version of Wasmi.

Severity

This issue is marked as low severity because:

- It only occurs with the `multi-value` proposal, which is disabled by present-day Soroban.
- It is an isolated issue with no indication of broader risk for to the MVP or present-day Soroban.

Appendix

This appendix contains additional technical documentation produced during the audit, as well as an overview of all `unsafe` Rust code inspected as part of the code review.

Appendix: Wasmi Instruction Set Overview

Wasmi Instruction Set Description

Source: Wasmi source code `wasmi::engine::bytecode::Instruction` (v0.36.0)

The instructions of Wasmi byte code are declared and described in the `bytecode` module. This is a reorganised overview of all instructions, categorised in a manner similar to how Wasm instructions are presented in the Wasm specification.

- Special purpose instructions - not present in Wasm
- Numeric instructions
- Vector instructions: None
- Reference instructions
- Parametric instructions
- Variable instructions
- Table instructions
- Memory instructions
- Control instructions

Generally speaking, the Wasmi instruction set contains more variants of the instructions found in Wasm. These variants are needed because Wasmi uses *registers* for local variables; therefore many instruction exists in a variant for *immediate* values (constants) as well as with registers. Also, most instructions that manipulate data require source and destination registers as parameters.

Special-Purpose Instructions

Instruction Parameters for Other Instructions

`Const32(value)`, `I64Const32(value)`, and `F64Const32(value)` are used to provide 32 bit encoded constants (of type `I32`, `I64`, or `F64`, respectively) to preceeding instructions.

To pass a *list of registers* (of arbitrary length) to a preceeding instruction, the following instruction parameters are used:

Instruction	Contents
`Register(r0)`	1 register
`Register2(r0,r1)`	2 registers
`Register3(r0,r1,r2)`	3 registers
`RegisterList(r0,r1,r2)`	3 registers, indicating that more registers follow

In order to provide more than 3 registers, a number of `RegisterList` instructions is followed by a final `Register`, `Register2` or `Register3` one.

For indexes that refer to tables, data segments, or element segments, there are specialised instruction parameters:

Instruction	Contents
`TableIdx(idx)`	A table index, consisting of 4 8-bit unsigned integral numbers
`DataSegmentIdx(idx)`	A data segment index (32-bit unsigned integral number)
`ElementSegmentIdx(idx)`	A data segment index referring to an element segment

Copying and Filling Registers

Wasmi requires utility instructions to copy values between registers or write immediate values into registers.

These instructions typically use a `RegisterSpan` of contiguous registers as the copy target, unless it is a single register (`r0`).

Instruction	Description
`Copy(r0,r1)`	copies contents of `r1` to `r0`
`Copy2(span,r1,r2)`	copies `r1` and `r2` to the given target `span`
`CopyImm32(r0,value)`	copies immediate 32-bit `value` to `r0`
`CopyI64Imm32(r0,value)`	copies immediate 32-bit `value` to `r0` as an `I64`
`CopyF64Imm32(r0,value)`	copies immediate 32-bit `value` to `r0` as an `F64`
`CopySpan(span,span2,len)`	copies `len` values from source `span2` to target `span`
`CopyMany(span,[r1,r2])`	copies more than 2 registers (2 in instruction, more following) to target `span`. Must be followed by a *list of instructions* as instruction parameters

`CopySpan` and `CopyMany` also have variants `CopySpanNonOverlapping` and `CopyManyNonOverlapping`. These variants assume that the given target registers are not read (as source registers) after having been written to, when performing the copy operation in a forward pass through the spans.

Fuel Consumption (if enabled)

Wasmi supports measuring execution effort by a special instruction `ConsumeFuel(fuel)` which carries an unsigned 32-bit `BlockFuel` integral number, described as a resource measure of executing a *basic block*.

Numeric Instructions

As in Wasm, arithmetic and other numeric instructions are grouped by underlying *numeric types* (`I` for integral numbers vs. `F` for floating-point decimals), as well as by the respective size in bits (32 or 64 in the instruction set). This numeric type prefix can be `I32` , `I64` , `F32` or `F64` .

Likewise, variants for signed or unsigned interpretation of the arguments exist equivalently to the ones in Wasm (infix `S` for signed or `U` for unsigned).

NB the description here is simplified in that it does not consider `NaN` or `Infinity` arguments. The full semantics of the operations on `F32` and `F64` when applied to `NaN` or `Infinity` values is intended to be the same as in Wasm but not reproduced here.

Comparison instructions take three arguments, commonly *registers* (denoted `r0` , `r1` , and `r2` below). The return value is stored in `r0` .

`I32` and `I64` comparisons have variants with `Imm16` suffix, which contain an immediate 16-bit encoded value instead of `r2` to serve as the second argument.

Instruction	Description
`I<SIZE>32Eq(r0,r1,r2)`	`r0 <- r1 == r2` Variants: `I<SIZE>EqImm16(r0,r1,imm)`
`I<SIZE>32Ne(r0,r1,r2)`	`r0 <- r1 != r2` Variants: `I<SIZE>NeImm16(r0,r1,imm)`
`I<SIZE>32LtS(r0,r1,r2)`	`r0 <- r1 < r2` (signed) Variants: `I<SIZE>LtSImm16(r0,r1,imm)`
`I<SIZE>32LtU(r0,r1,r2)`	`r0 <- r1 < r2` (unsigned) Variants: `I<SIZE>LtUImm16(r0,r1,imm)`
`I<SIZE>32GtS(r0,r1,r2)`	`r0 <- r1 > r2` (signed) Variants: `I<SIZE>GtSImm16(r0,r1,imm)`
`I<SIZE>32GtU(r0,r1,r2)`	`r0 <- r1 > r2` (unsigned) Variants: `I<SIZE>GtUImm16(r0,r1,imm)`
`I<SIZE>32LeS(r0,r1,r2)`	`r0 <- r1 <= r2` (signed) Variants: `I<SIZE>LeSImm16(r0,r1,imm)`
`I<SIZE>32LeU(r0,r1,r2)`	`r0 <- r1 <= r2` (unsigned) Variants: `I<SIZE>LeUImm16(r0,r1,imm)`
`I<SIZE>32GeS(r0,r1,r2)`	`r0 <- r1 >= r2` (signed) Variants: `I<SIZE>GeSImm16(r0,r1,imm)`
`I<SIZE>32GeU(r0,r1,r2)`	`r0 <- r1 >= r2` (unsigned) Variants: `I<SIZE>GeUImm16(r0,r1,imm)`

where `<SIZE>` is either `32` or `64` .

For floating point decimals, there are no variants for immediate arguments and no unsigned interpretation exists. Consequently, there are fewer instructions.

Instruction	Description
`F<SIZE>Eq(r0,r1,r2)`	`r0 <- r1 == r2`
`F<SIZE>Ne(r0,r1,r2)`	`r0 <- r1 != r2`
`F<SIZE>Lt(r0,r1,r2)`	`r0 <- r1 < r2`
`F<SIZE>Le(r0,r1,r2)`	`r0 <- r1 <= r2`
`F<SIZE>Gt(r0,r1,r2)`	`r0 <- r1 > r2`
`F<SIZE>Ge(r0,r1,r2)`	`r0 <- r1 >= r2`

where `<SIZE>` is either `32` or `64`.

Unary operations take two registers, `r0` and `r1`, as arguments, and store the return value in `r0`.

Instruction	Description
`I<SIZE>Clz(r0,r1)`	`r0 <-` amount of leading zeros in the binary representation of `r1`
`I<SIZE>Ctz(r0,r1)`	`r0 <-` amount of trailing zeros in the binary representation of `r1`
`I<SIZE>Popcnt(r0,r1)`	`r0 <-` amount of `1` s in the binary representation of `r1`

where `<SIZE>` is either `32` or `64`.

Instruction	Description (simplified)
`F<SIZE>Abs(r0,r1)`	`r0 <-` absolute value of FP decimal value in `r1`
`F<SIZE>Neg(r0,r1)`	`r0 <- -r1`
`F<SIZE>Ceil(r0,r1)`	`r0 <-` smallest integral number larger than FP decimal value in `r1`
`F<SIZE>Floor(r0,r1)`	`r0 <-` largest integral number smaller than FP decimal value in `r1`
`F<SIZE>Trunc(r0,r1)`	`r0 <-` `r1` truncated towards zero
`F<SIZE>Nearest(r0,r1)`	`r0 <-` `r1` rounded towards the nearest integral, preferring *even* numbers
`F<SIZE>Sqrt(r0,r1)`	`r0 <-` square root of value in `r1`

where `<SIZE>` is either `32` or `64`.

Binary Operations take 3 registers arguments, commonly *registers* (denoted `r0`, `r1`, and `r2` below). The return value is always stored in `r0`.

As for the comparison operations, binary operations for integral numbers come with variants (with `Imm16` suffix) where the register argument `r2` is replaced by an *immediate* 16-bit encoded value. In addition, the variants with suffix `Imm16Rev` exist for the non-commutative operations, which replace the register argument `r1`.

Again, `<SIZE>` is either `32` or `64` in all tables in this subsection.

Instruction	Description
`I<SIZE>Add(r0,r1,r2)`	`r0 <- r1 + r2` Variants: `I<SIZE>AddImm16(r0,r1,imm)`
`I<SIZE>Sub(r0,r1,r2)`	`r0 <- r1 - r2` Variants: `I<SIZE>SubImm16Rev(r0,r1,imm)`
`I<SIZE>Mul(r0,r1,r2)`	`r0 <- r1 * r2` Variants: `I<SIZE>MulImm16(r0,r1,imm)`
`I<SIZE>DivS(r0,r1,r2)`	`r0 <- r1 / r2` (signed truncated) Variants: `I<SIZE>DivSImm16(r0,r1,imm)`, `I<SIZE>DivSImm16Rev(r0,imm,r2)`
`I<SIZE>DivU(r0,r1,r2)`	`r0 <- r1 / r2` (unsigned truncated) Variants: `I<SIZE>DivUImm16(r0,r1,imm)`, `I<SIZE>DivUImm16Rev(r0,imm,r2)`
`I<SIZE>RemS(r0,r1,r2)`	`r0 <- r1 % r2` (signed truncated) Variants: `I<SIZE>RemSImm16(r0,r1,imm)`, `I<SIZE>RemSImm16Rev(r0,imm,r2)`
`I<SIZE>RemU(r0,r1,r2)`	`r0 <- r1 % r2` (unsigned truncated) Variants: `I<SIZE>RemUImm16(r0,r1,imm)`, `I<SIZE>RemUImm16Rev(r0,imm,r2)`

Instruction	Description
`I<SIZE>And(r0,r1,r2)`	`r0 <- r1 & r2` (bitwise) Variants: `I<SIZE>AndImm16(r0,r1,imm)`
`I<SIZE>Or(r0,r1,r2)`	`r0 <- r1 | r2` (bitwise) Variants: `I<SIZE>OrImm16(r0,r1,imm)`
`I<SIZE>Xor(r0,r1,r2)`	`r0 <- r1 ^ r2` (bitwise) Variants: `I<SIZE>XorImm16(r0,r1,imm)`

There are special *fused* instructions combining bitwise operations and test whether the result is zero:

Instruction	Description
`I<SIZE>AndEqz(r0,r1,r2)`	`r0 <- 0 == r1 & r2` (fused & and test) Variants: `I<SIZE>AndEqzImm16(r0,r1,imm)`
`I<SIZE>OrEqz(r0,r1,r2)`	`r0 <- 0 == r1 | r2` (fused | and test) Variants: `I<SIZE>OrEqzImm16(r0,r1,imm)`
`I<SIZE>XorEqz(r0,r1,r2)`	`r0 <- 0 == r1 ^ r2` (fused ^ and test) Variants: `I<SIZE>XorEqzImm16(r0,r1,imm)`

Instruction	Description
`I<SIZE>Shl(r0,r1,r2)`	`r0 <- r1 << r2` (logical) Variants: `I<SIZE>ShlImm(r0,r1,imm)`, `I<SIZE>ShlImm16Rev(r0,imm,r2)`
`I<SIZE>ShrU(r0,r1,r2)`	`r0 <- r1 >> r2` (logical unsigned) Variants: `I<SIZE>ShrUImm(r0,r1,imm)`, `I<SIZE>ShrUImm16Rev(r0,imm,r2)`
`I<SIZE>ShrS(r0,r1,r2)`	`r0 <- r1 >> r2` (logical signed) Variants: `I<SIZE>ShrSImm(r0,r1,imm)`, `I<SIZE>ShrSImm16Rev(r0,imm,r2)`
`I<SIZE>Rotl(r0,r1,r2)`	`r0 <-` bits of `r1` rotated left by `r2 mod <SIZE>` Variants: `I<SIZE>RotlImm(r0,r1,imm)`, `I<SIZE>RotlImm16Rev(r0,imm,r2)`
`I<SIZE>Rotr(r0,r1,r2)`	`r0 <-` bits of `r1` rotated right by `r2 mod <SIZE>` Variants: `I<SIZE>RotrImm(r0,r1,imm)`, `I<SIZE>RotrImm16Rev(r0,imm,r2)`

Instruction	Description
`F<SIZE>Add(r0,r1,r2)`	`r0 <- r1 + r2`
`F<SIZE>Sub(r0,r1,r2)`	`r0 <- r1 - r2`
`F<SIZE>Mul(r0,r1,r2)`	`r0 <- r1 * r2`
`F<SIZE>Div(r0,r1,r2)`	`r0 <- r1 / r2`
`F<SIZE>Min(r0,r1,r2)`	`r0 <- if r1 < r2 then r1 else r2`
`F<SIZE>Max(r0,r1,r2)`	`r0 <- if r1 < r2 then r2 else r1`
`F<SIZE>Copysign(r0,r1,r2)`	`r0 <- if sgn(r1) == sgn(r2) then r1 else r1 * (-1)` Variant: `F<SIZE>CopysignImm(r0,r1,sign)`

Conversions between numeric types follow the respective Wasm instruction set.

Instruction	Description
`I32WrapI64(r0,r1,r2)`	`r0 <- r1 mod 2^32`
`I64ExtendI32S(r0,r1,r2)`	`r0 <- r1` sign-extended to 64 bit
`I64ExtendI32U(r0,r1,r2)`	`r0 <- r1` extended to 64 bit (prefixing zeros, no sign)
`I<N>Extend<M>S(r0,r1,r2)`	`r0 <- r1` sign-extended to `<N>` bits from Wasm "sign-extension" proposal

where `<N>` is 32 or 64, `<M>` is 8, 16, or 32, and `<N> > <M>`.

Instruction	Description (simplified)
`I<N>TruncF<M>S(r0,r1,r2)`	`r0 <- trunc(r1)` if in range `[-2^(<N>-1)..2^(<N>-1) - 1]`
`I<N>TruncF<M>U(r0,r1,r2)`	`r0 <- trunc(r1)` if in range `[0..2^<N>-1]`
`F<N>ConvertI<M>S(r0,r1,r2)`	`r0 <- float(r1)` (as defined in Wasm)
`F<N>ConvertI<M>U(r0,r1,r2)`	`r0 <- float(signed(r1))` (as defined in Wasm)
`I<N>TruncSatF<M>S(r0,r1,r2)`	as `I<N>TruncF<M>S` but returns 0 for `NaN` and max/min value for infinity from Wasm "non-trapping float-to-int conversions" proposal
`I<N>TruncSatF<M>U(r0,r1,r2)`	as `I<N>TruncF<M>U` but returns 0 for `NaN` and max/min value for infinity from Wasm "non-trapping float-to-int conversions" proposal

where `<N>` and `<M>` are 32 or 64.

Instruction	Description
`F32DemoteF64(r0,r1,r2)`	`Double` to `Float` conversion (maybe infinity)
`F64PromoteF32(r0,r1,r2)`	`Float` to `Double` extension (identical value)

Reference Instructions

Producing a function reference works as in Wasm:

Instruction	Description
`RefFunc(r0,r1)`	`r0 <- ref r1`

No `null` check exists for function references, as there is no stack (the value to check will be in a register).

Parametric Instructions

The Wasm `drop` instruction does not exist because no stack is managed. The `select` instruction exists in several variants which each require to appear in a certain context (of instruction parameters following).

A register or constant value must follow as a separate instruction parameter for the following two variants:

Instruction	Description
`Select(r0,r1,r2)`	`r0 <- if r1 then r2 else <NEXT>` A `Register` or `Const` instruction parameter providing `<NEXT>` must follow
`SelectRev(r0,r1,r2)`	`r0 <- if r1 then <NEXT> else r2` A `Register` or `Const` instruction parameter providing `<NEXT>` must follow

For the `Imm` variants, the respective select instruction must appear *pairwise*, with two arguments each: one register argument `r0`, and one immediate argument `imm`. The arguments are indexed by instruction 1 or 2 below.

Instruction	Description
`SelectImm32(r0,imm)`	`r0(1) <- if r0(2) then imm(1) else imm(2)` `imm(_)` are immediate 32-bit integers.
`SelectI64Imm32(r0,imm)`	`r0(1) <- if r0(2) then imm(1) else imm(2)` `imm(_)` are immediate 64-bit integers that fit into 32 bit.
`SelectF64Imm32(r0,imm)`	`r0(1) <- if r0(2) then imm(1) else imm(2)` `imm(_)` are immediate 64-bit floats that fit into 32 bit.

Variable Instructions

In Wasmi, local variables are held in registers. Therefore, `get` and `set` for variables only apply to *global* variables (of initialised modules).

Instruction	Description
`GlobalGet(r0,r1)`	`r0 <-` value of global variable at `r1`
`GlobalSet(r0,r1)`	Set global variable at `r0` to value in `r1` Variants: `GlobalSetI32Imm16(r0,imm)`, `GlobalSetI64Imm16(r0,imm)`

Table Instructions

In comparison to Wasm, the Wasmi **table element access** instructions are similar but require the table index to be provided in a subsequent instruction parameter.

Instruction	Description
`TableGet(r0,r1)`	reads table `tidx` at index in `r1` into `r0` An instruction parameter `TableIdx tidx` must follow
`TableGetImm(r0,idx)`	reads table `tidx` at index `idx` into `r0` An instruction parameter `TableIdx tidx` must follow
`TableSet(r0,r1)`	writes value in `r1` into table `tidx` at index in `r0` An instruction parameter `TableIdx tidx` must follow
`TableSetAt(idx,r1)`	writes value in `r1` into table `tidx` at index `idx` An instruction parameter `TableIdx tidx` must follow
`TableSize(r0,tidx)`	stores size of table `tidx` in `r0`

`TableGetImm` and `TableSetAt` contain an immediate index `idx`, while `TableGet` and `TableSet` provide the index in a register argument.

There are a number of variants for **copying, filling, and initialising tables**.

`TableCopy*` instructions copy data between tables. All variants use three arguments `dst`, `src`, and `len` and additional instruction parameters for destination and source of the copy. They vary in whether arguments are registers or constant (16-bit encoded) values.

Instruction	Description
`TableCopy(r0,r1,r2)`	copies `*r2` (len) many values from table `srcTidx` starting at index in `r1` to table `destTidx` starting at index in `r0` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyTo(dst,r1,r2)`	copies `*r2` (len) many values from table `srcTidx` starting at index in `r1` to table `destTidx` starting at index `dst` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyFrom(r0,src,r2)`	copies `*r2` (len) many values from table `srcTidx` starting at index `src` to table `destTidx` starting at index in `r0` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyFromTo(dst,src,r2)`	copies `*r2` (len) many values from table `srcTidx` starting at index `src` to table `destTidx` starting at index `dst` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyExact(r0,r1,len)`	copies `len` many values from table `srcTidx` starting at index in `r1` to table `destTidx` starting at index in `r0` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyToExact(dst,r1,len)`	copies `len` values from table `srcTidx` starting at index in `r1` to table `destTidx` starting at index `dst` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyFromExact(r0,src,len)`	copies `len` values from table `srcTidx` starting at index `src` to table `destTidx` starting at index in `r0` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow
`TableCopyFromToExact(dst,src,len)`	copies `len` values from table `srcTidx` starting at index `src` to table `destTidx` starting at index `dst` Two `TableIdx` instruction parameters for `destTidx` and `srcTidx` must follow

As in Wasm, `TableInit*` instructions copy data from an element segment to a table. Like `TableCopy`, all `TableInit*` instructions have `dst`, `src`, and `len` arguments and differ in which ones are provided as registers or as (16-bit encoded) constants. `TableInit*` instructions require a destination `TableIdx` and a source `ElementIdx` instruction parameter to follow.

Instruction	Description
`TableInit(r0,r1,r2)`	copies `*r2` (len) many values from segment `srcEidx` starting at index in `r1` to table `destTidx` starting at index in `r0` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitTo(dst,r1,r2)`	copies `*r2` (len) many values from segment `srcEidx` starting at index in `r1` to table `destTidx` starting at index `dst` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitFrom(r0,src,r2)`	copies `*r2` (len) many values from segment `srcEidx` starting at index `src` to table `destTidx` starting at index in `r0` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitFromTo(dst,src,r2)`	copies `*r2` (len) many values from segment `srcEidx` starting at index `src` to table `destTidx` starting at index `dst` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitExact(r0,r1,len)`	copies `len` many values from segment `srcEidx` starting at index in `r1` to table `destTidx` starting at index in `r0` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitToExact(dst,r1,len)`	copies `len` values from segment `srcEidx` starting at index in `r1` to table `destTidx` starting at index `dst` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitFromExact(r0,src,len)`	copies `len` values from segment `srcEidx` starting at index `src` to table `destTidx` starting at index in `r0` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow
`TableInitFromToExact(dst,src,len)`	copies `len` values from segment `srcEidx` starting at index `src` to table `destTidx` starting at index `dst` A `TableIdx destTidx` and an `ElementIdx srcEIdx` instruction must follow

`TableFill` instructions use arguments `dst`, `len`, and `value`. The `value` to fill in is always provided in a register, while `dst` and `len` may be either from registers or as constants. Again, a `TableIdx` must follow to indicate the table to modify.

Instruction	Description
`TableFill(r0,r1,r2)`	writes value in `r2` `*r1` (len) many times into table `tidx`, starting at index in `r0` A `TableIdx tidx` instruction parameter must follow
`TableFillAt(dst,r1,r2)`	writes value in `r2` `*r1` (len) many times into table `tidx`, starting at index `dst` A `TableIdx tidx` instruction parameter must follow
`TableFillExact(r0,len,r2)`	writes value in `r2` `len` many times into table `tidx`, starting at index in `r0` A `TableIdx tidx` instruction parameter must follow
`TableFillAtExact(dst,len,r2)`	writes value in `r2` `len` many times into table `tidx`, starting at index `dst` A `TableIdx tidx` instruction parameter must follow

The `TableGrow` instructions enlarge a table and return the old table size (or `-1`) in the first argument register (`r0`). The size to grow by is either from argument register `r1` or immediate, the value to fill new cells with is provided in register `r2`.

Instruction	Description
`TableGrow(r0,r1,r2)`	Enlarges table `tidx` by size in `r1`, filling with value from `r2`. Returns previous size in `r0` (or `-1` on errors) A `TableIdx tidx` instruction parameter must follow
`TableGrowImm(r0,sz,r2)`	Enlarges table `tidx` by `sz`, filling with value from `r2`. Returns previous size in `r0` (or `-1` on errors) A `TableIdx tidx` instruction parameter must follow

In Wasm, *element segments* in a module may be dropped to prevent further use (as a hint for possible optimisations). Correspondingly, Wasmi provides an instruction `ElemDrop` which carries an element/data segment index and discards the indicated segment.

Memory Instructions

Load instructions access memory at a given offset and load bytes representing a value of the target type (`I32`, `I64`, `F32`, `F64`) into the argument register (here `r0`). The 2nd argument may be a register (in which case a `Const32` instruction must follow to provide an offset value), or a constant offset. In the `Offset16` variant, a 3rd argument provides a constant offset which must be representable in 16 bits, from a register (2nd argument).

Instruction	Description
`<TYPE>Load(r0,r1)`	`r0 <-` value loaded from `r1 + <offset>` A `Const32 <offset>` instruction parameter must follow
`<TYPE>LoadAt(r0,offset)`	`r0 <-` value loaded from given `offset` (2nd arg.)
`<TYPE>LoadOffset16(r0,r1, offset)`	`r0 <-` value loaded from `r1 + offset` (2nd/3rd arg.)

where `<TYPE>` is `I32`, `I64`, `F32`, or `F64`.

For `I32` and `I64` loads, additional instructions (equivalent to the ones in Wasm) exist to load fewer bytes and extend the value's bit pattern appropriately (unsigned or signed as per `u` or `s` after `<M>`).

Instruction	Description
`I<N>Load<M>u`	as above, but reading `<M>` bits A `Const32 <offset>` instruction parameter must follow
`I<N>Load<M>s`	as above, but reading `<M>` bits, sign-extending A `Const32 <offset>` instruction parameter must follow
`I<N>Load<M>uAt`	as above, but reading `<M>` bits
`I<N>Load<M>sAt`	as above, but reading `<M>` bits, sign-extending
`I<N>Load<M>uOffset16`	as above, but reading `<M>` bits
`I<N>Load<M>sOffset16`	as above, but reading `<M>` bits, sign-extending

where `<N>` is 32 or 64, and `<M>` is 8, 16, or 32 (if `<N>` is 64).

For **Store instructions**, the two arguments `r0` and `offset` define the place to store. There are 5 variants that differ in:

- how the *stored value* is specified:
  - either in a following `Register` instruction parameter,
  - or as a register within the same instruction (limiting the size of `offset` to 16 bit)
  - or as a 16-bit encoded `value` within the same instruction (limiting the size of `offset` to 16 bit).
- whether the target address is constant
  - the target address can be in a register, combined with a given offset,
  - or given as a constant (in the for `StoreAt` variants).

Instruction	Description
`<TYPE>Store(r0,offset)`	store value in `r1` at address `r0 + <offset>` A `Register <r1>` instruction parameter must follow
`<TYPE>StoreOffset16(r0,r1,offset)`	store value in `r1` at address `r0 + <offset>`
`<TYPE>StoreOffset16Imm16(r0,value,offset)`	store `value` at address `r0 + offset`
`<TYPE>StoreAt(addr,reg)`	store value from `reg` at `addr`
`<TYPE>StoreAtImm16(addr,value)`	store `value` at `addr`

where `<TYPE>` is `I32`, `I64`, `F32`, or `F64`.

For the integral types, there are variants for storing smaller values (of 8 or 16 bit width for `I32` and 8, 16, or 32 bit width for `I64`), each with similar variants as the ones described above.

Instruction	Description
`I<N>Store<M>(r0,offset)`	as above, but truncating the value to bits A `Register r` instruction parameter must follow
`I<N>Store<M>Offset16(r0,r1,offset)`	as above, but truncating the value to bits
`I<N>Store<M>Offset16Imm16(r0,value,offset)`	as above, but truncating the value to bits
`I<N>Store<M>At(addr,reg)`	as above, but truncating the value to bits
`I<N>Store<M>AtImm16(addr, value)`	as above, but truncating the value to bits

where `<N>` is 32 or 64, and `<M>` is 8, 16, or 32 (if `<N>` is 64).

Store instructions for floating-point decimals come in similar variants but without the immediate value variants.

Instruction	Description
`F<N>Store(r0,offset)`	store value in `r1` at address `r0 + <offset>` A `Register <r1>` instruction parameter must follow
`F<N>StoreOffset16(r0,r1,offset)`	store value in `r1` at address `r0 + <offset>` The offset is small (16 bit)
`F<N>StoreAt(addr, r1)`	store value in `r1` at `addr`

where `<N>` is 32 or 64.

Memory management and initialisation instructions in Wasmi have different variants according to how the arguments are provided (in registers or as immediate values). No instruction parameters are required because Wasm (currently) limits each module's memories to exactly one.

Instruction	Description
`MemorySize(r0)`	`r0 <-` current size of memory (in 64K pages)
`MemoryGrow(r0,r1)`	grow memory by size in `r1`, return *old* size in `r0`
`MemoryGrowBy(r0,sz)`	grow memory by `sz`, return *old* size in `r0`

Instruction	Description
`MemoryCopy(r0,r1,r2)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `len` in `r2`, `src` in `r1`, `dst` in `r0`
`MemoryCopyTo(dst,r1,r2)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `len` in `r2`, `src` in `r1`
`MemoryCopyFrom(r0,src,r2)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `len` in `r2`, `dst` in `r0`
`MemoryCopyFromTo(dst,src,r2)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `len` in `r2`
`MemoryCopyExact(r0,r1,len)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `src` in `r1`, `dst` in `r0`
`MemoryCopyToExact(dst,r1,len)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `src` in `r1`
`MemoryCopyFromExact(r0,src,len)`	copies `len` bytes from `mem[src..]` to `mem[dst..]` `dst` in `r0`
`MemoryCopyFromToExact(dst,src,len)`	copies `len` bytes from `mem[src..]` to `mem[dst..]`

Instruction	Description
`MemoryFill(r0,r1,r2)`	writes 8-bit `val` into `mem[dst..dst+len]` `val` in `r1`, `len` in `r2`, `dst` in `r0`
`MemoryFillAt(dst,r1,r2)`	writes 8-bit `val` into `mem[dst..dst+len]` `val` in `r1`, `len` in `r2`
`MemoryFillImm(r0,val,r2)`	writes 8-bit `val` into `mem[dst..dst+len]` `len` in `r2`, `dst` in `r0`
`MemoryFillExact(r0,r1,len)`	writes 8-bit `val` into `mem[dst..dst+len]` `val` in `r1`, `dst` in `r0`
`MemoryFillAtImm(dst,val,r2)`	writes 8-bit `val` into `mem[dst..dst+len]` `len` in `r2`
`MemoryFillAtExact(dst,r1,len)`	writes 8-bit `val` into `mem[dst..dst+len]` `val` in `r1`
`MemoryFillImmExact(r0,val,len)`	writes 8-bit `val` into `mem[dst..dst+len]` `val` in `r1`, `len` in `r2`
`MemoryFillAtImmExact(dst,val,len)`	writes 8-bit `val` into `mem[dst..dst+len]`

The `MemoryInit` family of instructions copy data from a given data segment to memory, therefore a `DataSegmentIdx` must follow to indicate which segment to use.

Instruction	Description
`MemoryInit(r0,r1,r2)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `dst` in `r0`, `src` in `r1`, `len` in `r2` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitTo(dst,r1,r2)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `src` in `r1`, `len` in `r2` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitFrom(r0,src,r2)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `dst` in `r0`, `len` in `r2` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitFromTo(dst,src,r2)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `len` in `r2` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitExact(r0,r1,len)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `dst` in `r0`, `src` in `r1` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitToExact(dst,r1,len)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `src` in `r1` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitFromExact(r0,src,len)`	for data segment index `dseg`, copies `data[dseg][src..src+len]` to `mem[dst..dst+len]` `dst` in `r0` A `DataSegmentIdx dseg` instruction parameter must follow
`MemoryInitFromToExact(dst,src,len)`	for data segment index `dseg`, copies `data[dseg] [src..src+len]` to `mem[dst..dst+len]` a `DataSegmentIdx dseg` instruction parameter must follow

As in Wasm, data segments can be *dropped* to prevent further access (enabling optimisations) using the Wasmi instruction `DataDrop`, which carries a data segment index.

Control Instructions

The `Trap` instruction does what its name suggests: Execution fails with the given "trap code" indicating an error condition.

A wide variety of **branching instructions** exist in the Wasmi instruction set. A common trait to all of them is that they operate using an *offset* from the current instruction pointer (IP) rather than the nesting levels that Wasm uses in its respective `br` instruction.

Instruction	Description
`Branch(offset)`	modifies instruction pointer by adding `offset`

Many branch instructions were added to combine certain tests with a subsequent conditional branch on the result. Each of these instructions has either two argument registers and one (16-bit) `offset`, or one argument register, one (16-bit-encoded) immediate `value` argument, and one (16-bit) `offset`.

Instruction	Description
`Branch<TYPE><OP>(r0,r1,offset)`	adds `offset` to IP if `r0 <OP> r1`
`Branch<TYPE><OP>Imm(r0,val,offset)`	adds `offset` to IP if `r0 <OP> val`

where `<TYPE>` and `<OP>` are a type and a binary operation from the table below:

OP	Description	Types
`And`	bit-wise `&`	`I32`
`Or`	bit-wise `|`	`I32`
`Xor`	bit-wise `^` (xor)	`I32`
`AndEqz`	bit-wise `&` followed by comparing to zero	`I32`
`OrEqz`	bit-wise `|` followed by comparing to zero	`I32`
`XorEqz`	bit-wise `^` followed by comparing to zero	`I32`
`Eq`	`==` unsigned	`I32` , `I64`
`Ne`	`!=` unsigned	`I32` , `I64`
`LtS`	`<` signed	`I32` , `I64`
`LtU`	`<` unsigned	`I32` , `I64`
`LeS`	`<=` signed	`I32` , `I64`
`LeU`	`<=` unsigned	`I32` , `I64`
`GtS`	`>` signed	`I32` , `I64`
`GtU`	`>` unsigned	`I32` , `I64`
`GeS`	`>=` signed	`I32` , `I64`
`GeU`	`>=` unsigned	`I32` , `I64`

(Note the absense of `And` , `Or` , and `Xor` variants for `I64`).

For floating-point decimals, there are no immediate variants (and no signedness).

Instruction	Description
`Branch<TYPE><OP>(r0,r1,offset)`	adds `offset` to IP if `r0 <OP> r1`

OP	Description	Types
`Eq`	`==`	`F32` , `F64`
`Ne`	`!=`	`F32` , `F64`
`Lt`	`<`	`F32` , `F64`
`Le`	`<=`	`F32` , `F64`
`Gt`	`>`	`F32` , `F64`
`Ge`	`>=`	`F32` , `F64`

All these instructions rely on a special encoding with a *16-bit* offset.

For cases where the offset is too large to be encoded as a 16-bit value, a generic fall-back instruction exists, which encodes the `offset` (32-bit) *and* the operation to perform as a special combined `param` (eter), read from a third argument *register* (`r2`) of the instruction.

Instruction	Description
`BranchCmpFallback(r0,r1,r2)`	adds `offset` to IP if `r0 <OP> r1` `offset` and `<OP>` read from a parameter `param` passed in a third register argument

For multi-target branches, (Wasm instruction `br_table`), Wasmi uses the `BranchTable` instruction. This instruction in Wasmi contains the scrutinee register and the length of the branch table as arguments, and expects the respective following instructions (after an optional `Copy*` instruction) to be `Branch` or `Return*` instructions that constitute the branch table (appropriate amount indicated by the length, including the default).

Instruction	Description
`BranchTable(index,len_targets)`	selects branch/return instruction indicated by value in `index` register, from `len_targets` branch or return instructions that follow Next `len_targets` instructions expected to be `Branch` or `Return*` (includes default if `index` value out of range).

All **function call instructions** `Call*` have a register span `results` (of unknown length) to indicate where the function results should be stored. Functions are either referred-to by their function index `func`, or called *indirectly through a table*, indicating the function type by an index `func_type` into the surrounding module's known types. Wasmi distinguishes internal functions from imported ones, and uses special `Call*` variants (suffixed with `0`) for functions without arguments. If arguments are required, they are passed as a register list that *follows* the `Call*` instruction.

Instruction	Description
`CallInternal0(results,func)`	Calls an internal function (by index `func`) without arguments
`CallInternal(results,func)`	Calls an internal function (by index `func`) Followed by a register list for the arguments
`CallImported0(results,func)`	Calls an imported function without arguments
`CallImported(results,func)`	Calls an imported function (by index `func`) Followed by a register list for the arguments
`CallIndirect0(results, func_type)`	Calls a function indirectly through a table, without arguments. Followed by `CallIndirectParams` (or `Imm16` variant)
`CallIndirect(results, func_type)`	Calls an function indirectly through a table. Followed by `CallIndirectParams` (or `Imm16` variant), and a register list for the arguments

The indirect call uses the following instruction parameters to supply the table and index for the indirect call:

Instruction	Description
`CallIndirectParams(tIdx, reg)`	holds a table index `tIdx` and a register `reg` containing an index
`CallIndirectParamsImm16(tIdx,index)`	holds a table index `tIdx` and a 16-bit `index`

For tail-call optimisation, there are special `ReturnCall*` variants corresponding to the above instructions, which reuse the prior function call's `results` registers:

Instruction	Description
`ReturnCallInternal0(func)`	tail-call internal function (by index `func`) without arg.s
`ReturnCallInternal(func)`	tail-call internal function (by index `func`). Followed by a register list for the arguments
`ReturnCallImported0(func)`	tail-call imported function (by index `func`) without arg.s
`ReturnCallImported(func)`	tail-call imported function (by index `func`). Followed by a register list for the arguments
`ReturnCallIndirect0(func_type)`	tail-call a function from a table without arg.s. Followed by `CallIndirectParams` (or `~Imm16` variant)
`ReturnCallIndirect(func_type)`	tail-call a function from a table. Followed by `CallIndirectParams` (or `~Imm16` variant) and then a register list for the arg.s

For **returning from function calls**, variants exist to return a number of immediate values or registers.

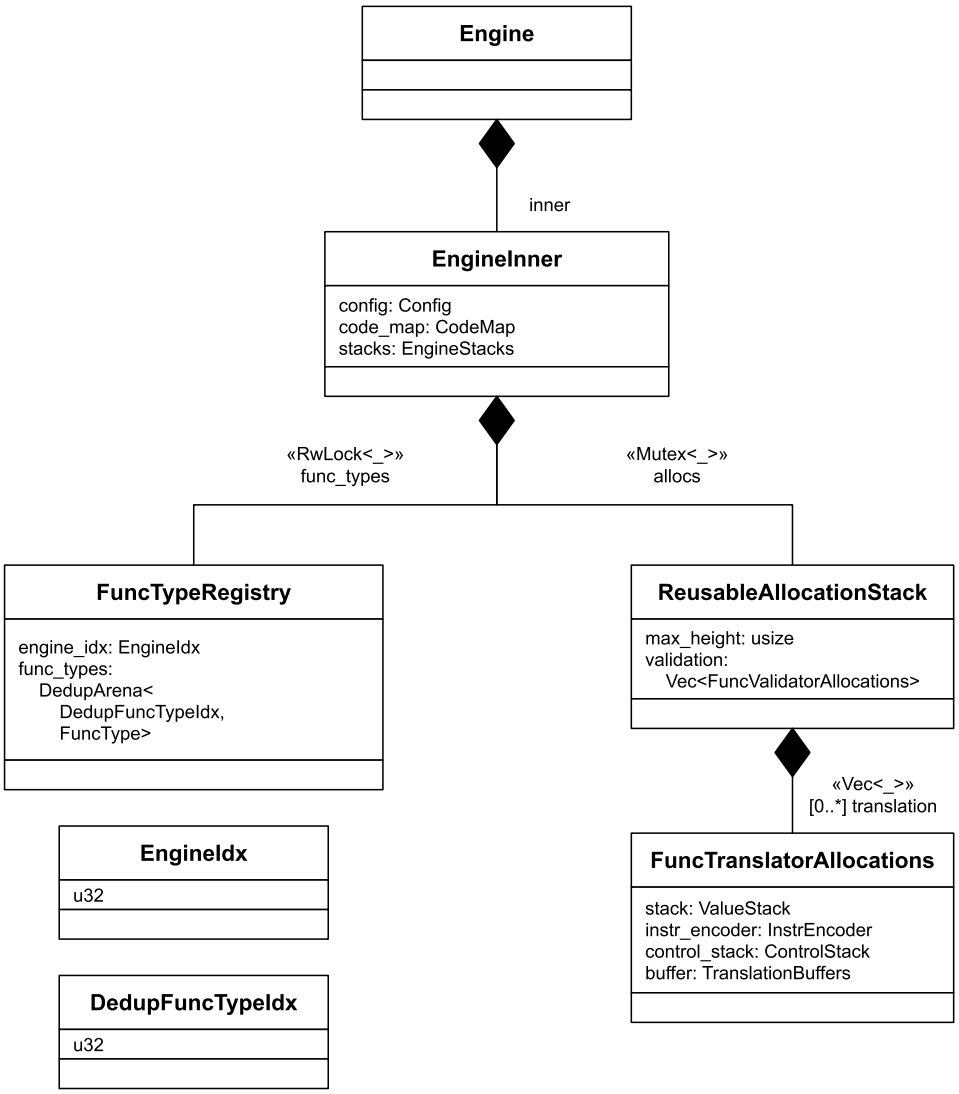
Instruction	Description
`Return`	Returns from a function without return value
`ReturnReg(r0)`	Returns value in register `r0`
`ReturnReg2([r0,r1])`	Returns values in registers `r0` and `r1`
`ReturnReg3([r0,r1,r2])`	Returns values in registers `r0`, `r1`, and `r2`
`ReturnImm32(value)`	Returns an immediate `I32` constant `value`
`ReturnI64Imm32(value)`	Returns an immediate `I64` constant `value` encoded in 32 bit
`ReturnF64Imm32(value)`	Returns an immediate `F64` constant `value` encoded in 32 bit
`ReturnSpan(iter)`	Returns more than 3 registers, given as a register span iterator `iter`
`ReturnMany(r0,r1,r2)`	Returns more than 3 registers, `r0`, `r1`, `r2`, and the ones from a following register list

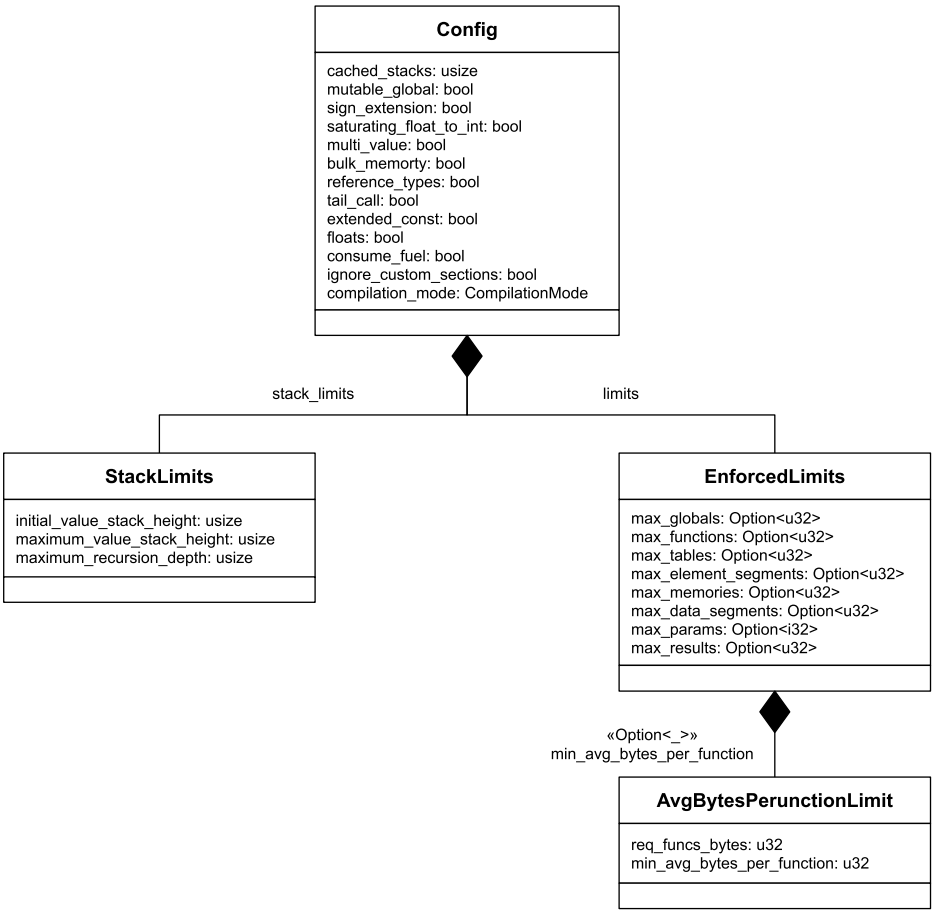
A special *conditional return* instruction (with the same variants as above) exists, which only returns if a given condition register contains a non-zero value.

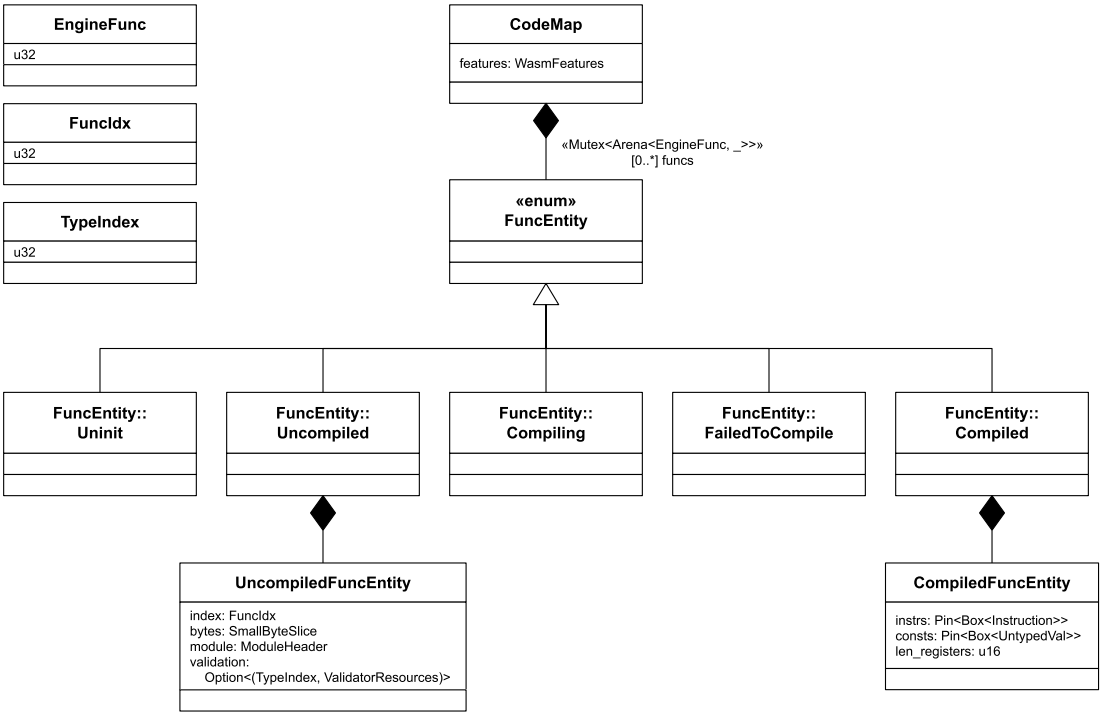
Instruction	Description
`ReturnNez(r0)`	If `r0` contains non-zero, returns from a function without return value
`ReturnNezReg(r0,r1)`	If `r0` contains non-zero, returns value in `r1`
`ReturnNezReg2(r0,r1,r2)`	If `r0` contains non-zero, returns values in registers `r0` and `r1`
`ReturnNezImm32(r0,val)`	If `r0` contains non-zero, returns an immediate `I32``constant` val`
`ReturnNezI64Imm32(r0,val)`	If `r0` contains non-zero, returns an immediate `I64` `val` encoded in 32 bit
`ReturnNezF64Imm32(r0,val)`	If `r0` contains non-zero, returns an immediate `F64` `val` encoded in 32 bit
`ReturnNezSpan(r0,iter)`	If `r0` contains non-zero, returns more than 2 registers, given as an iterator `iter`
`ReturnNezMany(r0,[r1,r2])`	If `r0` contains non-zero, returns more than 2 registers, `r1`, `r2`, and the ones following in a register list

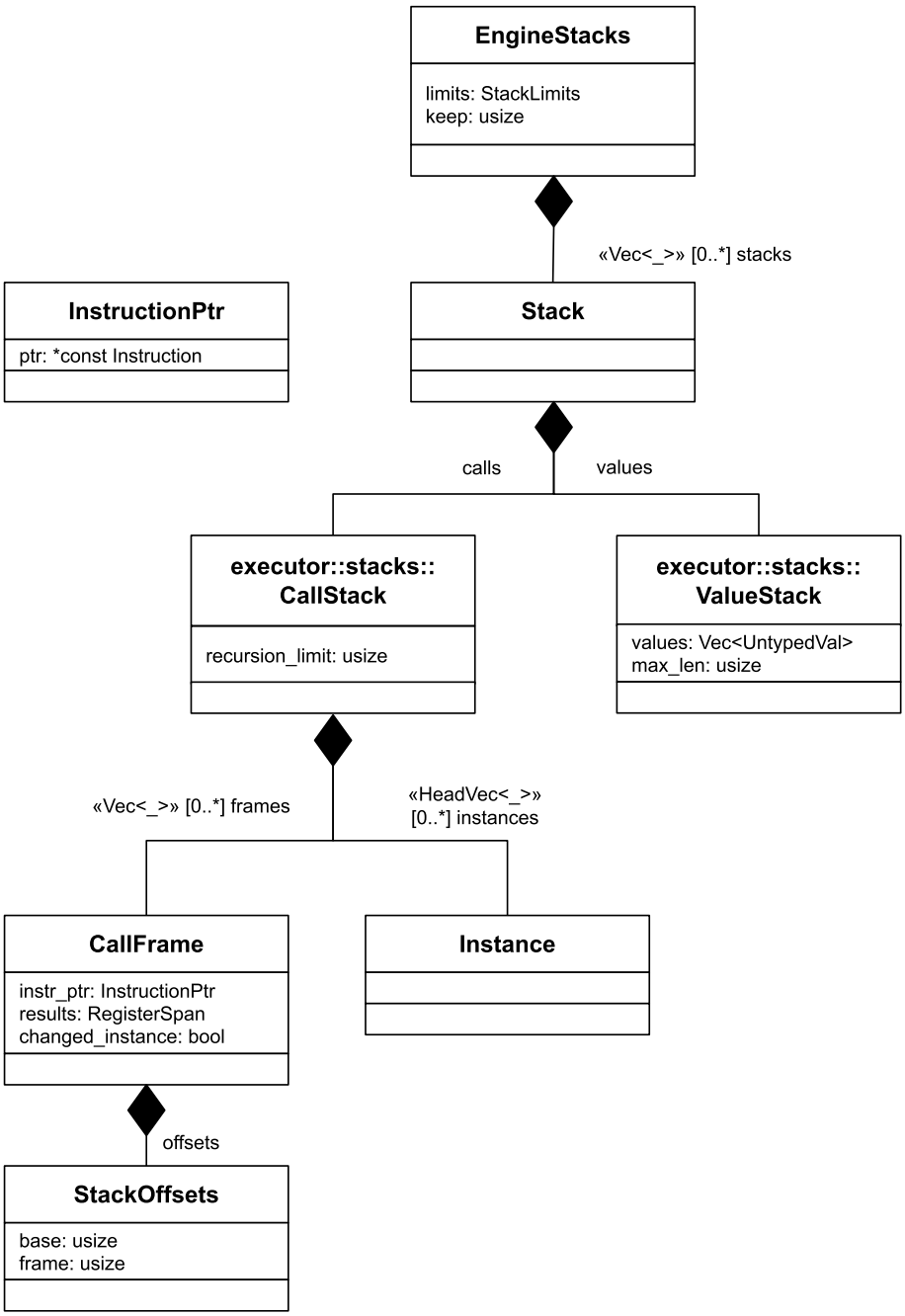
Note the absense of a variant with 3 registers (one register is needed for the r0).

Appendix: `Engine` Class Diagrams

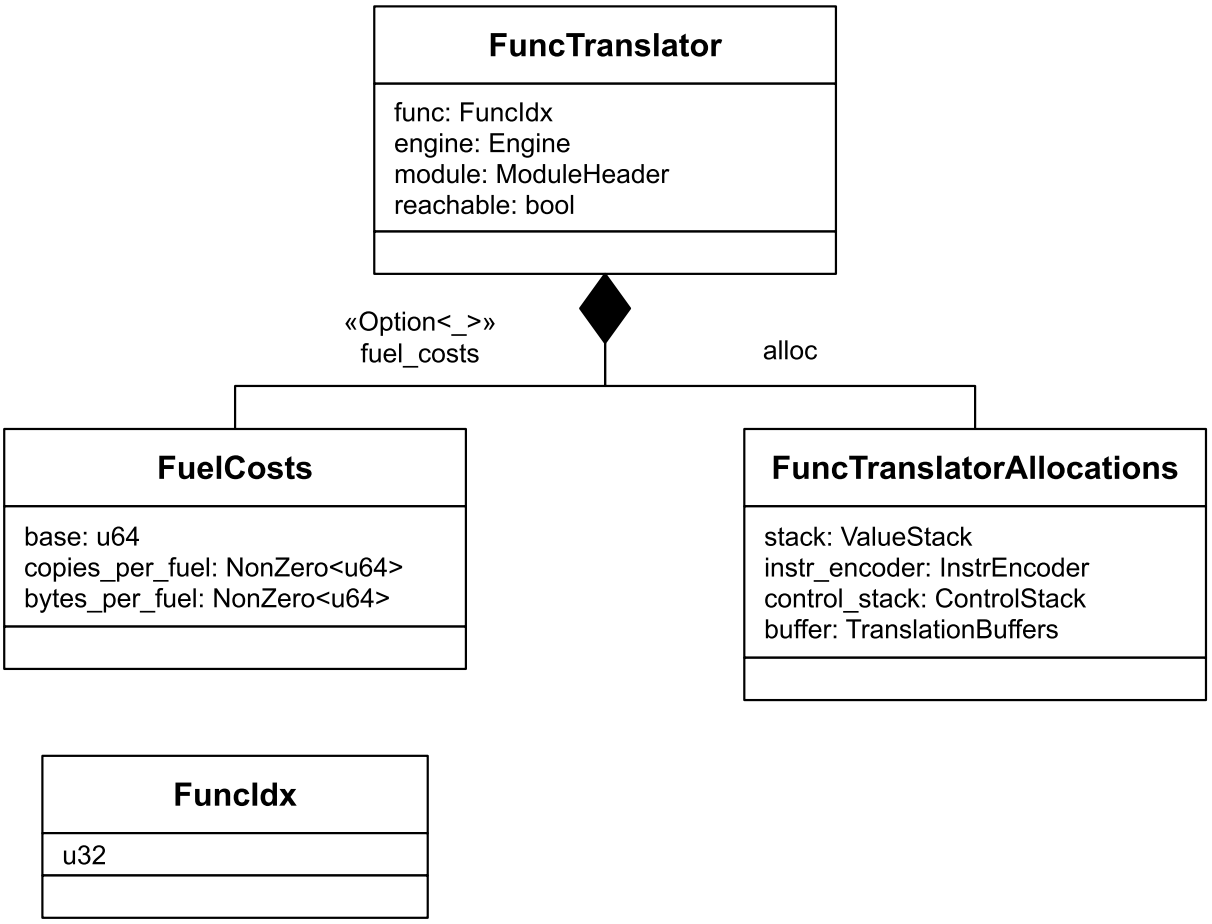


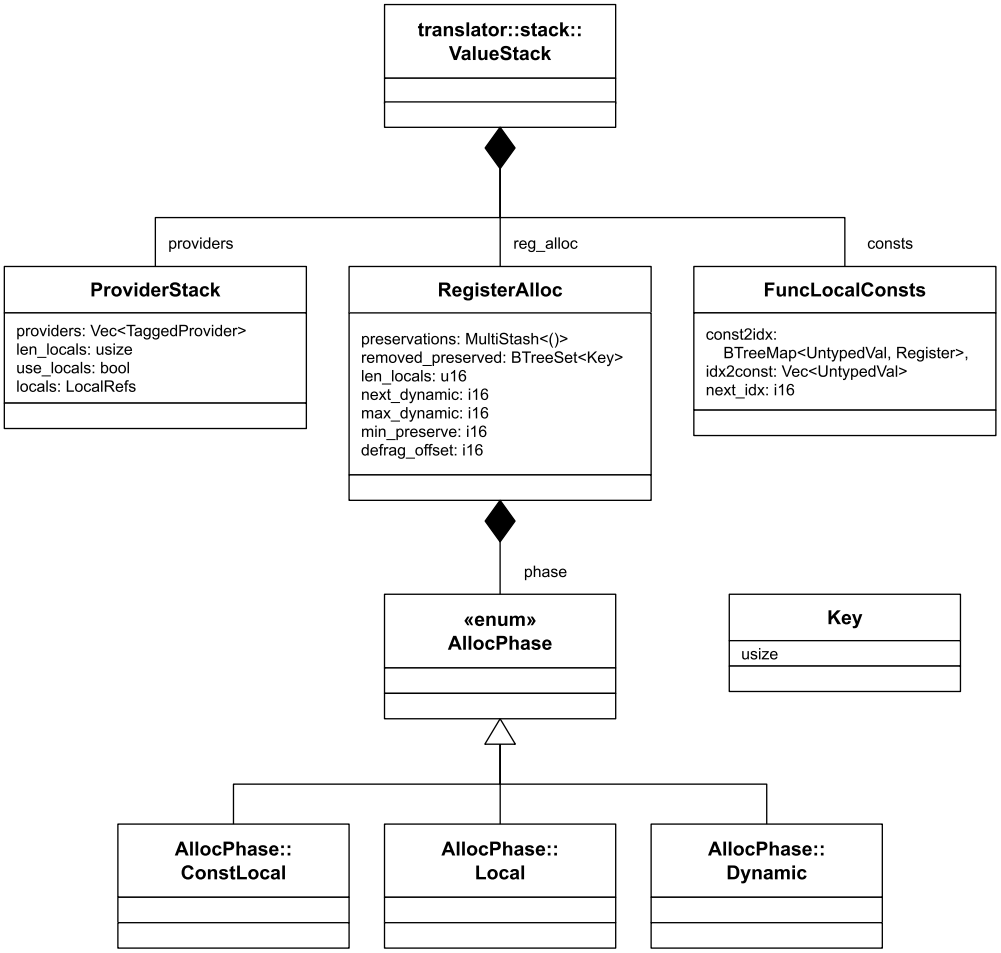

Config
cached_stacks: usize
mutable_global: bool
sign_extension: bool
saturating_float_to_int: bool
multi_value: bool
bulk_memorty: bool
reference_types: bool
tail_call: bool
extended_const: bool
floats: bool
consume_fuel: bool
ignore_custom_sections: bool
compilation_mode: CompilationMode
stack_limits
limits
StackLimits
initial_value_stack_height: usize
maximum_value_stack_height: usize
maximum_recursion_depth: usize
EnforcedLimits
max_globals: Option<u32>
max_functions: Option<u32>
max_tables: Option<u32>
max_element_segments: Option<u32>
max_memories: Option<u32>
max_data_segments: Option<u32>
max_params: Option<i32>
max_results: Option<u32>
«Option<_>»
min_avg_bytes_per_function
AvgBytesPerunctionLimit
req_funcs_bytes: u32
min_avg_bytes_per_function: u32

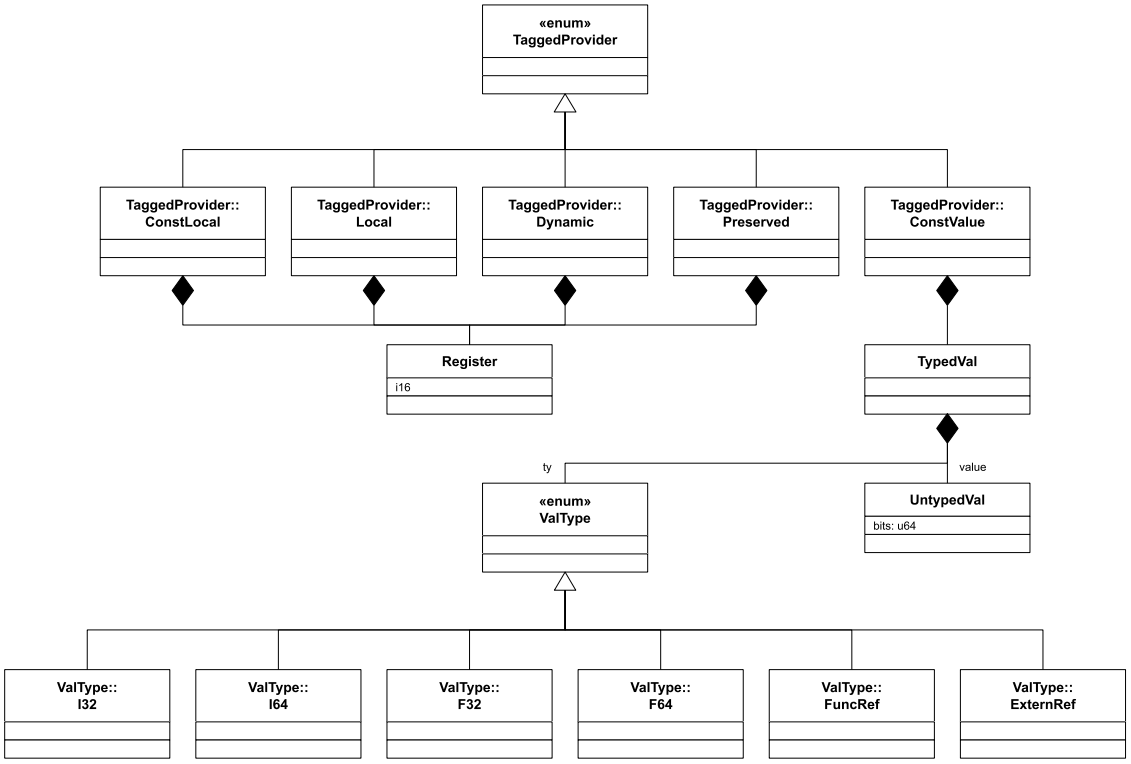

EngineFunc
u32
FuncIdx
u32
TypeIndex
u32
CodeMap
features: WasmFeatures
«Mutex<Arena<EngineFunc, _>>»
[0..*] funcs
«enum»
FuncEntity
FuncEntity::
Uninit
FuncEntity::
Uncompiled
FuncEntity::
Compiling
FuncEntity::
FailedToCompile
FuncEntity::
Compiled
UncompiledFuncEntity
index: FuncIdx
bytes: SmallByteSlice
module: ModuleHeader
validation:
Option<(TypeIndex, ValidatorResources)>
CompiledFuncEntity
instrs: Pin<Box<Instruction>>
consts: Pin<Box<UntypedVal>>
len_registers: u16

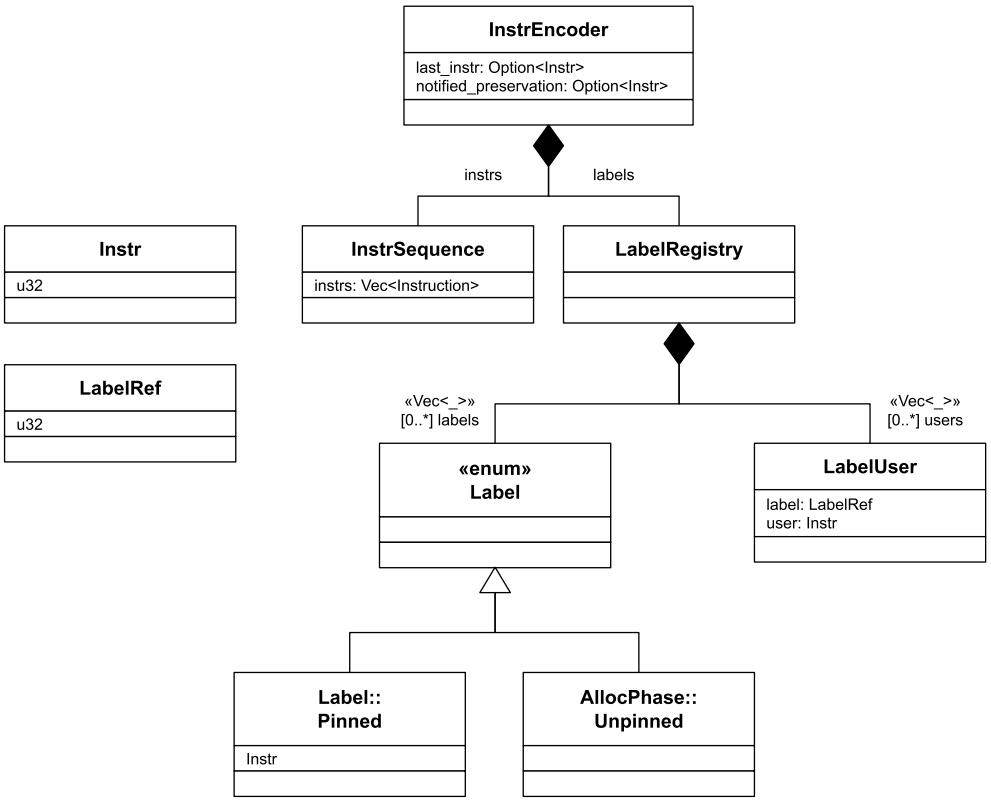

EngineStacks
limits: StackLimits
keep: usize
«Vec<_>» [0..*] stacks
InstructionPtr
ptr: *const Instruction
Stack
calls
values
executor::stacks::
CallStack
recursion_limit: usize
executor::stacks::
ValueStack
values: Vec<UntypedVal>
max_len: usize
«Vec<_>» [0..*] frames
«HeadVec<_>»
[0..*] instances
CallFrame
instr_ptr: InstructionPtr
results: RegisterSpan
changed_instance: bool
Instance
offsets
StackOffsets
base: usize
frame: usize

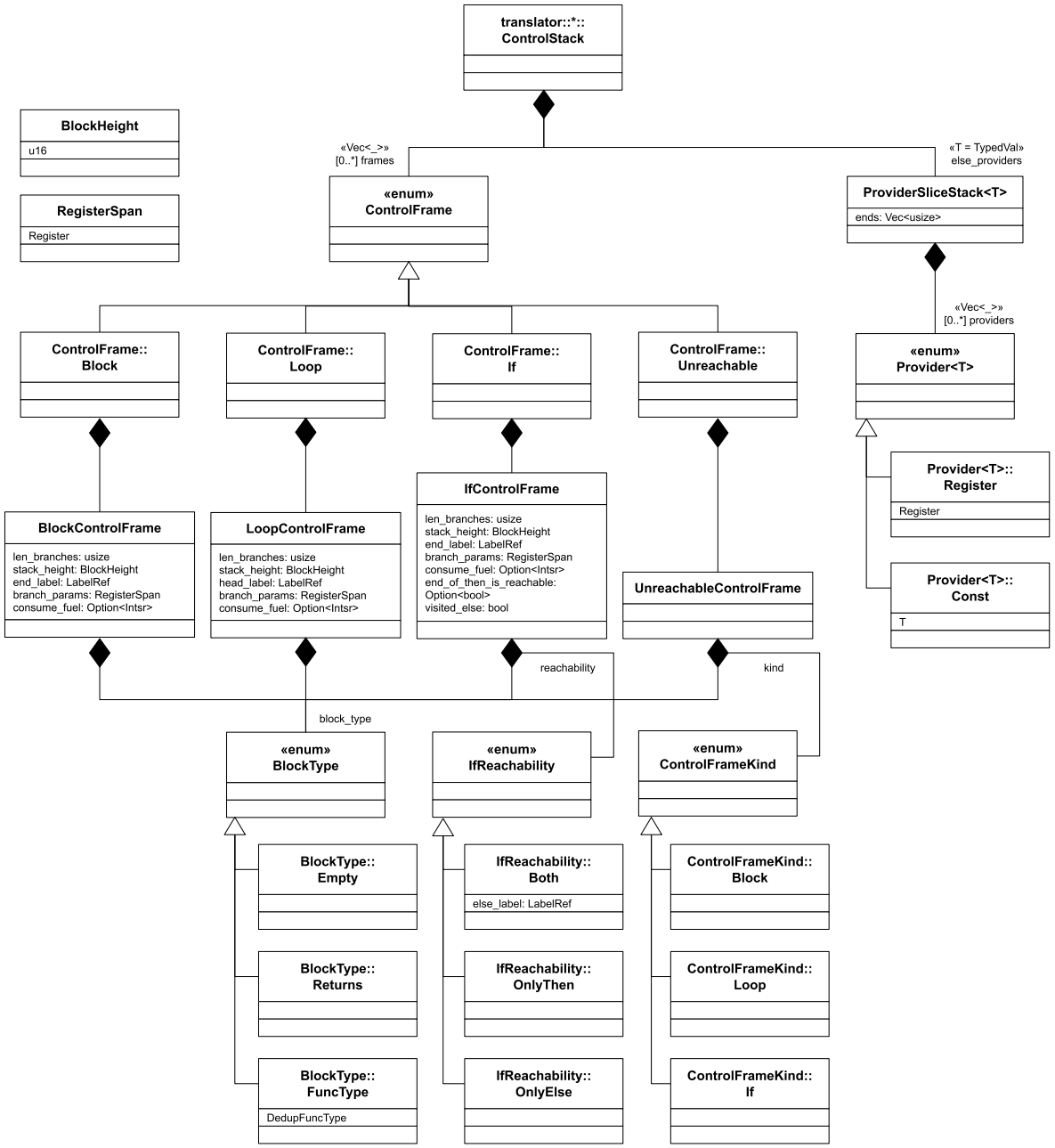
Appendix: `FuncTranslator` Class Diagrams

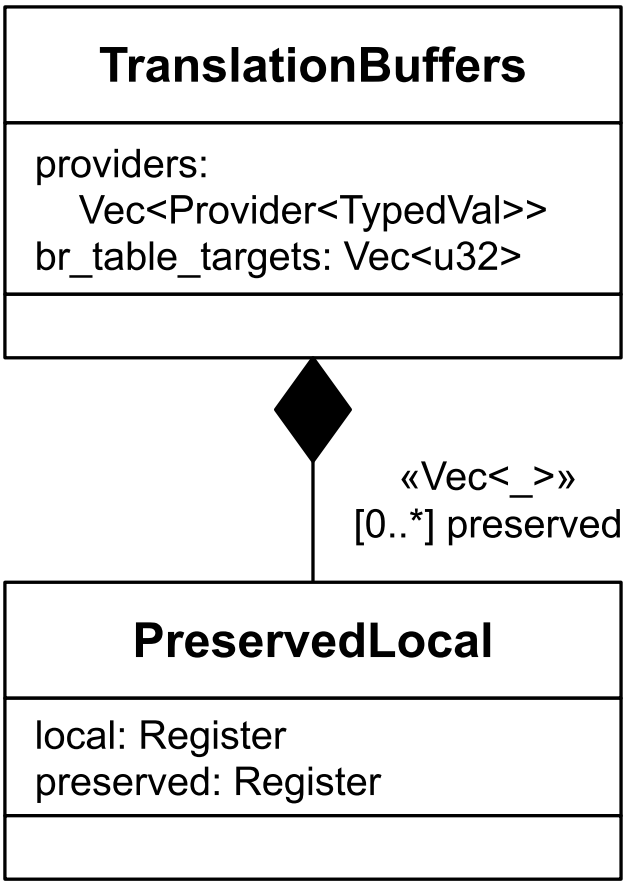



translator::stack::
ValueStack
providers
reg_alloc
consts
ProviderStack
providers: Vec<TaggedProvider>
len_locals: usize
use_locals: bool
locals: LocalRefs
RegisterAlloc
preservations: MultiStash<()>
removed_preserved: BTreeSet<Key>
len_locals: u16
next_dynamic: i16
max_dynamic: i16
min_preserve: i16
defrag_offset: i16
FuncLocalConsts
const2idx:
BTreeMap<UntypedVal, Register>,
idx2const: Vec<UntypedVal>
next_idx: i16
phase
«enum»
AllocPhase
Key
usize
AllocPhase::
ConstLocal
AllocPhase::
Local
AllocPhase::
Dynamic


«enum»
TaggedProvider
TaggedProvider::
ConstLocal
TaggedProvider::
Local
TaggedProvider::
Dynamic
TaggedProvider::
Preserved
TaggedProvider::
ConstValue
Register
i16
TypedVal
ty
value
«enum»
ValType
UntypedVal
bits: u64
ValType::
I32
ValType::
I64
ValType::
F32
ValType::
F64
ValType::
FuncRef
ValType::
ExternRef

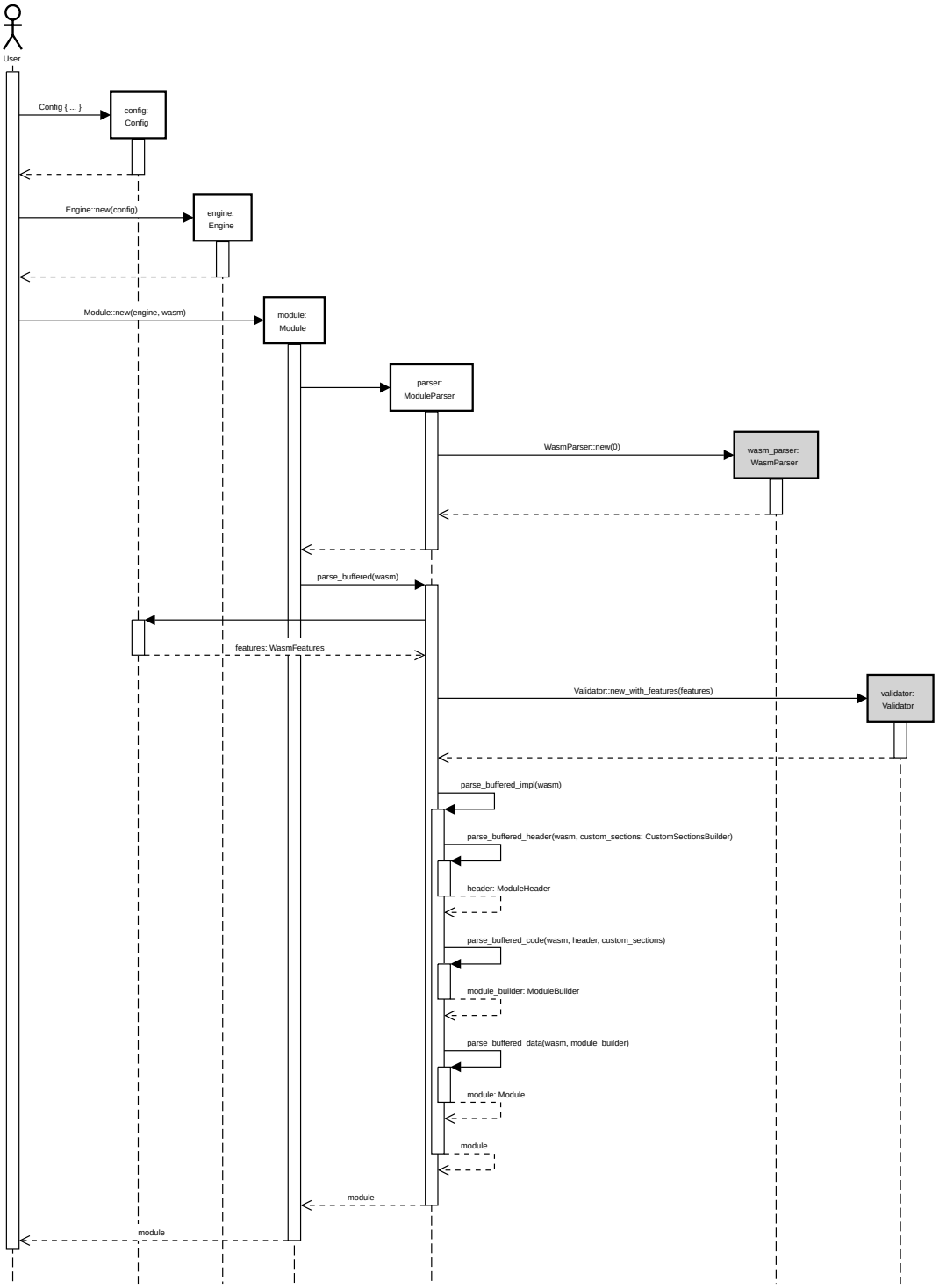

InstrEncoder
last_instr: Option<Instr>
notified_preservation: Option<Instr>
instrs
labels
Instr
u32
InstrSequence
instrs: Vec<Instruction>
LabelRegistry
LabelRef
u32
«Vec<_>»
[0..*] labels
«Vec<_>»
[0..*] users
«enum»
Label
LabelUser
label: LabelRef
user: Instr
Label::
Pinned
Instr
AllocPhase::
Unpinned


translator::*::
ControlStack
«Vec<_>»
[0..*] frames
«T = TypedVal»
else_providers
BlockHeight
u16
RegisterSpan
Register
«enum»
ControlFrame
ProviderSliceStack<T>
ends: Vec<usize>
«Vec<_>»
[0..*] providers
ControlFrame::
Block
ControlFrame::
Loop
ControlFrame::
If
ControlFrame::
Unreachable
«enum»
Provider<T>
Provider<T>::
Register
Register
Provider<T>::
Const
T
BlockControlFrame
len_branches: usize
stack_height: BlockHeight
end_label: LabelRef
branch_params: RegisterSpan
consume_fuel: Option<Intsr>
LoopControlFrame
len_branches: usize
stack_height: BlockHeight
head_label: LabelRef
branch_params: RegisterSpan
consume_fuel: Option<Intsr>
IfControlFrame
len_branches: usize
stack_height: BlockHeight
end_label: LabelRef
branch_params: RegisterSpan
consume_fuel: Option<Intsr>
end_of_then_is_reachable:
Option<bool>
visited_else: bool
UnreachableControlFrame
reachability
kind
block_type
«enum»
BlockType
«enum»
IfReachability
«enum»
ControlFrameKind
BlockType::
Empty
BlockType::
Returns
BlockType::
FuncType
DedupFuncType
IfReachability::
Both
else_label: LabelRef
IfReachability::
OnlyThen
IfReachability::
OnlyElse
ControlFrameKind::
Block
ControlFrameKind::
Loop
ControlFrameKind::
If

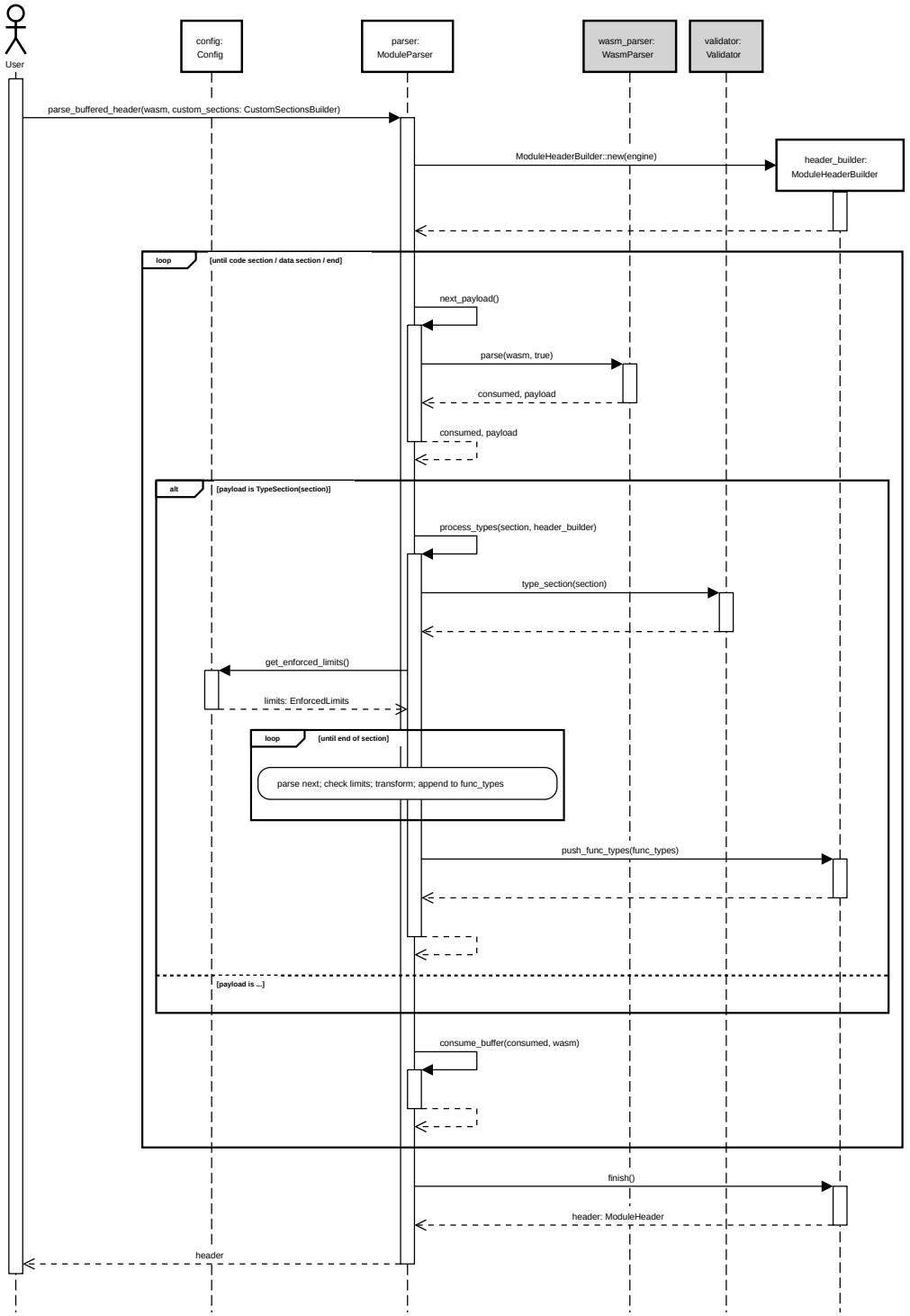

TranslationBuffers
providers:
Vec<Provider<TypedVal>>
br_table_targets: Vec<u32>
«Vec<_>»
[0..*] preserved
PreservedLocal
local: Register
preserved: Register

Appendix: Translation Sequence Diagrams

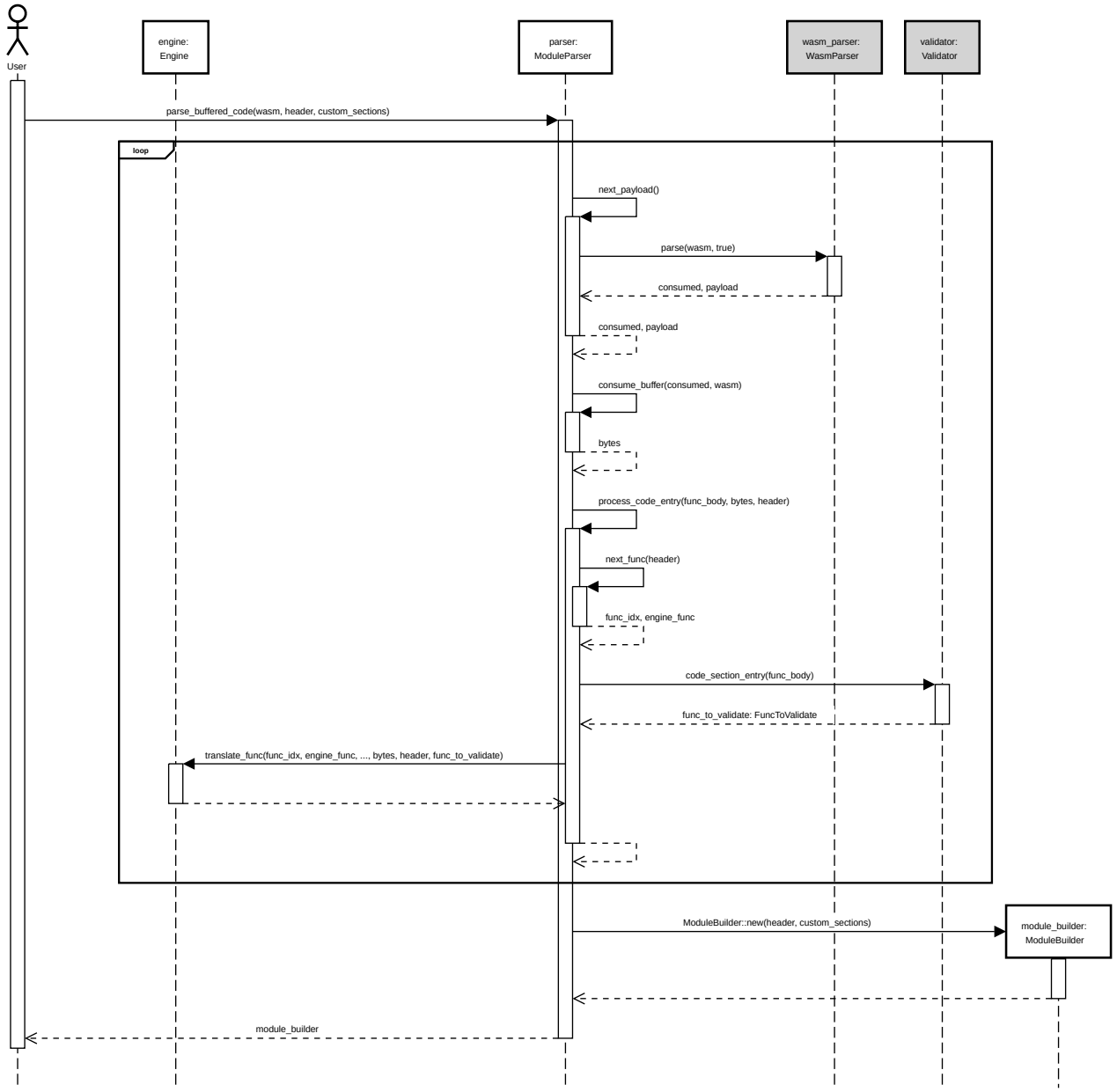
Module Parsing, Validation and Translation:
Overview



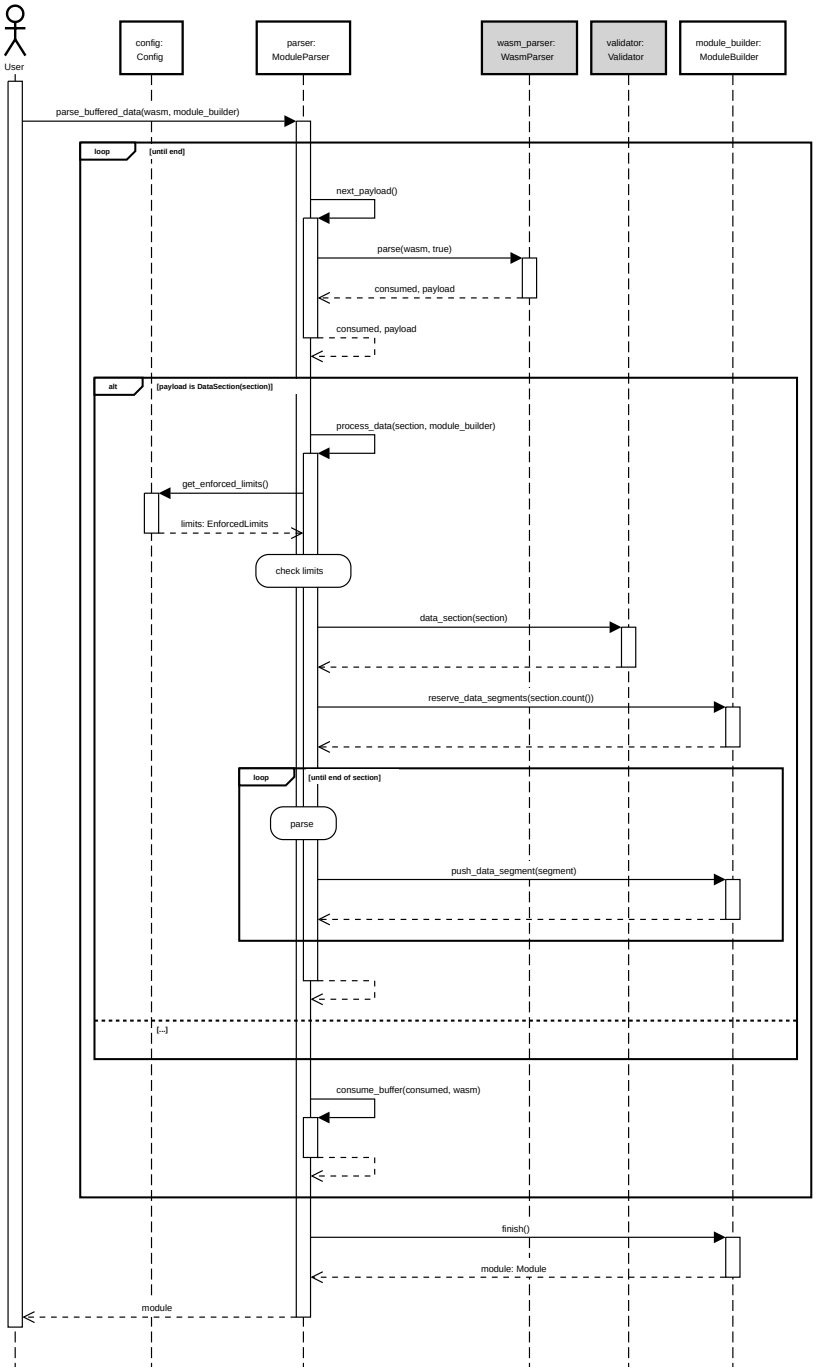
Module Parsing, Validation and Translation:
1. Header Parsing



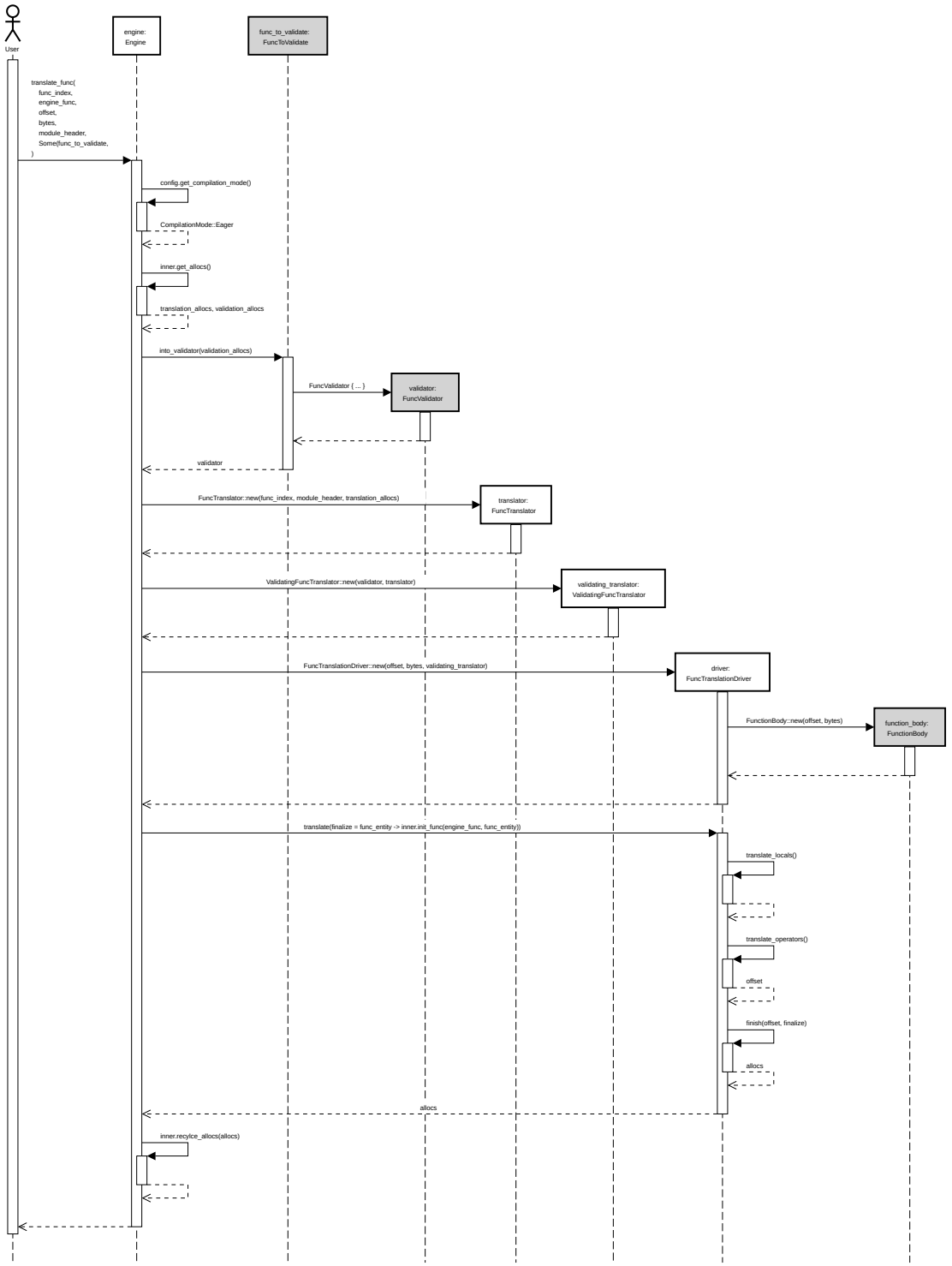
Module Parsing, Validation and Translation:
2. Code Parsing



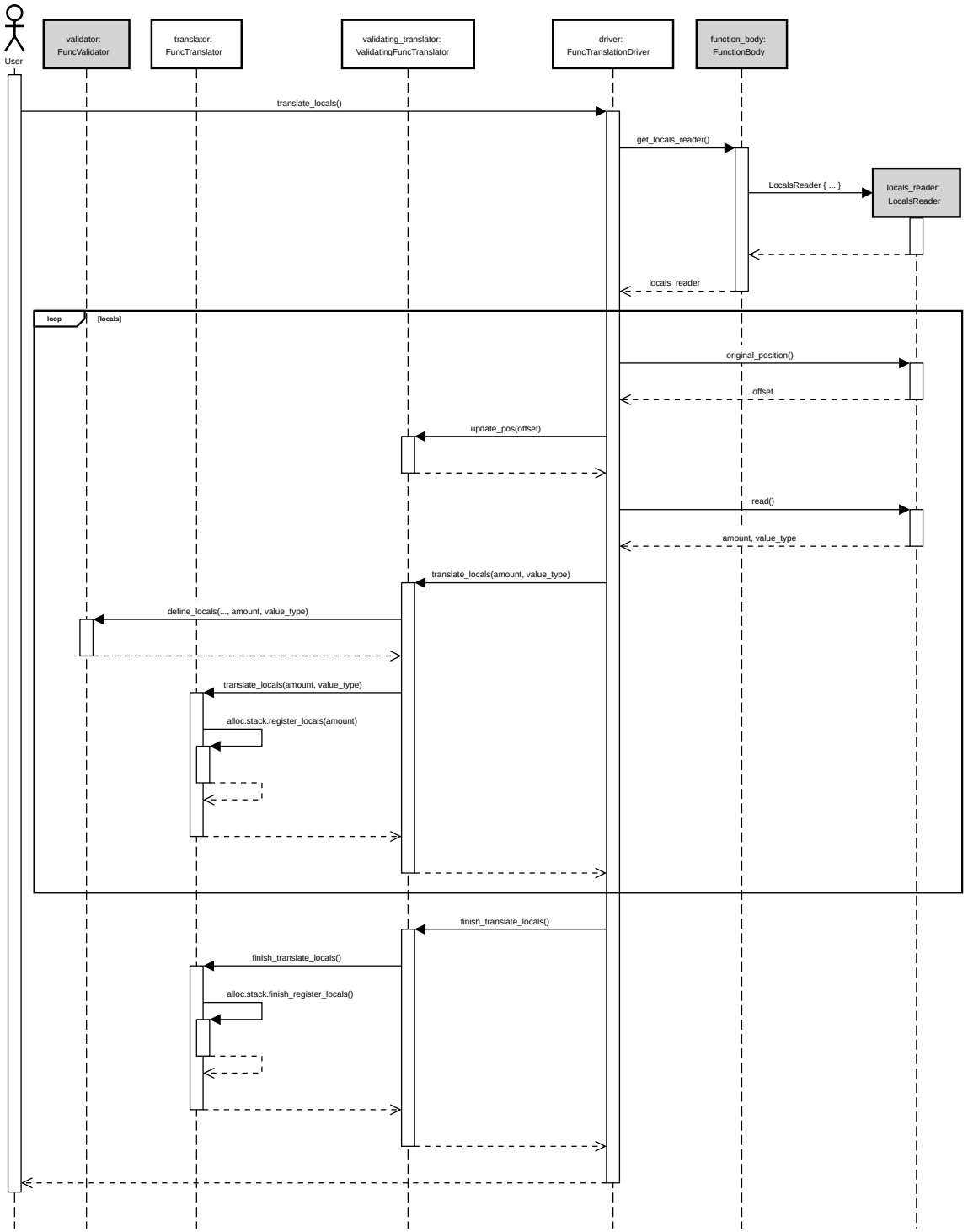
Module Parsing, Validation and Translation:
3. Data Parsing



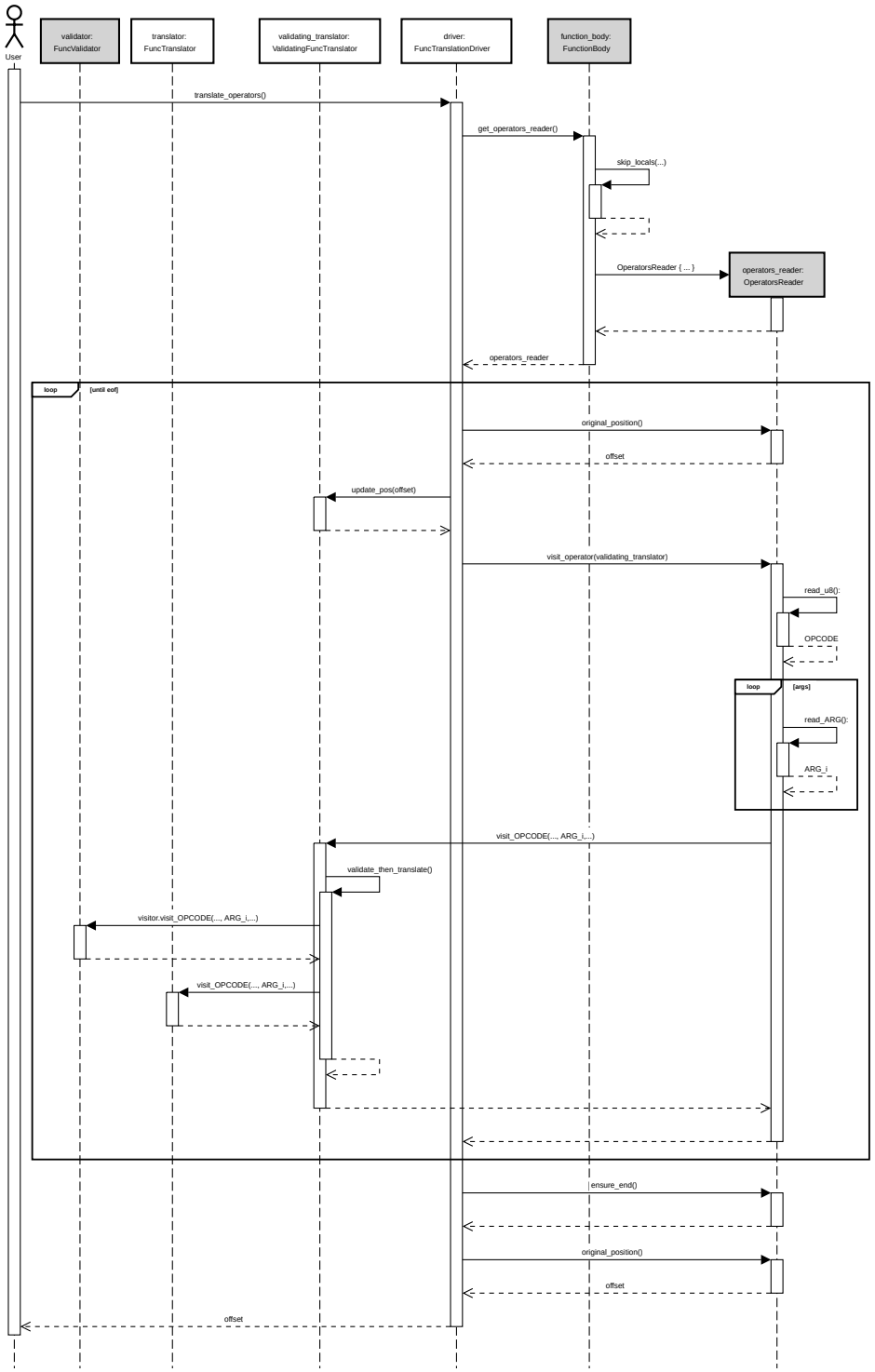
Function Parsing, Validation and Translation [CompilationMode::Eager]:
Overview



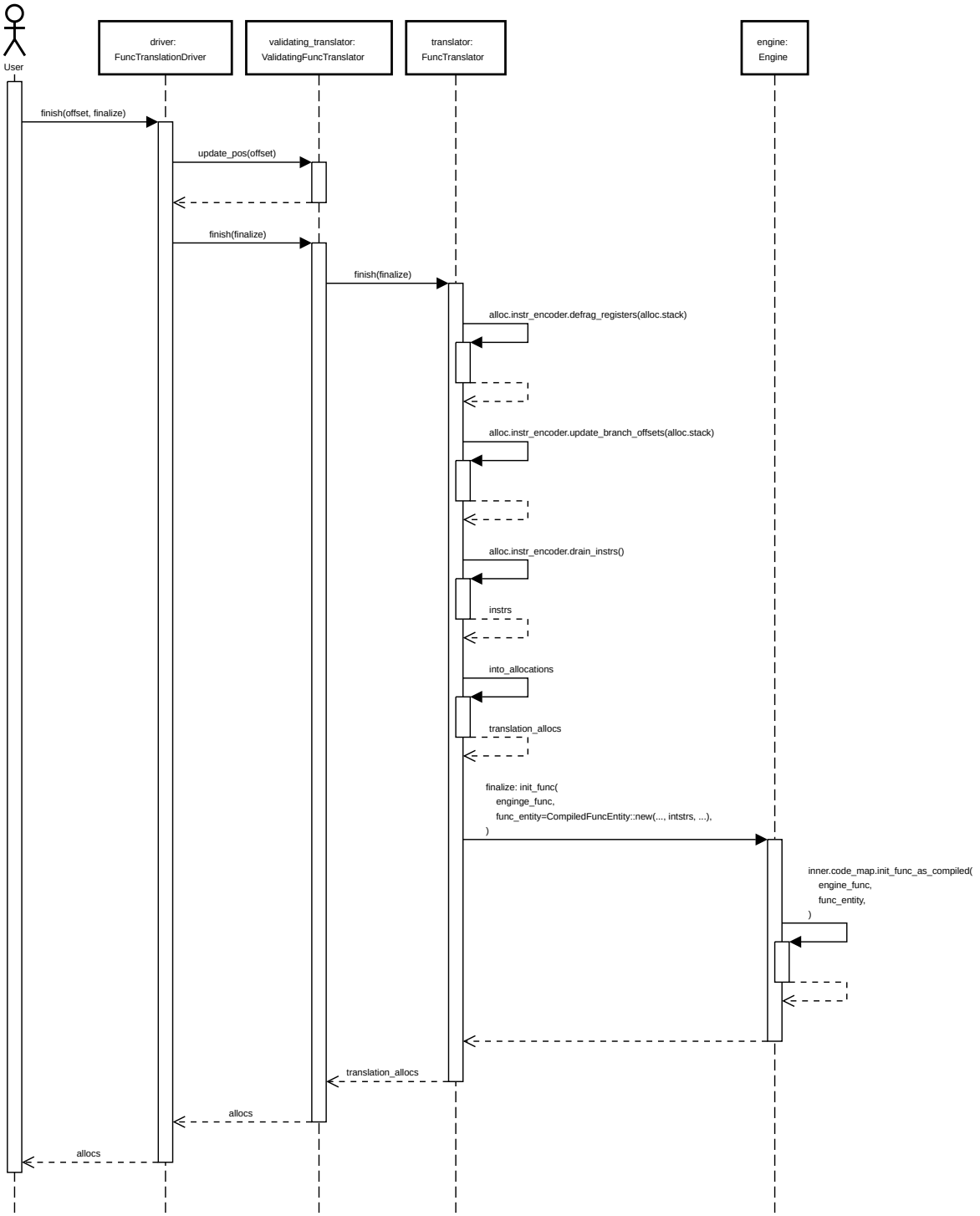
Function Parsing, Validation and Translation [CompilationMode::Eager]:
1. Local Translation



Function Parsing, Validation and Translation [CompilationMode::Eager]:
Operator Translation



Function Parsing, Validation and Translation [CompilationMode::Eager]:
Finalization



Appendix: Algorithmic Description of the Translator

Control Instructions

`nop`

Nothing to do.

`unreachable`

1. If the instruction is unreachable, return.
2. Else, encode a `trap` instruction.
3. Set reachability to false.

`block`

1. If the instruction is unreachable, push the control stack to track the scope of the block, then return.
2. Preserve all local variables: replace all local registers on the value stack by a preservation register allocated for the given local, and encode `copy*` instructions that copy locals to their preservations.
3. Create new label for the end of the block.
4. Allocate new dynamic registers for the block parameters and results.
5. Push the control stack.

`loop`

1. If the instruction is unreachable, push the control stack to track the scope of the block, then return.
2. Preserve all local variables.
3. Push dynamic registers for block parameters.
4. Encode copy instructions for the arguments.
5. Create new label for the head of the loop, then pin it.
6. Push the control stack.

`if`

1. If the instruction is unreachable, push the control stack to track the scope of the block, then return.
2. Preserve all local variables.
3. Create new label for the `end` of the `if-else-end` (or `if-end`).
4. Allocate new dynamic registers for the block parameters and results.
5. Process the condition.
   - If the condition is a constant, then only one of the branches is deemed reachable (`IfReachability`), administer which one.
     If only the `else` is deemed reachable, set reachability to false.
   - Otherwise, the condition is a register, and both branches are deemed reachable.
     1. Store the input parameters for processing in the `else` branch.
     2. Ensure that preservation registers are kept for the `else` branch.
     3. Create a new label for the `else` branch.
     4. Encode the branch instruction.
6. Push the control stack.

`else`

1. Pop the control stack.
   If the frame belongs to an unreachable `if` instruction, then the `else` block is also unreachable.
   Push the control stack to track the scope of the block, then return.
2. Administer that the `else` block has been visited. This indicates that the control frame is `if-else-end` and not `if-end`.
3. If the `then` branch was deemed reachable, administer the reachability of the end of the `then` block. This is important so that the reachability of the whole `if` can be established upon visiting the corresponding `end` instruction.
4. If the `else` branch was deemed reachable, and there is an associated label for it, it means that both both branches were deemed reachable.
   1. If the end of the `then` block was reachable, encode copy instructions for the `then` branch result, and encode the jump to the `end` of the `if`.

2. Set reachability to true.
3. Pin the `else` label.
4. Restore input parameters.

5. Set reachability based on which of the branches were deemed reachable.
   - The case for none of the branches is not possible.
   - The case for both branches has already been handled.
   - If only the `else` branch has been deemed reachable, then reachability is set to false. Set it to true.
   - If only the `then` branch has been deemed reachable, set reachability to false.

6. Push the control stack.

`end`

Pop the control stack. There are four cases based on the control frame.

`end` of an unreachable block

Nothing to do.

`end` of a `loop`

Nothing to do.

`end` of a `block`

1. If the control stack is empty, that means the `end` corresponds to the top-level function call. Encode the return instruction.
2. If the end of the block is reachable and is branched to, encode copy instructions for the block results.
3. Pin the block label.
4. If the block is branched to, push the result registers onto the value stack.
5. Set reachability to true if the current instruction is reachable or the block is branched to.

`end` of an `if`

There are five cases.

Case 1: No branches deemed reachable

This case is not possible.

Case 2: Both branches deemed reachable, `else` block exists

1. Compute reachability of the code following the `if-else-end`, based on the reachability of the end of the branches and whether the block has been branched to.
2. Pin the `else` label.
3. If the end of the `else` is reachable, encode copy instructions for the `else` branch result,
4. Pin the `end` label.
5. If the code following the `if-else-end` is reachable, push result registers onto the value stack.

Case 3: Both branches deemed reachable, `else` block missing

1. If the end of the `then` branch is reachable and the branch produced results, encode copy instructions for the `then` branch result, and encode the jump to the `end` of the `if`.
2. Pin the `else` label.
3. Pop the input parameters previously stored for the `else` branch.
4. If the block produces results, encode copies from those parameters to the output registers.
5. Pin the `end` label.
6. Push the result registers onto the value stack.
7. Set reachability to true.

Case 4: Only `then` deemed reachable

1. Compute reachability of the end of the `then` branch.
    - This value is already set iff the `else` branch has been visited.
    - Otherwise, its the reachability of the current instruction.
2. If the end of the `then` is reachable and the block has been branched to, encode copy instructions for the `then` branch result.
3. Pin the `end` label.
4. If the block has been branched to, push the output registers to the value stack.
5. Compute reachability of the code following the `if-else-end`, based on the reachability of the end of the `then` and whether the block has been branched to.

Case 5: Only `else` deemed reachable

This is symmetric to the previous case, the only difference being in how the reachability of the end of the block is computed:

- It is reachable if the current instruction is reachable, or the `else` has never been visited (i.e. it is an `if-end` instruction).

`br`

1. If the instruction is unreachable, return.
2. If the branch target is the top-level frame, encode a `return` instruction, then return.
3. Otherwise, increment the number of branches for the given frame.
4. Encode copy instructions for the target branch parameters (i.e. inputs for a `loop` frame, outputs otherwise).
5. Resolve the target frame's label.
6. Encode the branch instruction with the resolved offset.
7. Set reachability to false.

`br_if`

1. If the instruction is unreachable, return.
2. Pop the value stack for the condition.
    - If the value is zero, return.
    - If it is a non-zero constant, translate the instruction as `br`.
    - Otherwise, the value is a register, continue below.
3. If the branch target is the top-level frame, encode a `return_nez` instruction, then return.
4. Otherwise, increment the number of branches for the given frame.
5. If the target frame has no branch parameters, encode a `branch_nez` instruction, fusing with the previous instruction if possible.
6. Do the same if the values on top of the stack are already the branch parameters of the target branch.
7. Otherwise, create a new label, say, `L_skip`.
8. Encode a `branch_eqz` instruction to `L_skip`, fusing with the previous instruction if possible.
9. Encode copy instructions for the target branch parameters.
10. Resolve the target frame's label.
11. Encode the branch instruction with the resolved offset.
12. Pin `L_skip`.

`br_table`

1. If the instruction is unreachable, return.
2. Pop the value stack for the branch index.
3. If the instruction only has a default target, translate it as a `br` to the default target.
4. If the branch index is a constant, translate the instruction as a `br` to the given target.
5. If the target branches do not have branch parameters, then
    - Encode a `branch_table` instruction.
    - For each of the targets:
        - If it is the top-level frame, encode `return`.
        - Otherwise, resolve the corresponding label, bump branches to the frame and encode a `branch` with the resolved offset.
    - Set reachability to false.
6. Otherwise, for each unique branch target, introduce a label, say, `L_i`.
7. For each branch target, encode a `branch` to the corresponding `L_i`.
8. Pop a number of values from the value stack equal to the number of branch parameters.
9. For each `L_i`:
    - If `L_i` corresponds to the top-level frame, encode a `return` with the values.
    - Otherwise:
        - Pin `L_i`.
        - Bump the number of branches to the target frame.
        - Encode copies from the values to the branch parameters.
        - Encode a `branch` to the actual target.
10. Set reachability to false.

`return`

Pop the value stack the right number of types and encode a `return` instruction with the values.

`call`

1. If the instruction is unreachable, return.

2. Pop a number of values from the value stack equal to the number of parameters of the called function.
3. Allocate a register span for the results.
4. Based on whether the function is internal or imported, encode a `call_*` instruction with the result registers and the function index.
5. Encode register list instructions for the parameters.

`call_indirect`

1. If the instruction is unreachable, return.
2. Pop the index from the value stack.
3. Pop a number of values from the value stack equal to the number of parameters of the function type of the instruction.
4. Allocate a register span for the results.
5. Encode a `call_indirect` instruction with the result registers and the type index.
6. Encode a `call_indirect_params` instruction with the index and the table index.
7. Encode register list instructions for the parameters.

`return_call`, `return_call_indirect`

These are tail-call versions of the respective `call*` instructions. The difference in translation is that instead of `call*` they emit `return_call*` instructions with no results, and set reachability to false after translation.

Parametric Instructions

`drop`

1. If the instruction is unreachable, return.
2. Otherwise, pop an element from the value stack.

`select`

1. If the instruction is unreachable, return.
2. Pop `lhs`, `rhs` and `condition` from the value stack.

3. If `condition` is a constant, choose the corresponding value. If it is a dynamic or preservation register, encode a `copy` to a fresh dynamic register. Otherwise, just push the register back on the value stack and return.
4. If `lhs = rhs`, push `lhs` back onto the value stack and return.
5. Otherwise, encode the corresponding `select` instruction based on the types and whether the value is a register or constant.

Variable Instructions

`local.get`

1. If the instruction is unreachable, return.
2. Push the local register onto the value stack.

`local.set`

1. If the instruction is unreachable, return.
2. Pop the value from the value stack.
3. If the value is the local itself, return.
4. Preserve local register: if the local register is present on the value stack, allocate a new preservation register, and replace all occurrences of the local on the value stack by it.
5. An optimization is possible if:
    - The value is not a local or preservation register;
    - There is a previous instruction within the current basic block.
    - If the local has to be preserved, then:
        - The previous instruction has a small encoding (i.e. max `N` sub-instructions for some predefined `N`). Preservation and the optimization together require shifting the encoding (see below), the restriction on size bounds the translation overhead.
        - And the encoding does not use the preservation register as input.

    In this case:

    1. Relink the output of the previous instruction to the local variable.

2. If the local has to be preserved, insert a `copy` instruction from local register to preservation register (shifting the previous instruction).

6. Otherwise, if the local has to be preserved, encode a `copy` instruction from local register to preservation register.
7. Then encode a `copy` of the value to the local.

A note on local preservation

Preserving local registers requires replacing entries of the whole value stack, thus can be a costly operation. For small value stack sizes, Wasmi traverses the whole value stack to check for occurrences of the given local variable. If the value stack grows beyond a certain size however, implementation keeps track of value stack indexes for locals (`LocalRefs` data structure). In the worst case, the operation is linear in value stack size even in this case.

`local.tee`

1. If the instruction is unreachable, return.
2. Perform the translation logic for `local.set`.
3. Push the value set back onto the value stack.

`global.get`

1. If the instruction is unreachable, return.
2. Otherwise, if the global is a constant with an initializer expression, evaluate the initializer.
    - If the value is a numeric constant, push that on the value stack and return.
    - If the value is a function reference, process it as `ref.func` and return.
3. Allocate a dynamic register for the result.
4. Encode a `global_get` instruction.

`global.set`

1. If the instruction is unreachable, return.
2. Pop a value from the value stack.
3. Encode a `global_set*` instruction, based on value.

Reference Instructions

`ref.null`

1. If the instruction is unreachable, return.
2. Push the null value corresponding to the reference type (function vs. external) onto the value stack.

`ref.is_null`

1. If the instruction is unreachable, return.
2. Pop the value stack.
3. If the value is a const, push the constant result on the value stack.
4. Otherwise, encode an `i64_eq_imm16` instruction.

`ref.func`

1. If the instruction is unreachable, return.
2. Allocate a register for the result.
3. Encode a `ref_func` instruction.

Numeric Instructions

Algorithmically, translation of numeric instractions is straightforward. In the general case, operands are popped from the value stack, a dynamic regiter is puhed for the result, and the Wasmi instruction corresponding to the Wasm instruction and the operand types is encoded. However, the translation algorithm also attempts several optimizations:

- Constant propagation: if all operands are constants, evaluate the instruction, and push the result onto the value stack. Where possible, this is generalized to special cases involving register operands, e.g. instead of encoding `i32_add_imm16 x 0`, simply 0 is poushed onto the value stack.
- Peephole optimization: replace an instruction by an equivalent one. E.g. instead of `i32_sub_imm16 x 3`, instruction `i32_add_imm16 x (-3)` is encoded.
- Op-code fusion: the instructions is combined with the previous one. As a simple example, instead of translating to `i32_and ; i32_eqz`, a single `i32_and_eqz` instructions is

emitted.

Vector Instructions

Not supported.

Table Instructions

`table.get`

1. If the instruction is unreachable, return.
2. Pop the index from the value stack.
3. Allocate a new dynamic register for the result.
4. Encode a `table_get*` instruction corresponding to the index type (immediate / register).
5. Encode a `table_idx` instruction with the table index.

`table.set`

1. If the instruction is unreachable, return.
2. Pop the index and the value from the value stack.
3. If the value is a constant, allocate a register for it in the constant space.
4. Encode a `table_set*` instruction corresponding to the index type (immediate / register).
5. Encode a `table_idx` instruction with the table index.

`table.size`

1. If the instruction is unreachable, return.
2. Allocate a new dynamic register for the result.
3. Encode a `table_size` instruction.

`table.grow`

1. If the instruction is unreachable, return.
2. Pop the value and the delta from the value stack.
3. If the delta is zero, process as `table.size` then return.
4. If the value is a constant, allocate a register for it in the constant space.

5. Allocate a new dynamic register for the result.
6. Encode a `table_grow*` instruction corresponding to the delta type (immediate / register).
7. Encode a `table_idx` instruction with the table index.

`table.fill`

1. If the instruction is unreachable, return.
2. Pop the destination, value and length from the value stack.
3. If the value is a constant, allocate a register for it in the constant space.
4. Encode a `table_fill*` instruction corresponding to the destination and length type (immediate / register).
5. Encode a `table_idx` instruction with the table index.

`table.copy`

1. If the instruction is unreachable, return.
2. Pop the destination, source and length from the value stack.
3. Encode a `table_copy*` instruction corresponding to the destination, source and length type (immediate / register).
4. Encode a `table_idx` instruction with the destination table index.
5. Encode a `table_idx` instruction with the source table index.

`table.init`

1. If the instruction is unreachable, return.
2. Pop the destination, source and length from the value stack.
3. Encode a `table_init*` instruction corresponding to the destination, source and length type (immediate / register).
4. Encode a `table_idx` instruction with the table index.
5. Encode a `elem_idx` instruction with the element index.

`elem.drop`

1. If the instruction is unreachable, return.
2. Encode an `elem_drop` instruction.

Memory Instructions

`i<SIZE>.load(memarg)`, `f<SIZE>.load(memarg)`, `i<SIZE>.load<W>_<SIGNED>(memarg)`

Where `SIZE = 32,64`, `N = 8,16,32`, `SIGNED = s,u`. Implemented by `translate_load`.

1. If the instruction is unreachable, return.
2. Read `offset` from instruction argument.
3. Reserve a fresh register `result`.
4. If the `address` argument on the stack is a constant:
   - Ensure `address + offset` does not overflow `u32` (otherwise fail translation).
   - Emit `I<SIZE>LoadAt`, `F<SIZE>LoadAt`, `I<SIZE>Load<W><SIGNED>At` with `result` register and calculated `address + offset`.
5. Otherwise (i.e., `address` argument on stack is a register `reg`):
   - If `offset` fits into in 16 bit:
     - Emit `I<SIZE>LoadOffset16`, `F<SIZE>LoadOffset16`, `I<SIZE>Load<W><SIGNED>Offset16` with `result`, `reg`, and `offset`.
   - Otherwise:
     - Emit `I<SIZE>Load`, `F<SIZE>Load`, `I<SIZE>Load<W><SIGNED>` with `result` and `reg`.
     - Followed by a second instruction `Const32` with the `offset` value.

All cases use fuel cost for one `load` instruction.

`i<SIZE>.store(memarg)`, `i<SIZE>.store<N>(memarg)`

Where `SIZE = 32,64`, `N = 8,16,32`. Implemented by `translate_istore`.

1. If the instruction is unreachable, return.
2. Read `offset` from instruction argument.
3. There are four cases, depending on the nature of the top two stack elements `value` and `addr`:
   - Both `addr` and `value` are registers:
     - If `offset` can fit into 16 bit:

        - Emit `I<SIZE>StoreOffset16(addr,offset,value)`, `I<SIZE>Store<N>Offset16(addr,offset,value)`.
    - Otherwise:
        - Emit `I<SIZE>Store(addr,value)`, `I<SIZE>Store<N>(addr,value)`.
        - Followed by a second instruction `Const32` with the `offset` value.
- `addr` is a register and `value` is a constant:
    - If `offset` fits into 16 bit:
        - If `value` fits into 16 bit:
            - Emit `I<SIZE>StoreOffset16Imm(addr,offset,value)` (`I<SIZE>Store<N>Offset16Imm(addr,offset,value)`).
        - Otherwise:
            - Allocate a new function local constant for `value` on the value stack, use its register `valreg`.
            - Emit `I<SIZE>StoreOffset16(addr,offset,valreg)` (`I<SIZE>Store<N>Offset16(addr,offset,valreg)`).
    - Otherwise:
        - Allocate a new function local constant for `value` on the value stack, use its register `valreg`.
        - Emit `I<SIZE>Store(addr,offset)` (`I<SIZE>Store<N>(addr,offset)`).
        - Followed by a second instruction `Register(valreg)` to supply the `value` from the constant register.
- `addr` is a constant and `value` is a register:
    - Ensure `addr + offset` does not overflow `u32` (otherwise fail translation).
    - Emit `I<SIZE>StoreAt(addr + offset, value)` (`I<SIZE>Store<N>At(addr + offset, value)`).
- Both `addr` and `value` are constants:
    - Ensure `addr + offset` does not overflow `u32` (otherwise fail translation).
    - If `value` fits into 16 bit:
        - Emit `I<SIZE>StoreAtImm(addr + offset, value)` (`I<SIZE>Store<N>AtImm(addr + offset, value)`).

- Otherwise:
  - Allocate a new function local constant for `value` on the value stack, use its register `valreg`.
  - Emit `I<SIZE>StoreAt(addr + offset, valreg)` (`I<SIZE>Store<N>AtImm(addr + offset, valreg)`).

All cases use fuel cost for one `store` instruction.

`memory.size`

1. If the instruction is unreachable, return.
2. Translate directly to `MemorySize` (using a fresh register for the result), adding fuel cost for an `entity` instruction.

`memory.grow`

1. If the instruction is unreachable, return.
2. Translate directly to `MemoryGrow[By](size)` depending on `size` argument (using a fresh register for the result), adding fuel cost for an `entity` instruction.

`memory.init(data_index)`

1. If the instruction is unreachable, return.
2. Translate directly to `MemoryInit*(dst,src,len)` (depending on arguments `dst`, `src`, and `len`), adding fuel cost for an `entity` instruction.
3. Emit a `DataSegmentIdx` instruction.

`memory.copy`

1. If the instruction is unreachable, return.
2. Translate directly to `MemoryCopy*(dst,src,len)` (depending on arguments `dst`, `src`, and `len`), adding fuel cost for an `entity` instruction.

`memory.fill`

1. If the instruction is unreachable, return.

2. Translate directly to `MemoryFill*(dst,val,len)` (depending on arguments `dst`, `val`, and `len`), adding fuel cost for an `entity` instruction.

`data.drop(data_index)`

1. If the instruction is unreachable, return.
2. Translate directly to `DataDrop(data_index)`, adding fuel cost for an `entity` instruction.

Appendix: `unsafe` Rust Checklist

There are 112 usages of the `unsafe` keyword in wasmi v0.36.0 that are in scope of the audit. Below is a table that categorizes and analyses each occurrence. But first some definitions and context are required.

The unsafe code in Wasmi often involves the management of raw pointers. Raw pointers themselves are not inherently unsafe, and so long as they are *valid*, *aligned* correctly, and handled correctly according to the specific safety comments then no *undefined behaviour* will occur.

Validity
Validity of a pointer is best described in the the Rust documentation for rust/library/core/src/ptr/mod.rs:

> - For operations of size zero, every pointer is valid, including the null pointer.
>
> The following points are only concerned with non-zero-sized accesses.
>
> - A null pointer is never valid.
> - For a pointer to be valid, it is necessary, but not always sufficient, that the pointer be dereferenceable: the memory range of the given size starting at the pointer must all be within the bounds of a single allocated object. Note that in Rust, every (stack-allocated) variable is considered a separate allocated object.
> - All accesses performed by functions in this module are non-atomic in the sense of atomic operations used to synchronize between threads. This means it is undefined behavior to perform two concurrent accesses to the same location from different threads unless both accesses only read from memory. Notice that this explicitly includes read_volatile and write_volatile: Volatile accesses cannot be used for inter-thread synchronization.
> - The result of casting a reference to a pointer is valid for as long as the underlying object is live and no reference (just raw pointers) is used to access the same memory. That is, reference and pointer accesses cannot be interleaved.

Alignment

Alignment of a pointer is best described in the rust documentation for rust/library/core/src/ptr/mod.rs:

> Valid raw pointers as defined above are not necessarily properly aligned (where “proper” alignment is defined by the pointee type, i.e., *const T must be aligned to mem::align_of::()). However, most functions require their arguments to be properly aligned, and will explicitly state this requirement in their documentation. Notable exceptions to this are read_unaligned and write_unaligned.
>
> When a function requires proper alignment, it does so even if the access has size 0, i.e., even if memory is not actually touched.

Allocated Object

Above a pointer is described as being valid if the address is part of a memory range that is considered an "allocated object". Allocated Objects are best described in the rust documentation for rust/library/core/src/ptr/mod.rs:

> An allocated object is a subset of program memory which is addressable from Rust, and within which pointer arithmetic is possible. Examples of allocated objects include heap allocations, stack-allocated variables, statics, and consts. The safety preconditions of some Rust operations - such as offset and field projections (expr.field) - are defined in terms of the allocated objects on which they operate.
>
> An allocated object has a base address, a size, and a set of memory addresses. It is possible for an allocated object to have zero size, but such an allocated object will still have a base address. The base address of an allocated object is not necessarily unique. While it is currently the case that an allocated object always has a set of memory addresses which is fully contiguous (i.e., has no “holes”), there is no guarantee that this will not change in the future.

Undefined Behaviour

Undefined Behaviour (UB) in Rust is any code that exhibits behaviours defined in The Rust Reference chapter 16.2 Behavior considered undefined. Any code that triggers UB is

considered *unsound*. Safe code cannot trigger UB and so is automatically *sound*, while unsafe code is considered *sound* only if it cannot trigger UB.

Unsafe Rust Features

Unsafe Rust allows 5 features that safe rust does not have. Here is a legend to classify them, and another item for declaration, that will be used to categorise the type of `unsafe` in the analysis.

LEGEND:
DEREF - Dereference a raw pointer
CALL - Call an unsafe function or method
DECL - Declaration of an unsafe function or method
STATIC - Access or modify a mutable static variable
TRAIT - Implement an unsafe trait
UNION - Access fields of a `union`

Source	Type	Precondition	Notes
ExternRef::From#L125	UNION	`u64` bit pattern must be valid as `ExternRef`	size checked to be equal, contains `u32` (always valid) and *non-zero* `u32` (`ExternObject` type), `None` for zero ("niche" value [^1][^2]).
UntypedValue::From#L137	UNION	`ExternRef` bit pattern must be a valid `u64`	any bit patterns is valid
Externref::canonicalize#L168	UNION	`0_u64` must be a valid `ExternRef`	is valid (encoding a wrapped `None`) [^1]
/wasmi/src/memory/buffer.rs#L37	TRAIT	`Send` for `ByteBuffer` type	Pointer can be shared, rest is `Send`
ByteBuffer::data#L140	CALL	`slice::from_raw_parts`	`ByteBuffer` fields (`ptr` , `len`) consistent by construction
ByteBuffer::data_mut#L149	CALL	`slice::from_raw_parts`	`ByteBuffer` fields (`ptr` , `len`) consistent by construction
ByteBuffer::get_vec#L167	CALL	`Vec::from_raw_parts`	`ByteBuffer` fields (`ptr` , `len` , `capacity`) consistent by construction
/wasmi/src/memory/buffer.rs#L203	DEREF	dereferences a raw pointer (`core::ptr::addr_of_mut!`) for test	Test code only
/wasmi/src/memory/buffer.rs#L209	DEREF	dereferences a `static mut`	Test code only
/wasmi/src/memory/buffer.rs#L225	DEREF	dereferences a raw pointer (`core::ptr::addr_of_mut!`) for test	Test code only
/wasmi/src/memory/buffer.rs#L241	DEREF	dereferences a raw pointer (`core::ptr::addr_of_mut!`) for test	Test code only
/wasmi/src/engine/resumable.rs#L109	TRAIT	X	Safe since `InstructionPtr` safely implements `Sync` . More explanation in Safety comment
CodeMap::get_compiled#L297	CALL	Must be unreachable.	Reliant on guarantees from translator. See hint::unreachable_unchecked
CodeMap::adjust_cref_lifetime#L330	CALL	Data must not be moved or invalidated.	`CompiledFuncRef` only has reference to `Pin` data. No unsafe `Pin` methods are called and the data does not have interior mutability. As Safety comment mentions `CodeMap` is append only. See intrinsics.rs::transmute
CodeMap::get_uncompiled#L473	CALL	Must be unreachable.	unreachable by construction. See hint::unreachable_unchecked
CodeMap::compile#L627	CALL	memory must be initialised	Saftey comment needs to be added. Memory is initialised due to `result.write` called on both paths of match statement. See MaybeUninit::assume_init
Stack::merge_call_frames#L81	DECL	No `FrameRegisters` can reference the drained `CallFrame` , these pointers will be invalid.	Validity must be handled at call site. Not *strictly* required as no unsafe directly appears in the body, perhaps `ValueStack::Drain` should be unsafe although it does not call unsafe code directly either.
ValueStack::stack_ptr_at#L117	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	No checks, DECL unsafe so must be enforced at call site. All call sites provide argument of `CallFrame::base_offset()` , so soundness is dependent on `CallSack` data. See <*mut T>::add

Source	Type	Precondition	Notes
ValueStack::extend_by#L150	CALL	`new_len` must be less than capacity. Elements at `old_len..new_len` must be initialised.	Defaults for capacity are sound, although it is possible config with bad capacity, it should error in call to `reserve`. `new_len` is within range of capacity as `reserve` is called. All calls have `on_resize` either `do_nothing` or shift the stack pointer. Values are not initialised in this function, instead a slice of `MaybeUninit` is returned, this has appropriate handling of uninit values and no UB should be possible. See Vec::set_len
ValueStack::extend_by#L151	CALL	The slice data must be valid for reads and writes. The data must be initialised. Access to data must be exclusive to slice for lifetime of slice. len * size_of::() cannot overflow isize.	The slice data is valid as the line above ensures it is within vec capacity. Slice data is not initialised and is `MaybeUninit`, and so call site should enforce correctness as mentioned above. Not clear access is exclusive at this point, however the only possible accesses that would violate this would be raw pointers. Not clear that size could not overflow, extra checks should be added to ensure. Note: All calls have `on_resize` either `do_nothing` or shift the stack pointer. See slice::from_raw_parts_mut
ValueStack::drop#L177	CALL	`new_len` must be less than capacity. Elements at `old_len..new_len` must be initialised.	`new_len` is in range. Assuming values were already initialised, shortening the length is valid - other unsafe functions responsibility ensure they are initialised. See Vec::set_len
ValueStack::drop_return#L188	CALL	`index` must be in bounds of slice	This appears to be able to be violated, a check should be added here. See slice::get_unchecked
ValueStack::drop_return#L190	CALL	The slice data must be valid for reads. The data must be initialised. Access to data must be read only for other references to slice for lifetime of slice. len * size_of::() cannot overflow isize.	If the check from above is included then the validity and initialisation should be sound. Not clear if there is other access and is shared read only, these would need to occur through raw pointers though. Not clear that size could not overflow, extra checks should be added to ensure. See slice::from_raw_parts
ValueStack::truncate#L203	CALL	`new_len` must be less than capacity. Elements at `old_len..new_len` must be initialised.	`new_len` is in range. Assuming values were already initialised, shortening the length is valid - other unsafe functions responsibility ensure they are initialised. See Vec::set_len
FrameParams::init_next#L368	CALL	`ptr::write`: `FrameParams.range.start` must be valid for writes, and must be properly aligned. `<*mut T>::add`: Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	`ptr::write`: Validity and alignment must be established by calling context, function is declared unsafe. `<*mut T>::add`: Offset in bytes cannot overflow isize as count is 1. If `FrameParams.range.end <= FramePar` is already the case, offset may point to a different allocated object or wrap, since function is declared unsafe this is the responsibility of calling context to enforce. See ptr::write. See <*mut T>::add

Source	Type	Precondition	Notes
FrameParams::init_zeroes#L378	CALL	`ptr::write` : `FrameParams.range.start` must be valid for writes, and must be properly aligned. `<*mut T>::add` : Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	`ptr::write` : Pointer alignment and validity not explicitly enforced in funciton, but all call sites follow `ValueStack::alloc_call_frame` which is a valid context. <*mut T>::add: Offset in bytes cannot overflow isize as count is 1. `debug_assert` and calling context enforce these predicates, but the `debug_assert` could be added to extra checks. See FrameParams::init_next
FrameRegisters::get#L409	CALL	`ptr::read` : `src` must point to initialised memory. `src` must be valid for read. `src` must be aligned. `FrameRegisters::register_offset` : `count * size_of::<UntypedVal>()` must not overflow `isize` . Original pointer and offset pointer must be part of same allocated object without wrapping.	All `ptr::read` invariants (initialised, valid, aligned) are dependent on `FrameRegisters::register_offset` invariants being upheld, which (since the function is declared unsafe) must be determined by call site (only `Executor::get_register`). See ptr::read. See FrameRegisters::register_offset
FrameRegisters::set#L419	CALL	`ptr::write` : `dst` must be valid for writes. `dst` must be aligned. `FrameRegisters::register_offset` : `count * size_of::<UntypedVal>()` must not overflow `isize` . Original pointer and offset pointer must be part of same allocated object without wrapping.	All `ptr::write` invariants (valid, aligned) are dependent on `FrameRegisters::register_offset` invariants being upheld, which (since funtion is declared unsafe) must be determined by call site. See ptr::write. See FrameRegisters::register_offset
FrameRegisters::register_offset#L424	DECL	X	X
FrameRegisters::register_offset#L425	CALL	`count * size_of::<UntypedVal>()` must not overflow `isize` . Original pointer and offset pointer must be part of same allocated object without wrapping.	By [C5] analysis, and given that `count` is converted i16, `count` maximum possible is `i16::MAX` which will not overflow for 32bit and larger wordsize. If wordsize is 16bit, then the `count <= 4095` for soundness. Unclear that the offset would be part of same allocated object without wrapping, since funciton is marked unsafe this would need to determined by call site `FrameRegisters::{get
CachedInstance::as_ref#L80	DECL	X	X
CachedInstance::as_ref#L81	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See NonNull<T>::as_ref
CachedInstance::update_memory#L101	DECL	X	X
CachedInstance::update_memory#L102	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_func#L115	DECL	X	X
CachedInstance::get_func#L116	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_memory#L126	DECL	X	X
CachedInstance::get_memory#L127	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_table#L137	DECL	X	X

Source	Type	Precondition	Notes
CachedInstance::get_table#L138	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_global#L148	DECL	X	X
CachedInstance::get_global#L149	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_data_segment#L159	DECL	X	X
CachedInstance::get_data_segment#L160	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_element_segment#L170	DECL	X	X
CachedInstance::get_element_segment#L171	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See CachedInstance::as_ref#81
CachedInstance::get_func_type_dedup#L181	DECL	X	X
CachedInstance::get_func_type_dedup#L182	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe.
CachedMemory::data#L233	DECL	X	X
CachedMemory::data#L234	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe. See NonNull<T>::as_ref
CachedMemory::data_mut#L243	DECL	X	X
CachedMemory::data_mut#L244	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, function is declared unsafe.
CachedGlobal::get#L306	DECL	X	X
CachedGlobal::get#L309	CALL	`src` must point to initialised memory. `src` must be valid for read. `src` must be aligned.	Must be established at call site, function is declared unsafe. See ptr::read
CachedGlobal::set#L318	DECL	X	X
CachedGlobal::set#L321	CALL	`dst` must be valid for writes. dst` must be aligned.	Must be established at call site, function is declared unsafe. See ptr::write
EngineExecutor::execute_root_func#L211	CALL	`ptr::write` : `FrameParams.range.start` must be valid for writes, and must be properly aligned. `<*mut T>::add` : Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	`ptr::write` : The access is properly aligned and is valid for writes, but it should be noted if there are constants allocated in the frame they are overwritten. <*mut T>::add: Offset in bytes cannot overflow isize as count is 1. Previous call to `ValueStack::alloc_call_frame` ensures valid range, therefore offset is in same allocated object without wrapping. See FrameParams::init_next
EngineExecutor::resume_func#L272	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset` , extra checks are recommended. See ValueStack::stack_ptr_at
EngineExecutor::resume_func#L276	CALL	From `FrameRegisters::register_offset` : count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set

Source	Type	Precondition	Notes
Executor::new#L119	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset`, extra checks are recommended. See ValueStack::stack_ptr_at
Executor::get_entity!#L882	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Transitively calls `CachedEntity::get_*`. Unclear that invariants would hold, must be established at call site, expanded functions are not labelled as unsafe, so extra checks are recommended. See CachedInstance::as_ref
Executor::get_register#L923	CALL	From `FrameRegisters::register_offset`: `count * size_of::<UntypedVal>()` must not overflow `isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, and it depends on the callsite of which there are many. This funciton should be declared as unsafe or there should be extra checks enabled. See FrameRegisters::get
Executor::set_register#L941	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, and it depends on the callsite of which there are many. This function should be declared as unsafe or there shoul dbe extra checks enabled. See FrameRegisters::set
Executor::frame_stack_ptr_impl#L993	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset`, extra checks are recommended. See ValueStack::stack_ptr_at
Executor::execute_load_extend#L39	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Must be established at call site, however it the function isn't marked as unsafe, extra checks should be added to ensure the invariants are upheld.
Executor::dispatch_compiled_func#L224	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset`, extra checks are recommended. Part of a closure. See ValueStack::stack_ptr_at
Executor::copy_regs#L272	CALL	`ptr::write`: `FrameParams.range.start` must be valid for writes, and must be properly aligned. `<*mut T>::add`: Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	`Executor::copy_regs` soundness depends on call site, which is always the call site of `Executor::copy_call_params` which takes `uninit_params:FrameParams` as argument. In particular is the responsibility of the call site to allocate enough range for all calls `Executor::copy_reg` to be sound. Two calls to `Executor::copy_call_params` exist, one preceeded by `ValueStack::extend_by`, the other by `ValueStack::alloc_call_frame`, it is not clear that these allocate enough range and extra checks should be added. `ptr::write`: `<*mut T>::add`: See FrameParams::init_next
Executor::prepare_compiled_func_call#L317	CALL	No `FrameRegisters` can reference the drained `CallFrame`, these pointers will be invalid.	Not clear that the `CallFrame` on the head of the stack cannot have dangling pointers to it. This would depend on the context of the call site, if possible extra checks should be added. See Stack::merge_call_frames

Source	Type	Precondition	Notes
Executor::execute_host_func#L508	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset`, extra checks are recommended. See ValueStack::stack_ptr_at
Executor::execute_host_func#L534	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set
Executor::return_caller_results#L77	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	Dependent on validity of `CallFrame.base_offset`, extra checks are recommended. See ValueStack::stack_ptr_at
Executor::execute_return_value#L107	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set
Executor::execute_return_reg_n_impl#L152	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set
Executor::execute_return_span#L202	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set
Executor::execut_return_many_impl#L232	CALL	From `FrameRegisters::register_offset`: count * size_of:: () `must not overflow` isize`. Original pointer and offset pointer must be part of same allocated object without wrapping.	Unclear that these invariants hold, requires extra checks. See FrameRegisters::set
Executor::execute_global_get#L16	CALL	`src` must point to initialised memory. `src` must be valid for read. `src` must be aligned.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedGlobal::get
Executor::execute_global_set_impl#L72	CALL	`dst` must be valid for writes. dst` must be aligned.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedGlobal::set
Executor::execute_store_wrap#L55	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedMemory::data_mut
Executor::execute_memory_grow_impl#L90	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Unclear that these invariants hold, it must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedInstance::update_memory

Source	Type	Precondition	Notes
Executor::execute_memory_copy_impl#L234	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedMemory::data_mut
Executor::execute_memory_fill_impl#L381	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedMemory::data_mut
Executor::execute_memory_init_impl#L530	CALL	Pointer must be aligned. Must be "dereferenceable". Referent must be initialised.	Unclear that these invariants hold, must be established at call site however function is not declared unsafe. Extra checks are recommended. See CachedMemory::data_mut
/wasmi/src/engine/bytecode/instr_ptr.rs#L18	TRAIT	X	Read only on Send data
InstructionPtr::offset#L39	CALL	ptr always in bounds and ptr + offset <= i32::MAX	Safety comment does not mention potential to wrap i32, debug_assert can be added to enforce bound. See <*const T>::offset
InstructionPtr::add#L47	CALL	Starting pointer and ending pointer must be part of same allocated object. Offset in bytes cannot overflow isize. Computing offset cannot involve wrapping.	As above. See <*const T>::add
InstructionPtr::get#L62	DEREF	X	Safety comment asserts deref should be safe.
/wasmi/src/engine/bytecode/immediate.rs#L130	CALL	Value must be non-zero	Correct by PhantomData containing non-zero type. See NonZero::new_unchecked
/wasmi/src/engine/bytecode/immediate.rs#L138	CALL	Value must be non-zero	Correct by PhantomData containing non-zero type. See NonZero::new_unchecked
/wasmi/src/engine/bytecode/immediate.rs#L146	CALL	Value must be non-zero	Correct by PhantomData containing non-zero type. See NonZero::new_unchecked
/wasmi/src/engine/bytecode/immediate.rs#L154	CALL	Value must be non-zero	Correct by PhantomData containing non-zero type. See NonZero::new_unchecked
Module::new_unchecked#L256	DECL	input bytecode must be valid and consistent with config	only used for benchmark tests, see benches.rs
Module::new_unchecked#L258	CALL		see parse_buffered_unchecked
Module::new_streaming_unchecked#L281	DECL	input bytecode stream must be valid and consistent with config	never called within codebase
Module::new_streaming_unchecked#L286	CALL		see parse_streaming_unchecked
CustomSectionsIter::next#L125	CALL	str::from_utf8_unchecked	(Safety comment) read bytes constructed from a string before
Module::parse_streaming#L79	CALL		see Module::parse_streaming_impl, validator provided before call
Module::parse_streaming_unchecked	DECL	input bytecode stream must be valid and consistent with config	
Module::parse_streaming_unchecked#L95	CALL		see Module::parse_streaming_impl
Module::parse_streaming_impl	DECL	either input bytecode stream is valid/consistent with config, or validator is provided	
Module::parse_buffered#L24	CALL		see Module::parse_buffered_impl, validator provided before call

Source	Type	Precondition	Notes
Module::parse_buffered_unchecked	DECL	input bytecode must be valid and consistent with config	
Module::parse_buffered_unchecked #L40	CALL		see Module::parse_buffered_impl
Module::parse_buffered_impl	DECL	either input bytecode is valid/consistent with config, or validator is provided	
FuncRef::From#L43	UNION	`u64` bit pattern must be valid as `RefType`	size checked to be equal, contains `u32` (always valid) and *non-zero* `u32` (`Func` type), `None` for zero ("niche" value [^1][^2]).
UntypedVal::From#L55	UNION	`FuncRef` bit pattern must be a valid `u64`	any bit patterns is valid
FuncRef::canonicalize#L78	UNION	`0_u64` must be a valid `FuncRef`	is valid (encoding a wrapped `None`) [^1]
LenOrderStr::From#L172	CALL	`transmute` s a `str` to wrapper struct	wrapper struct declared `repr(transparent)`
/collections/src/arena/mod.rs#L38	TRAIT	`Send` for `Arena` allocation data	Safe if contained `T` is `Send`
/collections/src/arena/mod.rs#L41	TRAIT	`Sync` for `Arena` allocation data	Safe if contained `T` is `Sync` . It is however declared `Send` instead (reported).
/collections/src/arena/component_vec.rs#L16	TRAIT	`Send` for `ComponentVec` vector type	Safe if contained `T` is `Sync`
/collections/src/arena/component_vec.rs#L19	TRAIT	`Sync` for `ComponentVec` vector type	Safe if contained `T` is `Sync`
/wasmi/benches/benches.rs#L189	CALL	bytecode input must be valid and consistent with config	only called on known bytecode in repository

[^1] https://github.com/rust-lang/rust/pull/60300

[^2] https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#niche