WebAssembly Specification

Release 2.0 + Threads (Draft 2023-10-10)

WebAssembly Community Group

Andreas Rossberg (editor)

Oct 10, 2023

Contents

CHAPTER 1

Introduction

1.1 Introduction

WebAssembly (abbreviated Wasm[2]) is a *safe, portable, low-level code format* designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well.

WebAssembly is an open standard developed by a W3C Community Group[1].

This document describes version 2.0 + Threads (Draft 2023-10-10) of the core WebAssembly standard. It is intended that it will be superseded by new incremental releases with additional features in the future.

1.1.1 Design Goals

The design goals of WebAssembly are the following:

- Fast, safe, and portable *semantics*:
  - **Fast**: executes with near native code performance, taking advantage of capabilities common to all contemporary hardware.
  - **Safe**: code is validated and executes in a memory-safe[3], sandboxed environment preventing data corruption or security breaches.
  - **Well-defined**: fully and precisely defines valid programs and their behavior in a way that is easy to reason about informally and formally.
  - **Hardware-independent**: can be compiled on all modern architectures, desktop or mobile devices and embedded systems alike.
  - **Language-independent**: does not privilege any particular language, programming model, or object model.
  - **Platform-independent**: can be embedded in browsers, run as a stand-alone VM, or integrated in other environments.

---

[2] A contraction of "WebAssembly", not an acronym, hence not using all-caps.

[1] https://www.w3.org/community/webassembly/

[3] No program can break WebAssembly's memory model. Of course, it cannot guarantee that an unsafe language compiling to WebAssembly does not corrupt its own memory layout, e.g. inside WebAssembly's linear memory.

  - **Open**: programs can interoperate with their environment in a simple and universal manner.
- Efficient and portable *representation*:
  - **Compact**: has a binary format that is fast to transmit by being smaller than typical text or native code formats.
  - **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.
  - **Efficient**: can be decoded, validated, and compiled in a fast single pass, equally with either just-in-time (JIT) or ahead-of-time (AOT) compilation.
  - **Streamable**: allows decoding, validation, and compilation to begin as soon as possible, before all data has been seen.
  - **Parallelizable**: allows decoding, validation, and compilation to be split into many independent parallel tasks.
  - **Portable**: makes no architectural assumptions that are not broadly supported across modern hardware.

WebAssembly code is also intended to be easy to inspect and debug, especially in environments like web browsers, but such features are beyond the scope of this specification.

1.1.2 Scope

At its core, WebAssembly is a *virtual instruction set architecture (virtual ISA)*. As such, it has many use cases and can be embedded in many different environments. To encompass their variety and enable maximum reuse, the WebAssembly specification is split and layered into several documents.

This document is concerned with the core ISA layer of WebAssembly. It defines the instruction set, binary encoding, validation, and execution semantics, as well as a textual representation. It does not, however, define how WebAssembly programs can interact with a specific environment they execute in, nor how they are invoked from such an environment.

Instead, this specification is complemented by additional documents defining interfaces to specific embedding environments such as the Web. These will each define a WebAssembly *application programming interface (API)* suitable for a given environment.

1.1.3 Security Considerations

WebAssembly provides no ambient access to the computing environment in which code is executed. Any interaction with the environment, such as I/O, access to resources, or operating system calls, can only be performed by invoking functions provided by the embedder and imported into a WebAssembly module. An embedder can establish security policies suitable for a respective environment by controlling or limiting which functional capabilities it makes available for import. Such considerations are an embedder's responsibility and the subject of API definitions for a specific environment.

Because WebAssembly is designed to be translated into machine code running directly on the host's hardware, it is potentially vulnerable to side channel attacks on the hardware level. In environments where this is a concern, an embedder may have to put suitable mitigations into place to isolate WebAssembly computations.

1.1.4 Dependencies

WebAssembly depends on two existing standards:

- IEEE 754[4], for the representation of floating-point data and the semantics of respective numeric operations.
- Unicode[5], for the representation of import/export names and the text format.

However, to make this specification self-contained, relevant aspects of the aforementioned standards are defined and formalized as part of this specification, such as the binary representation and rounding of floating-point values, and the value range and UTF-8 encoding of Unicode characters.

Note: The aforementioned standards are the authoritative source of all respective definitions. Formalizations given in this specification are intended to match these definitions. Any discrepancy in the syntax or semantics described is to be considered an error.

1.2 Overview

1.2.1 Concepts

WebAssembly encodes a low-level, assembly-like programming language. This language is structured around the following concepts.

Values WebAssembly provides only four basic *number types*. These are integers and IEEE 754[6] numbers, each in 32 and 64 bit width. 32 bit integers also serve as Booleans and as memory addresses. The usual operations on these types are available, including the full matrix of conversions between them. There is no distinction between signed and unsigned integer types. Instead, integers are interpreted by respective operations as either unsigned or signed in two's complement representation.

In addition to these basic number types, there is a single 128 bit wide vector type representing different types of packed data. The supported representations are 4 32-bit, or 2 64-bit IEEE 754[7] numbers, or different widths of packed integer values, specifically 2 64-bit integers, 4 32-bit integers, 8 16-bit integers, or 16 8-bit integers.

Finally, values can consist of opaque *references* that represent pointers towards different sorts of entities. Unlike with other types, their size or representation is not observable.

Instructions The computational model of WebAssembly is based on a *stack machine*. Code consists of sequences of *instructions* that are executed in order. Instructions manipulate values on an implicit *operand stack*[8] and fall into two main categories. *Simple* instructions perform basic operations on data. They pop arguments from the operand stack and push results back to it. *Control* instructions alter control flow. Control flow is *structured*, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.

Traps Under some conditions, certain instructions may produce a *trap*, which immediately aborts execution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

Functions Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns a sequence of values as results. Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable *local variables* that are usable as virtual registers.

---

[4] https://ieeexplore.ieee.org/document/8766229

[5] https://www.unicode.org/versions/latest/

[6] https://ieeexplore.ieee.org/document/8766229

[7] https://ieeexplore.ieee.org/document/8766229

[8] In practice, implementations need not maintain an actual operand stack. Instead, the stack can be viewed as a set of anonymous registers that are implicitly referenced by instructions. The type system ensures that the stack height, and thus any referenced register, is always known statically.

Tables A *table* is an array of opaque values of a particular *element type*. It allows programs to select such values indirectly through a dynamic index operand. Currently, the only available element type is an untyped function reference or a reference to an external host value. Thereby, a program can call functions indirectly through a dynamic index into a table. For example, this allows emulating function pointers by way of table indices.

Linear Memory A *linear memory* is a contiguous, mutable array of raw bytes. Such a memory is created with an initial size but can be grown dynamically. A program can load and store values from/to a linear memory at any byte address (including unaligned). Integer loads and stores can specify a *storage size* which is smaller than the size of the respective value type. A trap occurs if an access is not within the bounds of the current memory size.

Modules A WebAssembly binary takes the form of a *module* that contains definitions for functions, tables, and linear memories, as well as mutable or immutable *global variables*. Definitions can also be *imported*, specifying a module/name pair and a suitable type. Each definition can optionally be *exported* under one or more names. In addition to definitions, modules can define initialization data for their memories or tables that takes the form of *segments* copied to given offsets. They can also define a *start function* that is automatically executed.

Embedder A WebAssembly implementation will typically be *embedded* into a *host* environment. This environment defines how loading of modules is initiated, how imports are provided (including host-side definitions), and how exports can be accessed. However, the details of any particular embedding are beyond the scope of this specification, and will instead be provided by complementary, environment-specific API definitions.

1.2.2 Semantic Phases

Conceptually, the semantics of WebAssembly is divided into three phases. For each part of the language, the specification specifies each of them.

Decoding WebAssembly modules are distributed in a *binary format*. *Decoding* processes that format and converts it into an internal representation of a module. In this specification, this representation is modelled by *abstract syntax*, but a real implementation could compile directly to machine code instead.

Validation A decoded module has to be *valid*. Validation checks a number of well-formedness conditions to guarantee that the module is meaningful and safe. In particular, it performs *type checking* of functions and the instruction sequences in their bodies, ensuring for example that the operand stack is used consistently.

Execution Finally, a valid module can be *executed*. Execution can be further divided into two phases:

Instantiation. A module *instance* is the dynamic representation of a module, complete with its own state and execution stack. Instantiation executes the module body itself, given definitions for all its imports. It initializes globals, memories and tables and invokes the module's start function if defined. It returns the instances of the module's exports.

Invocation. Once instantiated, further WebAssembly computations can be initiated by *invoking* an exported function on a module instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

CHAPTER 2

Structure

2.1 Conventions

WebAssembly is a programming language that has multiple concrete representations (its binary format and the text format). Both map to a common structure. For conciseness, this structure is described in the form of an *abstract syntax*. All parts of this specification are defined in terms of this abstract syntax.

2.1.1 Grammar Notation

The following conventions are adopted in defining grammar rules for abstract syntax.

- Terminal symbols (atoms) are written in sans-serif font or in symbolic form: $\mathsf{i32}, \mathsf{end}, \rightarrow, [,]$.
- Nonterminal symbols are written in italic font: $\mathit{valtype}, \mathit{instr}$.
- A^n is a sequence of $n \geq 0$ iterations of A.
- A^* is a possibly empty sequence of iterations of A. (This is a shorthand for A^n used where n is not relevant.)
- A^+ is a non-empty sequence of iterations of A. (This is a shorthand for A^n where $n \geq 1$.)
- $A^?$ is an optional occurrence of A. (This is a shorthand for A^n where $n \leq 1$.)
- Productions are written $\mathit{sym} ::= A_1 \mid \dots \mid A_n$.
- Large productions may be split into multiple definitions, indicated by ending the first one with explicit ellipses, $\mathit{sym} ::= A_1 \mid \dots$, and starting continuations with ellipses, $\mathit{sym} ::= \dots \mid A_2$.
- Some productions are augmented with side conditions in parentheses, "(if $\mathit{condition}$)", that provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production, then all those occurrences must have the same instantiation. (This is a shorthand for a side condition requiring multiple different variables to be equal.)

2.1.2 Auxiliary Notation

When dealing with syntactic constructs the following notation is also used:

- ϵ denotes the empty sequence.
- $|s|$ denotes the length of a sequence s.
- $s[i]$ denotes the i-th element of a sequence s, starting from 0.
- $s[i:n]$ denotes the sub-sequence $s[i]~\dots~s[i+n-1]$ of a sequence s.
- $s~\mathrm{with}~[i] = A$ denotes the same sequence as s, except that the i-th element is replaced with A.
- $s~\mathrm{with}~[i:n] = A^n$ denotes the same sequence as s, except that the sub-sequence $s[i:n]$ is replaced with A^n.
- $\mathrm{concat}(s^*)$ denotes the flat sequence formed by concatenating all sequences s_i in s^*.

Moreover, the following conventions are employed:

- The notation x^n, where x is a non-terminal symbol, is treated as a meta variable ranging over respective sequences of x (similarly for x^*, x^+, $x^?$).
- When given a sequence x^n, then the occurrences of x in a sequence written $(A_1~x~A_2)^n$ are assumed to be in point-wise correspondence with x^n (similarly for x^*, x^+, $x^?$). This implicitly expresses a form of mapping syntactic constructions over a sequence.

Productions of the following form are interpreted as *records* that map a fixed set of fields field_i to "values" A_i, respectively:

$$r ~::=~ \{\mathsf{field}_1~A_1, \mathsf{field}_2~A_2, \dots\}$$

The following notation is adopted for manipulating such records:

- $r.\mathsf{field}$ denotes the contents of the field component of r.
- $r~\mathrm{with}~\mathsf{field} = A$ denotes the same record as r, except that the contents of the field component is replaced with A.
- $r_1 \oplus r_2$ denotes the composition of two records with the same fields of sequences by appending each sequence point-wise:

$$\{\mathsf{field}_1~A_1^*, \mathsf{field}_2~A_2^*, \dots\} \oplus \{\mathsf{field}_1~B_1^*, \mathsf{field}_2~B_2^*, \dots\} = \{\mathsf{field}_1~A_1^*~B_1^*, \mathsf{field}_2~A_2^*~B_2^*, \dots\}$$

- $\bigoplus r^*$ denotes the composition of a sequence of records, respectively; if the sequence is empty, then all fields of the resulting record are empty.

The update notation for sequences and records generalizes recursively to nested components accessed by "paths" $\mathit{pth} ::= ([\dots] \mid .\mathsf{field})^+$:

- $s~\mathrm{with}~[i]~\mathit{pth} = A$ is short for $s~\mathrm{with}~[i] = (s[i]~\mathrm{with}~\mathit{pth} = A)$,
- $r~\mathrm{with}~\mathsf{field}~\mathit{pth} = A$ is short for $r~\mathrm{with}~\mathsf{field} = (r.\mathsf{field}~\mathrm{with}~\mathit{pth} = A)$,

where $r~\mathrm{with}~.\mathsf{field} = A$ is shortened to $r~\mathrm{with}~\mathsf{field} = A$.

2.1.3 Vectors

Vectors are bounded sequences of the form A^n (or A^*), where the A can either be values or complex constructions. A vector can have at most $2^{32} - 1$ elements.

$$\mathit{vec}(A) \quad ::= \quad A^n \quad (\text{if } n < 2^{32})$$

2.2 Values

WebAssembly programs operate on primitive numeric *values*. Moreover, in the definition of programs, immutable sequences of values occur to represent more complex data, such as text strings or other vectors.

2.2.1 Bytes

The simplest form of value are raw uninterpreted *bytes*. In the abstract syntax they are represented as hexadecimal literals.

$$\mathit{byte} \quad ::= \quad \mathtt{0x00} \mid \dots \mid \mathtt{0xFF}$$

Conventions

- The meta variable b ranges over bytes.
- Bytes are sometimes interpreted as natural numbers $n < 256$.

2.2.2 Integers

Different classes of *integers* with different value ranges are distinguished by their *bit width* N and by whether they are *unsigned* or *signed*.

$$\begin{array}{lll} \mathit{uN} & ::= & 0 \mid 1 \mid \dots \mid 2^N{-}1 \\ \mathit{sN} & ::= & -2^{N-1} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid 2^{N-1}{-}1 \\ \mathit{iN} & ::= & \mathit{uN} \end{array}$$

The class iN defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. In the abstract syntax, they are represented as unsigned values. However, some operations convert them to signed based on a two's complement interpretation.

Note: The main integer types occurring in this specification are $\mathit{u32}$, $\mathit{u64}$, $\mathit{s32}$, $\mathit{s64}$, $\mathit{i8}$, $\mathit{i16}$, $\mathit{i32}$, $\mathit{i64}$. However, other sizes occur as auxiliary constructions, e.g., in the definition of floating-point numbers.

Conventions

- The meta variables m, n, i range over integers.
- Numbers may be denoted by simple arithmetics, as in the grammar above. In order to distinguish arithmetics like 2^N from sequences like $(1)^N$, the latter is distinguished with parentheses.

2.2.3 Floating-Point

Floating-point data represents 32 or 64 bit values that correspond to the respective binary formats of the IEEE 754[9] standard (Section 3.3).

Every value has a *sign* and a *magnitude*. Magnitudes can either be expressed as *normal* numbers of the form $m_0.m_1m_2\dots m_M \cdot 2^e$, where e is the exponent and m is the *significand* whose most significant bit m_0 is 1, or as a *subnormal* number where the exponent is fixed to the smallest possible value and m_0 is 0; among the subnormals are positive and negative zero values. Since the significands are binary values, normals are represented in the form $(1 + m \cdot 2^{-M}) \cdot 2^e$, where M is the bit width of m; similarly for subnormals.

---

[9] https://ieeexplore.ieee.org/document/8766229

Possible magnitudes also include the special values ∞ (infinity) and nan (*NaN*, not a number). NaN values have a *payload* that describes the mantissa bits in the underlying binary representation. No distinction is made between signalling and quiet NaNs.

$$\begin{array}{llll}
fN & ::= & +fN\mathit{mag} \mid -fN\mathit{mag} & \\
fN\mathit{mag} & ::= & (1 + uM \cdot 2^{-M}) \cdot 2^e & (\text{if } -2^{E-1} + 2 \leq e \leq 2^{E-1} - 1) \\
& | & (0 + uM \cdot 2^{-M}) \cdot 2^e & (\text{if } e = -2^{E-1} + 2) \\
& | & \infty & \\
& | & \mathsf{nan}(n) & (\text{if } 1 \leq n < 2^M)
\end{array}$$

where $M = \mathrm{signif}(N)$ and $E = \mathrm{expon}(N)$ with

$$\begin{array}{lcllcl}
\mathrm{signif}(32) & = & 23 & \qquad \mathrm{expon}(32) & = & 8 \\
\mathrm{signif}(64) & = & 52 & \qquad \mathrm{expon}(64) & = & 11
\end{array}$$

A *canonical NaN* is a floating-point value $\pm\mathsf{nan}(\mathrm{canon}_N)$ where canon_N is a payload whose most significant bit is 1 while all others are 0:

$$\mathrm{canon}_N = 2^{\mathrm{signif}(N)-1}$$

An *arithmetic NaN* is a floating-point value $\pm\mathsf{nan}(n)$ with $n \geq \mathrm{canon}_N$, such that the most significant bit is 1 while all others are arbitrary.

Note: In the abstract syntax, subnormals are distinguished by the leading 0 of the significand. The exponent of subnormals has the same value as the smallest possible exponent of a normal number. Only in the binary representation the exponent of a subnormal is encoded differently than the exponent of any normal number.

The notion of canonical NaN defined here is unrelated to the notion of canonical NaN that the IEEE 754[10] standard (Section 3.5.2) defines for decimal interchange formats.

Conventions

- The meta variable z ranges over floating-point values where clear from context.

2.2.4 Vectors

Numeric vectors are 128-bit values that are processed by vector instructions (also known as *SIMD* instructions, single instruction multiple data). They are represented in the abstract syntax using *i128*. The interpretation of lane types (integer or floating-point numbers) and lane sizes are determined by the specific instruction operating on them.

2.2.5 Names

Names are sequences of *characters*, which are *scalar values* as defined by Unicode[11] (Section 2.4).

$$\begin{array}{lll}
\mathit{name} & ::= & \mathit{char}^* \qquad (\text{if } |\mathrm{utf8}(\mathit{char}^*)| < 2^{32}) \\
\mathit{char} & ::= & \mathrm{U{+}00} \mid \dots \mid \mathrm{U{+}D7FF} \mid \mathrm{U{+}E000} \mid \dots \mid \mathrm{U{+}10FFFF}
\end{array}$$

Due to the limitations of the binary format, the length of a name is bounded by the length of its UTF-8 encoding.

---

[10] https://ieeexplore.ieee.org/document/8766229

[11] https://www.unicode.org/versions/latest/

Convention

- Characters (Unicode scalar values) are sometimes used interchangeably with natural numbers $n < 1114112$.

2.3 Types

Various entities in WebAssembly are classified by types. Types are checked during validation, instantiation, and possibly execution.

2.3.1 Number Types

Number types classify numeric values.

$$\mathit{numtype} \quad ::= \quad \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$$

The types i32 and i64 classify 32 and 64 bit integers, respectively. Integers are not inherently signed or unsigned, their interpretation is determined by individual operations.

The types f32 and f64 classify 32 and 64 bit floating-point data, respectively. They correspond to the respective binary floating-point representations, also known as *single* and *double* precision, as defined by the IEEE 754[12] standard (Section 3.3).

Number types are *transparent*, meaning that their bit patterns can be observed. Values of number type can be stored in memories.

Conventions

- The notation $|t|$ denotes the *bit width* of a number type t. That is, $|\mathsf{i32}| = |\mathsf{f32}| = 32$ and $|\mathsf{i64}| = |\mathsf{f64}| = 64$.

2.3.2 Vector Types

Vector types classify vectors of numeric values processed by vector instructions (also known as *SIMD* instructions, single instruction multiple data).

$$\mathit{vectype} \quad ::= \quad \mathsf{v128}$$

The type v128 corresponds to a 128 bit vector of packed integer or floating-point data. The packed data can be interpreted as signed or unsigned integers, single or double precision floating-point values, or a single 128 bit type. The interpretation is determined by individual operations.

Vector types, like number types are *transparent*, meaning that their bit patterns can be observed. Values of vector type can be stored in memories.

Conventions

- The notation $|t|$ for bit width extends to vector types as well, that is, $|\mathsf{v128}| = 128$.

---

[12] https://ieeexplore.ieee.org/document/8766229

2.3.3 Reference Types

Reference types classify first-class references to objects in the runtime store.

$$\mathit{reftype} \quad ::= \quad \mathsf{funcref} \mid \mathsf{externref}$$

The type funcref denotes the infinite union of all references to functions, regardless of their function types.

The type externref denotes the infinite union of all references to objects owned by the embedder and that can be passed into WebAssembly under this type.

Reference types are *opaque*, meaning that neither their size nor their bit pattern can be observed. Values of reference type can be stored in tables.

2.3.4 Value Types

Value types classify the individual values that WebAssembly code can compute with and the values that a variable accepts. They are either number types, vector types, or reference types.

$$\mathit{valtype} \quad ::= \quad \mathit{numtype} \mid \mathit{vectype} \mid \mathit{reftype}$$

Conventions

- The meta variable t ranges over value types or subclasses thereof where clear from context.

2.3.5 Result Types

Result types classify the result of executing instructions or functions, which is a sequence of values, written with brackets.

$$\mathit{resulttype} \quad ::= \quad [\mathit{vec}(\mathit{valtype})]$$

2.3.6 Function Types

Function types classify the signature of functions, mapping a vector of parameters to a vector of results. They are also used to classify the inputs and outputs of instructions.

$$\mathit{functype} \quad ::= \quad \mathit{resulttype} \to \mathit{resulttype}$$

2.3.7 Limits

Limits classify the size range of resizeable storage associated with memory types and table types.

$$\mathit{limits} \quad ::= \quad \{\mathsf{min}~\mathit{u32}, \mathsf{max}~\mathit{u32}^?\}$$

If no maximum is given, the respective storage can grow to any size.

2.3.8 Memory Types

Memory types classify linear memories and their size range.

$$\begin{array}{llll} \mathit{memtype} & ::= & \mathit{limits}~\mathit{share} \\ \mathit{share} & ::= & \mathsf{shared} ~|~ \mathsf{unshared} \end{array}$$

The limits constrain the minimum and optionally the maximum size of a memory. The limits are given in units of page size. The memory type also determines whether this memory is shared.

2.3.9 Table Types

Table types classify tables over elements of reference type within a size range.

$$\begin{array}{llll} \mathit{tabletype} & ::= & \mathit{limits}~\mathit{reftype} \end{array}$$

Like memories, tables are constrained by limits for their minimum and optionally maximum size. The limits are given in numbers of entries.

Note: In future versions of WebAssembly, additional element types may be introduced.

2.3.10 Global Types

Global types classify global variables, which hold a value and can either be mutable or immutable.

$$\begin{array}{llll} \mathit{globaltype} & ::= & \mathit{mut}~\mathit{valtype} \\ \mathit{mut} & ::= & \mathsf{const} ~|~ \mathsf{var} \end{array}$$

2.3.11 External Types

External types classify imports and external values with their respective types.

$$\begin{array}{llll} \mathit{externtype} & ::= & \mathsf{func}~\mathit{functype} ~|~ \mathsf{table}~\mathit{tabletype} ~|~ \mathsf{mem}~\mathit{memtype} ~|~ \mathsf{global}~\mathit{globaltype} \end{array}$$

Conventions

The following auxiliary notation is defined for sequences of external types. It filters out entries of a specific kind in an order-preserving fashion:

- $\mathrm{funcs}(\mathit{externtype}^*) = [\mathit{functype} ~|~ (\mathsf{func}~\mathit{functype}) \in \mathit{externtype}^*]$
- $\mathrm{tables}(\mathit{externtype}^*) = [\mathit{tabletype} ~|~ (\mathsf{table}~\mathit{tabletype}) \in \mathit{externtype}^*]$
- $\mathrm{mems}(\mathit{externtype}^*) = [\mathit{memtype} ~|~ (\mathsf{mem}~\mathit{memtype}) \in \mathit{externtype}^*]$
- $\mathrm{globals}(\mathit{externtype}^*) = [\mathit{globaltype} ~|~ (\mathsf{global}~\mathit{globaltype}) \in \mathit{externtype}^*]$

2.4 Instructions

WebAssembly code consists of sequences of *instructions*. Its computational model is based on a *stack machine* in that instructions manipulate values on an implicit *operand stack*, consuming (popping) argument values and producing or returning (pushing) result values.

In addition to dynamic operands from the stack, some instructions also have static *immediate* arguments, typically indices or type annotations, which are part of the instruction itself.

Some instructions are structured in that they bracket nested sequences of instructions.

The following sections group instructions into a number of different categories.

2.4.1 Numeric Instructions

Numeric instructions provide basic operations over numeric values of specific type. These operations closely match respective operations available in hardware.

$$
\begin{array}{llll}
\mathit{nn}, \mathit{mm} & ::= & \mathsf{32} ~|~ \mathsf{64} \\
\mathit{sx} & ::= & \mathsf{u} ~|~ \mathsf{s} \\
\mathit{instr} & ::= & \mathsf{i}\mathit{nn}\mathsf{.const}~\mathit{u}\mathit{nn} ~|~ \mathsf{f}\mathit{nn}\mathsf{.const}~\mathit{f}\mathit{nn} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{iunop} ~|~ \mathsf{f}\mathit{nn}\mathsf{.}\mathit{funop} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{ibinop} ~|~ \mathsf{f}\mathit{nn}\mathsf{.}\mathit{fbinop} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{itestop} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{irelop} ~|~ \mathsf{f}\mathit{nn}\mathsf{.}\mathit{frelop} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.extend8\_s} ~|~ \mathsf{i}\mathit{nn}\mathsf{.extend16\_s} ~|~ \mathsf{i64.extend32\_s} \\
 & | & \mathsf{i32.wrap\_i64} ~|~ \mathsf{i64.extend\_i32\_}\mathit{sx} ~|~ \mathsf{i}\mathit{nn}\mathsf{.trunc\_f}\mathit{mm}\mathsf{\_}\mathit{sx} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.trunc\_sat\_f}\mathit{mm}\mathsf{\_}\mathit{sx} \\
 & | & \mathsf{f32.demote\_f64} ~|~ \mathsf{f64.promote\_f32} ~|~ \mathsf{f}\mathit{nn}\mathsf{.convert\_i}\mathit{mm}\mathsf{\_}\mathit{sx} \\
 & | & \mathsf{i}\mathit{nn}\mathsf{.reinterpret\_f}\mathit{nn} ~|~ \mathsf{f}\mathit{nn}\mathsf{.reinterpret\_i}\mathit{nn} \\
 & | & \dots \\
\mathit{iunop} & ::= & \mathsf{clz} ~|~ \mathsf{ctz} ~|~ \mathsf{popcnt} \\
\mathit{ibinop} & ::= & \mathsf{add} ~|~ \mathsf{sub} ~|~ \mathsf{mul} ~|~ \mathsf{div\_}\mathit{sx} ~|~ \mathsf{rem\_}\mathit{sx} \\
 & | & \mathsf{and} ~|~ \mathsf{or} ~|~ \mathsf{xor} ~|~ \mathsf{shl} ~|~ \mathsf{shr\_}\mathit{sx} ~|~ \mathsf{rotl} ~|~ \mathsf{rotr} \\
\mathit{funop} & ::= & \mathsf{abs} ~|~ \mathsf{neg} ~|~ \mathsf{sqrt} ~|~ \mathsf{ceil} ~|~ \mathsf{floor} ~|~ \mathsf{trunc} ~|~ \mathsf{nearest} \\
\mathit{fbinop} & ::= & \mathsf{add} ~|~ \mathsf{sub} ~|~ \mathsf{mul} ~|~ \mathsf{div} ~|~ \mathsf{min} ~|~ \mathsf{max} ~|~ \mathsf{copysign} \\
\mathit{itestop} & ::= & \mathsf{eqz} \\
\mathit{irelop} & ::= & \mathsf{eq} ~|~ \mathsf{ne} ~|~ \mathsf{lt\_}\mathit{sx} ~|~ \mathsf{gt\_}\mathit{sx} ~|~ \mathsf{le\_}\mathit{sx} ~|~ \mathsf{ge\_}\mathit{sx} \\
\mathit{frelop} & ::= & \mathsf{eq} ~|~ \mathsf{ne} ~|~ \mathsf{lt} ~|~ \mathsf{gt} ~|~ \mathsf{le} ~|~ \mathsf{ge}
\end{array}
$$

Numeric instructions are divided by number type. For each type, several subcategories can be distinguished:

- *Constants*: return a static constant.
- *Unary Operations*: consume one operand and produce one result of the respective type.
- *Binary Operations*: consume two operands and produce one result of the respective type.
- *Tests*: consume one operand of the respective type and produce a Boolean integer result.
- *Comparisons*: consume two operands of the respective type and produce a Boolean integer result.
- *Conversions*: consume a value of one type and produce a result of another (the source type of the conversion is the one after the “_”).

Some integer instructions come in two flavors, where a signedness annotation sx distinguishes whether the operands are to be interpreted as unsigned or signed integers. For the other integer instructions, the use of two’s complement for the signed interpretation means that they behave the same regardless of signedness.

Conventions

Occasionally, it is convenient to group operators together according to the following grammar shorthands:

$$\begin{array}{lll}
\mathit{unop} & ::= & \mathit{iunop} \mid \mathit{funop} \mid \mathsf{extend}N\mathsf{\_s} \\
\mathit{binop} & ::= & \mathit{ibinop} \mid \mathit{fbinop} \\
\mathit{testop} & ::= & \mathit{itestop} \\
\mathit{relop} & ::= & \mathit{irelop} \mid \mathit{frelop} \\
\mathit{cvtop} & ::= & \mathsf{wrap} \mid \mathsf{extend} \mid \mathsf{trunc} \mid \mathsf{trunc\_sat} \mid \mathsf{convert} \mid \mathsf{demote} \mid \mathsf{promote} \mid \mathsf{reinterpret}
\end{array}$$

2.4.2 Vector Instructions

Vector instructions (also known as *SIMD* instructions, *single instruction multiple data*) provide basic operations over values of vector type.

$$\begin{array}{lll}
\mathit{ishape} & ::= & \mathsf{i8x16} \mid \mathsf{i16x8} \mid \mathsf{i32x4} \mid \mathsf{i64x2} \\
\mathit{fshape} & ::= & \mathsf{f32x4} \mid \mathsf{f64x2} \\
\mathit{shape} & ::= & \mathit{ishape} \mid \mathit{fshape} \\
\mathit{half} & ::= & \mathsf{low} \mid \mathsf{high} \\
\mathit{laneidx} & ::= & \mathit{u8}
\end{array}$$

$$
\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{v128.const}~\mathit{i128} \\
& | & \mathsf{v128}.\mathit{vvunop} \\
& | & \mathsf{v128}.\mathit{vvbinop} \\
& | & \mathsf{v128}.\mathit{vvternop} \\
& | & \mathsf{v128}.\mathit{vvtestop} \\
& | & \mathsf{i8x16.shuffle}~\mathit{laneidx}^{16} \\
& | & \mathsf{i8x16.swizzle} \\
& | & \mathit{shape}.\mathsf{splat} \\
& | & \mathsf{i8x16.extract\_lane\_}\mathit{sx}~\mathit{laneidx} \mid \mathsf{i16x8.extract\_lane\_}\mathit{sx}~\mathit{laneidx} \\
& | & \mathsf{i32x4.extract\_lane}~\mathit{laneidx} \mid \mathsf{i64x2.extract\_lane}~\mathit{laneidx} \\
& | & \mathit{fshape}.\mathsf{extract\_lane}~\mathit{laneidx} \\
& | & \mathit{shape}.\mathsf{replace\_lane}~\mathit{laneidx} \\
& | & \mathsf{i8x16}.\mathit{virelop} \mid \mathsf{i16x8}.\mathit{virelop} \mid \mathsf{i32x4}.\mathit{virelop} \\
& | & \mathsf{i64x2.eq} \mid \mathsf{i64x2.ne} \mid \mathsf{i64x2.lt\_s} \mid \mathsf{i64x2.gt\_s} \mid \mathsf{i64x2.le\_s} \mid \mathsf{i64x2.ge\_s} \\
& | & \mathit{fshape}.\mathit{vfrelop} \\
& | & \mathit{ishape}.\mathit{viunop} \mid \mathsf{i8x16.popcnt} \\
& | & \mathsf{i16x8.q15mulr\_sat\_s} \\
& | & \mathsf{i32x4.dot\_i16x8\_s} \\
& | & \mathit{fshape}.\mathit{vfunop} \\
& | & \mathit{ishape}.\mathit{vitestop} \\
& | & \mathit{ishape}.\mathsf{bitmask} \\
& | & \mathsf{i8x16.narrow\_i16x8\_}\mathit{sx} \mid \mathsf{i16x8.narrow\_i32x4\_}\mathit{sx} \\
& | & \mathsf{i16x8.extend\_}\mathit{half}\mathsf{\_i8x16\_}\mathit{sx} \mid \mathsf{i32x4.extend\_}\mathit{half}\mathsf{\_i16x8\_}\mathit{sx} \\
& | & \mathsf{i64x2.extend\_}\mathit{half}\mathsf{\_i32x4\_}\mathit{sx} \\
& | & \mathit{ishape}.\mathit{vishiftop} \\
& | & \mathit{ishape}.\mathit{vibinop} \\
& | & \mathsf{i8x16}.\mathit{viminmaxop} \mid \mathsf{i16x8}.\mathit{viminmaxop} \mid \mathsf{i32x4}.\mathit{viminmaxop} \\
& | & \mathsf{i8x16}.\mathit{visatbinop} \mid \mathsf{i16x8}.\mathit{visatbinop} \\
& | & \mathsf{i16x8.mul} \mid \mathsf{i32x4.mul} \mid \mathsf{i64x2.mul} \\
& | & \mathsf{i8x16.avgr\_u} \mid \mathsf{i16x8.avgr\_u} \\
& | & \mathsf{i16x8.extmul\_}\mathit{half}\mathsf{\_i8x16\_}\mathit{sx} \mid \mathsf{i32x4.extmul\_}\mathit{half}\mathsf{\_i16x8\_}\mathit{sx} \mid \mathsf{i64x2.extmul\_}\mathit{half}\mathsf{\_i32x4\_}\mathit{sx} \\
& | & \mathsf{i16x8.extadd\_pairwise\_i8x16\_}\mathit{sx} \mid \mathsf{i32x4.extadd\_pairwise\_i16x8\_}\mathit{sx} \\
& | & \mathit{fshape}.\mathit{vfbinop} \\
& | & \mathsf{i32x4.trunc\_sat\_f32x4\_}\mathit{sx} \mid \mathsf{i32x4.trunc\_sat\_f64x2\_}\mathit{sx}\mathsf{\_zero} \\
& | & \mathsf{f32x4.convert\_i32x4\_}\mathit{sx} \mid \mathsf{f32x4.demote\_f64x2\_zero} \\
& | & \mathsf{f64x2.convert\_low\_i32x4\_}\mathit{sx} \mid \mathsf{f64x2.promote\_low\_f32x4} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{lll}
\mathit{vvunop} & ::= & \mathsf{not} \\
\mathit{vvbinop} & ::= & \mathsf{and} \mid \mathsf{andnot} \mid \mathsf{or} \mid \mathsf{xor} \\
\mathit{vvternop} & ::= & \mathsf{bitselect} \\
\mathit{vvtestop} & ::= & \mathsf{any\_true} \\
\mathit{vitestop} & ::= & \mathsf{all\_true} \\
\mathit{virelop} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt\_}\mathit{sx} \mid \mathsf{gt\_}\mathit{sx} \mid \mathsf{le\_}\mathit{sx} \mid \mathsf{ge\_}\mathit{sx} \\
\mathit{vfrelop} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} \\
\mathit{viunop} & ::= & \mathsf{abs} \mid \mathsf{neg} \\
\mathit{vibinop} & ::= & \mathsf{add} \mid \mathsf{sub} \\
\mathit{viminmaxop} & ::= & \mathsf{min\_}\mathit{sx} \mid \mathsf{max\_}\mathit{sx} \\
\mathit{visatbinop} & ::= & \mathsf{add\_sat\_}\mathit{sx} \mid \mathsf{sub\_sat\_}\mathit{sx} \\
\mathit{vishiftop} & ::= & \mathsf{shl} \mid \mathsf{shr\_}\mathit{sx} \\
\mathit{vfunop} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\
\mathit{vfbinop} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{pmin} \mid \mathsf{pmax}
\end{array}
$$

Vector instructions have a naming convention involving a prefix that determines how their operands will be interpreted. This prefix describes the *shape* of the operand, written $t\mathsf{x}N$, and consisting of a packed numeric type t and the number of *lanes* N of that type. Operations are performed point-wise on the values of each lane.

Note: For example, the shape i32x4 interprets the operand as four $\mathit{i32}$ values, packed into an $\mathit{i128}$. The bitwidth of the numeric type t times N always is 128.

Instructions prefixed with v128 do not involve a specific interpretation, and treat the $\mathsf{v128}$ as an $\mathit{i128}$ value or a vector of 128 individual bits.

Vector instructions can be grouped into several subcategories:

- *Constants*: return a static constant.
- *Unary Operations*: consume one $\mathsf{v128}$ operand and produce one $\mathsf{v128}$ result.
- *Binary Operations*: consume two $\mathsf{v128}$ operands and produce one $\mathsf{v128}$ result.
- *Ternary Operations*: consume three $\mathsf{v128}$ operands and produce one $\mathsf{v128}$ result.
- *Tests*: consume one $\mathsf{v128}$ operand and produce a Boolean integer result.
- *Shifts*: consume a $\mathsf{v128}$ operand and a $\mathit{i32}$ operand, producing one $\mathsf{v128}$ result.
- *Splats*: consume a value of numeric type and produce a $\mathsf{v128}$ result of a specified shape.
- *Extract lanes*: consume a $\mathsf{v128}$ operand and return the numeric value in a given lane.
- *Replace lanes*: consume a $\mathsf{v128}$ operand and a numeric value for a given lane, and produce a $\mathsf{v128}$ result.

Some vector instructions have a signedness annotation sx which distinguishes whether the elements in the operands are to be interpreted as unsigned or signed integers. For the other vector instructions, the use of two's complement for the signed interpretation means that they behave the same regardless of signedness.

Conventions

Occasionally, it is convenient to group operators together according to the following grammar shorthands:

$$
\begin{array}{llll}
\mathit{vunop} & ::= & \mathit{viunop} ~|~ \mathit{vfunop} ~|~ \mathsf{popcnt} \\
\mathit{vbinop} & ::= & \mathit{vibinop} ~|~ \mathit{vfbinop} \\
 & | & \mathit{viminmaxop} ~|~ \mathit{visatbinop} \\
 & | & \mathsf{mul} ~|~ \mathsf{avgr\_u} ~|~ \mathsf{q15mulr\_sat\_s} \\
\mathit{vtestop} & ::= & \mathit{vitestop} \\
\mathit{vrelop} & ::= & \mathit{virelop} ~|~ \mathit{vfrelop} \\
\mathit{vcvtop} & ::= & \mathsf{extend} ~|~ \mathsf{trunc\_sat} ~|~ \mathsf{convert} ~|~ \mathsf{demote} ~|~ \mathsf{promote}
\end{array}
$$

2.4.3 Reference Instructions

Instructions in this group are concerned with accessing references.

$$
\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
 & | & \mathsf{ref.null}~\mathit{reftype} \\
 & | & \mathsf{ref.is\_null} \\
 & | & \mathsf{ref.func}~\mathit{funcidx}
\end{array}
$$

These instructions produce a null value, check for a null value, or produce a reference to a given function, respectively.

2.4.4 Parametric Instructions

Instructions in this group can operate on operands of any value type.

$$\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{drop} \\
& | & \mathsf{select}~(\mathit{valtype}^*)^?
\end{array}$$

The drop instruction simply throws away a single operand.

The select instruction selects one of its first two operands based on whether its third operand is zero or not. It may include a value type determining the type of these operands. If missing, the operands must be of numeric type.

Note: In future versions of WebAssembly, the type annotation on select may allow for more than a single value being selected at the same time.

2.4.5 Variable Instructions

Variable instructions are concerned with access to local or global variables.

$$\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{local.get}~\mathit{localidx} \\
& | & \mathsf{local.set}~\mathit{localidx} \\
& | & \mathsf{local.tee}~\mathit{localidx} \\
& | & \mathsf{global.get}~\mathit{globalidx} \\
& | & \mathsf{global.set}~\mathit{globalidx}
\end{array}$$

These instructions get or set the values of variables, respectively. The local.tee instruction is like local.set but also returns its argument.

2.4.6 Table Instructions

Instructions in this group are concerned with tables table.

$$\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{table.get}~\mathit{tableidx} \\
& | & \mathsf{table.set}~\mathit{tableidx} \\
& | & \mathsf{table.size}~\mathit{tableidx} \\
& | & \mathsf{table.grow}~\mathit{tableidx} \\
& | & \mathsf{table.fill}~\mathit{tableidx} \\
& | & \mathsf{table.copy}~\mathit{tableidx}~\mathit{tableidx} \\
& | & \mathsf{table.init}~\mathit{tableidx}~\mathit{elemidx} \\
& | & \mathsf{elem.drop}~\mathit{elemidx}
\end{array}$$

The table.get and table.set instructions load or store an element in a table, respectively.

The table.size instruction returns the current size of a table. The table.grow instruction grows table by a given delta and returns the previous size, or -1 if enough space cannot be allocated. It also takes an initialization value for the newly allocated entries.

The table.fill instruction sets all entries in a range to a given value.

The table.copy instruction copies elements from a source table region to a possibly overlapping destination region; the first index denotes the destination. The table.init instruction copies elements from a passive element segment into a table. The elem.drop instruction prevents further use of a passive element segment. This instruction is intended to be used as an optimization hint. After an element segment is dropped its elements can no longer be retrieved, so the memory used by this segment may be freed.

An additional instruction that accesses a table is the control instruction call_indirect.

2.4.7 Memory Instructions

Instructions in this group are concerned with linear memory.

$$
\begin{array}{lll}
\mathit{memarg} & ::= & \{\mathsf{offset}~\mathit{u32}, \mathsf{align}~\mathit{u32}\} \\
\mathit{ww} & ::= & 8 ~|~ 16 ~|~ 32 ~|~ 64 \\
\mathit{instr} & ::= & \dots \\
& | & \mathsf{i}\mathit{nn}\mathsf{.load}~\mathit{memarg} ~|~ \mathsf{f}\mathit{nn}\mathsf{.load}~\mathit{memarg} ~|~ \mathsf{v128.load}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.store}~\mathit{memarg} ~|~ \mathsf{f}\mathit{nn}\mathsf{.store}~\mathit{memarg} ~|~ \mathsf{v128.store}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.load8\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}\mathsf{.load16\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{i64.load32\_}\mathit{sx}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.store8}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}\mathsf{.store16}~\mathit{memarg} ~|~ \mathsf{i64.store32}~\mathit{memarg} \\
& | & \mathsf{v128.load8x8\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{v128.load16x4\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{v128.load32x2\_}\mathit{sx}~\mathit{memarg} \\
& | & \mathsf{v128.load32\_zero}~\mathit{memarg} ~|~ \mathsf{v128.load64\_zero}~\mathit{memarg} \\
& | & \mathsf{v128.load}\mathit{ww}\mathsf{\_splat}~\mathit{memarg} \\
& | & \mathsf{v128.load}\mathit{ww}\mathsf{\_lane}~\mathit{memarg}~\mathit{laneidx} ~|~ \mathsf{v128.store}\mathit{ww}\mathsf{\_lane}~\mathit{memarg}~\mathit{laneidx} \\
& | & \mathsf{memory.size} \\
& | & \mathsf{memory.grow} \\
& | & \mathsf{memory.fill} \\
& | & \mathsf{memory.copy} \\
& | & \mathsf{memory.init}~\mathit{dataidx} \\
& | & \mathsf{data.drop}~\mathit{dataidx}
\end{array}
$$

Memory is accessed with load and store instructions for the different number types. They all take a *memory immediate* memarg that contains an address *offset* and the expected *alignment* (expressed as the exponent of a power of 2). Integer loads and stores can optionally specify a *storage size* that is smaller than the bit width of the respective value type. In the case of loads, a sign extension mode sx is then required to select appropriate behavior.

Vector loads can specify a shape that is half the bit width of v128. Each lane is half its usual size, and the sign extension mode sx then specifies how the smaller lane is extended to the larger lane. Alternatively, vector loads can perform a *splat*, such that only a single lane of the specified storage size is loaded, and the result is duplicated to all lanes.

The static address offset is added to the dynamic address operand, yielding a 33 bit *effective address* that is the zero-based index at which the memory is accessed. All values are read and written in little endian[13] byte order. A trap results if any of the accessed memory bytes lies outside the address range implied by the memory's current size.

Note: Future versions of WebAssembly might provide memory instructions with 64 bit address ranges.

The memory.size instruction returns the current size of a memory. The memory.grow instruction grows memory by a given delta and returns the previous size, or -1 if enough memory cannot be allocated. Both instructions operate in units of page size.

The memory.fill instruction sets all values in a region to a given byte. The memory.copy instruction copies data from a source memory region to a possibly overlapping destination region. The memory.init instruction copies data from a passive data segment into a memory. The data.drop instruction prevents further use of a passive data segment. This instruction is intended to be used as an optimization hint. After a data segment is dropped its data can no longer be retrieved, so the memory used by this segment may be freed.

Note: In the current version of WebAssembly, all memory instructions implicitly operate on memory index 0. This restriction may be lifted in future versions.

---

[13] https://en.wikipedia.org/wiki/Endianness#Little-endian

2.4.8 Atomic Memory Instructions

Instructions in this group are concerned with accessing linear memory atomically.

$$
\begin{array}{lll}
\mathit{atop} & ::= & \mathsf{add} ~|~ \mathsf{sub} ~|~ \mathsf{and} ~|~ \mathsf{or} ~|~ \mathsf{xor} ~|~ \mathsf{xchg} \\
\mathit{instr} & ::= & \dots \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.load}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.store}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.load8\_u}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}\mathsf{.atomic.load16\_u}~\mathit{memarg} ~|~ \mathsf{i64.atomic.load32\_u}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.store8}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}\mathsf{.atomic.store16}~\mathit{memarg} ~|~ \mathsf{i64.atomic.store32}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw.}\mathit{atop}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw8.}\mathit{atop}\mathsf{\_u}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw16.}\mathit{atop}\mathsf{\_u}~\mathit{memarg} \\
& | & \mathsf{i64.atomic.rmw32.}\mathit{atop}\mathsf{\_u}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw.cmpxchg}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw8.cmpxchg\_u}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.atomic.rmw16.cmpxchg\_u}~\mathit{memarg} \\
& | & \mathsf{i64.atomic.rmw32.cmpxchg\_u}~\mathit{memarg} \\
& | & \mathsf{memory.atomic.notify}~\mathit{memarg} \\
& | & \mathsf{memory.atomic.wait}\mathit{nn}~\mathit{memarg} \\
& | & \mathsf{atomic.fence}
\end{array}
$$

Memory is accessed atomically using the atomic.load, atomic.store, and atomic.rmw instructions. All instructions take a *memory immediate* memarg, just like their non-atomic equivalents. Unlike non-atomic memory access instructions, only integer value types can be used. Also unlike non-atomic memory access instructions, there are no sign extension modes; atomic memory accesses are always zero-extending.

The atomic.rmw instructions are read-modify-write[14] instructions. They each have an *atomic operator*, which specifies how memory will be modified. Each instruction returns the value read from memory before modification. The xchg operator doesn't use the read value, but instead stores its argument unmodified. The cmpxchg operator is similar, but only performs this action conditionally, if the read value is equal to a provided comparison argument. All other atomic operators have the same behavior as the binary operator of the same name.

The memory.atomic.wait, memory.atomic.notify, and atomic.fence instructions provide primitive synchronization between threads. The memory.atomic.wait instructions atomically load a value from the calculated effective address and compare it to an expected value. If they are equal, the thread is then suspended until a given timeout expires or another thread wakes it. The memory.atomic.notify instruction wakes threads that are waiting on a given address, up to a given maximum. The atomic.fence instruction takes no operands, and returns nothing. It is intended to preserve the synchronization guarantees of the fence operators of higher-level languages. Unlike other atomic operators, it does not target a particular linear memory.

2.4.9 Control Instructions

Instructions in this group affect the flow of control.

$$
\begin{array}{lll}
\mathit{blocktype} & ::= & \mathit{typeidx} ~|~ \mathit{valtype}^? \\
\mathit{instr} & ::= & \dots \\
& | & \mathsf{nop} \\
& | & \mathsf{unreachable} \\
& | & \mathsf{block}~\mathit{blocktype}~\mathit{instr}^*~\mathsf{end} \\
& | & \mathsf{loop}~\mathit{blocktype}~\mathit{instr}^*~\mathsf{end} \\
& | & \mathsf{if}~\mathit{blocktype}~\mathit{instr}^*~\mathsf{else}~\mathit{instr}^*~\mathsf{end} \\
& | & \mathsf{br}~\mathit{labelidx} \\
& | & \mathsf{br\_if}~\mathit{labelidx} \\
& | & \mathsf{br\_table}~\mathit{vec}(\mathit{labelidx})~\mathit{labelidx} \\
& | & \mathsf{return} \\
& | & \mathsf{call}~\mathit{funcidx} \\
& | & \mathsf{call\_indirect}~\mathit{tableidx}~\mathit{typeidx}
\end{array}
$$

---

[14] https://en.wikipedia.org/wiki/Read-modify-write

The nop instruction does nothing.

The unreachable instruction causes an unconditional trap.

The block, loop and if instructions are *structured* instructions. They bracket nested sequences of instructions, called *blocks*, terminated with, or separated by, end or else pseudo-instructions. As the grammar prescribes, they must be well-nested.

A structured instruction can consume *input* and produce *output* on the operand stack according to its annotated *block type*. It is given either as a type index that refers to a suitable function type, or as an optional value type inline, which is a shorthand for the function type $[] \to [valtype^?]$.

Each structured control instruction introduces an implicit *label*. Labels are targets for branch instructions that reference them with label indices. Unlike with other index spaces, indexing of labels is relative by nesting depth, that is, label 0 refers to the innermost structured control instruction enclosing the referring branch instruction, while increasing indices refer to those farther out. Consequently, labels can only be referenced from *within* the associated structured control instruction. This also implies that branches can only be directed outwards, "breaking" from the block of the control construct they target. The exact effect depends on that control construct. In case of block or if it is a *forward jump*, resuming execution after the matching end. In case of loop it is a *backward jump* to the beginning of the loop.

Note: This enforces *structured control flow*. Intuitively, a branch targeting a block or if behaves like a break statement in most C-like languages, while a branch targeting a loop behaves like a continue statement.

Branch instructions come in several flavors: br performs an unconditional branch, br_if performs a conditional branch, and br_table performs an indirect branch through an operand indexing into the label vector that is an immediate to the instruction, or to a default target if the operand is out of bounds. The return instruction is a shortcut for an unconditional branch to the outermost block, which implicitly is the body of the current function. Taking a branch *unwinds* the operand stack up to the height where the targeted structured control instruction was entered. However, branches may additionally consume operands themselves, which they push back on the operand stack after unwinding. Forward branches require operands according to the output of the targeted block's type, i.e., represent the values produced by the terminated block. Backward branches require operands according to the input of the targeted block's type, i.e., represent the values consumed by the restarted block.

The call instruction invokes another function, consuming the necessary arguments from the stack and returning the result values of the call. The call_indirect instruction calls a function indirectly through an operand indexing into a table that is denoted by a table index and must have type funcref. Since it may contain functions of heterogeneous type, the callee is dynamically checked against the function type indexed by the instruction's second immediate, and the call is aborted with a trap if it does not match.

2.4.10 Expressions

Function bodies, initialization values for globals, and offsets of element or data segments are given as expressions, which are sequences of instructions terminated by an end marker.

$$expr ::= instr^* \ \mathsf{end}$$

In some places, validation restricts expressions to be *constant*, which limits the set of allowable instructions.

2.5 Modules

WebAssembly programs are organized into *modules*, which are the unit of deployment, loading, and compilation. A module collects definitions for types, functions, tables, memories, and globals. In addition, it can declare imports and exports and provide initialization in the form of data and element segments, or a start function.

$$\begin{array}{lll} \mathit{module} & ::= & \{ \quad \mathsf{types}~\mathit{vec}(\mathit{functype}), \\ & & \quad \mathsf{funcs}~\mathit{vec}(\mathit{func}), \\ & & \quad \mathsf{tables}~\mathit{vec}(\mathit{table}), \\ & & \quad \mathsf{mems}~\mathit{vec}(\mathit{mem}), \\ & & \quad \mathsf{globals}~\mathit{vec}(\mathit{global}), \\ & & \quad \mathsf{elems}~\mathit{vec}(\mathit{elem}), \\ & & \quad \mathsf{datas}~\mathit{vec}(\mathit{data}), \\ & & \quad \mathsf{start}~\mathit{start}^?, \\ & & \quad \mathsf{imports}~\mathit{vec}(\mathit{import}), \\ & & \quad \mathsf{exports}~\mathit{vec}(\mathit{export}) \quad \} \end{array}$$

Each of the vectors – and thus the entire module – may be empty.

2.5.1 Indices

Definitions are referenced with zero-based *indices*. Each class of definition has its own *index space*, as distinguished by the following classes.

$$\begin{array}{lll} \mathit{typeidx} & ::= & \mathit{u32} \\ \mathit{funcidx} & ::= & \mathit{u32} \\ \mathit{tableidx} & ::= & \mathit{u32} \\ \mathit{memidx} & ::= & \mathit{u32} \\ \mathit{globalidx} & ::= & \mathit{u32} \\ \mathit{elemidx} & ::= & \mathit{u32} \\ \mathit{dataidx} & ::= & \mathit{u32} \\ \mathit{localidx} & ::= & \mathit{u32} \\ \mathit{labelidx} & ::= & \mathit{u32} \end{array}$$

The index space for functions, tables, memories and globals includes respective imports declared in the same module. The indices of these imports precede the indices of other definitions in the same index space.

Element indices reference element segments and data indices reference data segments.

The index space for locals is only accessible inside a function and includes the parameters of that function, which precede the local variables.

Label indices reference structured control instructions inside an instruction sequence.

Conventions

- The meta variable l ranges over label indices.
- The meta variables x, y range over indices in any of the other index spaces.
- The notation $\mathrm{idx}(A)$ denotes the set of indices from index space idx occurring free in A. Sometimes this set is reinterpreted as the vector of its elements.

Note: For example, if instr^* is $(\mathsf{data.drop}~x)(\mathsf{memory.init}~y)$, then $\mathrm{dataidx}(\mathit{instr}^*) = \{x, y\}$, or equivalently, the vector $x~y$.

2.5.2 Types

The types component of a module defines a vector of function types.

All function types used in a module must be defined in this component. They are referenced by type indices.

Note: Future versions of WebAssembly may add additional forms of type definitions.

2.5.3 Functions

The funcs component of a module defines a vector of *functions* with the following structure:

$$func \quad ::= \quad \{\mathsf{type}\ typeidx, \mathsf{locals}\ vec(valtype), \mathsf{body}\ expr\}$$

The type of a function declares its signature by reference to a type defined in the module. The parameters of the function are referenced through 0-based local indices in the function's body; they are mutable.

The locals declare a vector of mutable local variables and their types. These variables are referenced through local indices in the function's body. The index of the first local is the smallest index not referencing a parameter.

The body is an instruction sequence that upon termination must produce a stack matching the function type's result type.

Functions are referenced through function indices, starting with the smallest index not referencing a function import.

2.5.4 Tables

The tables component of a module defines a vector of *tables* described by their table type:

$$table \quad ::= \quad \{\mathsf{type}\ tabletype\}$$

A table is a vector of opaque values of a particular reference type. The min size in the limits of the table type specifies the initial size of that table, while its max, if present, restricts the size to which it can grow later.

Tables can be initialized through element segments.

Tables are referenced through table indices, starting with the smallest index not referencing a table import. Most constructs implicitly reference table index 0.

2.5.5 Memories

The mems component of a module defines a vector of *linear memories* (or *memories* for short) as described by their memory type:

$$mem \quad ::= \quad \{\mathsf{type}\ memtype\}$$

A memory is a vector of raw uninterpreted bytes. The min size in the limits of the memory type specifies the initial size of that memory, while its max, if present, restricts the size to which it can grow later. Both are in units of page size.

Memories can be initialized through data segments.

Memories are referenced through memory indices, starting with the smallest index not referencing a memory import. Most constructs implicitly reference memory index 0.

Note: In the current version of WebAssembly, at most one memory may be defined or imported in a single module, and *all* constructs implicitly reference this memory 0. This restriction may be lifted in future versions.

2.5.6 Globals

The globals component of a module defines a vector of *global variables* (or *globals* for short):

$$global \ ::= \ \{\mathsf{type}\ globaltype, \mathsf{init}\ expr\}$$

Each global stores a single value of the given global type. Its type also specifies whether a global is immutable or mutable. Moreover, each global is initialized with an init value given by a constant initializer expression.

Globals are referenced through global indices, starting with the smallest index not referencing a global import.

2.5.7 Element Segments

The initial contents of a table is uninitialized. *Element segments* can be used to initialize a subrange of a table from a static vector of elements.

The elems component of a module defines a vector of element segments. Each element segment defines a reference type and a corresponding list of constant element expressions.

Element segments have a mode that identifies them as either *passive*, *active*, or *declarative*. A passive element segment's elements can be copied to a table using the table.init instruction. An active element segment copies its elements into a table during instantiation, as specified by a table index and a constant expression defining an offset into that table. A declarative element segment is not available at runtime but merely serves to forward-declare references that are formed in code with instructions like ref.func.

$$\begin{array}{lcl} elem & ::= & \{\mathsf{type}\ reftype, \mathsf{init}\ vec(expr), \mathsf{mode}\ elemmode\} \\ elemmode & ::= & \mathsf{passive} \\ & | & \mathsf{active}\ \{\mathsf{table}\ tableidx, \mathsf{offset}\ expr\} \\ & | & \mathsf{declarative} \end{array}$$

The offset is given by a constant expression.

Element segments are referenced through element indices.

2.5.8 Data Segments

The initial contents of a memory are zero bytes. *Data segments* can be used to initialize a range of memory from a static vector of bytes.

The datas component of a module defines a vector of data segments.

Like element segments, data segments have a mode that identifies them as either *passive* or *active*. A passive data segment's contents can be copied into a memory using the memory.init instruction. An active data segment copies its contents into a memory during instantiation, as specified by a memory index and a constant expression defining an offset into that memory.

$$\begin{array}{lcl} data & ::= & \{\mathsf{init}\ vec(byte), \mathsf{mode}\ datamode\} \\ datamode & ::= & \mathsf{passive} \\ & | & \mathsf{active}\ \{\mathsf{memory}\ memidx, \mathsf{offset}\ expr\} \end{array}$$

Data segments are referenced through data indices.

Note: In the current version of WebAssembly, at most one memory is allowed in a module. Consequently, the only valid $memidx$ is 0.

2.5.9 Start Function

The start component of a module declares the function index of a *start function* that is automatically invoked when the module is instantiated, after tables and memories have been initialized.

$$\begin{array}{lll} \mathit{start} & ::= & \{\mathsf{func}~\mathit{funcidx}\} \end{array}$$

Note: The start function is intended for initializing the state of a module. The module and its exports are not accessible externally before this initialization has completed.

2.5.10 Exports

The exports component of a module defines a set of *exports* that become accessible to the host environment once the module has been instantiated.

$$\begin{array}{llll} \mathit{export} & ::= & \{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{exportdesc}\} \\ \mathit{exportdesc} & ::= & \mathsf{func}~\mathit{funcidx} \\ & | & \mathsf{table}~\mathit{tableidx} \\ & | & \mathsf{mem}~\mathit{memidx} \\ & | & \mathsf{global}~\mathit{globalidx} \end{array}$$

Each export is labeled by a unique name. Exportable definitions are functions, tables, memories, and globals, which are referenced through a respective descriptor.

Conventions

The following auxiliary notation is defined for sequences of exports, filtering out indices of a specific kind in an order-preserving fashion:

- $\mathrm{funcs}(\mathit{export}^*) = [\mathit{funcidx} \mid \mathsf{func}~\mathit{funcidx} \in (\mathit{export}.\mathsf{desc})^*]$
- $\mathrm{tables}(\mathit{export}^*) = [\mathit{tableidx} \mid \mathsf{table}~\mathit{tableidx} \in (\mathit{export}.\mathsf{desc})^*]$
- $\mathrm{mems}(\mathit{export}^*) = [\mathit{memidx} \mid \mathsf{mem}~\mathit{memidx} \in (\mathit{export}.\mathsf{desc})^*]$
- $\mathrm{globals}(\mathit{export}^*) = [\mathit{globalidx} \mid \mathsf{global}~\mathit{globalidx} \in (\mathit{export}.\mathsf{desc})^*]$

2.5.11 Imports

The imports component of a module defines a set of *imports* that are required for instantiation.

$$\begin{array}{llll} \mathit{import} & ::= & \{\mathsf{module}~\mathit{name}, \mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{importdesc}\} \\ \mathit{importdesc} & ::= & \mathsf{func}~\mathit{typeidx} \\ & | & \mathsf{table}~\mathit{tabletype} \\ & | & \mathsf{mem}~\mathit{memtype} \\ & | & \mathsf{global}~\mathit{globaltype} \end{array}$$

Each import is labeled by a two-level name space, consisting of a module name and a name for an entity within that module. Importable definitions are functions, tables, memories, and globals. Each import is specified by a descriptor with a respective type that a definition provided during instantiation is required to match.

Every import defines an index in the respective index space. In each index space, the indices of imports go before the first index of any definition contained in the module itself.

Note: Unlike export names, import names are not necessarily unique. It is possible to import the same module/name pair multiple times; such imports may even have different type descriptions, including different kinds of entities. A module with such imports can still be instantiated depending on the specifics of how an embedder allows resolving and supplying imports. However, embedders are not required to support such overloading, and a WebAssembly module itself cannot implement an overloaded name.

CHAPTER 3

Validation

3.1 Conventions

Validation checks that a WebAssembly module is well-formed. Only valid modules can be instantiated.

Validity is defined by a *type system* over the abstract syntax of a module and its contents. For each piece of abstract syntax, there is a typing rule that specifies the constraints that apply to it. All rules are given in two *equivalent* forms:

1. In *prose*, describing the meaning in intuitive form.
2. In *formal notation*, describing the rule in mathematical form.[15]

Note: The prose and formal rules are equivalent, so that understanding of the formal notation is *not* required to read this specification. The formalism offers a more concise description in notation that is used widely in programming languages semantics and is readily amenable to mathematical proof.

In both cases, the rules are formulated in a *declarative* manner. That is, they only formulate the constraints, they do not define an algorithm. The skeleton of a sound and complete algorithm for type-checking instruction sequences according to this specification is provided in the appendix.

3.1.1 Contexts

Validity of an individual definition is specified relative to a *context*, which collects relevant information about the surrounding module and the definitions in scope:

- *Types*: the list of types defined in the current module.
- *Functions*: the list of functions declared in the current module, represented by their function type.
- *Tables*: the list of tables declared in the current module, represented by their table type.
- *Memories*: the list of memories declared in the current module, represented by their memory type.
- *Globals*: the list of globals declared in the current module, represented by their global type.

---

[15] The semantics is derived from the following article: Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman. Bringing the Web up to Speed with WebAssembly[16]. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM 2017.

[16] https://dl.acm.org/citation.cfm?doid=3062341.3062363

- *Element Segments*: the list of element segments declared in the current module, represented by their element type.
- *Data Segments*: the list of data segments declared in the current module, each represented by an ok entry.
- *Locals*: the list of locals declared in the current function (including parameters), represented by their value type.
- *Labels*: the stack of labels accessible from the current position, represented by their result type.
- *Return*: the return type of the current function, represented as an optional result type that is absent when no return is allowed, as in free-standing expressions.
- *References*: the list of function indices that occur in the module outside functions and can hence be used to form references inside them.

In other words, a context contains a sequence of suitable types for each index space, describing each defined entry in that space. Locals, labels and return type are only used for validating instructions in function bodies, and are left empty elsewhere. The label stack is the only part of the context that changes as validation of an instruction sequence proceeds.

More concretely, contexts are defined as records C with abstract syntax:

$$\begin{array}{llll} C & ::= & \{ & \mathsf{types}~ & \mathit{functype}^*, \\ &&& \mathsf{funcs}~ & \mathit{functype}^*, \\ &&& \mathsf{tables}~ & \mathit{tabletype}^*, \\ &&& \mathsf{mems}~ & \mathit{memtype}^*, \\ &&& \mathsf{globals}~ & \mathit{globaltype}^*, \\ &&& \mathsf{elems}~ & \mathit{reftype}^*, \\ &&& \mathsf{datas}~ & \mathrm{ok}^*, \\ &&& \mathsf{locals}~ & \mathit{valtype}^*, \\ &&& \mathsf{labels}~ & \mathit{resulttype}^*, \\ &&& \mathsf{return}~ & \mathit{resulttype}^?, \\ &&& \mathsf{refs}~ & \mathit{funcidx}^* ~\} \end{array}$$

In addition to field access written $C.\mathsf{field}$ the following notation is adopted for manipulating contexts:

- When spelling out a context, empty fields are omitted.
- $C, \mathsf{field}~A^*$ denotes the same context as C but with the elements A^* prepended to its field component sequence.

Note: Indexing notation like $C.\mathsf{labels}[i]$ is used to look up indices in their respective index space in the context. Context extension notation $C, \mathsf{field}~A$ is primarily used to locally extend *relative* index spaces, such as label indices. Accordingly, the notation is defined to append at the *front* of the respective sequence, introducing a new relative index 0 and shifting the existing ones.

3.1.2 Prose Notation

Validation is specified by stylised rules for each relevant part of the abstract syntax. The rules not only state constraints defining when a phrase is valid, they also classify it with a type. The following conventions are adopted in stating these rules.

- A phrase A is said to be "valid with type T" if and only if all constraints expressed by the respective rules are met. The form of T depends on what A is.

  Note: For example, if A is a function, then T is a function type; for an A that is a global, T is a global type; and so on.

- The rules implicitly assume a given context C.

- In some places, this context is locally extended to a context C' with additional entries. The formulation "Under context C', … *statement* …" is adopted to express that the following statement must apply under the assumptions embodied in the extended context.

3.1.3 Formal Notation

Note: This section gives a brief explanation of the notation for specifying typing rules formally. For the interested reader, a more thorough introduction can be found in respective text books.[17]

The proposition that a phrase A has a respective type T is written $A : T$. In general, however, typing is dependent on a context C. To express this explicitly, the complete form is a *judgement* $C \vdash A : T$, which says that $A : T$ holds under the assumptions encoded in C.

The formal typing rules use a standard approach for specifying type systems, rendering them into *deduction rules*. Every rule has the following general form:

$$\frac{\mathit{premise}_1 \qquad \mathit{premise}_2 \qquad \dots \qquad \mathit{premise}_n}{\mathit{conclusion}}$$

Such a rule is read as a big implication: if all premises hold, then the conclusion holds. Some rules have no premises; they are *axioms* whose conclusion holds unconditionally. The conclusion always is a judgment $C \vdash A : T$, and there is one respective rule for each relevant construct A of the abstract syntax.

Note: For example, the typing rule for the i32.add instruction can be given as an axiom:

$$\frac{}{C \vdash \mathsf{i32.add} : [\mathsf{i32}~\mathsf{i32}] \rightarrow [\mathsf{i32}]}$$

The instruction is always valid with type $[\mathsf{i32}~\mathsf{i32}] \rightarrow [\mathsf{i32}]$ (saying that it consumes two i32 values and produces one), independent of any side conditions.

An instruction like local.get can be typed as follows:

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.get}~x : [] \rightarrow [t]}$$

Here, the premise enforces that the immediate local index x exists in the context. The instruction produces a value of its respective type t (and does not consume any values). If $C.\mathsf{locals}[x]$ does not exist then the premise does not hold, and the instruction is ill-typed.

Finally, a structured instruction requires a recursive rule, where the premise is itself a typing judgement:

$$\frac{C \vdash \mathit{blocktype} : [t_1^*] \rightarrow [t_2^*] \qquad C, \mathsf{label}~[t_2^*] \vdash \mathit{instr}^* : [t_1^*] \rightarrow [t_2^*]}{C \vdash \mathsf{block}~\mathit{blocktype}~\mathit{instr}^*~\mathsf{end} : [t_1^*] \rightarrow [t_2^*]}$$

A block instruction is only valid when the instruction sequence in its body is. Moreover, the result type must match the block's annotation $\mathit{blocktype}$. If so, then the block instruction has the same type as the body. Inside the body an additional label of the corresponding result type is available, which is expressed by extending the context C with the additional label information for the premise.

---

[17] For example: Benjamin Pierce. Types and Programming Languages[18]. The MIT Press 2002

[18] https://www.cis.upenn.edu/~bcpierce/tapl/

3.2 Types

Most types are universally valid. However, restrictions apply to limits, which must be checked during validation. Moreover, block types are converted to plain function types for ease of processing.

3.2.1 Limits

Limits must have meaningful bounds that are within a given range.

$\{\mathsf{min}~n, \mathsf{max}~m^?\}$

- The value of n must not be larger than k.
- If the maximum $m^?$ is not empty, then:
  - Its value must not be larger than k.
  - Its value must not be smaller than n.
- Then the limit is valid within range k.

$$\frac{n \leq k \qquad (m \leq k)^? \qquad (n \leq m)^?}{\vdash \{\mathsf{min}~n, \mathsf{max}~m^?\} : k}$$

3.2.2 Block Types

Block types may be expressed in one of two forms, both of which are converted to plain function types by the following rules.

$\mathit{typeidx}$

- The type $C.\mathsf{types}[\mathit{typeidx}]$ must be defined in the context.
- Then the block type is valid as function type $C.\mathsf{types}[\mathit{typeidx}]$.

$$\frac{C.\mathsf{types}[\mathit{typeidx}] = \mathit{functype}}{C \vdash \mathit{typeidx} : \mathit{functype}}$$

$[\mathit{valtype}^?]$

- The block type is valid as function type $[] \rightarrow [\mathit{valtype}^?]$.

$$\frac{}{C \vdash [\mathit{valtype}^?] : [] \rightarrow [\mathit{valtype}^?]}$$

3.2.3 Function Types

Function types are always valid.

$[t_1^n] \rightarrow [t_2^m]$

- The function type is valid.

$$\frac{}{\vdash [t_1^*] \rightarrow [t_2^*]\ \mathrm{ok}}$$

3.2.4 Table Types

$limits\ reftype$

- The limits $limits$ must be valid within range $2^{32} - 1$.
- Then the table type is valid.

$$\frac{\vdash limits : 2^{32} - 1}{\vdash limits\ reftype\ \mathrm{ok}}$$

3.2.5 Memory Types

$limits$

- The limits $limits$ must be valid within range 2^{16}.
- Then the memory type is valid.

$$\frac{\vdash limits : 2^{16}}{\vdash limits\ share\ \mathrm{ok}}$$

3.2.6 Global Types

$mut\ valtype$

- The global type is valid.

$$\frac{}{\vdash mut\ valtype\ \mathrm{ok}}$$

3.2.7 External Types

$\mathsf{func}\ functype$

- The function type $functype$ must be valid.
- Then the external type is valid.

$$\frac{\vdash functype\ \mathrm{ok}}{\vdash \mathsf{func}\ functype\ \mathrm{ok}}$$

table *tabletype*

- The table type *tabletype* must be valid.
- Then the external type is valid.

$$\frac{\vdash \mathit{tabletype}~\mathrm{ok}}{\vdash \mathsf{table}~\mathit{tabletype}~\mathrm{ok}}$$

mem *memtype*

- The memory type *memtype* must be valid.
- Then the external type is valid.

$$\frac{\vdash \mathit{memtype}~\mathrm{ok}}{\vdash \mathsf{mem}~\mathit{memtype}~\mathrm{ok}}$$

global *globaltype*

- The global type *globaltype* must be valid.
- Then the external type is valid.

$$\frac{\vdash \mathit{globaltype}~\mathrm{ok}}{\vdash \mathsf{global}~\mathit{globaltype}~\mathrm{ok}}$$

3.2.8 Import Subtyping

When instantiating a module, external values must be provided whose types are *matched* against the respective external types classifying each import. In some cases, this allows for a simple form of subtyping (written “≤” formally), as defined here.

Limits

Limits $\{\mathsf{min}~n_1, \mathsf{max}~m_1^?\}$ match limits $\{\mathsf{min}~n_2, \mathsf{max}~m_2^?\}$ if and only if:

- n_1 is larger than or equal to n_2.
- Either:
  - $m_2^?$ is empty.
- Or:
  - Both $m_1^?$ and $m_2^?$ are non-empty.
  - m_1 is smaller than or equal to m_2.

$$\frac{n_1 \geq n_2}{\vdash \{\mathsf{min}~n_1, \mathsf{max}~m_1^?\} \leq \{\mathsf{min}~n_2, \mathsf{max}~\epsilon\}} \qquad \frac{n_1 \geq n_2 \qquad m_1 \leq m_2}{\vdash \{\mathsf{min}~n_1, \mathsf{max}~m_1\} \leq \{\mathsf{min}~n_2, \mathsf{max}~m_2\}}$$

Functions

An external type $\mathsf{func}~\mathit{functype}_1$ matches $\mathsf{func}~\mathit{functype}_2$ if and only if:

- Both $\mathit{functype}_1$ and $\mathit{functype}_2$ are the same.

$$\frac{}{\vdash \mathsf{func}~\mathit{functype} \leq \mathsf{func}~\mathit{functype}}$$

Tables

An external type $\mathsf{table}~(\mathit{limits}_1~\mathit{reftype}_1)$ matches $\mathsf{table}~(\mathit{limits}_2~\mathit{reftype}_2)$ if and only if:

- Limits limits_1 match limits_2.
- Both $\mathit{reftype}_1$ and $\mathit{reftype}_2$ are the same.

$$\frac{\vdash \mathit{limits}_1 \leq \mathit{limits}_2}{\vdash \mathsf{table}~(\mathit{limits}_1~\mathit{reftype}) \leq \mathsf{table}~(\mathit{limits}_2~\mathit{reftype})}$$

Memories

An external type $\mathsf{mem}~\mathit{limits}_1$ matches $\mathsf{mem}~\mathit{limits}_2$ if and only if:

- Limits limits_1 match limits_2.

$$\frac{\vdash \mathit{limits}_1 \leq \mathit{limits}_2}{\vdash \mathsf{mem}~\mathit{limits}_1 \leq \mathsf{mem}~\mathit{limits}_2}$$

Globals

An external type $\mathsf{global}~\mathit{globaltype}_1$ matches $\mathsf{global}~\mathit{globaltype}_2$ if and only if:

- Both $\mathit{globaltype}_1$ and $\mathit{globaltype}_2$ are the same.

$$\frac{}{\vdash \mathsf{global}~\mathit{globaltype} \leq \mathsf{global}~\mathit{globaltype}}$$

3.3 Instructions

Instructions are classified by *stack types* $[t_1^*] \rightarrow [t_2^*]$ that describe how instructions manipulate the operand stack.

$$\begin{array}{lll} \mathit{stacktype} & ::= & [\mathit{opdtype}^*] \rightarrow [\mathit{opdtype}^*] \\ \mathit{opdtype} & ::= & \mathit{valtype} \mid \bot \end{array}$$

The types describe the required input stack with *operand types* t_1^* that an instruction pops off and the provided output stack with result values of types t_2^* that it pushes back. Stack types are akin to function types, except that they allow individual operands to be classified as $\bot$ (*bottom*), indicating that the type is unconstrained. As an auxiliary notion, an operand type t_1 *matches* another operand type t_2, if t_1 is either $\bot$ or equal to t_2. This is extended to stack types in a point-wise manner.

$$\frac{}{\vdash t \leq t} \qquad \frac{}{\vdash \bot \leq t}$$

$$\frac{(\vdash t \leq t')^*}{\vdash [t^*] \leq [t'^*]}$$

Note: For example, the instruction $\mathsf{i32.add}$ has type $[\mathsf{i32}~\mathsf{i32}] \rightarrow [\mathsf{i32}]$, consuming two $\mathsf{i32}$ values and producing one.

Typing extends to instruction sequences $instr^*$. Such a sequence has a stack type $[t_1^*] \rightarrow [t_2^*]$ if the accumulative effect of executing the instructions is consuming values of types t_1^* off the operand stack and pushing new values of types t_2^*.

For some instructions, the typing rules do not fully constrain the type, and therefore allow for multiple types. Such instructions are called *polymorphic*. Two degrees of polymorphism can be distinguished:

- *value-polymorphic*: the value type t of one or several individual operands is unconstrained. That is the case for all parametric instructions like drop and select.
- *stack-polymorphic*: the entire (or most of the) stack type $[t_1^*] \rightarrow [t_2^*]$ of the instruction is unconstrained. That is the case for all control instructions that perform an *unconditional control transfer*, such as unreachable, br, br_table, and return.

In both cases, the unconstrained types or type sequences can be chosen arbitrarily, as long as they meet the constraints imposed for the surrounding parts of the program.

Note: For example, the select instruction is valid with type $[t~t~\mathsf{i32}] \rightarrow [t]$, for any possible number type t. Consequently, both instruction sequences

$$(\mathsf{i32.const}~1)~~(\mathsf{i32.const}~2)~~(\mathsf{i32.const}~3)~~\mathsf{select}$$

and

$$(\mathsf{f64.const}~1.0)~~(\mathsf{f64.const}~2.0)~~(\mathsf{i32.const}~3)~~\mathsf{select}$$

are valid, with t in the typing of select being instantiated to i32 or f64, respectively.

The unreachable instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$ for any possible sequences of operand types t_1^* and t_2^*. Consequently,

$$\mathsf{unreachable}~~\mathsf{i32.add}$$

is valid by assuming type $[] \rightarrow [\mathsf{i32}~\mathsf{i32}]$ for the unreachable instruction. In contrast,

$$\mathsf{unreachable}~~(\mathsf{i64.const}~0)~~\mathsf{i32.add}$$

is invalid, because there is no possible type to pick for the unreachable instruction that would make the sequence well-typed.

The Appendix describes a type checking algorithm that efficiently implements validation of instruction sequences as prescribed by the rules given here.

3.3.1 Numeric Instructions

$t.\mathsf{const}~c$

- The instruction is valid with type $[] \rightarrow [t]$.

$$\frac{}{C \vdash t.\mathsf{const}~c : [] \rightarrow [t]}$$

$t.\mathit{unop}$

- The instruction is valid with type $[t] \rightarrow [t]$.

$$\frac{}{C \vdash t.\mathit{unop} : [t] \rightarrow [t]}$$

$t.\mathit{binop}$

- The instruction is valid with type $[t\ t] \rightarrow [t]$.

$$\frac{}{C \vdash t.\mathit{binop} : [t\ t] \rightarrow [t]}$$

$t.\mathit{testop}$

- The instruction is valid with type $[t] \rightarrow [\mathsf{i32}]$.

$$\frac{}{C \vdash t.\mathit{testop} : [t] \rightarrow [\mathsf{i32}]}$$

$t.\mathit{relop}$

- The instruction is valid with type $[t\ t] \rightarrow [\mathsf{i32}]$.

$$\frac{}{C \vdash t.\mathit{relop} : [t\ t] \rightarrow [\mathsf{i32}]}$$

$t_2.\mathit{cvtop}\_t_1\_\mathit{sx}^?$

- The instruction is valid with type $[t_1] \rightarrow [t_2]$.

$$\frac{}{C \vdash t_2.\mathit{cvtop}\_t_1\_\mathit{sx}^? : [t_1] \rightarrow [t_2]}$$

3.3.2 Reference Instructions

$\mathsf{ref.null}\ t$

- The instruction is valid with type $[] \rightarrow [t]$.

$$\frac{}{C \vdash \mathsf{ref.null}\ t : [] \rightarrow [t]}$$

Note: In future versions of WebAssembly, there may be reference types for which no null reference is allowed.

$\mathsf{ref.is\_null}$

- The instruction is valid with type $[t] \rightarrow [\mathsf{i32}]$, for any reference type t.

$$\frac{t = \mathit{reftype}}{C \vdash \mathsf{ref.is\_null} : [t] \rightarrow [\mathsf{i32}]}$$

ref.func x

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- The function index x must be contained in $C.\mathsf{refs}$.
- The instruction is valid with type $[] \to [\mathsf{funcref}]$.

$$\frac{C.\mathsf{funcs}[x] = \mathit{functype} \qquad x \in C.\mathsf{refs}}{C \vdash \mathsf{ref.func}\ x : [] \to [\mathsf{funcref}]}$$

3.3.3 Vector Instructions

Vector instructions can have a prefix to describe the shape of the operand. Packed numeric types, i8 and i16, are not value types. An auxiliary function maps such packed type shapes to value types:

$$\begin{array}{lcl} \mathrm{unpacked}(\mathsf{i8x16}) &=& \mathsf{i32} \\ \mathrm{unpacked}(\mathsf{i16x8}) &=& \mathsf{i32} \\ \mathrm{unpacked}(t\mathsf{x}N) &=& t \end{array}$$

The following auxiliary function denotes the number of lanes in a vector shape, i.e., its *dimension*:

$$\mathrm{dim}(t\mathsf{x}N) \quad = \quad N$$

v128.const c

- The instruction is valid with type $[] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathsf{v128.const}\ c : [] \to [\mathsf{v128}]}$$

v128.vvunop

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvunop} : [\mathsf{v128}] \to [\mathsf{v128}]}$$

v128.$\mathit{vvbinop}$

- The instruction is valid with type $[\mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvbinop} : [\mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

v128.$\mathit{vvternop}$

- The instruction is valid with type $[\mathsf{v128}\ \mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvternop} : [\mathsf{v128}\ \mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathsf{v128}.\mathit{vvtestop}$

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{i32}]$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvtestop} : [\mathsf{v128}] \to [\mathsf{i32}]}$$

$\mathsf{i8x16.swizzle}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathsf{i8x16.swizzle} : [\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathsf{i8x16.shuffle}~\mathit{laneidx}^{16}$

- For all $\mathit{laneidx}_i$, in $\mathit{laneidx}^{16}$, $\mathit{laneidx}_i$ must be smaller than 32.
- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{(\mathit{laneidx} < 32)^{16}}{C \vdash \mathsf{i8x16.shuffle}~\mathit{laneidx}^{16} : [\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathsf{splat}$

- Let t be $\mathrm{unpacked}(\mathit{shape})$.
- The instruction is valid with type $[t] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathsf{splat} : [\mathrm{unpacked}(\mathit{shape})] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathsf{extract\_lane\_}\mathit{sx}^?~\mathit{laneidx}$

- The lane index $\mathit{laneidx}$ must be smaller than $\mathrm{dim}(\mathit{shape})$.
- The instruction is valid with type $[\mathsf{v128}] \to [\mathrm{unpacked}(\mathit{shape})]$.

$$\frac{\mathit{laneidx} < \mathrm{dim}(\mathit{shape})}{C \vdash t\mathsf{x}N.\mathsf{extract\_lane\_}\mathit{sx}^?~\mathit{laneidx} : [\mathsf{v128}] \to [\mathrm{unpacked}(\mathit{shape})]}$$

$\mathit{shape}.\mathsf{replace\_lane}~\mathit{laneidx}$

- The lane index $\mathit{laneidx}$ must be smaller than $\mathrm{dim}(\mathit{shape})$.
- Let t be $\mathrm{unpacked}(\mathit{shape})$.
- The instruction is valid with type $[\mathsf{v128}~t] \to [\mathsf{v128}]$.

$$\frac{\mathit{laneidx} < \mathrm{dim}(\mathit{shape})}{C \vdash \mathit{shape}.\mathsf{replace\_lane}~\mathit{laneidx} : [\mathsf{v128}~\mathrm{unpacked}(\mathit{shape})] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathit{vunop}$

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathit{vunop} : [\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathit{vbinop}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathit{vbinop} : [\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathit{vrelop}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathit{vrelop} : [\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{ishape}.\mathit{vishiftop}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{i32}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{ishape}.\mathit{vishiftop} : [\mathsf{v128}~\mathsf{i32}] \to [\mathsf{v128}]}$$

$\mathit{shape}.\mathit{vtestop}$

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{i32}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathit{vtestop} : [\mathsf{v128}] \to [\mathsf{i32}]}$$

$\mathit{shape}.\mathit{vcvtop}\_\mathit{half}^?\_\mathit{shape}\_\mathit{sx}^?\_\mathsf{zero}^?$

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{shape}.\mathit{vcvtop}\_\mathit{half}^?\_\mathit{shape}\_\mathit{sx}^?\_\mathsf{zero}^? : [\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{ishape}_1.\mathsf{narrow}\_\mathit{ishape}_2\_\mathit{sx}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{ishape}_1.\mathsf{narrow}\_\mathit{ishape}_2\_\mathit{sx} : [\mathsf{v128}~\mathsf{v128}] \to [\mathsf{v128}]}$$

$\mathit{ishape}.\mathsf{bitmask}$

- The instruction is valid with type $[\mathsf{v128}] \to [\mathsf{i32}]$.

$$\frac{}{C \vdash \mathit{ishape}.\mathsf{bitmask} : [\mathsf{v128}] \to [\mathsf{i32}]}$$

$\mathit{ishape}_1.\mathsf{dot}\_\mathit{ishape}_2\_\mathsf{s}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \rightarrow [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{ishape}_1.\mathsf{dot}\_\mathit{ishape}_2\_\mathsf{s} : [\mathsf{v128}~\mathsf{v128}] \rightarrow [\mathsf{v128}]}$$

$\mathit{ishape}_1.\mathsf{extmul}\_\mathit{half}\_\mathit{ishape}_2\_\mathit{sx}$

- The instruction is valid with type $[\mathsf{v128}~\mathsf{v128}] \rightarrow [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{ishape}_1.\mathsf{extmul}\_\mathit{half}\_\mathit{ishape}_2\_\mathit{sx} : [\mathsf{v128}~\mathsf{v128}] \rightarrow [\mathsf{v128}]}$$

$\mathit{ishape}_1.\mathsf{extadd\_pairwise}\_\mathit{ishape}_2\_\mathit{sx}$

- The instruction is valid with type $[\mathsf{v128}] \rightarrow [\mathsf{v128}]$.

$$\frac{}{C \vdash \mathit{ishape}_1.\mathsf{extadd\_pairwise}\_\mathit{ishape}_2\_\mathit{sx} : [\mathsf{v128}] \rightarrow [\mathsf{v128}]}$$

3.3.4 Parametric Instructions

drop

- The instruction is valid with type $[t] \rightarrow []$, for any operand type t.

$$\frac{}{C \vdash \mathsf{drop} : [t] \rightarrow []}$$

Note: Both drop and select without annotation are value-polymorphic instructions.

$\mathsf{select}~(t^*)^?$

- If t^* is present, then:
  - The length of t^* must be 1.
  - Then the instruction is valid with type $[t^*~t^*~\mathsf{i32}] \rightarrow [t^*]$.
- Else:
  - The instruction is valid with type $[t~t~\mathsf{i32}] \rightarrow [t]$, for any operand type t that matches some number type or vector type.

$$\frac{}{C \vdash \mathsf{select}~t : [t~t~\mathsf{i32}] \rightarrow [t]} \qquad \frac{\vdash t \leq \mathit{numtype}}{C \vdash \mathsf{select} : [t~t~\mathsf{i32}] \rightarrow [t]} \qquad \frac{\vdash t \leq \mathit{vectype}}{C \vdash \mathsf{select} : [t~t~\mathsf{i32}] \rightarrow [t]}$$

Note: In future versions of WebAssembly, select may allow more than one value per choice.

3.3.5 Variable Instructions

local.get x

- The local $C.\mathsf{locals}[x]$ must be defined in the context.
- Let t be the value type $C.\mathsf{locals}[x]$.
- Then the instruction is valid with type $[] \to [t]$.

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.get}\ x : [] \to [t]}$$

local.set x

- The local $C.\mathsf{locals}[x]$ must be defined in the context.
- Let t be the value type $C.\mathsf{locals}[x]$.
- Then the instruction is valid with type $[t] \to []$.

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.set}\ x : [t] \to []}$$

local.tee x

- The local $C.\mathsf{locals}[x]$ must be defined in the context.
- Let t be the value type $C.\mathsf{locals}[x]$.
- Then the instruction is valid with type $[t] \to [t]$.

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.tee}\ x : [t] \to [t]}$$

global.get x

- The global $C.\mathsf{globals}[x]$ must be defined in the context.
- Let $mut\ t$ be the global type $C.\mathsf{globals}[x]$.
- Then the instruction is valid with type $[] \to [t]$.

$$\frac{C.\mathsf{globals}[x] = mut\ t}{C \vdash \mathsf{global.get}\ x : [] \to [t]}$$

global.set x

- The global $C.\mathsf{globals}[x]$ must be defined in the context.
- Let $mut\ t$ be the global type $C.\mathsf{globals}[x]$.
- The mutability mut must be var.
- Then the instruction is valid with type $[t] \to []$.

$$\frac{C.\mathsf{globals}[x] = \mathsf{var}\ t}{C \vdash \mathsf{global.set}\ x : [t] \to []}$$

3.3.6 Table Instructions

table.get x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}~t$ be the table type $C.\mathsf{tables}[x]$.
- Then the instruction is valid with type $[\mathsf{i32}] \rightarrow [t]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}~t}{C \vdash \mathsf{table.get}~x : [\mathsf{i32}] \rightarrow [t]}$$

table.set x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}~t$ be the table type $C.\mathsf{tables}[x]$.
- Then the instruction is valid with type $[\mathsf{i32}~t] \rightarrow []$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}~t}{C \vdash \mathsf{table.set}~x : [\mathsf{i32}~t] \rightarrow []}$$

table.size x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Then the instruction is valid with type $[] \rightarrow [\mathsf{i32}]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{tabletype}}{C \vdash \mathsf{table.size}~x : [] \rightarrow [\mathsf{i32}]}$$

table.grow x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}~t$ be the table type $C.\mathsf{tables}[x]$.
- Then the instruction is valid with type $[t~\mathsf{i32}] \rightarrow [\mathsf{i32}]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}~t}{C \vdash \mathsf{table.grow}~x : [t~\mathsf{i32}] \rightarrow [\mathsf{i32}]}$$

table.fill x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}~t$ be the table type $C.\mathsf{tables}[x]$.
- Then the instruction is valid with type $[\mathsf{i32}~t~\mathsf{i32}] \rightarrow []$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}~t}{C \vdash \mathsf{table.fill}~x : [\mathsf{i32}~t~\mathsf{i32}] \rightarrow []}$$

$\mathsf{table.copy}\ x\ y$

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}_1\ t_1$ be the table type $C.\mathsf{tables}[x]$.
- The table $C.\mathsf{tables}[y]$ must be defined in the context.
- Let $\mathit{limits}_2\ t_2$ be the table type $C.\mathsf{tables}[y]$.
- The reference type t_1 must be the same as t_2.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \rightarrow []$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}_1\ t \qquad C.\mathsf{tables}[y] = \mathit{limits}_2\ t}{C \vdash \mathsf{table.copy}\ x\ y : [\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \rightarrow []}$$

$\mathsf{table.init}\ x\ y$

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}\ t_1$ be the table type $C.\mathsf{tables}[x]$.
- The element segment $C.\mathsf{elems}[y]$ must be defined in the context.
- Let t_2 be the reference type $C.\mathsf{elems}[y]$.
- The reference type t_1 must be the same as t_2.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \rightarrow []$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}\ t \qquad C.\mathsf{elems}[y] = t}{C \vdash \mathsf{table.init}\ x\ y : [\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \rightarrow []}$$

$\mathsf{elem.drop}\ x$

- The element segment $C.\mathsf{elems}[x]$ must be defined in the context.
- Then the instruction is valid with type $[] \rightarrow []$.

$$\frac{C.\mathsf{elems}[x] = t}{C \vdash \mathsf{elem.drop}\ x : [] \rightarrow []}$$

3.3.7 Memory Instructions

$t\mathsf{.load}\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must not be larger than the bit width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}] \rightarrow [t]$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} \leq |t|/8}{C \vdash t\mathsf{.load}\ \mathit{memarg} : [\mathsf{i32}] \rightarrow [t]}$$

$t.\mathsf{load}N\_sx\ memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}] \to [t]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq N/8}{C \vdash t.\mathsf{load}N\_sx\ memarg : [\mathsf{i32}] \to [t]}$$

$t.\mathsf{store}\ memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than the bit width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \to []$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq |t|/8}{C \vdash t.\mathsf{store}\ memarg : [\mathsf{i32}\ t] \to []}$$

$t.\mathsf{store}N\ memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \to []$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq N/8}{C \vdash t.\mathsf{store}N\ memarg : [\mathsf{i32}\ t] \to []}$$

$\mathsf{v128.load}N\mathsf{x}M\_sx\ memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8 \cdot M$.
- Then the instruction is valid with type $[\mathsf{i32}] \to [\mathsf{v128}]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq N/8 \cdot M}{C \vdash \mathsf{v128.load}N\mathsf{x}M\_sx\ memarg : [\mathsf{i32}] \to [\mathsf{v128}]}$$

$\mathsf{v128.load}N\_\mathsf{splat}\ memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}] \to [\mathsf{v128}]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq N/8}{C \vdash \mathsf{v128.load}N\_\mathsf{splat}\ memarg : [\mathsf{i32}] \to [\mathsf{v128}]}$$

v128.load$N$_zero $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}] \to [\mathsf{v128}]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} \leq N/8}{C \vdash \mathsf{v128.load}N\mathsf{\_zero}\ memarg : [\mathsf{i32}] \to [\mathsf{v128}]}$$

v128.load$N$_lane $memarg$ $laneidx$

- The lane index $laneidx$ must be smaller than $128/N$.
- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{laneidx < 128/N \qquad C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} < N/8}{C \vdash \mathsf{v128.load}N\mathsf{\_lane}\ memarg\ laneidx : [\mathsf{i32}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

v128.store$N$_lane $memarg$ $laneidx$

- The lane index $laneidx$ must be smaller than $128/N$.
- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must not be larger than $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{v128}] \to [\mathsf{v128}]$.

$$\frac{laneidx < 128/N \qquad C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} < N/8}{C \vdash \mathsf{v128.store}N\mathsf{\_lane}\ memarg\ laneidx : [\mathsf{i32}\ \mathsf{v128}] \to []}$$

memory.size

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[] \to [\mathsf{i32}]$.

$$\frac{C.\mathsf{mems}[0] = memtype}{C \vdash \mathsf{memory.size} : [] \to [\mathsf{i32}]}$$

memory.grow

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[\mathsf{i32}] \to [\mathsf{i32}]$.

$$\frac{C.\mathsf{mems}[0] = memtype}{C \vdash \mathsf{memory.grow} : [\mathsf{i32}] \to [\mathsf{i32}]}$$

memory.fill

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype}}{C \vdash \mathsf{memory.fill} : [\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []}$$

memory.copy

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype}}{C \vdash \mathsf{memory.copy} : [\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []}$$

memory.init x

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The data segment $C.\mathsf{datas}[x]$ must be defined in the context.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad C.\mathsf{datas}[x] = \mathrm{ok}}{C \vdash \mathsf{memory.init}\ x : [\mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []}$$

data.drop x

- The data segment $C.\mathsf{datas}[x]$ must be defined in the context.
- Then the instruction is valid with type $[] \to []$.

$$\frac{C.\mathsf{datas}[x] = \mathrm{ok}}{C \vdash \mathsf{data.drop}\ x : [] \to []}$$

3.3.8 Atomic Memory Instructions

t.atomic.load $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to the width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}] \to [t]$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = |t|/8}{C \vdash t.\mathsf{atomic.load}\ \mathit{memarg} : [\mathsf{i32}] \to [t]}$$

$t.\mathsf{atomic.load}N\_\mathsf{u}\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}] \rightarrow [t]$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = N/8}{C \vdash t.\mathsf{atomic.load}N\_\mathsf{u}\ \mathit{memarg} : [\mathsf{i32}] \rightarrow [t]}$$

$t.\mathsf{atomic.store}\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to the width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \rightarrow []$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = |t|/8}{C \vdash t.\mathsf{atomic.store}\ \mathit{memarg} : [\mathsf{i32}\ t] \rightarrow []}$$

$t.\mathsf{atomic.store}N\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \rightarrow []$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = N/8}{C \vdash t.\mathsf{atomic.store}N\ \mathit{memarg} : [\mathsf{i32}\ t] \rightarrow []}$$

$t.\mathsf{atomic.rmw}.\mathit{atop}\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to the width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \rightarrow [t]$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = |t|/8}{C \vdash t.\mathsf{atomic.rmw}.\mathit{atop}\ \mathit{memarg} : [\mathsf{i32}\ t] \rightarrow [t]}$$

$t.\mathsf{atomic.rmw}N.\mathit{atop}\_\mathsf{u}\ \mathit{memarg}$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{\mathit{memarg}.\mathsf{align}}$ must be equal to $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \rightarrow [t]$.

$$\frac{C.\mathsf{mems}[0] = \mathit{memtype} \qquad 2^{\mathit{memarg}.\mathsf{align}} = N/8}{C \vdash t.\mathsf{atomic.rmw}N.\mathit{atop}\_\mathsf{u}\ \mathit{memarg} : [\mathsf{i32}\ t] \rightarrow [t]}$$

t.atomic.rmw.cmpxchg $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must be equal to the width of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ t\ t] \to [t]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} = |t|/8}{C \vdash t.\mathsf{atomic.rmw.cmpxchg}\ memarg : [\mathsf{i32}\ t\ t] \to [t]}$$

t.atomic.rmw$N$.cmpxchg_u $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- The alignment $2^{memarg.\mathsf{align}}$ must be equal to $N/8$.
- Then the instruction is valid with type $[\mathsf{i32}\ t\ t] \to [t]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} = N/8}{C \vdash t.\mathsf{atomic.rmw}N\mathsf{.cmpxchg\_u}\ memarg : [\mathsf{i32}\ t\ t] \to [t]}$$

memory.atomic.notify $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Let $limits\ share$ be the memory type $C.\mathsf{mems}[0]$.
- The alignment $2^{memarg.\mathsf{align}}$ must be equal to 4.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}] \to [\mathsf{i32}]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} = 4}{C \vdash \mathsf{memory.atomic.notify}\ memarg : [\mathsf{i32}\ \mathsf{i32}] \to [\mathsf{i32}]}$$

memory.atomic.wait$N$ $memarg$

- The memory $C.\mathsf{mems}[0]$ must be defined in the context.
- Let $limits\ share$ be the memory type $C.\mathsf{mems}[0]$.
- The alignment $2^{memarg.\mathsf{align}}$ must be equal to N divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i}N\ \mathsf{i64}] \to [\mathsf{i32}]$.

$$\frac{C.\mathsf{mems}[0] = memtype \qquad 2^{memarg.\mathsf{align}} = N/8}{C \vdash \mathsf{memory.atomic.wait}N\ memarg : [\mathsf{i32}\ \mathsf{i}N\ \mathsf{i64}] \to [\mathsf{i32}]}$$

atomic.fence

- The instruction is valid with type $[] \to []$.

$$\frac{}{C \vdash \mathsf{atomic.fence} : [] \to []}$$

Note: The atomic.fence instruction may occur in modules which declare no memory without causing a validation error.

3.3.9 Control Instructions

nop

- The instruction is valid with type $[] \to []$.

$$\frac{}{C \vdash \mathsf{nop} : [] \to []}$$

unreachable

- The instruction is valid with type $[t_1^*] \to [t_2^*]$, for any sequences of operand types t_1^* and t_2^*.

$$\frac{}{C \vdash \mathsf{unreachable} : [t_1^*] \to [t_2^*]}$$

Note: The unreachable instruction is stack-polymorphic.

---

block *blocktype* *instr** end

- The block type must be valid as some function type $[t_1^*] \to [t_2^*]$.
- Let C' be the same context as C, but with the result type $[t_2^*]$ prepended to the labels vector.
- Under context C', the instruction sequence $instr^*$ must be valid with type $[t_1^*] \to [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*] \to [t_2^*]$.

$$\frac{C \vdash blocktype : [t_1^*] \to [t_2^*] \qquad C, \mathsf{labels}\,[t_2^*] \vdash instr^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{block}\ blocktype\ instr^*\ \mathsf{end} : [t_1^*] \to [t_2^*]}$$

Note: The notation $C, \mathsf{labels}\,[t^*]$ inserts the new label type at index 0, shifting all others.

---

loop *blocktype* *instr** end

- The block type must be valid as some function type $[t_1^*] \to [t_2^*]$.
- Let C' be the same context as C, but with the result type $[t_1^*]$ prepended to the labels vector.
- Under context C', the instruction sequence $instr^*$ must be valid with type $[t_1^*] \to [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*] \to [t_2^*]$.

$$\frac{C \vdash blocktype : [t_1^*] \to [t_2^*] \qquad C, \mathsf{labels}\,[t_1^*] \vdash instr^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{loop}\ blocktype\ instr^*\ \mathsf{end} : [t_1^*] \to [t_2^*]}$$

Note: The notation $C, \mathsf{labels}\,[t^*]$ inserts the new label type at index 0, shifting all others.

$\mathsf{if}~\mathit{blocktype}~\mathit{instr}_1^*~\mathsf{else}~\mathit{instr}_2^*~\mathsf{end}$

- The block type must be valid as some function type $[t_1^*] \to [t_2^*]$.
- Let C' be the same context as C, but with the result type $[t_2^*]$ prepended to the labels vector.
- Under context C', the instruction sequence instr_1^* must be valid with type $[t_1^*] \to [t_2^*]$.
- Under context C', the instruction sequence instr_2^* must be valid with type $[t_1^*] \to [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*~\mathsf{i32}] \to [t_2^*]$.

$$\frac{C \vdash \mathit{blocktype} : [t_1^*] \to [t_2^*] \qquad C,\mathsf{labels}\,[t_2^*] \vdash \mathit{instr}_1^* : [t_1^*] \to [t_2^*] \qquad C,\mathsf{labels}\,[t_2^*] \vdash \mathit{instr}_2^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{if}~\mathit{blocktype}~\mathit{instr}_1^*~\mathsf{else}~\mathit{instr}_2^*~\mathsf{end} : [t_1^*~\mathsf{i32}] \to [t_2^*]}$$

Note: The notation $C,\mathsf{labels}\,[t^*]$ inserts the new label type at index 0, shifting all others.

$\mathsf{br}~l$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t^*]$ be the result type $C.\mathsf{labels}[l]$.
- Then the instruction is valid with type $[t_1^*~t^*] \to [t_2^*]$, for any sequences of operand types t_1^* and t_2^*.

$$\frac{C.\mathsf{labels}[l] = [t^*]}{C \vdash \mathsf{br}~l : [t_1^*~t^*] \to [t_2^*]}$$

Note: The label index space in the context C contains the most recent label first, so that $C.\mathsf{labels}[l]$ performs a relative lookup as expected.

The br instruction is stack-polymorphic.

$\mathsf{br\_if}~l$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t^*]$ be the result type $C.\mathsf{labels}[l]$.
- Then the instruction is valid with type $[t^*~\mathsf{i32}] \to [t^*]$.

$$\frac{C.\mathsf{labels}[l] = [t^*]}{C \vdash \mathsf{br\_if}~l : [t^*~\mathsf{i32}] \to [t^*]}$$

Note: The label index space in the context C contains the most recent label first, so that $C.\mathsf{labels}[l]$ performs a relative lookup as expected.

$\mathsf{br\_table}~l^*~l_N$

- The label $C.\mathsf{labels}[l_N]$ must be defined in the context.
- For all l_i in l^*, the label $C.\mathsf{labels}[l_i]$ must be defined in the context.
- There must be a sequence t^* of operand types, such that:
  - For each operand type t_j in t^* and corresponding type t'_{Nj} in $C.\mathsf{labels}[l_N]$, t_j matches t'_{Nj}.
  - For all l_i in l^*, and for each operand type t_j in t^* and corresponding type t'_{ij} in $C.\mathsf{labels}[l_i]$, t_j matches t'_{ij}.

- Then the instruction is valid with type $[t_1^* \; t^* \; \mathsf{i32}] \rightarrow [t_2^*]$, for any sequences of operand types t_1^* and t_2^*.

$$\frac{(\vdash [t^*] \leq C.\mathsf{labels}[l])^* \qquad \vdash [t^*] \leq C.\mathsf{labels}[l_N]}{C \vdash \mathsf{br\_table} \; l^* \; l_N : [t_1^* \; t^* \; \mathsf{i32}] \rightarrow [t_2^*]}$$

Note: The label index space in the context C contains the most recent label first, so that $C.\mathsf{labels}[l_i]$ performs a relative lookup as expected.

The br_table instruction is stack-polymorphic.

return

- The return type $C.\mathsf{return}$ must not be absent in the context.
- Let $[t^*]$ be the result type of $C.\mathsf{return}$.
- Then the instruction is valid with type $[t_1^* \; t^*] \rightarrow [t_2^*]$, for any sequences of operand types t_1^* and t_2^*.

$$\frac{C.\mathsf{return} = [t^*]}{C \vdash \mathsf{return} : [t_1^* \; t^*] \rightarrow [t_2^*]}$$

Note: The return instruction is stack-polymorphic.

$C.\mathsf{return}$ is absent (set to ϵ) when validating an expression that is not a function body. This differs from it being set to the empty result type ($[\epsilon]$), which is the case for functions not returning anything.

call x

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- Then the instruction is valid with type $C.\mathsf{funcs}[x]$.

$$\frac{C.\mathsf{funcs}[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \mathsf{call} \; x : [t_1^*] \rightarrow [t_2^*]}$$

call_indirect x y

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits} \; t$ be the table type $C.\mathsf{tables}[x]$.
- The reference type t must be $\mathsf{funcref}$.
- The type $C.\mathsf{types}[y]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^*]$ be the function type $C.\mathsf{types}[y]$.
- Then the instruction is valid with type $[t_1^* \; \mathsf{i32}] \rightarrow [t_2^*]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits} \; \mathsf{funcref} \qquad C.\mathsf{types}[y] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \mathsf{call\_indirect} \; x \; y : [t_1^* \; \mathsf{i32}] \rightarrow [t_2^*]}$$

3.3.10 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

- The empty instruction sequence is valid with type $[t^*] \to [t^*]$, for any sequence of operand types t^*.

$$\frac{}{C \vdash \epsilon : [t^*] \to [t^*]}$$

Non-empty Instruction Sequence: $\mathit{instr}^*\ \mathit{instr}_N$

- The instruction sequence instr^* must be valid with type $[t_1^*] \to [t_2^*]$, for some sequences of operand types t_1^* and t_2^*.
- The instruction instr_N must be valid with type $[t^*] \to [t_3^*]$, for some sequences of operand types t^* and t_3^*.
- There must be a sequence of operand types t_0^*, such that $t_2^* = t_0^*\ t'^*$ where the type sequence t'^* is as long as t^*.
- For each operand type t'_i in t'^* and corresponding type t_i in t^*, t'_i matches t_i.
- Then the combined instruction sequence is valid with type $[t_1^*] \to [t_0^*\ t_3^*]$.

$$\frac{C \vdash \mathit{instr}^* : [t_1^*] \to [t_0^*\ t'^*] \qquad \vdash [t'^*] \leq [t^*] \qquad C \vdash \mathit{instr}_N : [t^*] \to [t_3^*]}{C \vdash \mathit{instr}^*\ \mathit{instr}_N : [t_1^*] \to [t_0^*\ t_3^*]}$$

3.3.11 Expressions

Expressions expr are classified by result types of the form $[t^*]$.

instr^* end

- The instruction sequence instr^* must be valid with some stack type $[] \to [t'^*]$.
- For each operand type t'_i in t'^* and corresponding value type t_i in t^*, t'_i matches t_i.
- Then the expression is valid with result type $[t^*]$.

$$\frac{C \vdash \mathit{instr}^* : [] \to [t'^*] \qquad \vdash [t'^*] \leq [t^*]}{C \vdash \mathit{instr}^*\ \mathsf{end} : [t^*]}$$

Constant Expressions

- In a *constant* expression instr^* end all instructions in instr^* must be constant.
- A constant instruction instr must be:
  - either of the form $t.\mathsf{const}\ c$,
  - or of the form ref.null,
  - or of the form ref.func x,
  - or of the form global.get x, in which case $C.\mathsf{globals}[x]$ must be a global type of the form const t.

$$\frac{(C \vdash \mathit{instr}\ \mathsf{const})^*}{C \vdash \mathit{instr}^*\ \mathsf{end}\ \mathsf{const}}$$

$$\frac{}{C \vdash t.\mathsf{const}\ c\ \mathsf{const}} \qquad \frac{}{C \vdash \mathsf{ref.null}\ t\ \mathsf{const}} \qquad \frac{}{C \vdash \mathsf{ref.func}\ x\ \mathsf{const}}$$

$$\frac{C.\mathsf{globals}[x] = \mathsf{const}\ t}{C \vdash \mathsf{global.get}\ x\ \mathsf{const}}$$

Note: Currently, constant expressions occurring in globals, element, or data segments are further constrained in that contained global.get instructions are only allowed to refer to *imported* globals. This is enforced in the validation rule for modules by constraining the context C accordingly.

The definition of constant expression may be extended in future versions of WebAssembly.

3.4 Modules

Modules are valid when all the components they contain are valid. Furthermore, most definitions are themselves classified with a suitable type.

3.4.1 Functions

Functions *func* are classified by function types of the form $[t_1^*] \rightarrow [t_2^*]$.

$\{\mathsf{type}\ x, \mathsf{locals}\ t^*, \mathsf{body}\ \mathit{expr}\}$

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^*]$ be the function type $C.\mathsf{types}[x]$.
- Let C' be the same context as C, but with:
  - locals set to the sequence of value types $t_1^*\ t^*$, concatenating parameters and locals,
  - labels set to the singular sequence containing only result type $[t_2^*]$.
  - return set to the result type $[t_2^*]$.
- Under the context C', the expression *expr* must be valid with type $[t_2^*]$.
- Then the function definition is valid with type $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{C.\mathsf{types}[x] = [t_1^*] \rightarrow [t_2^*] \qquad C, \mathsf{locals}\ t_1^*\ t^*, \mathsf{labels}\ [t_2^*], \mathsf{return}\ [t_2^*] \vdash \mathit{expr} : [t_2^*]}{C \vdash \{\mathsf{type}\ x, \mathsf{locals}\ t^*, \mathsf{body}\ \mathit{expr}\} : [t_1^*] \rightarrow [t_2^*]}$$

3.4.2 Tables

Tables *table* are classified by table types.

$\{\mathsf{type}~\mathit{tabletype}\}$

- The table type $\mathit{tabletype}$ must be valid.
- Then the table definition is valid with type $\mathit{tabletype}$.

$$\frac{\vdash \mathit{tabletype}~\mathrm{ok}}{C \vdash \{\mathsf{type}~\mathit{tabletype}\} : \mathit{tabletype}}$$

3.4.3 Memories

Memories mem are classified by memory types.

$\{\mathsf{type}~\mathit{memtype}\}$

- The memory type $\mathit{memtype}$ must be valid.
- Then the memory definition is valid with type $\mathit{memtype}$.

$$\frac{\vdash \mathit{memtype}~\mathrm{ok}}{C \vdash \{\mathsf{type}~\mathit{memtype}\} : \mathit{memtype}}$$

3.4.4 Globals

Globals global are classified by global types of the form $\mathit{mut}~t$.

$\{\mathsf{type}~\mathit{mut}~t, \mathsf{init}~\mathit{expr}\}$

- The global type $\mathit{mut}~t$ must be valid.
- The expression expr must be valid with result type $[t]$.
- The expression expr must be constant.
- Then the global definition is valid with type $\mathit{mut}~t$.

$$\frac{\vdash \mathit{mut}~t~\mathrm{ok} \qquad C \vdash \mathit{expr} : [t] \qquad C \vdash \mathit{expr}~\mathrm{const}}{C \vdash \{\mathsf{type}~\mathit{mut}~t, \mathsf{init}~\mathit{expr}\} : \mathit{mut}~t}$$

3.4.5 Element Segments

Element segments elem are classified by the reference type of their elements.

$\{\mathsf{type}~t, \mathsf{init}~e^*, \mathsf{mode}~\mathit{elemmode}\}$

- For each e_i in e^*:
  - The expression e_i must be valid with some result type $[t]$.
  - The expression e_i must be constant.
- The element mode $\mathit{elemmode}$ must be valid with reference type t.
- Then the element segment is valid with reference type t.

$$\frac{(C \vdash e : [t])^* \qquad (C \vdash e~\mathrm{const})^* \qquad C \vdash \mathit{elemmode} : t}{C \vdash \{\mathsf{type}~t, \mathsf{init}~e^*, \mathsf{mode}~\mathit{elemmode}\} : t}$$

$\mathsf{passive}$

- The element mode is valid with any reference type.

$$\frac{}{C \vdash \mathsf{passive} : \mathit{reftype}}$$

$\mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~\mathit{expr}\}$

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}~t$ be the table type $C.\mathsf{tables}[x]$.
- The expression expr must be valid with result type $[\mathsf{i32}]$.
- The expression expr must be constant.
- Then the element mode is valid with reference type t.

$$\frac{\begin{array}{@{}c@{}}C.\mathsf{tables}[x] = \mathit{limits}~t \\ C \vdash \mathit{expr} : [\mathsf{i32}] \qquad C \vdash \mathit{expr}~\mathrm{const}\end{array}}{C \vdash \mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~\mathit{expr}\} : t}$$

$\mathsf{declarative}$

- The element mode is valid with any reference type.

$$\frac{}{C \vdash \mathsf{declarative} : \mathit{reftype}}$$

3.4.6 Data Segments

Data segments data are not classified by any type but merely checked for well-formedness.

$\{\mathsf{init}~b^*, \mathsf{mode}~\mathit{datamode}\}$

- The data mode $\mathit{datamode}$ must be valid.
- Then the data segment is valid.

$$\frac{C \vdash \mathit{datamode}~\mathrm{ok}}{C \vdash \{\mathsf{init}~b^*, \mathsf{mode}~\mathit{datamode}\}~\mathrm{ok}}$$

$\mathsf{passive}$

- The data mode is valid.

$$\frac{}{C \vdash \mathsf{passive}~\mathrm{ok}}$$

$\mathsf{active}\ \{\mathsf{memory}\ x, \mathsf{offset}\ \mathit{expr}\}$

- The memory $C.\mathsf{mems}[x]$ must be defined in the context.
- The expression expr must be valid with result type $[\mathsf{i32}]$.
- The expression expr must be constant.
- Then the data mode is valid.

$$\frac{C.\mathsf{mems}[x] = \mathit{limits} \qquad C \vdash \mathit{expr} : [\mathsf{i32}] \qquad C \vdash \mathit{expr}\ \mathsf{const}}{C \vdash \mathsf{active}\ \{\mathsf{memory}\ x, \mathsf{offset}\ \mathit{expr}\}\ \mathrm{ok}}$$

3.4.7 Start Function

Start function declarations start are not classified by any type.

$\{\mathsf{func}\ x\}$

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- The type of $C.\mathsf{funcs}[x]$ must be $[] \to []$.
- Then the start function is valid.

$$\frac{C.\mathsf{funcs}[x] = [] \to []}{C \vdash \{\mathsf{func}\ x\}\ \mathrm{ok}}$$

3.4.8 Exports

Exports export and export descriptions $\mathit{exportdesc}$ are classified by their external type.

$\{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{exportdesc}\}$

- The export description $\mathit{exportdesc}$ must be valid with external type $\mathit{externtype}$.
- Then the export is valid with external type $\mathit{externtype}$.

$$\frac{C \vdash \mathit{exportdesc} : \mathit{externtype}}{C \vdash \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{exportdesc}\} : \mathit{externtype}}$$

$\mathsf{func}\ x$

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- Then the export description is valid with external type $\mathsf{func}\ C.\mathsf{funcs}[x]$.

$$\frac{C.\mathsf{funcs}[x] = \mathit{functype}}{C \vdash \mathsf{func}\ x : \mathsf{func}\ \mathit{functype}}$$

table x

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Then the export description is valid with external type table $C.\mathsf{tables}[x]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{tabletype}}{C \vdash \mathsf{table}\ x : \mathsf{table}\ \mathit{tabletype}}$$

mem x

- The memory $C.\mathsf{mems}[x]$ must be defined in the context.
- Then the export description is valid with external type mem $C.\mathsf{mems}[x]$.

$$\frac{C.\mathsf{mems}[x] = \mathit{memtype}}{C \vdash \mathsf{mem}\ x : \mathsf{mem}\ \mathit{memtype}}$$

global x

- The global $C.\mathsf{globals}[x]$ must be defined in the context.
- Then the export description is valid with external type global $C.\mathsf{globals}[x]$.

$$\frac{C.\mathsf{globals}[x] = \mathit{globaltype}}{C \vdash \mathsf{global}\ x : \mathsf{global}\ \mathit{globaltype}}$$

3.4.9 Imports

Imports import and import descriptions $\mathit{importdesc}$ are classified by external types.

$\{\mathsf{module}\ \mathit{name}_1, \mathsf{name}\ \mathit{name}_2, \mathsf{desc}\ \mathit{importdesc}\}$

- The import description $\mathit{importdesc}$ must be valid with type $\mathit{externtype}$.
- Then the import is valid with type $\mathit{externtype}$.

$$\frac{C \vdash \mathit{importdesc} : \mathit{externtype}}{C \vdash \{\mathsf{module}\ \mathit{name}_1, \mathsf{name}\ \mathit{name}_2, \mathsf{desc}\ \mathit{importdesc}\} : \mathit{externtype}}$$

func x

- The function $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^*]$ be the function type $C.\mathsf{types}[x]$.
- Then the import description is valid with type func $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{C.\mathsf{types}[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \mathsf{func}\ x : \mathsf{func}\ [t_1^*] \rightarrow [t_2^*]}$$

$\mathsf{table}~\mathit{tabletype}$

- The table type $\mathit{tabletype}$ must be valid.
- Then the import description is valid with type $\mathsf{table}~\mathit{tabletype}$.

$$\frac{\vdash \mathit{tabletype}~\mathrm{ok}}{C \vdash \mathsf{table}~\mathit{tabletype} : \mathsf{table}~\mathit{tabletype}}$$

$\mathsf{mem}~\mathit{memtype}$

- The memory type $\mathit{memtype}$ must be valid.
- Then the import description is valid with type $\mathsf{mem}~\mathit{memtype}$.

$$\frac{\vdash \mathit{memtype}~\mathrm{ok}}{C \vdash \mathsf{mem}~\mathit{memtype} : \mathsf{mem}~\mathit{memtype}}$$

$\mathsf{global}~\mathit{globaltype}$

- The global type $\mathit{globaltype}$ must be valid.
- Then the import description is valid with type $\mathsf{global}~\mathit{globaltype}$.

$$\frac{\vdash \mathit{globaltype}~\mathrm{ok}}{C \vdash \mathsf{global}~\mathit{globaltype} : \mathsf{global}~\mathit{globaltype}}$$

3.4.10 Modules

Modules are classified by their mapping from the external types of their imports to those of their exports.

A module is entirely *closed*, that is, its components can only refer to definitions that appear in the module itself. Consequently, no initial context is required. Instead, the context C for validation of the module's content is constructed from the definitions in the module.

- Let module be the module to validate.
- Let C be a context where:
  - $C.\mathsf{types}$ is $\mathit{module}.\mathsf{types}$,
  - $C.\mathsf{funcs}$ is $\mathrm{funcs}(\mathit{it}^*)$ concatenated with ft^*, with the import's external types it^* and the internal function types ft^* as determined below,
  - $C.\mathsf{tables}$ is $\mathrm{tables}(\mathit{it}^*)$ concatenated with tt^*, with the import's external types it^* and the internal table types tt^* as determined below,
  - $C.\mathsf{mems}$ is $\mathrm{mems}(\mathit{it}^*)$ concatenated with mt^*, with the import's external types it^* and the internal memory types mt^* as determined below,
  - $C.\mathsf{globals}$ is $\mathrm{globals}(\mathit{it}^*)$ concatenated with gt^*, with the import's external types it^* and the internal global types gt^* as determined below,
  - $C.\mathsf{elems}$ is rt^* as determined below,
  - $C.\mathsf{datas}$ is ok^n, where n is the length of the vector $\mathit{module}.\mathsf{datas}$,
  - $C.\mathsf{locals}$ is empty,
  - $C.\mathsf{labels}$ is empty,
  - $C.\mathsf{return}$ is empty.
  - $C.\mathsf{refs}$ is the set $\mathrm{funcidx}(\mathit{module}~\mathsf{with}~\mathsf{funcs} = \epsilon~\mathsf{with}~\mathsf{start} = \epsilon)$, i.e., the set of function indices occurring in the module, except in its functions or start function.

- Let C' be the context where:
  - $C'.\mathsf{globals}$ is the sequence $\mathrm{globals}(it^*)$,
  - $C'.\mathsf{funcs}$ is the same as $C.\mathsf{funcs}$,
  - $C'.\mathsf{refs}$ is the same as $C.\mathsf{refs}$,
  - all other fields are empty.
- For each $functype_i$ in $module.\mathsf{types}$, the function type $functype_i$ must be valid.
- Under the context C:
  - For each $func_i$ in $module.\mathsf{funcs}$, the definition $func_i$ must be valid with a function type ft_i.
  - If $module.\mathsf{start}$ is non-empty, then $module.\mathsf{start}$ must be valid.
  - For each $import_i$ in $module.\mathsf{imports}$, the segment $import_i$ must be valid with an external type it_i.
  - For each $export_i$ in $module.\mathsf{exports}$, the segment $export_i$ must be valid with external type et_i.
- Under the context C':
  - For each $table_i$ in $module.\mathsf{tables}$, the definition $table_i$ must be valid with a table type tt_i.
  - For each mem_i in $module.\mathsf{mems}$, the definition mem_i must be valid with a memory type mt_i.
  - For each $global_i$ in $module.\mathsf{globals}$, the definition $global_i$ must be valid with a global type gt_i.
  - For each $elem_i$ in $module.\mathsf{elems}$, the segment $elem_i$ must be valid with reference type rt_i.
  - For each $data_i$ in $module.\mathsf{datas}$, the segment $data_i$ must be valid.
- The length of $C.\mathsf{mems}$ must not be larger than 1.
- All export names $export_i.\mathsf{name}$ must be different.
- Let ft^* be the concatenation of the internal function types ft_i, in index order.
- Let tt^* be the concatenation of the internal table types tt_i, in index order.
- Let mt^* be the concatenation of the internal memory types mt_i, in index order.
- Let gt^* be the concatenation of the internal global types gt_i, in index order.
- Let rt^* be the concatenation of the reference types rt_i, in index order.
- Let it^* be the concatenation of external types it_i of the imports, in index order.
- Let et^* be the concatenation of external types et_i of the exports, in index order.
- Then the module is valid with external types $it^* \to et^*$.

$$\frac{\begin{array}{c}
(\vdash type~\mathrm{ok})^* \quad (C \vdash func : ft)^* \quad (C' \vdash table : tt)^* \quad (C' \vdash mem : mt)^* \quad (C' \vdash global : gt)^* \\
(C' \vdash elem : rt)^* \quad (C' \vdash data~\mathrm{ok})^n \quad (C \vdash start~\mathrm{ok})^? \quad (C \vdash import : it)^* \quad (C \vdash export : et)^* \\
ift^* = \mathrm{funcs}(it^*) \qquad itt^* = \mathrm{tables}(it^*) \qquad imt^* = \mathrm{mems}(it^*) \qquad igt^* = \mathrm{globals}(it^*) \\
x^* = \mathrm{funcidx}(module~\mathsf{with}~\mathsf{funcs} = \epsilon~\mathsf{with}~\mathsf{start} = \epsilon) \\
C = \{\mathsf{types}~type^*, \mathsf{funcs}~ift^*~ft^*, \mathsf{tables}~itt^*~tt^*, \mathsf{mems}~imt^*~mt^*, \mathsf{globals}~igt^*~gt^*, \mathsf{elems}~rt^*, \mathsf{datas}~\mathrm{ok}^n, \mathsf{refs}~x^*\} \\
C' = \{\mathsf{globals}~igt^*, \mathsf{funcs}~(C.\mathsf{funcs}), \mathsf{refs}~(C.\mathsf{refs})\} \qquad |C.\mathsf{mems}| \leq 1 \qquad (export.\mathsf{name})^*~\mathrm{disjoint} \\
module = \{\mathsf{types}~type^*, \mathsf{funcs}~func^*, \mathsf{tables}~table^*, \mathsf{mems}~mem^*, \mathsf{globals}~global^*, \\
\mathsf{elems}~elem^*, \mathsf{datas}~data^n, \mathsf{start}~start^?, \mathsf{imports}~import^*, \mathsf{exports}~export^*\}
\end{array}}{\vdash module : it^* \to et^*}$$

Note: Most definitions in a module – particularly functions – are mutually recursive. Consequently, the definition of the context C in this rule is recursive: it depends on the outcome of validation of the function, table, memory, and global definitions contained in the module, which itself depends on C. However, this recursion is just a specification device. All types needed to construct C can easily be determined from a simple pre-pass over the module that does not perform any actual validation.

Globals, however, are not recursive and not accessible within constant expressions when they are defined locally. The effect of defining the limited context C' for validating certain definitions is that they can only access functions and imported globals and nothing else.

Note: The restriction on the number of memories may be lifted in future versions of WebAssembly.

CHAPTER 4

Execution

4.1 Conventions

WebAssembly code is *executed* when instantiating a module or invoking an exported function on the resulting module instance.

Execution behavior is defined in terms of an *abstract machine* that models the *program state*. It includes a *stack*, which records operand values and control constructs, and an abstract *store* containing global state.

For each instruction, there is a rule that specifies the effect of its execution on the program state. Furthermore, there are rules describing the instantiation of a module. As with validation, all rules are given in two *equivalent* forms:

1. In *prose*, describing the execution in intuitive form.
2. In *formal notation*, describing the rule in mathematical form.[19]

Note: As with validation, the prose and formal rules are equivalent, so that understanding of the formal notation is *not* required to read this specification. The formalism offers a more concise description in notation that is used widely in programming languages semantics and is readily amenable to mathematical proof.

4.1.1 Prose Notation

Execution is specified by stylised, step-wise rules for each instruction of the abstract syntax. The following conventions are adopted in stating these rules.

- The execution rules implicitly assume a given store S.
- The execution rules also assume the presence of an implicit stack that is modified by *pushing* or *popping* values, labels, and frames.
- Certain rules require the stack to contain at least one frame. The most recent frame is referred to as the *current* frame.

---

[19] The semantics is derived from the following article: Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman. Bringing the Web up to Speed with WebAssembly[20]. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM 2017.

[20] https://dl.acm.org/citation.cfm?doid=3062341.3062363

- Both the store and the current frame are mutated by *replacing* some of their components. Such replacement is assumed to apply globally.
- The execution of an instruction may *trap*, in which case the entire computation is aborted and no further modifications to the store are performed by it. (Other computations can still be initiated afterwards.)
- The execution of an instruction may also end in a *jump* to a designated target, which defines the next instruction to execute.
- Execution can *enter* and *exit* instruction sequences that form blocks.
- Instruction sequences are implicitly executed in order, unless a trap or jump occurs.
- In various places the rules contain *assertions* expressing crucial invariants about the program state.

4.1.2 Formal Notation

Note: This section gives a brief explanation of the notation for specifying execution formally. For the interested reader, a more thorough introduction can be found in respective text books.[21]

The formal execution rules use a standard approach for specifying operational semantics, rendering them into *reduction rules*. Execution rules for each instruction have the following general form:

$$\mathit{configuration} \quad \hookrightarrow^{\mathit{act}^*} \quad \mathit{configuration}$$

A *configuration* is a syntactic description of a program state. Each rule specifies one *step* of execution, resulting in an altered configuration. In addition, steps which involve operations related to shared memory concurrency may emit a number of actions which are used to specify relevant concurrent behaviours (for further details, see Events and the Relaxed Memory Model). As long as there is at most one reduction rule applicable to a given configuration, reduction – and thereby execution – is *deterministic*. WebAssembly has only very few exceptions to this, which are noted explicitly in this specification.

For WebAssembly, a configuration typically is a tuple $(S; F; \mathit{instr}^*)$ consisting of the current store S, the call frame F of the current function, and the sequence of instructions that is to be executed. (A more precise definition is given later.)

To avoid unnecessary clutter, the store S and the frame F are omitted from reduction rules that do not touch them.

There is no separate representation of the stack. Instead, it is conveniently represented as part of the configuration's instruction sequence. In particular, values are defined to coincide with const instructions, and a sequence of const instructions can be interpreted as an operand "stack" that grows to the right.

Note: For example, the reduction rule for the i32.add instruction can be given as follows:

$$(\mathsf{i32.const}\ n_1)\ (\mathsf{i32.const}\ n_2)\ \mathsf{i32.add} \quad \hookrightarrow \quad (\mathsf{i32.const}\ (n_1 + n_2) \bmod 2^{32})$$

Per this rule, two const instructions and the add instruction itself are removed from the instruction stream and replaced with one new const instruction. This can be interpreted as popping two values off the stack and pushing the result.

When no result is produced, an instruction reduces to the empty sequence:

$$\mathsf{nop} \quad \hookrightarrow \quad \epsilon$$

Labels and frames are similarly defined to be part of an instruction sequence.

The order of reduction is determined by the definition of an appropriate evaluation context.

---

[21] For example: Benjamin Pierce. Types and Programming Languages[22]. The MIT Press 2002

[22] https://www.cis.upenn.edu/~bcpierce/tapl/

Reduction *terminates* when no more reduction rules are applicable. Soundness of the WebAssembly type system guarantees that this is only the case when the original instruction sequence has either been reduced to a sequence of const instructions, which can be interpreted as the values of the resulting operand stack, or if a trap occurred.

Note: For example, the following instruction sequence,

$$(\mathsf{f64.const}\ x_1)\ (\mathsf{f64.const}\ x_2)\ \mathsf{f64.neg}\ (\mathsf{f64.const}\ x_3)\ \mathsf{f64.add}\ \mathsf{f64.mul}$$

terminates after three steps:

$$\begin{array}{ll} & (\mathsf{f64.const}\ x_1)\ (\mathsf{f64.const}\ x_2)\ \mathsf{f64.neg}\ (\mathsf{f64.const}\ x_3)\ \mathsf{f64.add}\ \mathsf{f64.mul} \\ \hookrightarrow & (\mathsf{f64.const}\ x_1)\ (\mathsf{f64.const}\ x_4)\ (\mathsf{f64.const}\ x_3)\ \mathsf{f64.add}\ \mathsf{f64.mul} \\ \hookrightarrow & (\mathsf{f64.const}\ x_1)\ (\mathsf{f64.const}\ x_5)\ \mathsf{f64.mul} \\ \hookrightarrow & (\mathsf{f64.const}\ x_6) \end{array}$$

where $x_4 = -x_2$ and $x_5 = -x_2 + x_3$ and $x_6 = x_1 \cdot (-x_2 + x_3)$.

4.2 Runtime Structure

Store, stack, and other *runtime structure* forming the WebAssembly abstract machine, such as values or module instances, are made precise in terms of additional auxiliary syntax.

4.2.1 Values

WebAssembly computations manipulate *values* of either the four basic number types, i.e., integers and floating-point data of 32 or 64 bit width each, or vectors of 128 bit width, or of reference type.

In most places of the semantics, values of different types can occur. In order to avoid ambiguities, values are therefore represented with an abstract syntax that makes their type explicit. It is convenient to reuse the same notation as for the const instructions and ref.null producing them.

References other than null are represented with additional administrative instructions. They either are *function references*, pointing to a specific function address, or *external references* pointing to an uninterpreted form of extern address that can be defined by the embedder to represent its own objects.

$$\begin{array}{llll} \mathit{num} & ::= & \mathsf{i32.const}\ \mathit{i32} \\ & | & \mathsf{i64.const}\ \mathit{i64} \\ & | & \mathsf{f32.const}\ \mathit{f32} \\ & | & \mathsf{f64.const}\ \mathit{f64} \\ \mathit{vec} & ::= & \mathsf{v128.const}\ \mathit{i128} \\ \mathit{ref} & ::= & \mathsf{ref.null}\ t \\ & | & \mathsf{ref}\ \mathit{funcaddr} \\ & | & \mathsf{ref.extern}\ \mathit{externaddr} \\ \mathit{val} & ::= & \mathit{num} \mid \mathit{vec} \mid \mathit{ref} \end{array}$$

Note: Future versions of WebAssembly may add additional forms of reference.

Each value type has an associated *default value*; it is the respective value 0 for number types, 0 for vector types, and null for reference types.

$$\begin{array}{llll} \mathrm{default}_t & = & t.\mathsf{const}\ 0 & (\text{if } t = \mathit{numtype}) \\ \mathrm{default}_t & = & t.\mathsf{const}\ 0 & (\text{if } t = \mathit{vectype}) \\ \mathrm{default}_t & = & \mathsf{ref.null}\ t & (\text{if } t = \mathit{reftype}) \end{array}$$

Convention

- The meta variable r ranges over reference values where clear from context.

4.2.2 Results

A *result* is the outcome of a computation. It is either a sequence of values or a trap.

$$\begin{array}{llll} \mathit{result} & ::= & \mathit{val}^* \\ & | & \mathsf{trap} \end{array}$$

4.2.3 Store

The *store* represents all global state that can be manipulated by WebAssembly programs. It consists of the runtime representation of all *instances* of functions, tables, memories, and globals, element segments, and data segments that have been allocated during the life time of the abstract machine.[23]

It is an invariant of the semantics that no element or data instance is addressed from anywhere else but the owning module instances.

Syntactically, the store is defined as a record listing the existing instances of each category:

$$\begin{array}{llll} \mathit{store} & ::= & \{ & \mathsf{funcs}~\mathit{funcinst}^*, \\ &&& \mathsf{tables}~\mathit{tableinst}^*, \\ &&& \mathsf{mems}~\mathit{meminst}^*, \\ &&& \mathsf{globals}~\mathit{globalinst}^*, \\ &&& \mathsf{elems}~\mathit{eleminst}^*, \\ &&& \mathsf{datas}~\mathit{datainst}^* ~\} \end{array}$$

Convention

- The meta variable S ranges over stores where clear from context.

4.2.4 Addresses

Function instances, table instances, memory instances, and global instances, element instances, and data instances in the store are referenced with abstract *addresses*. These are simply indices into the respective store component. In addition, an embedder may supply an uninterpreted set of *host addresses*.

$$\begin{array}{lll} \mathit{addr} & ::= & 0 ~|~ 1 ~|~ 2 ~|~ \dots \\ \mathit{funcaddr} & ::= & \mathit{addr} \\ \mathit{tableaddr} & ::= & \mathit{addr} \\ \mathit{memaddr} & ::= & \mathit{addr} \\ \mathit{globaladdr} & ::= & \mathit{addr} \\ \mathit{elemaddr} & ::= & \mathit{addr} \\ \mathit{dataaddr} & ::= & \mathit{addr} \\ \mathit{externaddr} & ::= & \mathit{addr} \end{array}$$

An embedder may assign identity to exported store objects corresponding to their addresses, even where this identity is not observable from within WebAssembly code itself (such as for function instances or immutable globals).

Note: Addresses are *dynamic*, globally unique references to runtime objects, in contrast to indices, which are *static*, module-local references to their original definitions. A *memory address* $\mathit{memaddr}$ denotes the abstract address *of* a memory *instance* in the store, not an offset *inside* a memory instance.

---

[23] In practice, implementations may apply techniques like garbage collection to remove objects from the store that are no longer referenced. However, such techniques are not semantically observable, and hence outside the scope of this specification.

There is no specific limit on the number of allocations of store objects, hence logical addresses can be arbitrarily large natural numbers.

4.2.5 Time Stamps

In order to track the relative ordering in the execution of multiple threads and the occurrence of events, the semantics uses a notion of abstract *time stamps*.

$$\mathit{time} \quad ::= \quad \dots$$

Each time stamp denotes a discrete point in time, and is drawn from an infinite set. The shape of time stamps is not specified or observable.

Time stamps are associated with a total order – if a time stamp time_1 is *totally ordered before* time stamp time_2, this is written as $\mathit{time}_1 \prec_{\mathsf{tot}} \mathit{time}_2$.

Time stamps are also associated with a *happens before* partial order – if a time stamp time_1 *happens before* time_2, this is written as $\mathit{time}_1 \prec_{\mathsf{hb}} \mathit{time}_2$.

During the execution of WebAssembly code, time stamped events may be emitted which are related by one or both of these orderings, thus constraining the code's observable behaviours according to WebAssembly's relaxed memory model.

4.2.6 Module Instances

A *module instance* is the runtime representation of a module. It is created by instantiating a module, and collects runtime representations of all entities that are imported, defined, or exported by the module.

$$\begin{array}{llll}
\mathit{moduleinst} & ::= & \{ & \mathsf{types}~\mathit{functype}^*, \\
&&& \mathsf{funcaddrs}~\mathit{funcaddr}^*, \\
&&& \mathsf{tableaddrs}~\mathit{tableaddr}^*, \\
&&& \mathsf{memaddrs}~\mathit{memaddr}^*, \\
&&& \mathsf{globaladdrs}~\mathit{globaladdr}^*, \\
&&& \mathsf{elemaddrs}~\mathit{elemaddr}^*, \\
&&& \mathsf{dataaddrs}~\mathit{dataaddr}^*, \\
&&& \mathsf{exports}~\mathit{exportinst}^* ~\} \\
\end{array}$$

Each component references runtime instances corresponding to respective declarations from the original module – whether imported or defined – in the order of their static indices. Function instances, table instances, memory instances, and global instances are referenced with an indirection through their respective addresses in the store.

It is an invariant of the semantics that all export instances in a given module instance have different names.

4.2.7 Function Instances

A *function instance* is the runtime representation of a function. It effectively is a *closure* of the original function over the runtime module instance of its originating module. The module instance is used to resolve references to other definitions during execution of the function.

$$\begin{array}{llll}
\mathit{funcinst} & ::= & \{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\} \\
& | & \{\mathsf{type}~\mathit{functype}, \mathsf{hostcode}~\mathit{hostfunc}\} \\
\mathit{hostfunc} & ::= & \dots \\
\end{array}$$

A *host function* is a function expressed outside WebAssembly but passed to a module as an import. The definition and behavior of host functions are outside the scope of this specification. For the purpose of this specification, it is assumed that when invoked, a host function behaves non-deterministically, but within certain constraints that ensure the integrity of the runtime.

Note: Function instances are immutable, and their identity is not observable by WebAssembly code. However, the embedder might provide implicit or explicit means for distinguishing their addresses.

4.2.8 Table Instances

A *table instance* is the runtime representation of a table. It records its type and holds a vector of reference values.

$$\mathit{tableinst} ::= \{\mathsf{type}~\mathit{tabletype}, \mathsf{elem}~\mathit{vec}(\mathit{ref})\}$$

Table elements can be mutated through table instructions, the execution of an active element segment, or by external means provided by the embedder.

It is an invariant of the semantics that all table elements have a type equal to the element type of $\mathit{tabletype}$. It also is an invariant that the length of the element vector never exceeds the maximum size of $\mathit{tabletype}$, if present.

4.2.9 Memory Instances

A *memory instance* is the runtime representation of a linear memory. It records its original memory type and takes one of two different shapes depending on whether that type is shared or not. It is an invariant of the semantics that the shape always matches the type.

$$\begin{array}{lcl} \mathit{meminst} & ::= & \{\mathsf{type}~\mathit{limits}~\mathsf{unshared}, \mathsf{data}~\mathit{vec}(\mathit{byte})\} \\ & | & \{\mathsf{type}~\mathit{limits}~\mathsf{shared}\} \end{array}$$

The instance of a memory with unshared type holds a vector of bytes directly representing its state. The length of the vector always is a multiple of the WebAssembly *page size*, which is defined to be the constant 65536 – abbreviated $64\,\mathrm{Ki}$. The bytes can be mutated through memory instructions, the execution of an active data segment, or by external means provided by the embedder. It is an invariant of the semantics that the length of the byte vector, divided by page size, never exceeds the maximum size of $\mathit{memtype}$, if present.

For memories of shared type, no state is recorded in the instance itself.

4.2.10 Global Instances

A *global instance* is the runtime representation of a global variable. It records its type and holds an individual value.

$$\mathit{globalinst} ::= \{\mathsf{type}~\mathit{globaltype}, \mathsf{value}~\mathit{val}\}$$

The value of mutable globals can be mutated through variable instructions or by external means provided by the embedder.

It is an invariant of the semantics that the value has a type equal to the value type of $\mathit{globaltype}$.

4.2.11 Element Instances

An *element instance* is the runtime representation of an element segment. It holds a vector of references and their common type.

$$\mathit{eleminst} ::= \{\mathsf{type}~\mathit{reftype}, \mathsf{elem}~\mathit{vec}(\mathit{ref})\}$$

4.2.12 Data Instances

An *data instance* is the runtime representation of a data segment. It holds a vector of bytes.

$$\mathit{datainst} ::= \{\mathsf{data}\ \mathit{vec}(\mathit{byte})\}$$

4.2.13 Export Instances

An *export instance* is the runtime representation of an export. It defines the export's name and the associated external value.

$$\mathit{exportinst} ::= \{\mathsf{name}\ \mathit{name}, \mathsf{value}\ \mathit{externval}\}$$

4.2.14 External Values

An *external value* is the runtime representation of an entity that can be imported or exported. It is an address denoting either a function instance, table instance, memory instance, or global instances in the shared store.

$$\begin{array}{llll} \mathit{externval} & ::= & \mathsf{func}\ \mathit{funcaddr} \\ & | & \mathsf{table}\ \mathit{tableaddr} \\ & | & \mathsf{mem}\ \mathit{memaddr} \\ & | & \mathsf{global}\ \mathit{globaladdr} \end{array}$$

Conventions

The following auxiliary notation is defined for sequences of external values. It filters out entries of a specific kind in an order-preserving fashion:

- $\mathrm{funcs}(\mathit{externval}^*) = [\mathit{funcaddr} \mid (\mathsf{func}\ \mathit{funcaddr}) \in \mathit{externval}^*]$
- $\mathrm{tables}(\mathit{externval}^*) = [\mathit{tableaddr} \mid (\mathsf{table}\ \mathit{tableaddr}) \in \mathit{externval}^*]$
- $\mathrm{mems}(\mathit{externval}^*) = [\mathit{memaddr} \mid (\mathsf{mem}\ \mathit{memaddr}) \in \mathit{externval}^*]$
- $\mathrm{globals}(\mathit{externval}^*) = [\mathit{globaladdr} \mid (\mathsf{global}\ \mathit{globaladdr}) \in \mathit{externval}^*]$

4.2.15 Stack

Besides the store, most instructions interact with an implicit *stack*. The stack contains three kinds of entries:

- *Values*: the *operands* of instructions.
- *Labels*: active structured control instructions that can be targeted by branches.
- *Activations*: the *call frames* of active function calls.

These entries can occur on the stack in any order during the execution of a program. Stack entries are described by abstract syntax as follows.

Note: It is possible to model the WebAssembly semantics using separate stacks for operands, control constructs, and calls. However, because the stacks are interdependent, additional book keeping about associated stack heights would be required. For the purpose of this specification, an interleaved representation is simpler.

Values

Values are represented by themselves.

Labels

Labels carry an argument arity n and their associated branch *target*, which is expressed syntactically as an instruction sequence:

$$\mathit{label} \quad ::= \quad \mathsf{label}_n\{\mathit{instr}^*\}$$

Intuitively, instr^* is the *continuation* to execute when the branch is taken, in place of the original control construct.

Note: For example, a loop label has the form

$$\mathsf{label}_n\{\mathsf{loop}~\dots~\mathsf{end}\}$$

When performing a branch to this label, this executes the loop, effectively restarting it from the beginning. Conversely, a simple block label has the form

$$\mathsf{label}_n\{\epsilon\}$$

When branching, the empty continuation ends the targeted block, such that execution can proceed with consecutive instructions.

Activation Frames

Activation frames carry the return arity n of the respective function, hold the values of its locals (including arguments) in the order corresponding to their static local indices, and a reference to the function's own module instance:

$$\mathit{frame} \quad ::= \quad \{\mathsf{locals}~\mathit{val}^*, \mathsf{module}~\mathit{moduleinst}\}$$

The values of the locals are mutated by respective variable instructions.

Conventions

- The meta variable L ranges over labels where clear from context.
- The meta variable F ranges over frames where clear from context.
- The following auxiliary definition takes a block type and looks up the function type that it denotes in the current frame:

$$\begin{array}{lll}\mathrm{expand}_F(\mathit{typeidx}) &=& F.\mathsf{module}.\mathsf{types}[\mathit{typeidx}] \\ \mathrm{expand}_F([\mathit{valtype}^?]) &=& [] \rightarrow [\mathit{valtype}^?]\end{array}$$

4.2.16 Administrative Instructions

Note: This section is only relevant for the formal notation.

In order to express the reduction of traps, calls, and control instructions, the syntax of instructions is extended to include the following *administrative instructions*:

$$
\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{trap} \\
& | & \mathsf{ref}~\mathit{funcaddr} \\
& | & \mathsf{ref.extern}~\mathit{externaddr} \\
& | & \mathsf{invoke}~\mathit{funcaddr} \\
& | & \mathsf{label}_n\{\mathit{instr}^*\}~\mathit{instr}^*~\mathsf{end} \\
& | & \mathsf{frame}_n\{\mathit{frame}\}~\mathit{instr}^*~\mathsf{end} \\
& | & \mathsf{wait}'~\mathit{loc}~\mathit{s64} \\
& | & \mathsf{perform}~\mathit{act}^* \\
& | & \mathsf{host}~\mathit{resulttype}
\end{array}
$$

The trap instruction represents the occurrence of a trap. Traps are bubbled up through nested instruction sequences, ultimately reducing the entire program to a single trap instruction, signalling abrupt termination.

The ref instruction represents function reference values. Similarly, ref.extern represents external references.

The invoke instruction represents the imminent invocation of a function instance, identified by its address. It unifies the handling of different forms of calls.

The label and frame instructions model labels and frames "on the stack". Moreover, the administrative syntax maintains the nesting structure of the original structured control instruction or function body and their instruction sequences with an end marker. That way, the end of the inner instruction sequence is known when part of an outer sequence.

The wait' instruction models a thread suspending further execution as the result of executing a memory.atomic.wait instruction. If its $\mathit{s64}$ argument is non-negative, execution may resume after at least $\mathit{s64}$ nanoseconds. Otherwise, execution will only resume after loc is the target of a corresponding memory.atomic.notify instruction.

The perform instruction is used to perform an action at a precise point in execution. These actions are used by the Relaxed Memory Model to determine the behaviours that are observable in a concurrent execution.

The host instruction models the execution of the host environment.

Note: For example, the reduction rule for block is:

$$\mathsf{block}~[t^n]~\mathit{instr}^*~\mathsf{end} \quad\hookrightarrow\quad \mathsf{label}_n\{\epsilon\}~\mathit{instr}^*~\mathsf{end}$$

This replaces the block with a label instruction, which can be interpreted as "pushing" the label on the stack. When end is reached, i.e., the inner instruction sequence has been reduced to the empty sequence – or rather, a sequence of n const instructions representing the resulting values – then the label instruction is eliminated courtesy of its own reduction rule:

$$\mathsf{label}_m\{\mathit{instr}^*\}~\mathit{val}^n~\mathsf{end} \quad\hookrightarrow\quad \mathit{val}^n$$

This can be interpreted as removing the label from the stack and only leaving the locally accumulated operand values.

Block Contexts

In order to specify the reduction of branches, the following syntax of *block contexts* is defined, indexed by the count k of labels surrounding a *hole* $[\_]$ that marks the place where the next step of computation is taking place:

$$
\begin{array}{lll}
B^0 & ::= & \mathit{val}^*~[\_]~\mathit{instr}^* \\
B^{k+1} & ::= & \mathit{val}^*~\mathsf{label}_n\{\mathit{instr}^*\}~B^k~\mathsf{end}~\mathit{instr}^*
\end{array}
$$

This definition allows to index active labels surrounding a branch or return instruction.

Note: For example, the reduction of a simple branch can be defined as follows:

$$\mathsf{label}_0\{instr^*\}\ B^l[\mathsf{br}\ l]\ \mathsf{end} \quad\hookrightarrow\quad instr^*$$

Here, the hole [_] of the context is instantiated with a branch instruction. When a branch occurs, this rule replaces the targeted label and associated instruction sequence with the label's continuation. The selected label is identified through the label index l, which corresponds to the number of surrounding label instructions that must be hopped over – which is exactly the count encoded in the index of a block context.

4.2.17 Events

The interaction of a computation with the store is described through *events*. An event is a (possibly empty) set of *actions*, such as reads and writes, that are atomically performed by the execution of an individual instruction. Each event is annotated with two time stamps: the first records the time stamp of the event's immediate predecessor while the second uniquely identifies the event.

$$\begin{array}{llll}
evt & ::= & act^*\ \mathsf{at}\ time\ time \\
act & ::= & \mathsf{rd}_{ord}\ loc\ storeval\ \mathsf{notears}^? \\
 & | & \mathsf{wr}_{ord}\ loc\ storeval\ \mathsf{notears}^? \\
 & | & \mathsf{rmw}\ loc\ storeval\ storeval \\
 & | & \mathsf{wait}\ loc\ s64 \\
 & | & \mathsf{woken}\ loc \\
 & | & \mathsf{timeout}\ loc \\
 & | & \mathsf{notify}\ loc\ u32\ u32 \\
 & | & \mathsf{fence} \\
 & | & \mathsf{spawn} \\
 & | & hostact \\
ord & ::= & \mathsf{unord} \mid \mathsf{seqcst} \mid \mathsf{init} \\
loc & ::= & reg[u32] \\
reg & ::= & addr.fld \\
fld & ::= & \mathsf{len} \mid \mathsf{data} \\
storeval & ::= & val \mid byte^*
\end{array}$$

The access of *mutable* shared state is performed through the rd, wr, and rmw actions. Each action accesses an abstract *location*, which consists of an address of a shared memory instance, a symbolic *field* name in the respective object (either len for the size or data for the vector of bytes), and an offset index into the field.

In each case, read and write actions record the *store value* that has been read or written, which is either a regular value or a sequence of bytes, depending on the location accessed. An rmw event, performing an atomic read-modify-write access, records both the store values read (first) and written (second); it is an invariant of the semantics that both are either regular values of the same type or byte sequences of the same length.

rd and wr events are further annotated by a memory *ordering*, which describes whether the access is *unordered*, as e.g. performed by a regular load or store instruction, or *sequentially consistent*, as e.g. performed by atomic memory instructions. A third ordering, *initialisation*, is used to record the initial value taken by the memory location upon its creation. A rmw action always is sequentially consistent.

Note: Future versions of WebAssembly may introduce additional orderings.

Finally, a *host action* is an action performed outside of WebAssembly code. Its form and meaning is outside the scope of this specification.

Note: An embedder may define a custom set of host actions and respective ordering constraints to model other forms of interactions that are not expressible within WebAssembly, but whose ordering relative to WebAssembly events is relevant for the combined semantics.

Conventions

- The actions rd_{ord} and wr_{ord} are abbreviated to just rd and wr when ord is unord.
- A location may syntactically elide its $[u32]$ offset in the case that it is 0.

The following auxiliary definition is used to classify whether an access will *tear* - that is, whether it will observably decompose into bytewise accesses when it participates in a data race.

$$\begin{array}{lcll}
\mathrm{tearing}(iN', N, u32) & = & \mathsf{notears} & (\text{if } u32 \bmod N/8 = 0 \wedge N \leq 32) \\
\mathrm{tearing}(iN', N, u32) & = & \epsilon & (otherwise) \\
\mathrm{tearing}(fN', N, u32) & = & \epsilon &
\end{array}$$

Relations between time stamps are lifted to relations between events.

$$\begin{array}{lllllllll}
act_1^* & \mathtt{at} & time_p\ time_1 & \prec_{\mathsf{tot}} & act_2^*\ \mathtt{at}\ time'_p\ time_2 & = & time_1 & \prec_{\mathsf{tot}} & time_2 \\
act_1^* & \mathtt{at} & time_p\ time_1 & \prec_{\mathsf{hb}} & act_2^*\ \mathtt{at}\ time'_p\ time_2 & = & time_1 & \prec_{\mathsf{hb}} & time_2
\end{array}$$

Configurations

A thread is a computation over instructions that operates relative to a current frame referring to the module instance in which the current computation runs. The thread is annotated with the time it was last active.

A *global configuration* consists of the current store and a list of executing *threads*.

A *local configuration* describes the state of a single executing thread.

$$\begin{array}{lcl}
thread & ::= & frame; instr^*\ \mathtt{at}\ time \\
config & ::= & store; thread^* \\
lconfig & ::= & store; thread
\end{array}$$

A thread has *terminated* when its instruction sequence has been reduced to a result, that is, either a sequence of values or to a trap.

Convention

- The meta variable P ranges over threads where clear from context.

4.2.18 Reduction

Formally, WebAssembly computation is defined by two *small-step reduction* relations on global and local configurations that define how a single step of execution modifies these configurations, respectively.

Global Reduction

Global reduction is concerned with allocation in the global store and synchronization between multiple threads. It emits a (possibly empty) set of events that are produced by the corresponding step of computation.[24]

Formally, global reduction is a relation

$$config \hookrightarrow^{evt} config$$

defined by inductive rewrite rules on global configurations.

---

[24] The semantics of global configurations is derived from the following article: Conrad Watt, Andreas Rossberg, Jean Pichon-Pharabod. Weakening WebAssembly[25]. Proceedings of the ACM on Programming Languages (OOPSLA 2019). ACM 2019.

[25] https://dl.acm.org/citation.cfm?id=3360559

The following structural rule for global reduction delegates to local reduction for single thread execution:

$$\begin{array}{lll} S; P_1^*~(F; \mathit{instr}^*~\mathsf{at}~h)~P_2^* & \hookrightarrow^{\mathit{act}^*~\mathsf{at}~h~h'} & S'; P_1^*~(F'; \mathit{instr'}^*~\mathsf{at}~h')~P_2^* \\ & \multicolumn{2}{l}{(\mathrm{if}~S; F; \mathit{instr}^* \hookrightarrow^{\mathit{act}^*} S'; F'; \mathit{instr'}^*} \\ & \multicolumn{2}{l}{\wedge~h \prec_{\mathsf{hb}} h'} \\ & \multicolumn{2}{l}{\wedge~\mathsf{spawn} \notin \mathit{act}^*)} \end{array}$$

The following rule for global reduction describes the creation of a new thread by the host:

$$\begin{array}{lll} S; P_1^*~(F; \mathit{instr}^*~\mathsf{at}~h)~P_2^* & \hookrightarrow^{\epsilon~\mathsf{at}~h~h'} & S'; P_1^*~(F'; \mathit{instr'}^*~\mathsf{at}~h')~P_2^*~(\epsilon; (\mathsf{host}~[\epsilon])~\mathsf{at}~h'') \\ & \multicolumn{2}{l}{(\mathrm{if}~S; F; \mathit{instr}^* \hookrightarrow^{\mathsf{spawn}} S'; F'; \mathit{instr'}^*} \\ & \multicolumn{2}{l}{\wedge~h \prec_{\mathsf{hb}} h'} \\ & \multicolumn{2}{l}{\wedge~h \prec_{\mathsf{hb}} h''} \end{array}$$

Note: The time stamp h' indicates the point in time at which the computation step takes place, marking both the emitted atomic event and the updated time of the thread. This time stamp is chosen non-deterministically in the rule. However, the second side condition ensures that the time h of the last activity of the thread *happened before* h', thereby imposing *program order* for any events originating from the same thread.

Local Reduction

Local reduction defines the execution of individual instructions. Each execution step can perform a (possibly empty) set of actions.

Formally, this is described by a labelled relation

$$\mathit{lconfig} \hookrightarrow^{\mathit{act}^*} \mathit{lconfig}$$

To avoid unnecessary clutter, the following conventions are employed in the notation for local reduction rules:

- The configuration's store S is omitted from rules that do not touch it.
- The configuration's frame F is omitted from rules that do not touch it.

Evaluation Contexts

Finally, the following definition of *evaluation context* and associated structural rules enable reduction inside instruction sequences and administrative forms as well as the propagation of traps:

$$E ~~::=~~ [\_] ~|~ \mathit{val}^*~E~\mathit{instr}^* ~|~ \mathsf{label}_n\{\mathit{instr}^*\}~E~\mathsf{end}$$

$$\begin{array}{rcl} S; F; E[\mathit{instr}^*] & \hookrightarrow^{\mathit{act}^*} & S'; F'; E[\mathit{instr'}^*] \\ & & (\mathrm{if}~S; F; \mathit{instr}^* \hookrightarrow^{\mathit{act}^*} S'; F'; \mathit{instr'}^*) \\ S; F; \mathsf{frame}_n\{F'\}~\mathit{instr}^*~\mathsf{end} & \hookrightarrow^{\mathit{act}^*} & S'; F; \mathsf{frame}_n\{F''\}~\mathit{instr'}^*~\mathsf{end} \\ & & (\mathrm{if}~S; F'; \mathit{instr}^* \hookrightarrow^{\mathit{act}^*} S'; F''; \mathit{instr'}^*) \\ S; F; E[\mathsf{trap}] & \hookrightarrow & S; F; \mathsf{trap} \qquad (\mathrm{if}~E \neq [\_]) \\ S; F; \mathsf{frame}_n\{F'\}~\mathsf{trap}~\mathsf{end} & \hookrightarrow & S; F; \mathsf{trap} \end{array}$$

Reduction terminates when a thread's instruction sequence has been reduced to a result, that is, either a sequence of values or to a trap.

Note: The restriction on evaluation contexts rules out contexts like $[\_]$ and $\epsilon~[\_]~\epsilon$ for which $E[\mathsf{trap}] = \mathsf{trap}$.

For an example of reduction under evaluation contexts, consider the following instruction sequence.

$$(\mathsf{f64.const}~x_1)~(\mathsf{f64.const}~x_2)~\mathsf{f64.neg}~(\mathsf{f64.const}~x_3)~\mathsf{f64.add}~\mathsf{f64.mul}$$

This can be decomposed into $E[(\mathsf{f64.const}\ x_2)\ \mathsf{f64.neg}]$ where

$$E = (\mathsf{f64.const}\ x_1)\ [\_]\ (\mathsf{f64.const}\ x_3)\ \mathsf{f64.add}\ \mathsf{f64.mul}$$

Moreover, this is the *only* possible choice of evaluation context where the contents of the hole matches the left-hand side of a reduction rule.

4.3 Numerics

Numeric primitives are defined in a generic manner, by operators indexed over a bit width N.

Some operators are *non-deterministic*, because they can return one of several possible results (such as different NaN values). Technically, each operator thus returns a *set* of allowed values. For convenience, deterministic results are expressed as plain values, which are assumed to be identified with a respective singleton set.

Some operators are *partial*, because they are not defined on certain inputs. Technically, an empty set of results is returned for these inputs.

In formal notation, each operator is defined by equational clauses that apply in decreasing order of precedence. That is, the first clause that is applicable to the given arguments defines the result. In some cases, similar clauses are combined into one by using the notation $\pm$ or $\mp$. When several of these placeholders occur in a single clause, then they must be resolved consistently: either the upper sign is chosen for all of them or the lower sign.

Note: For example, the fcopysign operator is defined as follows:

$$\begin{array}{lcl}
\mathrm{fcopysign}_N(\pm p_1, \pm p_2) &=& \pm p_1 \\
\mathrm{fcopysign}_N(\pm p_1, \mp p_2) &=& \mp p_1
\end{array}$$

This definition is to be read as a shorthand for the following expansion of each clause into two separate ones:

$$\begin{array}{lcl}
\mathrm{fcopysign}_N(+p_1, +p_2) &=& +p_1 \\
\mathrm{fcopysign}_N(-p_1, -p_2) &=& -p_1 \\
\mathrm{fcopysign}_N(+p_1, -p_2) &=& -p_1 \\
\mathrm{fcopysign}_N(-p_1, +p_2) &=& +p_1
\end{array}$$

Numeric operators are lifted to input sequences by applying the operator element-wise, returning a sequence of results. When there are multiple inputs, they must be of equal length.

$$op(c_1^n, \dots, c_k^n) \quad = \quad op(c_1^n[0], \dots, c_k^n[0])\ \dots\ op(c_1^n[n-1], \dots, c_k^n[n-1])$$

Note: For example, the unary operator fabs, when given a sequence of floating-point values, return a sequence of floating-point results:

$$\mathrm{fabs}_N(z^n) \quad = \quad \mathrm{fabs}_N(z[0])\ \dots\ \mathrm{fabs}_N(z[n])$$

The binary operator iadd, when given two sequences of integers of the same length, n, return a sequence of integer results:

$$\mathrm{iadd}_N(i_1^n, i_2^n) \quad = \quad \mathrm{iadd}_N(i_1[0], i_2[0])\ \dots\ \mathrm{iadd}_N(i_1[n], i_2[n])$$

Conventions:

- The meta variable d is used to range over single bits.
- The meta variable p is used to range over (signless) magnitudes of floating-point values, including nan and ∞.

- The meta variable q is used to range over (signless) *rational* magnitudes, excluding nan or ∞.
- The notation f^{-1} denotes the inverse of a bijective function f.
- Truncation of rational values is written $\mathrm{trunc}(\pm q)$, with the usual mathematical definition:

$$\mathrm{trunc}(\pm q) \quad = \quad \pm i \qquad (\text{if } i \in \mathbb{N} \wedge +q - 1 < i \leq +q)$$

- Saturation of integers is written $\mathrm{sat\_u}_N(i)$ and $\mathrm{sat\_s}_N(i)$. The arguments to these two functions range over arbitrary signed integers.
  - Unsigned saturation, $\mathrm{sat\_u}_N(i)$ clamps i to between 0 and $2^N - 1$:

$$\begin{array}{lcll} \mathrm{sat\_u}_N(i) & = & 2^N - 1 & (\text{if } i > 2^N - 1) \\ \mathrm{sat\_u}_N(i) & = & 0 & (\text{if } i < 0) \\ \mathrm{sat\_u}_N(i) & = & i & (\text{otherwise}) \end{array}$$

  - Signed saturation, $\mathrm{sat\_s}_N(i)$ clamps i to between -2^{N-1} and $2^{N-1} - 1$:

$$\begin{array}{lcll} \mathrm{sat\_s}_N(i) & = & \mathrm{signed}_N^{-1}(-2^{N-1}) & (\text{if } i < -2^{N-1}) \\ \mathrm{sat\_s}_N(i) & = & \mathrm{signed}_N^{-1}(2^{N-1} - 1) & (\text{if } i > 2^{N-1} - 1) \\ \mathrm{sat\_s}_N(i) & = & i & (\text{otherwise}) \end{array}$$

4.3.1 Representations

Numbers and numeric vectors have an underlying binary representation as a sequence of bits:

$$\begin{array}{lcl} \mathrm{bits}_{\mathsf{i}N}(i) & = & \mathrm{ibits}_N(i) \\ \mathrm{bits}_{\mathsf{f}N}(z) & = & \mathrm{fbits}_N(z) \\ \mathrm{bits}_{\mathsf{v}N}(i) & = & \mathrm{ibits}_N(i) \end{array}$$

Each of these functions is a bijection, hence they are invertible.

Integers

Integers are represented as base two unsigned numbers:

$$\mathrm{ibits}_N(i) \quad = \quad d_{N-1}\ \ldots\ d_0 \qquad (i = 2^{N-1} \cdot d_{N-1} + \cdots + 2^0 \cdot d_0)$$

Boolean operators like $\wedge$, $\vee$, or $\veebar$ are lifted to bit sequences of equal length by applying them pointwise.

Floating-Point

Floating-point values are represented in the respective binary format defined by IEEE 754[26] (Section 3.4):

$$\begin{array}{lcl} \mathrm{fbits}_N(\pm(1 + m \cdot 2^{-M}) \cdot 2^e) & = & \mathrm{fsign}(\pm)\ \mathrm{ibits}_E(e + \mathrm{fbias}_N)\ \mathrm{ibits}_M(m) \\ \mathrm{fbits}_N(\pm(0 + m \cdot 2^{-M}) \cdot 2^e) & = & \mathrm{fsign}(\pm)\ (0)^E\ \mathrm{ibits}_M(m) \\ \mathrm{fbits}_N(\pm\infty) & = & \mathrm{fsign}(\pm)\ (1)^E\ (0)^M \\ \mathrm{fbits}_N(\pm\mathsf{nan}(n)) & = & \mathrm{fsign}(\pm)\ (1)^E\ \mathrm{ibits}_M(n) \\ \mathrm{fbias}_N & = & 2^{E-1} - 1 \\ \mathrm{fsign}(+) & = & 0 \\ \mathrm{fsign}(-) & = & 1 \end{array}$$

where $M = \mathrm{signif}(N)$ and $E = \mathrm{expon}(N)$.

---

[26] https://ieeexplore.ieee.org/document/8766229

Vectors

Numeric vectors have the same underlying representation as an $i128$. They can also be interpreted as a sequence of numeric values packed into a v128 with a particular $shape$.

$$\begin{array}{lll} \mathrm{lanes}_{t\mathsf{x}N}(c) & = & c_0~\dots~c_{N-1} \\ \quad (\mathrm{where}~B & = & |t|/8 \\ \qquad \wedge~b^{16} & = & \mathrm{bytes}_{i128}(c) \\ \qquad \wedge~c_i & = & \mathrm{bytes}_t^{-1}(b^{16}[i \cdot B : B])) \end{array}$$

These functions are bijections, so they are invertible.

Storage

When a number is stored into memory, it is converted into a sequence of bytes in little endian[27] byte order:

$$\begin{array}{lll} \mathrm{bytes}_t(i) & = & \mathrm{littleendian}(\mathrm{bits}_t(i)) \\ \mathrm{littleendian}(\epsilon) & = & \epsilon \\ \mathrm{littleendian}(d^8~{d'}^*~) & = & \mathrm{littleendian}({d'}^*)~\mathrm{ibits}_8^{-1}(d^8) \end{array}$$

Again these functions are invertible bijections.

4.3.2 Integer Operations

Sign Interpretation

Integer operators are defined on iN values. Operators that use a signed interpretation convert the value using the following definition, which takes the two's complement when the value lies in the upper half of the value range (i.e., its most significant bit is 1):

$$\begin{array}{lllll} \mathrm{signed}_N(i) & = & i & & (0 \le i < 2^{N-1}) \\ \mathrm{signed}_N(i) & = & i - 2^N & & (2^{N-1} \le i < 2^N) \end{array}$$

This function is bijective, and hence invertible.

Boolean Interpretation

The integer result of predicates – i.e., tests and relational operators – is defined with the help of the following auxiliary function producing the value 1 or 0 depending on a condition.

$$\begin{array}{llll} \mathrm{bool}(C) & = & 1 & (\mathrm{if}~C) \\ \mathrm{bool}(C) & = & 0 & (\mathrm{otherwise}) \end{array}$$

$\mathrm{iadd}_N(i_1, i_2)$

- Return the result of adding i_1 and i_2 modulo 2^N.

$$\mathrm{iadd}_N(i_1, i_2) \quad = \quad (i_1 + i_2) \bmod 2^N$$

---

[27] https://en.wikipedia.org/wiki/Endianness#Little-endian

$\mathrm{isub}_N(i_1, i_2)$

- Return the result of subtracting i_2 from i_1 modulo 2^N.

$$\mathrm{isub}_N(i_1, i_2) \quad = \quad (i_1 - i_2 + 2^N) \bmod 2^N$$

$\mathrm{imul}_N(i_1, i_2)$

- Return the result of multiplying i_1 and i_2 modulo 2^N.

$$\mathrm{imul}_N(i_1, i_2) \quad = \quad (i_1 \cdot i_2) \bmod 2^N$$

$\mathrm{idiv\_u}_N(i_1, i_2)$

- If i_2 is 0, then the result is undefined.
- Else, return the result of dividing i_1 by i_2, truncated toward zero.

$$\begin{array}{lcl} \mathrm{idiv\_u}_N(i_1, 0) & = & \{\} \\ \mathrm{idiv\_u}_N(i_1, i_2) & = & \mathrm{trunc}(i_1/i_2) \end{array}$$

Note: This operator is partial.

$\mathrm{idiv\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- If j_2 is 0, then the result is undefined.
- Else if j_1 divided by j_2 is 2^{N-1}, then the result is undefined.
- Else, return the result of dividing j_1 by j_2, truncated toward zero.

$$\begin{array}{lcll} \mathrm{idiv\_s}_N(i_1, 0) & = & \{\} & \\ \mathrm{idiv\_s}_N(i_1, i_2) & = & \{\} & (\text{if } \mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2) = 2^{N-1}) \\ \mathrm{idiv\_s}_N(i_1, i_2) & = & \mathrm{signed}_N^{-1}(\mathrm{trunc}(\mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2))) & \end{array}$$

Note: This operator is partial. Besides division by 0, the result of $(-2^{N-1})/(-1) = +2^{N-1}$ is not representable as an N-bit signed integer.

$\mathrm{irem\_u}_N(i_1, i_2)$

- If i_2 is 0, then the result is undefined.
- Else, return the remainder of dividing i_1 by i_2.

$$\begin{array}{lcl} \mathrm{irem\_u}_N(i_1, 0) & = & \{\} \\ \mathrm{irem\_u}_N(i_1, i_2) & = & i_1 - i_2 \cdot \mathrm{trunc}(i_1/i_2) \end{array}$$

Note: This operator is partial.

As long as both operators are defined, it holds that $i_1 = i_2 \cdot \mathrm{idiv\_u}(i_1, i_2) + \mathrm{irem\_u}(i_1, i_2)$.

$\mathrm{irem\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- If i_2 is 0, then the result is undefined.
- Else, return the remainder of dividing j_1 by j_2, with the sign of the dividend j_1.

$$\begin{array}{lcl}
\mathrm{irem\_s}_N(i_1, 0) &=& \{\} \\
\mathrm{irem\_s}_N(i_1, i_2) &=& \mathrm{signed}_N^{-1}(j_1 - j_2 \cdot \mathrm{trunc}(j_1/j_2)) \\
&& (\mathrm{where}~j_1 = \mathrm{signed}_N(i_1) \wedge j_2 = \mathrm{signed}_N(i_2))
\end{array}$$

Note: This operator is partial.

As long as both operators are defined, it holds that $i_1 = i_2 \cdot \mathrm{idiv\_s}(i_1, i_2) + \mathrm{irem\_s}(i_1, i_2)$.

$\mathrm{inot}_N(i)$

- Return the bitwise negation of i.

$$\mathrm{inot}_N(i) \quad = \quad \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i) \veebar \mathrm{ibits}_N(2^N - 1))$$

$\mathrm{iand}_N(i_1, i_2)$

- Return the bitwise conjunction of i_1 and i_2.

$$\mathrm{iand}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \wedge \mathrm{ibits}_N(i_2))$$

$\mathrm{iandnot}_N(i_1, i_2)$

- Return the bitwise conjunction of i_1 and the bitwise negation of i_2.

$$\mathrm{iandnot}_N(i_1, i_2) \quad = \quad \mathrm{iand}_N(i_1, \mathrm{inot}_N(i_2))$$

$\mathrm{ior}_N(i_1, i_2)$

- Return the bitwise disjunction of i_1 and i_2.

$$\mathrm{ior}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \vee \mathrm{ibits}_N(i_2))$$

$\mathrm{ixor}_N(i_1, i_2)$

- Return the bitwise exclusive disjunction of i_1 and i_2.

$$\mathrm{ixor}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \veebar \mathrm{ibits}_N(i_2))$$

$\mathrm{ishl}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of shifting i_1 left by k bits, modulo 2^N.

$$\mathrm{ishl}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(d_2^{N-k}\ 0^k) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_1^k\ d_2^{N-k} \wedge k = i_2 \bmod N)$$

$\mathrm{ishr\_u}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of shifting i_1 right by k bits, extended with 0 bits.

$$\mathrm{ishr\_u}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(0^k\ d_1^{N-k}) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_1^{N-k}\ d_2^k \wedge k = i_2 \bmod N)$$

$\mathrm{ishr\_s}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of shifting i_1 right by k bits, extended with the most significant bit of the original value.

$$\mathrm{ishr\_s}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(d_0^{k+1}\ d_1^{N-k-1}) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_0\ d_1^{N-k-1}\ d_2^k \wedge k = i_2 \bmod N)$$

$\mathrm{irotl}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of rotating i_1 left by k bits.

$$\mathrm{irotl}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(d_2^{N-k}\ d_1^k) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_1^k\ d_2^{N-k} \wedge k = i_2 \bmod N)$$

$\mathrm{irotr}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of rotating i_1 right by k bits.

$$\mathrm{irotr}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(d_2^k\ d_1^{N-k}) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_1^{N-k}\ d_2^k \wedge k = i_2 \bmod N)$$

$\mathrm{iclz}_N(i)$

- Return the count of leading zero bits in i; all bits are considered leading zeros if i is 0.

$$\mathrm{iclz}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = 0^k\ (1\ d^*)^?)$$

$\mathrm{ictz}_N(i)$

- Return the count of trailing zero bits in i; all bits are considered trailing zeros if i is 0.

$$\mathrm{ictz}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = (d^*\ 1)^?\ 0^k)$$

$\mathrm{ipopcnt}_N(i)$

- Return the count of non-zero bits in i.

$$\mathrm{ipopcnt}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = (0^*\ 1)^k\ 0^*)$$

$\mathrm{ieqz}_N(i)$

- Return 1 if i is zero, 0 otherwise.

$$\mathrm{ieqz}_N(i) \quad = \quad \mathrm{bool}(i = 0)$$

$\mathrm{ieq}_N(i_1, i_2)$

- Return 1 if i_1 equals i_2, 0 otherwise.

$$\mathrm{ieq}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 = i_2)$$

$\mathrm{ine}_N(i_1, i_2)$

- Return 1 if i_1 does not equal i_2, 0 otherwise.

$$\mathrm{ine}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \neq i_2)$$

$\mathrm{ilt\_u}_N(i_1, i_2)$

- Return 1 if i_1 is less than i_2, 0 otherwise.

$$\mathrm{ilt\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 < i_2)$$

$\mathrm{ilt\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is less than j_2, 0 otherwise.

$$\mathrm{ilt\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) < \mathrm{signed}_N(i_2))$$

$\mathrm{igt\_u}_N(i_1, i_2)$

- Return 1 if i_1 is greater than i_2, 0 otherwise.

$$\mathrm{igt\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 > i_2)$$

$\mathrm{igt\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is greater than j_2, 0 otherwise.

$$\mathrm{igt\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) > \mathrm{signed}_N(i_2))$$

$\mathrm{ile\_u}_N(i_1, i_2)$

- Return 1 if i_1 is less than or equal to i_2, 0 otherwise.

$$\mathrm{ile\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \leq i_2)$$

$\mathrm{ile\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is less than or equal to j_2, 0 otherwise.

$$\mathrm{ile\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) \leq \mathrm{signed}_N(i_2))$$

$\mathrm{ige\_u}_N(i_1, i_2)$

- Return 1 if i_1 is greater than or equal to i_2, 0 otherwise.

$$\mathrm{ige\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \geq i_2)$$

$\mathrm{ige\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is greater than or equal to j_2, 0 otherwise.

$$\mathrm{ige\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) \geq \mathrm{signed}_N(i_2))$$

$\mathrm{iextend}M\mathrm{\_s}_N(i)$

- Return $\mathrm{extend}^{\mathsf{s}}{}_{M,N}(i)$.

$$\mathrm{iextend}M\mathrm{\_s}_N(i) \quad = \quad \mathrm{extend}^{\mathsf{s}}{}_{M,N}(i)$$

$\mathrm{ibitselect}_N(i_1, i_2, i_3)$

- Let j_1 be the bitwise conjunction of i_1 and i_3.
- Let j_3' be the bitwise negation of i_3.
- Let j_2 be the bitwise conjunction of i_2 and j_3'.
- Return the bitwise disjunction of j_1 and j_2.

$$\mathrm{ibitselect}_N(i_1, i_2, i_3) \quad = \quad \mathrm{ior}_N(\mathrm{iand}_N(i_1, i_3), \mathrm{iand}_N(i_2, \mathrm{inot}_N(i_3)))$$

$\mathrm{iabs}_N(i)$

- Let j be the signed interpretation of i.
- If j is greater than or equal to 0, then return i.
- Else return the negation of j, modulo 2^N.

$$\begin{array}{lcll}
\mathrm{iabs}_N(i) & = & i & (\text{if } \mathrm{signed}_N(i) \geq 0) \\
\mathrm{iabs}_N(i) & = & -\mathrm{signed}_N(i) \bmod 2^N & (\text{otherwise})
\end{array}$$

$\mathrm{ineg}_N(i)$

- Return the result of negating i, modulo 2^N.

$$\mathrm{ineg}_N(i) \quad = \quad (2^N - i) \bmod 2^N$$

$\mathrm{imin\_u}_N(i_1, i_2)$

- Return i_1 if $\mathrm{ilt\_u}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll}
\mathrm{imin\_u}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{ilt\_u}_N(i_1, i_2) = 1) \\
\mathrm{imin\_u}_N(i_1, i_2) & = & i_2 & (\text{otherwise})
\end{array}$$

$\mathrm{imin\_s}_N(i_1, i_2)$

- Return i_1 if $\mathrm{ilt\_s}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll}
\mathrm{imin\_s}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{ilt\_s}_N(i_1, i_2) = 1) \\
\mathrm{imin\_s}_N(i_1, i_2) & = & i_2 & (\text{otherwise})
\end{array}$$

$\mathrm{imax\_u}_N(i_1, i_2)$

- Return i_1 if $\mathrm{igt\_u}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll}
\mathrm{imax\_u}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{igt\_u}_N(i_1, i_2) = 1) \\
\mathrm{imax\_u}_N(i_1, i_2) & = & i_2 & (\text{otherwise})
\end{array}$$

$\mathrm{imax\_s}_N(i_1, i_2)$

- Return i_1 if $\mathrm{igt\_s}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll}
\mathrm{imax\_s}_N(i_1, i_2) &=& i_1 & (\mathrm{if}\ \mathrm{igt\_s}_N(i_1, i_2) = 1) \\
\mathrm{imax\_s}_N(i_1, i_2) &=& i_2 & (\mathrm{otherwise})
\end{array}$$

$\mathrm{iaddsat\_u}_N(i_1, i_2)$

- Let i be the result of adding i_1 and i_2.
- Return $\mathrm{sat\_u}_N(i)$.

$$\mathrm{iaddsat\_u}_N(i_1, i_2) = \mathrm{sat\_u}_N(i_1 + i_2)$$

$\mathrm{iaddsat\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1
- Let j_2 be the signed interpretation of i_2
- Let j be the result of adding j_1 and j_2.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{iaddsat\_s}_N(i_1, i_2) = \mathrm{sat\_s}_N(\mathrm{signed}_N(i_1) + \mathrm{signed}_N(i_2))$$

$\mathrm{isubsat\_u}_N(i_1, i_2)$

- Let i be the result of subtracting i_2 from i_1.
- Return $\mathrm{sat\_u}_N(i)$.

$$\mathrm{isubsat\_u}_N(i_1, i_2) = \mathrm{sat\_u}_N(i_1 - i_2)$$

$\mathrm{isubsat\_s}_N(i_1, i_2)$

- Let j_1 be the signed interpretation of i_1
- Let j_2 be the signed interpretation of i_2
- Let j be the result of subtracting j_2 from j_1.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{isubsat\_s}_N(i_1, i_2) = \mathrm{sat\_s}_N(\mathrm{signed}_N(i_1) - \mathrm{signed}_N(i_2))$$

$\mathrm{iavgr\_u}_N(i_1, i_2)$

- Let j be the result of adding i_1, i_2, and 1.
- Return the result of dividing j by 2, truncated toward zero.

$$\mathrm{iavgr\_u}_N(i_1, i_2) = \mathrm{trunc}((i_1 + i_2 + 1)/2)$$

$\mathrm{iq15mulrsat\_s}_N(i_1, i_2)$

- Return the result of $\mathrm{sat\_s}_N(\mathrm{ishr\_s}_N(i_1 \cdot i_2 + 2^{14}, 15))$.

$$\mathrm{iq15mulrsat\_s}_N(i_1, i_2) \quad = \quad \mathrm{sat\_s}_N(\mathrm{ishr\_s}_N(i_1 \cdot i_2 + 2^{14}, 15))$$

4.3.3 Floating-Point Operations

Floating-point arithmetic follows the IEEE 754[28] standard, with the following qualifications:

- All operators use round-to-nearest ties-to-even, except where otherwise specified. Non-default directed rounding attributes are not supported.
- Following the recommendation that operators propagate NaN payloads from their operands is permitted but not required.
- All operators use "non-stop" mode, and floating-point exceptions are not otherwise observable. In particular, neither alternate floating-point exception handling attributes nor operators on status flags are supported. There is no observable difference between quiet and signalling NaNs.

Note: Some of these limitations may be lifted in future versions of WebAssembly.

Rounding

Rounding always is round-to-nearest ties-to-even, in correspondence with IEEE 754[29] (Section 4.3.1).

An *exact* floating-point number is a rational number that is exactly representable as a floating-point number of given bit width N.

A *limit* number for a given floating-point bit width N is a positive or negative number whose magnitude is the smallest power of 2 that is not exactly representable as a floating-point number of width N (that magnitude is 2^{128} for $N = 32$ and 2^{1024} for $N = 64$).

A *candidate* number is either an exact floating-point number or a positive or negative limit number for the given bit width N.

A *candidate pair* is a pair z_1, z_2 of candidate numbers, such that no candidate number exists that lies between the two.

A real number r is converted to a floating-point value of bit width N as follows:

- If r is 0, then return $+0$.
- Else if r is an exact floating-point number, then return r.
- Else if r greater than or equal to the positive limit, then return $+\infty$.
- Else if r is less than or equal to the negative limit, then return $-\infty$.
- Else if z_1 and z_2 are a candidate pair such that $z_1 < r < z_2$, then:
  - If $|r - z_1| < |r - z_2|$, then let z be z_1.
  - Else if $|r - z_1| > |r - z_2|$, then let z be z_2.
  - Else if $|r - z_1| = |r - z_2|$ and the significand of z_1 is even, then let z be z_1.
  - Else, let z be z_2.
- If z is 0, then:

---

[28] https://ieeexplore.ieee.org/document/8766229

[29] https://ieeexplore.ieee.org/document/8766229

  - If $r < 0$, then return -0.
  - Else, return $+0$.
- Else if z is a limit number, then:
  - If $r < 0$, then return $-\infty$.
  - Else, return $+\infty$.
- Else, return z.

$$\begin{array}{lcll}
\mathrm{float}_N(0) & = & +0 & \\
\mathrm{float}_N(r) & = & r & (\mathrm{if}\ r \in \mathrm{exact}_N) \\
\mathrm{float}_N(r) & = & +\infty & (\mathrm{if}\ r \geq +\mathrm{limit}_N) \\
\mathrm{float}_N(r) & = & -\infty & (\mathrm{if}\ r \leq -\mathrm{limit}_N) \\
\mathrm{float}_N(r) & = & \mathrm{closest}_N(r, z_1, z_2) & (\mathrm{if}\ z_1 < r < z_2 \wedge (z_1, z_2) \in \mathrm{candidatepair}_N) \\
\\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_1) & (\mathrm{if}\ |r - z_1| < |r - z_2|) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_2) & (\mathrm{if}\ |r - z_1| > |r - z_2|) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_1) & (\mathrm{if}\ |r - z_1| = |r - z_2| \wedge \mathrm{even}_N(z_1)) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_2) & (\mathrm{if}\ |r - z_1| = |r - z_2| \wedge \mathrm{even}_N(z_2))
\end{array}$$

$$\begin{array}{lcll}
\mathrm{rectify}_N(r, \pm\mathrm{limit}_N) & = & \pm\infty & \\
\mathrm{rectify}_N(r, 0) & = & +0 & (r \geq 0) \\
\mathrm{rectify}_N(r, 0) & = & -0 & (r < 0) \\
\mathrm{rectify}_N(r, z) & = & z &
\end{array}$$

where:

$$\begin{array}{lcl}
\mathrm{exact}_N & = & fN \cap \mathbb{Q} \\
\mathrm{limit}_N & = & 2^{2^{\mathrm{expon}(N)-1}} \\
\mathrm{candidate}_N & = & \mathrm{exact}_N \cup \{+\mathrm{limit}_N, -\mathrm{limit}_N\} \\
\mathrm{candidatepair}_N & = & \{(z_1, z_2) \in \mathrm{candidate}_N^2 \mid z_1 < z_2 \wedge \forall z \in \mathrm{candidate}_N, z \leq z_1 \vee z \geq z_2\} \\
\mathrm{even}_N((d + m \cdot 2^{-M}) \cdot 2^e) & \Leftrightarrow & m \bmod 2 = 0 \\
\mathrm{even}_N(\pm\mathrm{limit}_N) & \Leftrightarrow & \mathrm{true}
\end{array}$$

NaN Propagation

When the result of a floating-point operator other than fneg, fabs, or fcopysign is a NaN, then its sign is non-deterministic and the payload is computed as follows:

- If the payload of all NaN inputs to the operator is canonical (including the case that there are no NaN inputs), then the payload of the output is canonical as well.
- Otherwise the payload is picked non-deterministically among all arithmetic NaNs; that is, its most significant bit is 1 and all others are unspecified.

This non-deterministic result is expressed by the following auxiliary function producing a set of allowed outputs from a set of inputs:

$$\begin{array}{lcll}
\mathrm{nans}_N\{z^*\} & = & \{+\mathsf{nan}(n), -\mathsf{nan}(n) \mid n = \mathrm{canon}_N\} & (\mathrm{if}\ \forall \mathsf{nan}(n) \in z^*,\ n = \mathrm{canon}_N) \\
\mathrm{nans}_N\{z^*\} & = & \{+\mathsf{nan}(n), -\mathsf{nan}(n) \mid n \geq \mathrm{canon}_N\} & (\mathrm{otherwise})
\end{array}$$

$\mathrm{fadd}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities of opposite signs, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return that infinity.
- Else if either z_1 or z_2 is an infinity, then return that infinity.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of equal sign, then return that zero.
- Else if either z_1 or z_2 is a zero, then return the other operand.
- Else if both z_1 and z_2 are values with the same magnitude but opposite signs, then return positive zero.
- Else return the result of adding z_1 and z_2, rounded to the nearest representable value.

$$\begin{array}{lcl}
\mathrm{fadd}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fadd}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fadd}_N(\pm\infty, \mp\infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fadd}_N(\pm\infty, \pm\infty) & = & \pm\infty \\
\mathrm{fadd}_N(z_1, \pm\infty) & = & \pm\infty \\
\mathrm{fadd}_N(\pm\infty, z_2) & = & \pm\infty \\
\mathrm{fadd}_N(\pm 0, \mp 0) & = & +0 \\
\mathrm{fadd}_N(\pm 0, \pm 0) & = & \pm 0 \\
\mathrm{fadd}_N(z_1, \pm 0) & = & z_1 \\
\mathrm{fadd}_N(\pm 0, z_2) & = & z_2 \\
\mathrm{fadd}_N(\pm q, \mp q) & = & +0 \\
\mathrm{fadd}_N(z_1, z_2) & = & \mathrm{float}_N(z_1 + z_2)
\end{array}$$

$\mathrm{fsub}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities of equal signs, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of opposite sign, then return z_1.
- Else if z_1 is an infinity, then return that infinity.
- Else if z_2 is an infinity, then return that infinity negated.
- Else if both z_1 and z_2 are zeroes of equal sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return z_1.
- Else if z_2 is a zero, then return z_1.
- Else if z_1 is a zero, then return z_2 negated.
- Else if both z_1 and z_2 are the same value, then return positive zero.
- Else return the result of subtracting z_2 from z_1, rounded to the nearest representable value.

$$\begin{array}{lll}
\mathrm{fsub}_N(\pm\mathsf{nan}(n), z_2) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fsub}_N(z_1, \pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fsub}_N(\pm\infty, \pm\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fsub}_N(\pm\infty, \mp\infty) &=& \pm\infty \\
\mathrm{fsub}_N(z_1, \pm\infty) &=& \mp\infty \\
\mathrm{fsub}_N(\pm\infty, z_2) &=& \pm\infty \\
\mathrm{fsub}_N(\pm 0, \pm 0) &=& +0 \\
\mathrm{fsub}_N(\pm 0, \mp 0) &=& \pm 0 \\
\mathrm{fsub}_N(z_1, \pm 0) &=& z_1 \\
\mathrm{fsub}_N(\pm 0, \pm q_2) &=& \mp q_2 \\
\mathrm{fsub}_N(\pm q, \pm q) &=& +0 \\
\mathrm{fsub}_N(z_1, z_2) &=& \mathrm{float}_N(z_1 - z_2)
\end{array}$$

Note: Up to the non-determinism regarding NaNs, it always holds that $\mathrm{fsub}_N(z_1, z_2) = \mathrm{fadd}_N(z_1, \mathrm{fneg}_N(z_2))$.

$\mathrm{fmul}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if one of z_1 and z_2 is a zero and the other an infinity, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return positive infinity.
- Else if both z_1 and z_2 are infinities of opposite sign, then return negative infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with equal sign, then return positive infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with opposite sign, then return negative infinity.
- Else if both z_1 and z_2 are zeroes of equal sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return negative zero.
- Else return the result of multiplying z_1 and z_2, rounded to the nearest representable value.

$$\begin{array}{lll}
\mathrm{fmul}_N(\pm\mathsf{nan}(n), z_2) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fmul}_N(z_1, \pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fmul}_N(\pm\infty, \pm 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm\infty, \mp 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm 0, \pm\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm 0, \mp\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm\infty, \pm\infty) &=& +\infty \\
\mathrm{fmul}_N(\pm\infty, \mp\infty) &=& -\infty \\
\mathrm{fmul}_N(\pm q_1, \pm\infty) &=& +\infty \\
\mathrm{fmul}_N(\pm q_1, \mp\infty) &=& -\infty \\
\mathrm{fmul}_N(\pm\infty, \pm q_2) &=& +\infty \\
\mathrm{fmul}_N(\pm\infty, \mp q_2) &=& -\infty \\
\mathrm{fmul}_N(\pm 0, \pm 0) &=& +0 \\
\mathrm{fmul}_N(\pm 0, \mp 0) &=& -0 \\
\mathrm{fmul}_N(z_1, z_2) &=& \mathrm{float}_N(z_1 \cdot z_2)
\end{array}$$

$\mathrm{fdiv}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are zeroes, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if z_1 is an infinity and z_2 a value with equal sign, then return positive infinity.
- Else if z_1 is an infinity and z_2 a value with opposite sign, then return negative infinity.
- Else if z_2 is an infinity and z_1 a value with equal sign, then return positive zero.
- Else if z_2 is an infinity and z_1 a value with opposite sign, then return negative zero.
- Else if z_1 is a zero and z_2 a value with equal sign, then return positive zero.
- Else if z_1 is a zero and z_2 a value with opposite sign, then return negative zero.
- Else if z_2 is a zero and z_1 a value with equal sign, then return positive infinity.
- Else if z_2 is a zero and z_1 a value with opposite sign, then return negative infinity.
- Else return the result of dividing z_1 by z_2, rounded to the nearest representable value.

$$\begin{array}{lcl}
\mathrm{fdiv}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fdiv}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fdiv}_N(\pm\infty, \pm\infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm\infty, \mp\infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm 0, \pm 0) & = & \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm 0, \mp 0) & = & \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm\infty, \pm q_2) & = & +\infty \\
\mathrm{fdiv}_N(\pm\infty, \mp q_2) & = & -\infty \\
\mathrm{fdiv}_N(\pm q_1, \pm\infty) & = & +0 \\
\mathrm{fdiv}_N(\pm q_1, \mp\infty) & = & -0 \\
\mathrm{fdiv}_N(\pm 0, \pm q_2) & = & +0 \\
\mathrm{fdiv}_N(\pm 0, \mp q_2) & = & -0 \\
\mathrm{fdiv}_N(\pm q_1, \pm 0) & = & +\infty \\
\mathrm{fdiv}_N(\pm q_1, \mp 0) & = & -\infty \\
\mathrm{fdiv}_N(z_1, z_2) & = & \mathrm{float}_N(z_1 / z_2)
\end{array}$$

$\mathrm{fmin}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if either z_1 or z_2 is a negative infinity, then return negative infinity.
- Else if either z_1 or z_2 is a positive infinity, then return the other value.
- Else if both z_1 and z_2 are zeroes of opposite signs, then return negative zero.
- Else return the smaller value of z_1 and z_2.

$$\begin{array}{lcll}
\mathrm{fmin}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} & \\
\mathrm{fmin}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} & \\
\mathrm{fmin}_N(+\infty, z_2) & = & z_2 & \\
\mathrm{fmin}_N(-\infty, z_2) & = & -\infty & \\
\mathrm{fmin}_N(z_1, +\infty) & = & z_1 & \\
\mathrm{fmin}_N(z_1, -\infty) & = & -\infty & \\
\mathrm{fmin}_N(\pm 0, \mp 0) & = & -0 & \\
\mathrm{fmin}_N(z_1, z_2) & = & z_1 & (\text{if } z_1 \leq z_2) \\
\mathrm{fmin}_N(z_1, z_2) & = & z_2 & (\text{if } z_2 \leq z_1)
\end{array}$$

$\mathrm{fmax}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if either z_1 or z_2 is a positive infinity, then return positive infinity.
- Else if either z_1 or z_2 is a negative infinity, then return the other value.
- Else if both z_1 and z_2 are zeroes of opposite signs, then return positive zero.
- Else return the larger value of z_1 and z_2.

$$\begin{array}{lcll}
\mathrm{fmax}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fmax}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fmax}_N(+\infty, z_2) & = & +\infty \\
\mathrm{fmax}_N(-\infty, z_2) & = & z_2 \\
\mathrm{fmax}_N(z_1, +\infty) & = & +\infty \\
\mathrm{fmax}_N(z_1, -\infty) & = & z_1 \\
\mathrm{fmax}_N(\pm 0, \mp 0) & = & +0 \\
\mathrm{fmax}_N(z_1, z_2) & = & z_1 & (\mathrm{if}\ z_1 \geq z_2) \\
\mathrm{fmax}_N(z_1, z_2) & = & z_2 & (\mathrm{if}\ z_2 \geq z_1)
\end{array}$$

$\mathrm{fcopysign}_N(z_1, z_2)$

- If z_1 and z_2 have the same sign, then return z_1.
- Else return z_1 with negated sign.

$$\begin{array}{lcl}
\mathrm{fcopysign}_N(\pm p_1, \pm p_2) & = & \pm p_1 \\
\mathrm{fcopysign}_N(\pm p_1, \mp p_2) & = & \mp p_1
\end{array}$$

$\mathrm{fabs}_N(z)$

- If z is a NaN, then return z with positive sign.
- Else if z is an infinity, then return positive infinity.
- Else if z is a zero, then return positive zero.
- Else if z is a positive value, then z.
- Else return z negated.

$$\begin{array}{lcl}
\mathrm{fabs}_N(\pm\mathsf{nan}(n)) & = & +\mathsf{nan}(n) \\
\mathrm{fabs}_N(\pm\infty) & = & +\infty \\
\mathrm{fabs}_N(\pm 0) & = & +0 \\
\mathrm{fabs}_N(\pm q) & = & +q
\end{array}$$

$\mathrm{fneg}_N(z)$

- If z is a NaN, then return z with negated sign.
- Else if z is an infinity, then return that infinity negated.
- Else if z is a zero, then return that zero negated.
- Else return z negated.

$$\begin{array}{lcl}
\mathrm{fneg}_N(\pm\mathsf{nan}(n)) & = & \mp\mathsf{nan}(n) \\
\mathrm{fneg}_N(\pm\infty) & = & \mp\infty \\
\mathrm{fneg}_N(\pm 0) & = & \mp 0 \\
\mathrm{fneg}_N(\pm q) & = & \mp q
\end{array}$$

$\mathrm{fsqrt}_N(z)$

- If z is a NaN, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is negative infinity, then return an element of $\mathrm{nans}_N\{\}$.
- Else if z is positive infinity, then return positive infinity.
- Else if z is a zero, then return that zero.
- Else if z has a negative sign, then return an element of $\mathrm{nans}_N\{\}$.
- Else return the square root of z.

$$\begin{array}{lcl}
\mathrm{fsqrt}_N(\pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} \\
\mathrm{fsqrt}_N(-\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fsqrt}_N(+\infty) &=& +\infty \\
\mathrm{fsqrt}_N(\pm 0) &=& \pm 0 \\
\mathrm{fsqrt}_N(-q) &=& \mathrm{nans}_N\{\} \\
\mathrm{fsqrt}_N(+q) &=& \mathrm{float}_N\left(\sqrt{q}\right)
\end{array}$$

$\mathrm{fceil}_N(z)$

- If z is a NaN, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is an infinity, then return z.
- Else if z is a zero, then return z.
- Else if z is smaller than 0 but greater than -1, then return negative zero.
- Else return the smallest integral value that is not smaller than z.

$$\begin{array}{lcll}
\mathrm{fceil}_N(\pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} & \\
\mathrm{fceil}_N(\pm\infty) &=& \pm\infty & \\
\mathrm{fceil}_N(\pm 0) &=& \pm 0 & \\
\mathrm{fceil}_N(-q) &=& -0 & (\mathrm{if}\ -1 < -q < 0) \\
\mathrm{fceil}_N(\pm q) &=& \mathrm{float}_N(i) & (\mathrm{if}\ \pm q \leq i < \pm q + 1)
\end{array}$$

$\mathrm{ffloor}_N(z)$

- If z is a NaN, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is an infinity, then return z.
- Else if z is a zero, then return z.
- Else if z is greater than 0 but smaller than 1, then return positive zero.
- Else return the largest integral value that is not larger than z.

$$\begin{array}{lcll}
\mathrm{ffloor}_N(\pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} & \\
\mathrm{ffloor}_N(\pm\infty) &=& \pm\infty & \\
\mathrm{ffloor}_N(\pm 0) &=& \pm 0 & \\
\mathrm{ffloor}_N(+q) &=& +0 & (\mathrm{if}\ 0 < +q < 1) \\
\mathrm{ffloor}_N(\pm q) &=& \mathrm{float}_N(i) & (\mathrm{if}\ \pm q - 1 < i \leq \pm q)
\end{array}$$

$\mathrm{ftrunc}_N(z)$

- If z is a NaN, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is an infinity, then return z.
- Else if z is a zero, then return z.
- Else if z is greater than 0 but smaller than 1, then return positive zero.
- Else if z is smaller than 0 but greater than -1, then return negative zero.
- Else return the integral value with the same sign as z and the largest magnitude that is not larger than the magnitude of z.

$$\begin{array}{lcll}
\mathrm{ftrunc}_N(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} & \\
\mathrm{ftrunc}_N(\pm\infty) & = & \pm\infty & \\
\mathrm{ftrunc}_N(\pm 0) & = & \pm 0 & \\
\mathrm{ftrunc}_N(+q) & = & +0 & (\mathrm{if}\ 0 < +q < 1) \\
\mathrm{ftrunc}_N(-q) & = & -0 & (\mathrm{if}\ -1 < -q < 0) \\
\mathrm{ftrunc}_N(\pm q) & = & \mathrm{float}_N(\pm i) & (\mathrm{if}\ +q-1 < i \leq +q)
\end{array}$$

$\mathrm{fnearest}_N(z)$

- If z is a NaN, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is an infinity, then return z.
- Else if z is a zero, then return z.
- Else if z is greater than 0 but smaller than or equal to 0.5, then return positive zero.
- Else if z is smaller than 0 but greater than or equal to -0.5, then return negative zero.
- Else return the integral value that is nearest to z; if two values are equally near, return the even one.

$$\begin{array}{lcll}
\mathrm{fnearest}_N(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} & \\
\mathrm{fnearest}_N(\pm\infty) & = & \pm\infty & \\
\mathrm{fnearest}_N(\pm 0) & = & \pm 0 & \\
\mathrm{fnearest}_N(+q) & = & +0 & (\mathrm{if}\ 0 < +q \leq 0.5) \\
\mathrm{fnearest}_N(-q) & = & -0 & (\mathrm{if}\ -0.5 \leq -q < 0) \\
\mathrm{fnearest}_N(\pm q) & = & \mathrm{float}_N(\pm i) & (\mathrm{if}\ |i-q| < 0.5) \\
\mathrm{fnearest}_N(\pm q) & = & \mathrm{float}_N(\pm i) & (\mathrm{if}\ |i-q| = 0.5 \wedge i\ \mathrm{even})
\end{array}$$

$\mathrm{feq}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 0.
- Else if both z_1 and z_2 are zeroes, then return 1.
- Else if both z_1 and z_2 are the same value, then return 1.
- Else return 0.

$$\begin{array}{lcl}
\mathrm{feq}_N(\pm\mathsf{nan}(n), z_2) & = & 0 \\
\mathrm{feq}_N(z_1, \pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{feq}_N(\pm 0, \mp 0) & = & 1 \\
\mathrm{feq}_N(z_1, z_2) & = & \mathrm{bool}(z_1 = z_2)
\end{array}$$

$\mathrm{fne}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 1.
- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if both z_1 and z_2 are the same value, then return 0.
- Else return 1.

$$\begin{array}{lcl}
\mathrm{fne}_N(\pm\mathsf{nan}(n), z_2) & = & 1 \\
\mathrm{fne}_N(z_1, \pm\mathsf{nan}(n)) & = & 1 \\
\mathrm{fne}_N(\pm 0, \mp 0) & = & 0 \\
\mathrm{fne}_N(z_1, z_2) & = & \mathrm{bool}(z_1 \neq z_2)
\end{array}$$

$\mathrm{flt}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 0.
- Else if z_1 is positive infinity, then return 0.
- Else if z_1 is negative infinity, then return 1.
- Else if z_2 is positive infinity, then return 1.
- Else if z_2 is negative infinity, then return 0.
- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if z_1 is smaller than z_2, then return 1.
- Else return 0.

$$\begin{array}{lcl}
\mathrm{flt}_N(\pm\mathsf{nan}(n), z_2) & = & 0 \\
\mathrm{flt}_N(z_1, \pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{flt}_N(z, z) & = & 0 \\
\mathrm{flt}_N(+\infty, z_2) & = & 0 \\
\mathrm{flt}_N(-\infty, z_2) & = & 1 \\
\mathrm{flt}_N(z_1, +\infty) & = & 1 \\
\mathrm{flt}_N(z_1, -\infty) & = & 0 \\
\mathrm{flt}_N(\pm 0, \mp 0) & = & 0 \\
\mathrm{flt}_N(z_1, z_2) & = & \mathrm{bool}(z_1 < z_2)
\end{array}$$

$\mathrm{fgt}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 0.
- Else if z_1 is positive infinity, then return 1.
- Else if z_1 is negative infinity, then return 0.
- Else if z_2 is positive infinity, then return 0.
- Else if z_2 is negative infinity, then return 1.
- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if z_1 is larger than z_2, then return 1.
- Else return 0.

$$
\begin{array}{lll}
\mathrm{fgt}_N(\pm\mathsf{nan}(n), z_2) &=& 0 \\
\mathrm{fgt}_N(z_1, \pm\mathsf{nan}(n)) &=& 0 \\
\mathrm{fgt}_N(z, z) &=& 0 \\
\mathrm{fgt}_N(+\infty, z_2) &=& 1 \\
\mathrm{fgt}_N(-\infty, z_2) &=& 0 \\
\mathrm{fgt}_N(z_1, +\infty) &=& 0 \\
\mathrm{fgt}_N(z_1, -\infty) &=& 1 \\
\mathrm{fgt}_N(\pm 0, \mp 0) &=& 0 \\
\mathrm{fgt}_N(z_1, z_2) &=& \mathrm{bool}(z_1 > z_2)
\end{array}
$$

$\mathrm{fle}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 1.
- Else if z_1 is positive infinity, then return 0.
- Else if z_1 is negative infinity, then return 1.
- Else if z_2 is positive infinity, then return 1.
- Else if z_2 is negative infinity, then return 0.
- Else if both z_1 and z_2 are zeroes, then return 1.
- Else if z_1 is smaller than or equal to z_2, then return 1.
- Else return 0.

$$
\begin{array}{lll}
\mathrm{fle}_N(\pm\mathsf{nan}(n), z_2) &=& 0 \\
\mathrm{fle}_N(z_1, \pm\mathsf{nan}(n)) &=& 0 \\
\mathrm{fle}_N(z, z) &=& 1 \\
\mathrm{fle}_N(+\infty, z_2) &=& 0 \\
\mathrm{fle}_N(-\infty, z_2) &=& 1 \\
\mathrm{fle}_N(z_1, +\infty) &=& 1 \\
\mathrm{fle}_N(z_1, -\infty) &=& 0 \\
\mathrm{fle}_N(\pm 0, \mp 0) &=& 1 \\
\mathrm{fle}_N(z_1, z_2) &=& \mathrm{bool}(z_1 \leq z_2)
\end{array}
$$

$\mathrm{fge}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 1.
- Else if z_1 is positive infinity, then return 1.
- Else if z_1 is negative infinity, then return 0.
- Else if z_2 is positive infinity, then return 0.
- Else if z_2 is negative infinity, then return 1.
- Else if both z_1 and z_2 are zeroes, then return 1.
- Else if z_1 is smaller than or equal to z_2, then return 1.
- Else return 0.

$$\begin{array}{lcl}
\mathrm{fge}_N(\pm\mathsf{nan}(n), z_2) &=& 0 \\
\mathrm{fge}_N(z_1, \pm\mathsf{nan}(n)) &=& 0 \\
\mathrm{fge}_N(z, z) &=& 1 \\
\mathrm{fge}_N(+\infty, z_2) &=& 1 \\
\mathrm{fge}_N(-\infty, z_2) &=& 0 \\
\mathrm{fge}_N(z_1, +\infty) &=& 0 \\
\mathrm{fge}_N(z_1, -\infty) &=& 1 \\
\mathrm{fge}_N(\pm 0, \mp 0) &=& 1 \\
\mathrm{fge}_N(z_1, z_2) &=& \mathrm{bool}(z_1 \geq z_2)
\end{array}$$

$\mathrm{fpmin}_N(z_1, z_2)$

- If z_2 is less than z_1 then return z_2.
- Else return z_1.

$$\begin{array}{lcll}
\mathrm{fpmin}_N(z_1, z_2) &=& z_2 & (\text{if } \mathrm{flt}_N(z_2, z_1) = 1) \\
\mathrm{fpmin}_N(z_1, z_2) &=& z_1 & (\text{otherwise})
\end{array}$$

$\mathrm{fpmax}_N(z_1, z_2)$

- If z_1 is less than z_2 then return z_2.
- Else return z_1.

$$\begin{array}{lcll}
\mathrm{fpmax}_N(z_1, z_2) &=& z_2 & (\text{if } \mathrm{flt}_N(z_1, z_2) = 1) \\
\mathrm{fpmax}_N(z_1, z_2) &=& z_1 & (\text{otherwise})
\end{array}$$

4.3.4 Conversions

$\mathrm{extend}^{\mathsf{u}}{}_{M,N}(i)$

- Return i.

$$\mathrm{extend}^{\mathsf{u}}{}_{M,N}(i) \quad = \quad i$$

Note: In the abstract syntax, unsigned extension just reinterprets the same value.

$\mathrm{extend}^{\mathsf{s}}{}_{M,N}(i)$

- Let j be the signed interpretation of i of size M.
- Return the two's complement of j relative to size N.

$$\mathrm{extend}^{\mathsf{s}}{}_{M,N}(i) \quad = \quad \mathrm{signed}_N^{-1}(\mathrm{signed}_M(i))$$

$\mathrm{wrap}_{M,N}(i)$

- Return i modulo 2^N.

$$\mathrm{wrap}_{M,N}(i) \quad = \quad i \bmod 2^N$$

$\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(z)$

- If z is a NaN, then the result is undefined.
- Else if z is an infinity, then the result is undefined.
- Else if z is a number and $\mathrm{trunc}(z)$ is a value within range of the target type, then return that value.
- Else the result is undefined.

$$\begin{array}{lcll}
\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(\pm \mathsf{nan}(n)) & = & \{\} & \\
\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(\pm \infty) & = & \{\} & \\
\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(\pm q) & = & \mathrm{trunc}(\pm q) & (\mathrm{if}\ -1 < \mathrm{trunc}(\pm q) < 2^N) \\
\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(\pm q) & = & \{\} & (\mathrm{otherwise})
\end{array}$$

Note: This operator is partial. It is not defined for NaNs, infinities, or values for which the result is out of range.

$\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(z)$

- If z is a NaN, then the result is undefined.
- Else if z is an infinity, then the result is undefined.
- If z is a number and $\mathrm{trunc}(z)$ is a value within range of the target type, then return that value.
- Else the result is undefined.

$$\begin{array}{lcll}
\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(\pm \mathsf{nan}(n)) & = & \{\} & \\
\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(\pm \infty) & = & \{\} & \\
\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(\pm q) & = & \mathrm{trunc}(\pm q) & (\mathrm{if}\ -2^{N-1} - 1 < \mathrm{trunc}(\pm q) < 2^{N-1}) \\
\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(\pm q) & = & \{\} & (\mathrm{otherwise})
\end{array}$$

Note: This operator is partial. It is not defined for NaNs, infinities, or values for which the result is out of range.

$\mathrm{trunc\_sat\_u}_{M,N}(z)$

- If z is a NaN, then return 0.
- Else if z is negative infinity, then return 0.
- Else if z is positive infinity, then return $2^N - 1$.
- Else, return $\mathrm{sat\_u}_N(\mathrm{trunc}(z))$.

$$\begin{array}{lcl}
\mathrm{trunc\_sat\_u}_{M,N}(\pm \mathsf{nan}(n)) & = & 0 \\
\mathrm{trunc\_sat\_u}_{M,N}(-\infty) & = & 0 \\
\mathrm{trunc\_sat\_u}_{M,N}(+\infty) & = & 2^N - 1 \\
\mathrm{trunc\_sat\_u}_{M,N}(z) & = & \mathrm{sat\_u}_N(\mathrm{trunc}(z))
\end{array}$$

$\mathrm{trunc\_sat\_s}_{M,N}(z)$

- If z is a NaN, then return 0.
- Else if z is negative infinity, then return -2^{N-1}.
- Else if z is positive infinity, then return $2^{N-1} - 1$.
- Else, return $\mathrm{sat\_s}_N(\mathrm{trunc}(z))$.

$$\begin{array}{lcl}
\mathrm{trunc\_sat\_s}_{M,N}(\pm \mathsf{nan}(n)) & = & 0 \\
\mathrm{trunc\_sat\_s}_{M,N}(-\infty) & = & -2^{N-1} \\
\mathrm{trunc\_sat\_s}_{M,N}(+\infty) & = & 2^{N-1} - 1 \\
\mathrm{trunc\_sat\_s}_{M,N}(z) & = & \mathrm{sat\_s}_N(\mathrm{trunc}(z))
\end{array}$$

$\mathrm{promote}_{M,N}(z)$

- If z is a canonical NaN, then return an element of $\mathrm{nans}_N\{\}$ (i.e., a canonical NaN of size N).
- Else if z is a NaN, then return an element of $\mathrm{nans}_N\{\pm \mathsf{nan}(1)\}$ (i.e., any arithmetic NaN of size N).
- Else, return z.

$$\begin{array}{lcll}
\mathrm{promote}_{M,N}(\pm \mathsf{nan}(n)) & = & \mathrm{nans}_N\{\} & (\text{if } n = \mathrm{canon}_N) \\
\mathrm{promote}_{M,N}(\pm \mathsf{nan}(n)) & = & \mathrm{nans}_N\{+\mathsf{nan}(1)\} & (\text{otherwise}) \\
\mathrm{promote}_{M,N}(z) & = & z &
\end{array}$$

$\mathrm{demote}_{M,N}(z)$

- If z is a canonical NaN, then return an element of $\mathrm{nans}_N\{\}$ (i.e., a canonical NaN of size N).
- Else if z is a NaN, then return an element of $\mathrm{nans}_N\{\pm \mathsf{nan}(1)\}$ (i.e., any NaN of size N).
- Else if z is an infinity, then return that infinity.
- Else if z is a zero, then return that zero.
- Else, return $\mathrm{float}_N(z)$.

$$\begin{array}{lcll}
\mathrm{demote}_{M,N}(\pm \mathsf{nan}(n)) & = & \mathrm{nans}_N\{\} & (\text{if } n = \mathrm{canon}_N) \\
\mathrm{demote}_{M,N}(\pm \mathsf{nan}(n)) & = & \mathrm{nans}_N\{+\mathsf{nan}(1)\} & (\text{otherwise}) \\
\mathrm{demote}_{M,N}(\pm \infty) & = & \pm \infty & \\
\mathrm{demote}_{M,N}(\pm 0) & = & \pm 0 & \\
\mathrm{demote}_{M,N}(\pm q) & = & \mathrm{float}_N(\pm q) &
\end{array}$$

$\mathrm{convert}^{\mathsf{u}}{}_{M,N}(i)$

- Return $\mathrm{float}_N(i)$.

$$\mathrm{convert}^{\mathsf{u}}{}_{M,N}(i) \quad = \quad \mathrm{float}_N(i)$$

$\mathrm{convert}^{\mathsf{s}}{}_{M,N}(i)$

- Let j be the signed interpretation of i.
- Return $\mathrm{float}_N(j)$.

$$\mathrm{convert}^{\mathsf{s}}{}_{M,N}(i) \quad = \quad \mathrm{float}_N(\mathrm{signed}_M(i))$$

$\mathrm{reinterpret}_{t_1,t_2}(c)$

- Let d^* be the bit sequence $\mathrm{bits}_{t_1}(c)$.
- Return the constant c' for which $\mathrm{bits}_{t_2}(c') = d^*$.

$$\mathrm{reinterpret}_{t_1,t_2}(c) \quad = \quad \mathrm{bits}_{t_2}^{-1}(\mathrm{bits}_{t_1}(c))$$

$\mathrm{narrow}^{\mathsf{s}}{}_{M,N}(i)$

- Let j be the signed interpretation of i of size M.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{narrow}^{\mathsf{s}}{}_{M,N}(i) \quad = \quad \mathrm{sat\_s}_N(\mathrm{signed}_M(i))$$

$\mathrm{narrow}^{\mathsf{u}}{}_{M,N}(i)$

- Let j be the signed interpretation of i of size M.
- Return $\mathrm{sat\_u}_N(j)$.

$$\mathrm{narrow}^{\mathsf{u}}{}_{M,N}(i) \quad = \quad \mathrm{sat\_u}_N(\mathrm{signed}_M(i))$$

4.4 Instructions

WebAssembly computation is performed by executing individual instructions.

4.4.1 Numeric Instructions

Numeric instructions are defined in terms of the generic numeric operators. The mapping of numeric instructions to their underlying operators is expressed by the following definition:

$$\begin{array}{lcl} op_{\mathsf{i}N}(i_1,\dots,i_k) & = & \mathrm{i}op_N(i_1,\dots,i_k) \\ op_{\mathsf{f}N}(z_1,\dots,z_k) & = & \mathrm{f}op_N(z_1,\dots,z_k) \\ op_{\mathsf{v}N}(i_1,\dots,i_k) & = & \mathrm{i}op_N(i_1,\dots,i_k) \end{array}$$

And for conversion operators:

$$cvtop^{sx^?}_{t_1,t_2}(c) \quad = \quad cvtop^{sx^?}_{|t_1|,|t_2|}(c)$$

Where the underlying operators are partial, the corresponding instruction will trap when the result is not defined. Where the underlying operators are non-deterministic, because they may return one of multiple possible NaN values, so are the corresponding instructions.

Note: For example, the result of instruction `i32.add` applied to operands i_1, i_2 invokes $\mathsf{add}_{\mathsf{i32}}(i_1, i_2)$, which maps to the generic $\mathrm{iadd}_{32}(i_1, i_2)$ via the above definition. Similarly, `i64.trunc_f32_s` applied to z invokes $\mathsf{trunc}^{\mathsf{s}}_{\mathsf{f32},\mathsf{i64}}(z)$, which maps to the generic $\mathrm{trunc}^{\mathsf{s}}{}_{32,64}(z)$.

$t.\mathsf{const}\ c$

1. Push the value $t.\mathsf{const}\ c$ to the stack.

Note: No formal reduction rule is required for this instruction, since const instructions already are values.

$t.\mathit{unop}$

1. Assert: due to validation, a value of value type t is on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
3. If $\mathit{unop}_t(c_1)$ is defined, then:
   a. Let c be a possible result of computing $\mathit{unop}_t(c_1)$.
   b. Push the value $t.\mathsf{const}\ c$ to the stack.
4. Else:
   a. Trap.

$$\begin{array}{lcll}
(t.\mathsf{const}\ c_1)\ t.\mathit{unop} & \hookrightarrow & (t.\mathsf{const}\ c) & (\mathrm{if}\ c \in \mathit{unop}_t(c_1)) \\
(t.\mathsf{const}\ c_1)\ t.\mathit{unop} & \hookrightarrow & \mathsf{trap} & (\mathrm{if}\ \mathit{unop}_t(c_1) = \{\})
\end{array}$$

$t.\mathit{binop}$

1. Assert: due to validation, two values of value type t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. If $\mathit{binop}_t(c_1, c_2)$ is defined, then:
   a. Let c be a possible result of computing $\mathit{binop}_t(c_1, c_2)$.
   b. Push the value $t.\mathsf{const}\ c$ to the stack.
5. Else:
   a. Trap.

$$\begin{array}{lcll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & (t.\mathsf{const}\ c) & (\mathrm{if}\ c \in \mathit{binop}_t(c_1, c_2)) \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & \mathsf{trap} & (\mathrm{if}\ \mathit{binop}_t(c_1, c_2) = \{\})
\end{array}$$

$t.\mathit{testop}$

1. Assert: due to validation, a value of value type t is on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
3. Let c be the result of computing $\mathit{testop}_t(c_1)$.
4. Push the value $\mathsf{i32.const}\ c$ to the stack.

$$(t.\mathsf{const}\ c_1)\ t.\mathit{testop} \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad (\mathrm{if}\ c = \mathit{testop}_t(c_1))$$

$t.\mathit{relop}$

1. Assert: due to validation, two values of value type t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. Let c be the result of computing $\mathit{relop}_t(c_1, c_2)$.
5. Push the value $\mathsf{i32.const}\ c$ to the stack.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{relop} \quad\hookrightarrow\quad (\mathsf{i32.const}\ c) \qquad (\text{if } c = \mathit{relop}_t(c_1, c_2))$$

$t_2.\mathit{cvtop}\_t_1\_\mathit{sx}^?$

1. Assert: due to validation, a value of value type t_1 is on the top of the stack.
2. Pop the value $t_1.\mathsf{const}\ c_1$ from the stack.
3. If $\mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c_1)$ is defined:
    a. Let c_2 be a possible result of computing $\mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c_1)$.
    b. Push the value $t_2.\mathsf{const}\ c_2$ to the stack.
4. Else:
    a. Trap.

$$\begin{array}{lcll}
(t_1.\mathsf{const}\ c_1)\ t_2.\mathit{cvtop}\_t_1\_\mathit{sx}^? & \hookrightarrow & (t_2.\mathsf{const}\ c_2) & (\text{if } c_2 \in \mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c_1)) \\
(t_1.\mathsf{const}\ c_1)\ t_2.\mathit{cvtop}\_t_1\_\mathit{sx}^? & \hookrightarrow & \mathsf{trap} & (\text{if } \mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c_1) = \{\})
\end{array}$$

4.4.2 Reference Instructions

$\mathsf{ref.null}\ t$

1. Push the value $\mathsf{ref.null}\ t$ to the stack.

Note: No formal reduction rule is required for this instruction, since the ref.null instruction is already a value.

$\mathsf{ref.is\_null}$

1. Assert: due to validation, a reference value is on the top of the stack.
2. Pop the value val from the stack.
3. If val is $\mathsf{ref.null}\ t$, then:
    a. Push the value $\mathsf{i32.const}\ 1$ to the stack.
4. Else:
    a. Push the value $\mathsf{i32.const}\ 0$ to the stack.

$$\begin{array}{lcll}
\mathit{val}\ \mathsf{ref.is\_null} & \hookrightarrow & (\mathsf{i32.const}\ 1) & (\text{if } \mathit{val} = \mathsf{ref.null}\ t) \\
\mathit{val}\ \mathsf{ref.is\_null} & \hookrightarrow & (\mathsf{i32.const}\ 0) & (\text{otherwise})
\end{array}$$

ref.func x

1. Let F be the current frame.
2. Assert: due to validation, F.module.funcaddrs$[x]$ exists.
3. Let a be the function address F.module.funcaddrs$[x]$.
4. Push the value ref a to the stack.

$$F;(\mathsf{ref.func}\ x) \quad \hookrightarrow \quad F;(\mathsf{ref}\ a) \qquad (\text{if } a = F.\mathsf{module.funcaddrs}[x])$$

4.4.3 Vector Instructions

Most vector instructions are defined in terms of generic numeric operators applied lane-wise based on the shape.

$$op_{t\mathsf{x}N}(n_1,\dots,n_k) \quad = \quad \mathrm{lanes}^{-1}_{t\mathsf{x}N}(op_t(\mathrm{lanes}_{t\mathsf{x}N}(n_1)\ \dots\ \mathrm{lanes}_{t\mathsf{x}N}(n_k))$$

Note: For example, the result of instruction i32x4.add applied to operands i_1, i_2 invokes $\mathrm{add}_{\mathsf{i32x4}}(i_1, i_2)$, which maps to $\mathrm{lanes}^{-1}_{\mathsf{i32x4}}(\mathrm{add}_{\mathsf{i32}}(i_1^+, i_2^+))$, where i_1^+ and i_2^+ are sequences resulting from invoking $\mathrm{lanes}_{\mathsf{i32x4}}(i_1)$ and $\mathrm{lanes}_{\mathsf{i32x4}}(i_2)$ respectively.

v128.const c

1. Push the value v128.const c to the stack.

Note: No formal reduction rule is required for this instruction, since const instructions coincide with values.

v128.*vvunop*

1. Assert: due to validation, a value of value type v128 is on the top of the stack.
2. Pop the value v128.const c_1 from the stack.
3. Let c be the result of computing $vvunop_{\mathsf{v128}}(c_1)$.
4. Push the value v128.const c to the stack.

$$(\mathsf{v128.const}\ c_1)\ \mathsf{v128}.vvunop \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad (\text{if } c = vvunop_{\mathsf{v128}}(c_1))$$

v128.*vvbinop*

1. Assert: due to validation, two values of value type v128 are on the top of the stack.
2. Pop the value v128.const c_2 from the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let c be the result of computing $vvbinop_{\mathsf{v128}}(c_1, c_2)$.
5. Push the value v128.const c to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathsf{v128}.vvbinop \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad (\text{if } c = vvbinop_{\mathsf{v128}}(c_1, c_2))$$

$\mathsf{v128}.\mathit{vvternop}$

1. Assert: due to validation, three values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_3$ from the stack.
3. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
4. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
5. Let c be the result of computing $\mathit{vvternop}_{\mathsf{v128}}(c_1, c_2, c_3)$.
6. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{v128.const}\ c_3)\ \mathsf{v128}.\mathit{vvternop} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad (\text{if } c = \mathit{vvternop}_{\mathsf{v128}}(c_1, c_2, c_3))$$

$\mathsf{v128.any\_true}$

1. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
3. Let i be the result of computing $\mathrm{ine}_{128}(c_1, 0)$.
4. Push the value $\mathsf{i32.const}\ i$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ \mathsf{v128.any\_true} \quad\hookrightarrow\quad (\mathsf{i32.const}\ i) \qquad (\text{if } i = \mathrm{ine}_{128}(c_1, 0))$$

$\mathsf{i8x16.swizzle}$

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
3. Let i^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_2)$.
4. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
5. Let j^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_1)$.
6. Let c^* be the concatenation of the two sequences j^* and 0^{240}.
7. Let c' be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i8x16}}(c^*[i^*[0]] \dots c^*[i^*[15]])$.
8. Push the value $\mathsf{v128.const}\ c'$ onto the stack.

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathsf{i8x16.swizzle} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c') \\ \quad (\text{if } i^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_2) \\ \quad \wedge\ c^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_1)\ 0^{240} \\ \quad \wedge\ c' = \mathrm{lanes}^{-1}_{\mathsf{i8x16}}(c^*[i^*[0]] \dots c^*[i^*[15]]))\end{array}$$

$\mathsf{i8x16.shuffle}\ x^*$

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Assert: due to validation, for all x_i in x^* it holds that $x_i < 32$.
3. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
4. Let i_2^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_2)$.
5. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
6. Let i_1^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_1)$.
7. Let i^* be the concatenation of the two sequences i_1^* and i_2^*.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i8x16}}(i^*[x^*[0]] \dots i^*[x^*[15]])$.

9. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{i8x16.shuffle}\ x^*) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)\\ \quad(\mathrm{if}\ i^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_1)\ \mathrm{lanes}_{\mathsf{i8x16}}(c_2)\\ \quad\ \wedge c = \mathrm{lanes}^{-1}_{\mathsf{i8x16}}(i^*[x^*[0]] \dots i^*[x^*[15]]))\end{array}$$

shape.splat

1. Let t be the type $\mathrm{unpacked}(\mathit{shape})$.
2. Assert: due to validation, a value of value type t is on the top of the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. Let N be the integer $\mathrm{dim}(\mathit{shape})$.
5. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathit{shape}}(c_1^N)$.
6. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$(t.\mathsf{const}\ c_1)\ \mathit{shape}.\mathsf{splat} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad (\mathrm{if}\ t = \mathrm{unpacked}(\mathit{shape}) \wedge c = \mathrm{lanes}^{-1}_{\mathit{shape}}(c_1^{\mathrm{dim}(\mathit{shape})}))$$

$t_1\mathsf{x}N.\mathsf{extract\_lane\_}\mathit{sx}^?\ x$

1. Assert: due to validation, $x < N$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}N}(c_1)$.
5. Let t_2 be the type $\mathrm{unpacked}(t_1\mathsf{x}N)$.
6. Let c_2 be the result of computing $\mathrm{extend}^{\mathit{sx}^?}_{t_1,t_2}(i^*[x])$.
7. Push the value $t_2.\mathsf{const}\ c_2$ to the stack.

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (t_1\mathsf{x}N.\mathsf{extract\_lane}\ x) \quad\hookrightarrow\quad (t_2.\mathsf{const}\ c_2)\\ \quad(\mathrm{if}\ t_2 = \mathrm{unpacked}(t_1\mathsf{x}N)\\ \quad\ \wedge c_2 = \mathrm{extend}^{\mathit{sx}^?}_{t_1,t_2}(\mathrm{lanes}_{t_1\mathsf{x}N}(c_1)[x]))\end{array}$$

$\mathit{shape}.\mathsf{replace\_lane}\ x$

1. Assert: due to validation, $x < \mathrm{dim}(\mathit{shape})$.
2. Let t_1 be the type $\mathrm{unpacked}(\mathit{shape})$.
3. Assert: due to validation, a value of value type t_1 is on the top of the stack.
4. Pop the value $t_1.\mathsf{const}\ c_1$ from the stack.
5. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
6. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
7. Let i^* be the result of computing $\mathrm{lanes}_{\mathit{shape}}(c_2)$.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathit{shape}}(i^*\ \mathrm{with}\ [x] = c_1)$.
9. Push $\mathsf{v128.const}\ c$ on the stack.

$$\begin{array}{l}(t_1.\mathsf{const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathit{shape}.\mathsf{replace\_lane}\ x) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)\\ \quad(\mathrm{if}\ i^* = \mathrm{lanes}_{\mathit{shape}}(c_2)\\ \quad\ \wedge c = \mathrm{lanes}^{-1}_{\mathit{shape}}(i^*\ \mathrm{with}\ [x] = c_1))\end{array}$$

$\mathit{shape}.\mathit{vunop}$

1. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
3. Let c be the result of computing $\mathit{vunop}_{\mathit{shape}}(c_1)$.
4. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ \mathsf{v128}.\mathit{vunop} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad (\mathrm{if}\ c = \mathit{vunop}_{\mathit{shape}}(c_1))$$

$\mathit{shape}.\mathit{vbinop}$

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. If $\mathit{vbinop}_{\mathit{shape}}(c_1, c_2)$ is defined:
    a. Let c be a possible result of computing $\mathit{vbinop}_{\mathit{shape}}(c_1, c_2)$.
    b. Push the value $\mathsf{v128.const}\ c$ to the stack.
5. Else:
    a. Trap.

$$\begin{array}{lcll}
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathit{shape}.\mathit{vbinop} & \hookrightarrow & (\mathsf{v128.const}\ c) & (\mathrm{if}\ c \in \mathit{vbinop}_{\mathit{shape}}(c_1, c_2)) \\
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathit{shape}.\mathit{vbinop} & \hookrightarrow & \mathsf{trap} & (\mathrm{if}\ \mathit{vbinop}_{\mathit{shape}}(c_1, c_2) = \{\})
\end{array}$$

$\mathsf{t}\mathsf{x}N.\mathit{vrelop}$

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i_1^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_1)$.
5. Let i_2^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_2)$.
6. Let i^* be the result of computing $\mathit{vrelop}_t(i_1^*, i_2^*)$.
7. Let j^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{1,|t|}(i^*)$.
8. Let c be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}^{-1}(j^*)$.
9. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{l}
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ t\mathsf{x}N.\mathit{vrelop} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \\
\quad (\mathrm{if}\ c = \mathrm{lanes}_{t\mathsf{x}N}^{-1}(\mathrm{extend}^{\mathsf{s}}{}_{1,|t|}(\mathit{vrelop}_t(\mathrm{lanes}_{t\mathsf{x}N}(c_1), \mathrm{lanes}_{t\mathsf{x}N}(c_2)))))
\end{array}$$

$t\mathsf{x}N.\mathit{vishiftop}$

1. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $\mathsf{i32.const}\ s$ from the stack.
3. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
4. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
5. Let i^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_1)$.
6. Let j^* be the result of computing $\mathit{vishiftop}_t(i^*, s^N)$.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t\mathsf{x}N}(j^*)$.
8. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}\ c_1)\ (\mathsf{i32.const}\ s)\ t\mathsf{x}N.\mathit{vishiftop} & \hookrightarrow & (\mathsf{v128.const}\ c) \\
\quad (\mathrm{if}\ i^* = \mathrm{lanes}_{t\mathsf{x}N}(c_1) & & \\
\quad\ \wedge c = \mathrm{lanes}^{-1}_{t\mathsf{x}N}(\mathit{vishiftop}_t(i^*, s^N))) & &
\end{array}$$

$\mathit{shape}.\mathsf{all\_true}$

1. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
3. Let i_1^* be the result of computing $\mathrm{lanes}_{\mathit{shape}}(c_1)$.
4. Let i be the result of computing $\mathrm{bool}(\bigwedge(i_1 \neq 0)^*)$.
5. Push the value $\mathsf{i32.const}\ i$ onto the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}\ c_1)\ \mathit{shape}.\mathsf{all\_true} & \hookrightarrow & (\mathsf{i32.const}\ i) \\
\quad (\mathrm{if}\ i_1^* = \mathrm{lanes}_{\mathit{shape}}(c) & & \\
\quad\ \wedge i = \mathrm{bool}(\bigwedge(i_1 \neq 0)^*)) & &
\end{array}$$

$t\mathsf{x}N.\mathsf{bitmask}$

1. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
3. Let i_1^N be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c)$.
4. Let B be the bit width $|t|$ of value type t.
5. Let i_2^N be the result of computing $\mathrm{ilt\_s}_B(i_1^N, 0^N)$.
6. Let j^* be the concatenation of the two sequences i_2^N and 0^{32-N}.
7. Let c be the result of computing $\mathrm{ibits}^{-1}_{32}(j^*)$.
8. Push the value $\mathsf{i32.const}\ c$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ t\mathsf{x}N.\mathsf{bitmask} \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad (\mathrm{if}\ c = \mathrm{ibits}^{-1}_{32}(\mathrm{ilt\_s}_{|t|}(\mathrm{lanes}_{t\mathsf{x}N}(c), 0^N)))$$

$t_2\mathsf{x}N.\mathsf{narrow}\_t_1\mathsf{x}M\_sx$

1. Assert: due to syntax, $N = 2 \cdot M$.
2. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
3. Pop the value $\mathsf{v128.const}~c_2$ from the stack.
4. Let i_2^M be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_2)$.
5. Let d_2^M be the result of computing $\mathrm{narrow}^{sx}_{|t_1|,|t_2|}(i_2^M)$.
6. Pop the value $\mathsf{v128.const}~c_1$ from the stack.
7. Let i_1^M be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
8. Let d_1^M be the result of computing $\mathrm{narrow}^{sx}_{|t_1|,|t_2|}(i_1^M)$.
9. Let j^N be the concatenation of the two sequences d_1^M and d_2^M.
10. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(j^N)$.
11. Push the value $\mathsf{v128.const}~c$ onto the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}~c_1)~(\mathsf{v128.const}~c_2)~t_2\mathsf{x}N.\mathsf{narrow}\_t_1\mathsf{x}M\_sx & \hookrightarrow & (\mathsf{v128.const}~c) \\
\quad (\mathrm{if}~d_1^M = \mathrm{narrow}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)) & & \\
\quad \wedge~d_2^M = \mathrm{narrow}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_2)) & & \\
\quad \wedge~c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(d_1^M~d_2^M)) & &
\end{array}$$

$t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_sx$

1. Assert: due to syntax, $N = M$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}~c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. Let j^* be the result of computing $\mathit{vcvtop}^{sx}_{|t_1|,|t_2|}(i^*)$.
6. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(j^*)$.
7. Push the value $\mathsf{v128.const}~c$ onto the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}~c_1)~t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_sx & \hookrightarrow & (\mathsf{v128.const}~c) \\
\quad (\mathrm{if}~c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathit{vcvtop}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)))) & &
\end{array}$$

$t_2\mathsf{x}N.\mathit{vcvtop}\_\mathit{half}\_t_1\mathsf{x}M\_sx^?$

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}~c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. If half is low, then:
    a. Let j^* be the sequence $i^*[0:N]$.
6. Else:
    a. Let j^* be the sequence $i^*[N:N]$.
7. Let k^* be the result of computing $\mathit{vcvtop}^{sx^?}_{|t_1|,|t_2|}(j^*)$.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.

9. Push the value v128.const c onto the stack.

$$\begin{array}{l} (\mathsf{v128.const}~c_1)~t_2\mathsf{x}N.\mathit{vcvtop}\_\mathit{half}\_t_1\mathsf{x}M\_\mathit{sx}^? \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \\ \qquad (\mathrm{if}~c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathit{vcvtop}^{\mathit{sx}^?}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)[\mathit{half}(0,N) : N]))) \end{array}$$

where:

$$\begin{array}{lcl} \mathsf{low}(x,y) &=& x \\ \mathsf{high}(x,y) &=& y \end{array}$$

$t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_\mathit{sx}\_\mathsf{zero}$

1. Assert: due to syntax, $N = 2 \cdot M$.
2. Assert: due to validation, a value of value type v128 is on the top of the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. Let j^* be the result of computing $\mathit{vcvtop}^{\mathit{sx}}_{|t_1|,|t_2|}(i^*)$.
6. Let k^* be the concatenation of the two sequences j^* and 0^M.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.
8. Push the value v128.const c onto the stack.

$$\begin{array}{l} (\mathsf{v128.const}~c_1)~t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_\mathit{sx}\_\mathsf{zero} \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \\ \qquad (\mathrm{if}~c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathit{vcvtop}^{\mathit{sx}}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1))~0^M)) \end{array}$$

i32x4.dot_i16x8_s

1. Assert: due to validation, two values of value type v128 are on the top of the stack.
2. Pop the value v128.const c_2 from the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let i_1^* be the result of computing $\mathrm{lanes}_{\mathsf{i16x8}}(c_1)$.
5. Let j_1^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{16,32}(i_1^*)$.
6. Let i_2^* be the result of computing $\mathrm{lanes}_{\mathsf{i16x8}}(c_2)$.
7. Let j_2^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{16,32}(i_2^*)$.
8. Let $(k_1~k_2)^*$ be the result of computing $\mathrm{imul}_{32}(j_1^*, j_2^*)$.
9. Let k^* be the result of computing $\mathrm{iadd}_{32}(k_1, k_2)^*$.
10. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i32x4}}(k^*)$.
11. Push the value v128.const c onto the stack.

$$\begin{array}{l} (\mathsf{v128.const}~c_1)~(\mathsf{v128.const}~c_2)~\mathsf{i32x4.dot\_i16x8\_s} \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \\ \qquad (\mathrm{if}~(i_1~i_2)^* = \mathrm{imul}_{32}(\mathrm{extend}^{\mathsf{s}}{}_{16,32}(\mathrm{lanes}_{\mathsf{i16x8}}(c_1)), \mathrm{extend}^{\mathsf{s}}{}_{16,32}(\mathrm{lanes}_{\mathsf{i16x8}}(c_2))) \\ \qquad \wedge~j^* = \mathrm{iadd}_{32}(i_1, i_2)^* \\ \qquad \wedge~c = \mathrm{lanes}^{-1}_{\mathsf{i32x4}}(j^*)) \end{array}$$

$t_2\mathsf{x}N.\mathsf{extmul\_}half\mathsf{\_}t_1\mathsf{x}M\mathsf{\_}sx$

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, two values of value type v128 are on the top of the stack.
3. Pop the value v128.const c_2 from the stack.
4. Pop the value v128.const c_1 from the stack.
5. Let i_1^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
6. Let i_2^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_2)$.
7. If $half$ is low, then:
   a. Let j_1^* be the sequence $i_1^*[0 : N]$.
   b. Let j_2^* be the sequence $i_2^*[0 : N]$.
8. Else:
   a. Let j_1^* be the sequence $i_1^*[N : N]$.
   b. Let j_2^* be the sequence $i_2^*[N : N]$.
9. Let k_1^* be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(j_1^*)$.
10. Let k_2^* be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(j_2^*)$.
11. Let k^* be the result of computing $\mathrm{imul}_{t_2\mathsf{x}N}(k_1^*, k_2^*)$.
12. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.
13. Push the value v128.const c onto the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ t_2\mathsf{x}N.\mathsf{extmul\_}half\mathsf{\_}t_1\mathsf{x}M\mathsf{\_}sx & \hookrightarrow & (\mathsf{v128.const}\ c) \\
\multicolumn{3}{r}{(\mathrm{if}\ i^* = \mathrm{lanes}_{t_1\mathsf{x}M}(c_1)[half(0, N) : N]} \\
\multicolumn{3}{r}{\wedge\ j^* = \mathrm{lanes}_{t_1\mathsf{x}M}(c_2)[half(0, N) : N]} \\
\multicolumn{3}{r}{\wedge\ c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathrm{imul}_{t_2\mathsf{x}N}(\mathrm{extend}^{sx}_{|t_1|,|t_2|}(i^*), \mathrm{extend}^{sx}_{|t_1|,|t_2|}(j^*))))}
\end{array}$$

where:

$$\begin{array}{lcl}
\mathsf{low}(x, y) & = & x \\
\mathsf{high}(x, y) & = & y
\end{array}$$

$t_2\mathsf{x}N.\mathsf{extadd\_pairwise\_}t_1\mathsf{x}M\mathsf{\_}sx$

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, a value of value type v128 is on the top of the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. Let $(j_1\ j_2)^*$ be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(i^*)$.
6. Let k^* be the result of computing $\mathrm{iadd}_N(j_1, j_2)^*$.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.
8. Push the value v128.const c to the stack.

$$\begin{array}{lcl}
(\mathsf{v128.const}\ c_1)\ t_2\mathsf{x}N.\mathsf{extadd\_pairwise\_}t_1\mathsf{x}M\mathsf{\_}sx & \hookrightarrow & (\mathsf{v128.const}\ c) \\
\quad (\mathrm{if}\ (i_1\ i_2)^* = \mathrm{extend}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)) && \\
\quad \wedge\ j^* = \mathrm{iadd}_N(i_1, i_2)^* && \\
\quad \wedge\ c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(j^*)) &&
\end{array}$$

4.4.4 Parametric Instructions

drop

1. Assert: due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.

$$\mathit{val}~\mathsf{drop} \quad \hookrightarrow \quad \epsilon$$

select $(t^*)^?$

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. Assert: due to validation, two more values (of the same value type) are on the top of the stack.
4. Pop the value val_2 from the stack.
5. Pop the value val_1 from the stack.
6. If c is not 0, then:
   a. Push the value val_1 back to the stack.
7. Else:
   a. Push the value val_2 back to the stack.

$$\begin{array}{lcll}
\mathit{val}_1~\mathit{val}_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & \mathit{val}_1 & (\text{if } c \neq 0) \\
\mathit{val}_1~\mathit{val}_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & \mathit{val}_2 & (\text{if } c = 0)
\end{array}$$

Note: In future versions of WebAssembly, select may allow more than one value per choice.

4.4.5 Variable Instructions

local.get x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{locals}[x]$ exists.
3. Let val be the value $F.\mathsf{locals}[x]$.
4. Push the value val to the stack.

$$F; (\mathsf{local.get}~x) \quad \hookrightarrow \quad F; \mathit{val} \qquad (\text{if } F.\mathsf{locals}[x] = \mathit{val})$$

local.set x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{locals}[x]$ exists.
3. Assert: due to validation, a value is on the top of the stack.
4. Pop the value val from the stack.
5. Replace $F.\mathsf{locals}[x]$ with the value val.

$$F; \mathit{val}~(\mathsf{local.set}~x) \quad \hookrightarrow \quad F'; \epsilon \qquad (\text{if } F' = F~\mathsf{with}~\mathsf{locals}[x] = \mathit{val})$$

local.tee x

1. Assert: due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. Push the value val to the stack.
4. Push the value val to the stack.
5. Execute the instruction local.set x.

$$val\ (\mathsf{local.tee}\ x) \quad \hookrightarrow \quad val\ val\ (\mathsf{local.set}\ x)$$

global.get x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{globaladdrs}[x]$ exists.
3. Let a be the global address $F.\mathsf{module}.\mathsf{globaladdrs}[x]$.
4. Assert: due to validation, $S.\mathsf{globals}[a]$ exists.
5. Let $glob$ be the global instance $S.\mathsf{globals}[a]$.
6. Let val be the value $glob.\mathsf{value}$.
7. Push the value val to the stack.

$$\begin{array}{l}S; F; (\mathsf{global.get}\ x) \quad \hookrightarrow \quad S; F; val \\ \qquad (\text{if } S.\mathsf{globals}[F.\mathsf{module}.\mathsf{globaladdrs}[x]].\mathsf{value} = val)\end{array}$$

global.set x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{globaladdrs}[x]$ exists.
3. Let a be the global address $F.\mathsf{module}.\mathsf{globaladdrs}[x]$.
4. Assert: due to validation, $S.\mathsf{globals}[a]$ exists.
5. Let $glob$ be the global instance $S.\mathsf{globals}[a]$.
6. Assert: due to validation, a value is on the top of the stack.
7. Pop the value val from the stack.
8. Replace $glob.\mathsf{value}$ with the value val.

$$\begin{array}{l}S; F; val\ (\mathsf{global.set}\ x) \quad \hookrightarrow \quad S'; F; \epsilon \\ \qquad (\text{if } S' = S \text{ with } \mathsf{globals}[F.\mathsf{module}.\mathsf{globaladdrs}[x]].\mathsf{value} = val)\end{array}$$

Note: Validation ensures that the global is, in fact, marked as mutable.

4.4.6 Table Instructions

table.get x

1. Let F be the current frame.
2. Assert: due to validation, F.module.tableaddrs$[x]$ exists.
3. Let a be the table address F.module.tableaddrs$[x]$.
4. Assert: due to validation, S.tables$[a]$ exists.
5. Let tab be the table instance S.tables$[a]$.
6. Assert: due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value i32.const i from the stack.
8. If i is not smaller than the length of tab.elem, then:
   a. Trap.
9. Let val be the value tab.elem$[i]$.
10. Push the value val to the stack.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & S; F; val \\
\qquad (\text{if } S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}[i] = val) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}$$

table.set x

1. Let F be the current frame.
2. Assert: due to validation, F.module.tableaddrs$[x]$ exists.
3. Let a be the table address F.module.tableaddrs$[x]$.
4. Assert: due to validation, S.tables$[a]$ exists.
5. Let tab be the table instance S.tables$[a]$.
6. Assert: due to validation, a reference value is on the top of the stack.
7. Pop the value val from the stack.
8. Assert: due to validation, a value of value type i32 is on the top of the stack.
9. Pop the value i32.const i from the stack.
10. If i is not smaller than the length of tab.elem, then:
    a. Trap.
11. Replace the element tab.elem$[i]$ with val.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x) & \hookrightarrow & S'; F; \epsilon \\
\qquad (\text{if } S' = S \text{ with } \mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}[i] = val) \\
S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x) & \hookrightarrow & S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}$$

table.size x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{tableaddrs}[x]$ exists.
3. Let a be the table address $F.\mathsf{module}.\mathsf{tableaddrs}[x]$.
4. Assert: due to validation, $S.\mathsf{tables}[a]$ exists.
5. Let tab be the table instance $S.\mathsf{tables}[a]$.
6. Let sz be the length of $tab.\mathsf{elem}$.
7. Push the value $\mathsf{i32}.\mathsf{const}\ sz$ to the stack.

$$\begin{array}{l}S; F; (\mathsf{table.size}\ x) \quad \hookrightarrow \quad S; F; (\mathsf{i32}.\mathsf{const}\ sz) \\ \qquad (\mathrm{if}\ |S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}| = sz)\end{array}$$

table.grow x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{tableaddrs}[x]$ exists.
3. Let a be the table address $F.\mathsf{module}.\mathsf{tableaddrs}[x]$.
4. Assert: due to validation, $S.\mathsf{tables}[a]$ exists.
5. Let tab be the table instance $S.\mathsf{tables}[a]$.
6. Let sz be the length of $S.\mathsf{tables}[a]$.
7. Assert: due to validation, a value of value type i32 is on the top of the stack.
8. Pop the value $\mathsf{i32}.\mathsf{const}\ n$ from the stack.
9. Assert: due to validation, a reference value is on the top of the stack.
10. Pop the value val from the stack.
11. Let err be the $i32$ value $2^{32} - 1$, for which $\mathrm{signed}_{32}(err)$ is -1.
12. Either:
    a. If growing tab by n entries with initialization value val succeeds, then:
        i. Push the value $\mathsf{i32}.\mathsf{const}\ sz$ to the stack.
    b. Else:
        i. Push the value $\mathsf{i32}.\mathsf{const}\ err$ to the stack.
13. Or:
    a. push the value $\mathsf{i32}.\mathsf{const}\ err$ to the stack.

$$\begin{array}{lcl}S; F; val\ (\mathsf{i32}.\mathsf{const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & S'; F; (\mathsf{i32}.\mathsf{const}\ sz) \\ \quad (\mathrm{if}\ F.\mathsf{module}.\mathsf{tableaddrs}[x] = a \\ \quad \wedge\ sz = |S.\mathsf{tables}[a].\mathsf{elem}| \\ \quad \wedge\ S' = S\ \mathsf{with}\ \mathsf{tables}[a] = \mathrm{growtable}(S.\mathsf{tables}[a], n, val)) \\ S; F; val\ (\mathsf{i32}.\mathsf{const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & S; F; (\mathsf{i32}.\mathsf{const}\ \mathrm{signed}_{32}^{-1}(-1))\end{array}$$

Note: The table.grow instruction is non-deterministic. It may either succeed, returning the old table size sz, or fail, returning -1. Failure *must* occur if the referenced table instance has a maximum size defined that would be exceeded. However, failure *can* occur in other cases as well. In practice, the choice depends on the resources available to the embedder.

table.fill x

1. Let F be the current frame.
2. Assert: due to validation, F.module.tableaddrs$[x]$ exists.
3. Let ta be the table address F.module.tableaddrs$[x]$.
4. Assert: due to validation, S.tables$[ta]$ exists.
5. Let tab be the table instance S.tables$[ta]$.
6. Assert: due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value i32.const n from the stack.
8. Assert: due to validation, a reference value is on the top of the stack.
9. Pop the value val from the stack.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value i32.const i from the stack.
12. If $i + n$ is larger than the length of tab.elem, then:
    a. Trap.
12. If n is 0, then:
    a. Return.
13. Push the value i32.const i to the stack.
14. Push the value val to the stack.
15. Execute the instruction table.set x.
16. Push the value i32.const $(i + 1)$ to the stack.
17. Push the value val to the stack.
18. Push the value i32.const $(n - 1)$ to the stack.
19. Execute the instruction table.fill x.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\mathrm{if}\ i + n > |S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}|) \\
S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ 0)\ (\mathsf{table.fill}\ x) \quad \hookrightarrow \quad S; F; \epsilon \\
\qquad (\mathrm{otherwise}) \\
S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n+1)\ (\mathsf{table.fill}\ x) \quad \hookrightarrow \\
\qquad S; F; (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x) \\
\qquad\qquad (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) \\
\qquad (\mathrm{otherwise})
\end{array}$$

table.copy x y

1. Let F be the current frame.
2. Assert: due to validation, F.module.tableaddrs$[x]$ exists.
3. Let ta_x be the table address F.module.tableaddrs$[x]$.
4. Assert: due to validation, S.tables$[ta_x]$ exists.
5. Let tab_x be the table instance S.tables$[ta_x]$.
6. Assert: due to validation, F.module.tableaddrs$[y]$ exists.
7. Let ta_y be the table address F.module.tableaddrs$[y]$.

8. Assert: due to validation, S.tables$[ta_y]$ exists.
9. Let tab_y be the table instance S.tables$[ta_y]$.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value i32.const n from the stack.
12. Assert: due to validation, a value of value type i32 is on the top of the stack.
13. Pop the value i32.const s from the stack.
14. Assert: due to validation, a value of value type i32 is on the top of the stack.
15. Pop the value i32.const d from the stack.
16. If $s + n$ is larger than the length of tab_y.elem or $d + n$ is larger than the length of tab_x.elem, then:
    a. Trap.
17. If $n = 0$, then:

a. Return.

18. If $d \leq s$, then:

a. Push the value i32.const d to the stack.

b. Push the value i32.const s to the stack.

c. Execute the instruction table.get y.

d. Execute the instruction table.set x.

e. Assert: due to the earlier check against the table size, $d + 1 < 2^{32}$.

f. Push the value i32.const $(d + 1)$ to the stack.

g. Assert: due to the earlier check against the table size, $s + 1 < 2^{32}$.

h. Push the value i32.const $(s + 1)$ to the stack.

19. Else:
    a. Assert: due to the earlier check against the table size, $d + n - 1 < 2^{32}$.
    b. Push the value i32.const $(d + n - 1)$ to the stack.
    c. Assert: due to the earlier check against the table size, $s + n - 1 < 2^{32}$.
    d. Push the value i32.const $(s + n - 1)$ to the stack.
    c. Execute the instruction table.get y.
    f. Execute the instruction table.set x.
    g. Push the value i32.const d to the stack.
    h. Push the value i32.const s to the stack.
20. Push the value i32.const $(n - 1)$ to the stack.
21. Execute the instruction table.copy x y.

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\mathrm{if}\ s + n > |S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[y]].\mathsf{elem}| & & \\
\quad\ \vee\ d + n > |S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}|) & & \\
S; F; (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{i32.const}\ 0)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & S; F; \epsilon \\
\quad (\mathrm{otherwise}) & & \\
S; F; (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{i32.const}\ n+1)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \\
\quad S; F; (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & \\
\qquad (\mathsf{i32.const}\ d+1)\ (\mathsf{i32.const}\ s+1)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & & \\
\quad (\mathrm{otherwise, if}\ d \leq s) & & \\
S; F; (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{i32.const}\ n+1)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \\
\quad S; F; (\mathsf{i32.const}\ d+n)\ (\mathsf{i32.const}\ s+n)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & \\
\qquad (\mathsf{i32.const}\ d)\ (\mathsf{i32.const}\ s)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & & \\
\quad (\mathrm{otherwise, if}\ d > s) & &
\end{array}
$$

table.init x y

1. Let F be the current frame.
2. Assert: due to validation, F.module.tableaddrs$[x]$ exists.
3. Let ta be the table address F.module.tableaddrs$[x]$.
4. Assert: due to validation, S.tables$[ta]$ exists.
5. Let tab be the table instance S.tables$[ta]$.
6. Assert: due to validation, F.module.elemaddrs$[y]$ exists.
7. Let ea be the element address F.module.elemaddrs$[y]$.
8. Assert: due to validation, S.elems$[ea]$ exists.
9. Let $elem$ be the element instance S.elems$[ea]$.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value i32.const n from the stack.
12. Assert: due to validation, a value of value type i32 is on the top of the stack.
13. Pop the value i32.const s from the stack.
14. Assert: due to validation, a value of value type i32 is on the top of the stack.
15. Pop the value i32.const d from the stack.
16. If $s + n$ is larger than the length of $elem$.elem or $d + n$ is larger than the length of tab.elem, then:
    a. Trap.
17. If $n = 0$, then:
    a. Return.
18. Let val be the reference value $elem$.elem$[s]$.
19. Push the value i32.const d to the stack.
20. Push the value val to the stack.
21. Execute the instruction table.set x.
22. Assert: due to the earlier check against the table size, $d + 1 < 2^{32}$.
23. Push the value i32.const $(d + 1)$ to the stack.
24. Assert: due to the earlier check against the segment size, $s + 1 < 2^{32}$.
25. Push the value i32.const $(s + 1)$ to the stack.

26. Push the value i32.const $(n-1)$ to the stack.
27. Execute the instruction table.init x y.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n)~(\mathsf{table.init}~x~y) &\hookrightarrow& S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~s + n > |S.\mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[y]].\mathsf{elem}| \\
\qquad \lor d + n > |S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}|) \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~0)~(\mathsf{table.init}~x~y) &\hookrightarrow& S; F; \epsilon \\
\qquad (\mathrm{otherwise}) \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n+1)~(\mathsf{table.init}~x~y) &\hookrightarrow& \\
\qquad S; F; (\mathsf{i32.const}~d)~\mathit{val}~(\mathsf{table.set}~x) \\
\qquad\qquad (\mathsf{i32.const}~d+1)~(\mathsf{i32.const}~s+1)~(\mathsf{i32.const}~n)~(\mathsf{table.init}~x~y) \\
\qquad (\mathrm{otherwise, if}~\mathit{val} = S.\mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[y]].\mathsf{elem}[s])
\end{array}$$

elem.drop x

1. Let F be the current frame.
2. Assert: due to validation, F.module.elemaddrs$[x]$ exists.
3. Let a be the element address F.module.elemaddrs$[x]$.
4. Assert: due to validation, S.elems$[a]$ exists.
5. Replace S.elems$[a]$.elem with ϵ.

$$\begin{array}{lcl}
S; F; (\mathsf{elem.drop}~x) &\hookrightarrow& S'; F; \epsilon \\
\qquad (\mathrm{if}~S' = S~\mathsf{with}~\mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[x]].\mathsf{elem} = \epsilon)
\end{array}$$

4.4.7 Memory Instructions

Note: The alignment memarg.align in load and store instructions does not affect the semantics. It is an indication that the offset ea at which the memory is accessed is intended to satisfy the property $\mathit{ea} \bmod 2^{\mathit{memarg}.\mathsf{align}} = 0$. A WebAssembly implementation can use this hint to optimize for the intended use. Unaligned access violating that property is still allowed and must succeed regardless of the annotation. However, it may be substantially slower on some hardware.

t.load memarg **and** t.load$N$_sx memarg

1. Let ord be unord.
2. Let F be the current frame.
3. Assert: due to validation, F.module.memaddrs$[0]$ exists.
4. Let a be the memory address F.module.memaddrs$[0]$.
5. Assert: due to validation, S.mems$[a]$ exists.
6. Let mem be the memory instance S.mems$[a]$.
7. Assert: due to validation, a value of value type i32 is on the top of the stack.
8. Pop the value i32.const i from the stack.
9. Let ea be the integer $i + \mathit{memarg}$.offset.
10. If N is not part of the instruction, then:

a. Let N be the bit width $|t|$ of number type t.

11. If both ord is seqcst and ea modulo $N/8$ is not equal to 0, then:

a. Trap.

12. If $\mathit{mem}.\mathsf{type}$ is $\mathit{limits}\ \mathsf{unshared}$, then:
    a. If $\mathit{ea} + N/8$ is larger than the length of $\mathit{mem}.\mathsf{data}$, then:
        i. Trap.
    b. Let b^* be the byte sequence $\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8]$.
13. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $\mathit{ea} + N/8$ is larger than n, then:
        i. Trap.
    c. Let $\mathit{notears}^?$ be $\mathrm{tearing}(t, N, \mathit{ea})$.
    d. Perform the action $(\mathsf{rd}_{\mathit{ord}}\ a.\mathsf{data}[\mathit{ea}]\ b^*\ \mathit{notears}^?)$ to read $N/8$ bytes b^* from data offset ea of the shared memory instance at memory address a.
14. Let c_N be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
15. Let c be the result of computing $\mathrm{extend}^{sx}_{N,|t|}(c_N)$.
16. Push the value $t.\mathsf{const}\ c$ to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}(N\_sx)^?\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; (t.\mathsf{const}\ c) \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared} \\
\qquad \wedge\ \mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}| \\
\qquad \wedge\ (\mathit{ord} = \mathsf{unord} \vee \mathit{ea} \bmod N/8 = 0) \\
\qquad \wedge\ c = \mathrm{extend}^{sx}_{N,|t|}(\mathrm{bytes}^{-1}_{iN}(\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8]))) \\
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}(N\_sx)^?\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise, if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared})
\end{array}
$$

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}(N\_sx)?\ \mathit{memarg}) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{\mathit{ord}}\ a.\mathsf{data}[\mathit{ea}]\ b^*\ \mathit{notears}^?)} \quad S; F; (t.\mathsf{const}\ c) \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad \wedge\ \mathit{ea} + N/8 \leq n \\
\qquad \wedge\ (\mathit{ord} = \mathsf{unord} \vee \mathit{ea} \bmod N/8 = 0) \\
\qquad \wedge\ c = \mathrm{extend}^{sx}_{N,|t|}(\mathrm{bytes}^{-1}_{iN}(b^*))) \\
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}(N\_sx)^?\ \mathit{memarg}) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} \quad S; F; \mathsf{trap} \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad \wedge\ (\mathit{ea} + N/8 > n \vee \mathit{ord} = \mathsf{seqcst} \wedge \mathit{ea} \bmod N/8 \neq 0))
\end{array}
$$

$$
\begin{array}{l}
(\text{where } \mathit{ord} = \mathsf{unord} \\
\qquad \wedge\ N = |t| \text{ if } N \text{ not present} \\
\qquad \wedge\ sx = \mathsf{u} \text{ if } sx \text{ not present} \\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ \mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge\ \mathit{ea} = i + \mathit{memarg}.\mathsf{offset} \\
\qquad \wedge\ \mathit{notears}^? = \mathrm{tearing}(t, N, \mathit{ea}))
\end{array}
$$

v128.load$M$xN_sx $memarg$

1. Let ord be unord.
2. Let F be the current frame.
3. Assert: due to validation, F.module.memaddrs$[0]$ exists.
4. Let a be the memory address F.module.memaddrs$[0]$.
5. Assert: due to validation, S.mems$[a]$ exists.
6. Let mem be the memory instance S.mems$[a]$.
7. Assert: due to validation, a value of value type i32 is on the top of the stack.
8. Pop the value i32.const i from the stack.
9. Let ea be the integer $i + memarg$.offset.
10. If mem.type is $limits$ unshared, then:
    a. If $ea + M \cdot N/8$ is larger than the length of mem.data, then:
        i. Trap.
    b. Let b^* be the byte sequence mem.data$[ea : M \cdot N/8]$.
11. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $ea + M \cdot N/8$ is larger than n, then:
        i. Trap.
    c. Perform the action $(\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)$ to read $M \cdot N/8$ bytes b^* from data offset ea of the shared memory instance at memory address a.
12. Let m_k be the integer for which $\mathrm{bytes}_{iM}(m_k) = b^*[k \cdot M/8 : M/8]$.
13. Let W be the integer $M \cdot 2$.
14. Let n_k be the result of computing $\mathrm{extend}^{sx}_{M,W}(m_k)$.
15. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathrm{i}W\mathrm{x}N}(n_0 \dots n_{N-1})$.
16. Push the value v128.const c to the stack.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ memarg) \quad \hookrightarrow \quad S; F; (\mathsf{v128.const}\ c)\\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{unshared}\\
\qquad \wedge\ ea + M \cdot N/8 \leq |mem.\mathsf{data}|\\
\qquad \wedge\ \mathrm{bytes}_{iM}(m_k) = mem.\mathsf{data}[ea + k \cdot M/8 : M/8]\\
\qquad \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}W\mathsf{x}N}(\mathrm{extend}^{sx}_{M,W}(m_0) \dots \mathrm{extend}^{sx}_{M,W}(m_{N-1})))\\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap}\\
\qquad (\mathrm{otherwise, if}\ mem.\mathsf{type} = limits\ \mathsf{unshared})
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ memarg) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)} \quad S; F; (\mathsf{v128.const}\ c)\\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared}\\
\qquad \wedge\ ea + M \cdot N/8 \leq n\\
\qquad \wedge\ \mathrm{bytes}_{iM}(m_k) = b^*[k \cdot M/8 : M/8]\\
\qquad \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}W\mathsf{x}N}(\mathrm{extend}^{sx}_{M,W}(m_0) \dots \mathrm{extend}^{sx}_{M,W}(m_{N-1})))\\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ memarg) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)} \quad S; F; \mathsf{trap}\\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared}\\
\qquad \wedge\ (ea + M \cdot N/8 > n)
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ ord = \mathsf{unord}\\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0]\\
\qquad \wedge\ mem = S.\mathsf{mems}[a]\\
\qquad \wedge\ ea = i + memarg.\mathsf{offset}\\
\qquad \wedge\ W = M \cdot 2)
\end{array}$$

v128.load$N$_splat $memarg$

1. Let ord be unord.
2. Let F be the current frame.
3. Assert: due to validation, F.module.memaddrs[0] exists.
4. Let a be the memory address F.module.memaddrs[0].
5. Assert: due to validation, S.mems[a] exists.
6. Let mem be the memory instance S.mems[a].
7. Assert: due to validation, a value of value type i32 is on the top of the stack.
8. Pop the value i32.const i from the stack.
9. Let ea be the integer $i + memarg$.offset.
10. If mem.type is $limits$ unshared, then:
    a. If $ea + N/8$ is larger than the length of mem.data, then:
        i. Trap.
    b. Let b^* be the byte sequence mem.data[$ea : N/8$].
11. Else:
    a. Perform the action (rd a.len n) to read the length n of the shared memory instance at memory address a.
    b. If $ea + N/8$ is larger than n, then:
        i. Trap.
    c. Perform the action (rd$_{ord}$ a.data[ea] b^*) to read $N/8$ bytes b^* from data offset ea of the shared memory instance at memory address a.
12. Let n be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.

13. Let L be the integer $128/N$.
14. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathrm{i}N\mathrm{x}L}(n^L)$.
15. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_splat}\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; (\mathsf{v128.const}\ c) \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared} \\
\qquad\ \wedge\ \mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}| \\
\qquad\ \wedge\ \mathrm{bytes}_{iN}(n) = \mathit{mem}.\mathsf{data}[\mathit{ea} : N/8] \\
\qquad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathrm{i}N\mathrm{x}L}(n^L)) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_splat}\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise, if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared})
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_splat}\ \mathit{memarg}) \quad \xhookrightarrow{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{\mathit{ord}}\ a.\mathsf{data}[\mathit{ea}]\ b^*)} \quad S; F; (t.\mathsf{const}\ c) \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad\ \wedge\ \mathit{ea} + N/8 \leq n \\
\qquad\ \wedge\ \mathrm{bytes}_{iN}(n) = b^* \\
\qquad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathrm{i}N\mathrm{x}L}(n^L)) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_splat}\ \mathit{memarg}) \quad \xhookrightarrow{(\mathsf{rd}\ a.\mathsf{len}\ n)} \quad S; F; \mathsf{trap} \\
\qquad (\text{if } \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad\ \wedge\ (\mathit{ea} + N/8 > n)
\end{array}$$

$$\begin{array}{l}
(\text{where } \mathit{ord} = \mathsf{unord} \\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ \mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge\ \mathit{ea} = i + \mathit{memarg}.\mathsf{offset})
\end{array}$$

v128.load$N$_zero memarg

1. Let ord be unord.
2. Let F be the current frame.
3. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
4. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
5. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
6. Let mem be the memory instance $S.\mathsf{mems}[a]$.
7. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
8. Pop the value $\mathsf{i32.const}\ i$ from the stack.
9. Let ea be the integer $i + \mathit{memarg}.\mathsf{offset}$.
10. If $\mathit{mem}.\mathsf{type}$ is $\mathit{limits}\ \mathsf{unshared}$, then:
    a. If $\mathit{ea} + N/8$ is larger than the length of $\mathit{mem}.\mathsf{data}$, then:
        i. Trap.
    b. Let b^* be the byte sequence $\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8]$.
11. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $\mathit{ea} + N/8$ is larger than n, then:
        i. Trap.

```
        c. Perform the action $(\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)$ to read $N/8$ bytes $b^*$ from data offset $ea$ of the shared memory instance at memory address $a$.
12. Let $n$ be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
13. Let $c$ be the result of computing $\mathrm{extend}^{\mathsf{u}}{}_{N,128}(n)$.
14. Push the value $\mathsf{v128.const}\ c$ to the stack.
```

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_zero}\ memarg) & \hookrightarrow & S; F; (\mathsf{v128.const}\ c) \\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{unshared} & & \\
\qquad \wedge\ ea + N/8 \leq |mem.\mathsf{data}| & & \\
\qquad \wedge\ \mathrm{bytes}_{iN}(n) = mem.\mathsf{data}[ea : N/8] & & \\
\qquad \wedge\ c = \mathrm{extend}^{\mathsf{u}}{}_{N,128}(n)) & & \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_zero}\ memarg) & \hookrightarrow & S; F; \mathsf{trap} \\
\qquad (\mathrm{otherwise, if}\ mem.\mathsf{type} = limits\ \mathsf{unshared}) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_zero}\ memarg) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)} & S; F; (t.\mathsf{const}\ c) \\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\qquad \wedge\ ea + N/8 \leq n & & \\
\qquad \wedge\ \mathrm{bytes}_{iN}(n) = b^* & & \\
\qquad \wedge\ c = \mathrm{extend}^{\mathsf{u}}{}_{N,128}(n)) & & \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\mathsf{\_zero}\ memarg) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} & S; F; \mathsf{trap} \\
\qquad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\qquad \wedge\ (ea + N/8 > n) & &
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ ord = \mathsf{unord} \\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ mem = S.\mathsf{mems}[a] \\
\qquad \wedge\ ea = i + memarg.\mathsf{offset})
\end{array}$$

$\mathsf{v128.load}N\mathsf{\_lane}\ memarg\ x$

```
1. Let $ord$ be $\mathsf{unord}$.
2. Let $F$ be the current frame.
3. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
4. Let $a$ be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
5. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
6. Let $mem$ be the memory instance $S.\mathsf{mems}[a]$.
7. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
8. Pop the value $\mathsf{v128.const}\ v$ from the stack.
9. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
10. Pop the value $\mathsf{i32.const}\ i$ from the stack.
11. Let $ea$ be the integer $i + memarg.\mathsf{offset}$.
12. If $mem.\mathsf{type}$ is $limits\ \mathsf{unshared}$, then:
    a. If $ea + N/8$ is larger than the length of $mem.\mathsf{data}$, then:
        i. Trap.
    b. Let $b^*$ be the byte sequence $mem.\mathsf{data}[ea : N/8]$.
13. Else:
```

   a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.

   b. If $ea + N/8$ is larger than n, then:

      i. Trap.

   c. Perform the action $(\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)$ to read $N/8$ bytes b^* from data offset ea of the shared memory instance at memory address a.

14. Let r be the constant for which $\mathrm{bytes}_{iN}(r) = b^*$.

15. Let L be $128/N$.

16. Let j^* be the result of computing $\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(v)$.

17. Let c be the result of computing $\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(j^*\ \mathrm{with}\ [x] = r)$.

18. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{\_lane}\ memarg\ x) & \hookrightarrow & S; F; (\mathsf{v128.const}\ c) \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{unshared} & & \\
\quad\ \wedge\ ea + N/8 \leq |mem.\mathsf{data}| & & \\
\quad\ \wedge\ \mathrm{bytes}_{iN}(n) = mem.\mathsf{data}[ea : N/8] & & \\
\quad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}L}(\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(v)\ \mathrm{with}\ [x] = r)) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{\_lane}\ memarg\ x) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\text{otherwise, if } mem.\mathsf{type} = limits\ \mathsf{unshared}) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{\_lane}\ memarg\ x) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{ord}\ a.\mathsf{data}[ea]\ b^*)} & S; F; (t.\mathsf{const}\ c) \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\quad\ \wedge\ ea + N/8 \leq n & & \\
\quad\ \wedge\ \mathrm{bytes}_{iN}(n) = b^* & & \\
\quad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}L}(\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(v)\ \mathrm{with}\ [x] = r)) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{\_lane}\ memarg\ x) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} & S; F; \mathsf{trap} \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\quad\ \wedge\ (ea + N/8 > n) & &
\end{array}$$

$$\begin{array}{l}
(\text{where } ord = \mathsf{unord} \\
\quad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\quad \wedge\ mem = S.\mathsf{mems}[a] \\
\quad \wedge\ ea = i + memarg.\mathsf{offset}) \\
\quad \wedge\ L = 128/N
\end{array}$$

$t.\mathsf{store}\ memarg$ and $t.\mathsf{store}N\ memarg$

1. Let ord be unord.

2. Let F be the current frame.

3. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.

4. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.

5. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.

6. Let mem be the memory instance $S.\mathsf{mems}[a]$.

7. Assert: due to validation, a value of value type t is on the top of the stack.

8. Pop the value $t.\mathsf{const}\ c$ from the stack.

9. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.

10. Pop the value $\mathsf{i32.const}\ i$ from the stack.

11. Let ea be the integer $i + memarg.\mathsf{offset}$.
12. If N is not part of the instruction, then:
    a. Let N be the bit width $|t|$ of number type t.
13. If both ord is seqcst and ea modulo $N/8$ is not equal to 0, then:

a. Trap.

15. Let n be the result of computing $\mathrm{wrap}_{|t|,N}(c)$.
16. Let b^* be the byte sequence $\mathrm{bytes}_{iN}(n)$.
17. If $mem.\mathsf{type}$ is $limits$ $\mathsf{unshared}$, then:
    a. If $ea + N/8$ is larger than the length of $mem.\mathsf{data}$, then:
        i. Trap.
    b. Replace the bytes $mem.\mathsf{data}[ea : N/8]$ with b^*.
18. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $ea + N/8$ is larger than n, then:
        i. Trap.
    c. Let $notears^?$ be $\mathrm{tearing}(t, N, ea)$.
    d. Perform the action $(\mathsf{wr}_{ord}\ a.\mathsf{data}[ea]\ b^*\ notears^?)$ to write the bytes b^* to data offset ea of the shared memory instance at memory address a.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N^?\ memarg) & \hookrightarrow & S'; F; \epsilon \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{unshared} & & \\
\quad\ \wedge\ ea + N/8 \leq |mem.\mathsf{data}| & & \\
\quad\ \wedge\ (ord = \mathsf{unord} \vee ea \bmod N/8 = 0) & & \\
\quad\ \wedge\ S' = S\ \mathsf{with}\ \mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[ea : N/8] = \mathrm{bytes}_{iN}(\mathrm{wrap}_{|t|,N}(c))) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N^?\ memarg) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\text{otherwise, if } mem.\mathsf{type} = limits\ \mathsf{unshared}) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N^?\ memarg) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{wr}_{ord}\ a.\mathsf{data}[ea]\ b^*)\ notears} & S; F; \epsilon \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\quad\ \wedge\ ea + N/8 \leq n & & \\
\quad\ \wedge\ (ord = \mathsf{unord} \vee ea \bmod N/8 = 0) & & \\
\quad\ \wedge\ b^* = \mathrm{bytes}_{iN}(\mathrm{wrap}_{|t|,N}(c))) & &
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N^?\ memarg) & \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} & S; F; \mathsf{trap} \\
\quad (\text{if } mem.\mathsf{type} = limits\ \mathsf{shared} & & \\
\quad\ \wedge\ (ea + N/8 > n \vee ord = \mathsf{seqcst} \wedge ea \bmod N/8 \neq 0)) & &
\end{array}$$

$$\begin{array}{l}
(\text{where } ord = \mathsf{unord} \\
\quad\ \wedge\ N = |t| \text{ if } N \text{ not present} \\
\quad\ \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\quad\ \wedge\ mem = S.\mathsf{mems}[a] \\
\quad\ \wedge\ ea = i + memarg.\mathsf{offset} \\
\quad\ \wedge\ notears^? = \mathrm{tearing}(t, N, ea))
\end{array}$$

v128.store$N$_lane $memarg$ x

1. Let ord be unord.
2. Let F be the current frame.
3. Assert: due to validation, F.module.memaddrs$[0]$ exists.
4. Let a be the memory address F.module.memaddrs$[0]$.
5. Assert: due to validation, S.mems$[a]$ exists.
6. Let mem be the memory instance S.mems$[a]$.
7. Assert: due to validation, a value of value type v128 is on the top of the stack.
8. Pop the value v128.const c from the stack.
9. Assert: due to validation, a value of value type i32 is on the top of the stack.
10. Pop the value i32.const i from the stack.
11. Let ea be the integer $i + memarg$.offset.
12. Let L be $128/N$.
13. Let j^* be the result of computing $\mathrm{lanes}_{\mathrm{i}N\times L}(c)$.
14. Let b^* be the result of computing $\mathrm{bytes}_{iN}(j^*[x])$.
15. If mem.type is $limits$ unshared, then:
    1. If $ea + N/8$ is larger than the length of mem.data, then:
        1. Trap.
    2. Replace the bytes mem.data$[ea : N/8]$ with b^*.
16. Else:
    1. Perform the action (rd a.len n) to read the length n of the shared memory instance at memory address a.
    2. If $ea + N/8$ is larger than n, then:
        1. Trap.
    3. Perform the action (wr_{ord} a.data$[ea]$ b^*) to write the bytes b^* to data offset ea of the shared memory instance at memory address a.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ \mathit{memarg}\ x) \quad \hookrightarrow \quad S'; F; \epsilon \\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared} \\
\qquad\ \wedge\ \mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}| \\
\qquad\ \wedge\ S' = S\ \mathsf{with}\ \mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[\mathit{ea} : N/8] = \mathrm{bytes}_{iN}(\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(c)[x])) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ \mathit{memarg}\ x) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\mathrm{otherwise, if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared})
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ \mathit{memarg}\ x) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{wr}_{\mathit{ord}}\ a.\mathsf{data}[\mathit{ea}]\ b^*)} \quad S; F; \epsilon \\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad\ \wedge\ \mathit{ea} + N/8 \leq n \\
\qquad\ \wedge\ b^* = \mathrm{bytes}_{iN}(\mathrm{lanes}_{\mathsf{i}N\mathsf{x}L}(c)[x])) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ \mathit{memarg}\ x) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} \quad S; F; \mathsf{trap} \\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad\ \wedge\ (\mathit{ea} + N/8 > n \vee \mathit{ord} = \mathsf{seqcst} \wedge \mathit{ea} \bmod N/8 \neq 0))
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ \mathit{ord} = \mathsf{unord} \\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ \mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge\ \mathit{ea} = i + \mathit{memarg}.\mathsf{offset} \\
\qquad \wedge\ L = 128/N)
\end{array}$$

memory.size

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. If $\mathit{mem}.\mathsf{type}$ is $\mathit{limits}\ \mathsf{unshared}$, then:
    a. Let n be the length of $\mathit{mem}.\mathsf{data}$.
7. Else:
    a. Perform the action $(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
8. Let sz be n divided by the page size.
9. Push the value $\mathsf{i32.const}\ \mathit{sz}$ to the stack.

$$\begin{array}{l}
S; F; \mathsf{memory.size} \quad \hookrightarrow \quad S; F; (\mathsf{i32.const}\ \mathit{sz}) \\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared} \\
\qquad\ \wedge\ |\mathit{mem}.\mathsf{data}| = \mathit{sz} \cdot 64\,\mathrm{Ki}
\end{array}$$

$$\begin{array}{l}
S; F; \mathsf{memory.size} \quad \hookrightarrow^{(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{len}\ n)} \quad S; F; (\mathsf{i32.const}\ \mathit{sz}) \\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\qquad\ \wedge\ n = \mathit{sz} \cdot 64\,\mathrm{Ki})
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ \mathit{mem} = S.\mathsf{mems}[a])
\end{array}$$

memory.grow

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32}.\mathsf{const}\ n$ from the stack.
8. Let err be the $\mathit{i32}$ value $2^{32} - 1$, for which $\mathrm{signed}_{32}(err)$ is -1.
9. If $mem.\mathsf{type}$ is limits $\mathsf{unshared}$, then:
    b. Let sz be the length of $S.\mathsf{mems}[a]$ divided by the page size.
    c. Either, try growing mem by n pages:
        i. If it succeeds, push the value $\mathsf{i32}.\mathsf{const}\ sz$ to the stack.
        ii. Else, push the value $\mathsf{i32}.\mathsf{const}\ err$ to the stack.
    d. Or, push the value $\mathsf{i32}.\mathsf{const}\ err$ to the stack.
10. Else:

a. Either successfully grow the memory:
    i. Let k be n multiplied by the page size.
    ii. Perform the action $(\mathsf{rmw}\ a.\mathsf{len}\ l\ (l + k))$ to update the current length l of the shared memory instance at memory address a to $l + k$.
    iii. Perform the action $(\mathsf{wr}\ a.\mathsf{data}[l]\ (0)^k)$ to append k zero bytes to the end of the shared memory instance at memory address a.
    iv. Let sz be l divided by the page size.
    v. Push the value $\mathsf{i32}.\mathsf{const}\ sz$ to the stack.

b. Or indicate failure:
    i. Perform the action $(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{len}\ l)$ to read the length l of the shared memory instance at memory address a.
    ii. Push the value $\mathsf{i32}.\mathsf{const}\ err$ to the stack.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ n)\ \mathsf{memory.grow} & \hookrightarrow & S'; F; (\mathsf{i32.const}\ sz) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = \mathit{limits}\ \mathsf{unshared} \\
\quad \wedge\ sz = |mem.\mathsf{data}|/64\,\mathrm{Ki} \\
\quad \wedge\ S' = S\ \mathsf{with}\ \mathsf{mems}[a] = \mathrm{growmem}(S.\mathsf{mems}[a], n)) \\
S; F; (\mathsf{i32.const}\ n)\ \mathsf{memory.grow} & \hookrightarrow & S; F; (\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1)) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = \mathit{limits}\ \mathsf{unshared})
\end{array}$$

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ n)\ \mathsf{memory.grow} & \hookrightarrow^{(\mathsf{rmw}\ a.\mathsf{len}\ l\ (l+k))\ (\mathsf{wr}\ a.\mathsf{data}[l]\ (0)^k)} & S; F; (\mathsf{i32.const}\ sz) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = \mathit{limits}\ \mathsf{shared} \\
\quad \wedge\ sz = l/64\,\mathrm{Ki} \\
\quad \wedge\ n = k/64\,\mathrm{Ki}) \\
S; F; (\mathsf{i32.const}\ n)\ \mathsf{memory.grow} & \hookrightarrow^{(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{len}\ n)} & S; F; (\mathsf{i32.const}\ -1) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = \mathit{limits}\ \mathsf{shared})
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\quad \wedge\ mem = S.\mathsf{mems}[a])
\end{array}$$

Note: The memory.grow instruction is non-deterministic, even on unshared memories. It may either succeed, returning the old memory size sz, or fail, returning -1. Failure *must* occur if the referenced memory instance has a maximum size defined that would be exceeded. However, failure *can* occur in other cases as well. In practice, the choice depends on the resources available to the embedder.

memory.fill

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
3. Let ma be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
4. Assert: due to validation, $S.\mathsf{mems}[ma]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[ma]$.
6. Assert: due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ n$ from the stack.
8. Assert: due to validation, a value of value type i32 is on the top of the stack.
9. Pop the value val from the stack.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value $\mathsf{i32.const}\ d$ from the stack.
12. If $mem.\mathsf{type}$ is limits unshared, then:
    a. If $d + n$ is larger than the length of $mem.\mathsf{data}$, then:
        i. Trap.
13. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ l)$ to read the length l of the shared memory instance at memory address a.
    b. If $d + n$ is larger than l, then:
        i. Trap.
14. If $n = 0$, then:
    a. Return.

15. Push the value i32.const d to the stack.
16. Push the value val to the stack.
17. Execute the instruction i32.store8 $\{$offset $0,$ align $0\}$.
18. Assert: due to the earlier check against the memory size, $d + 1 < 2^{32}$.
19. Push the value i32.const $(d + 1)$ to the stack.
20. Push the value val to the stack.
21. Push the value i32.const $(n - 1)$ to the stack.
22. Execute the instruction memory.fill.

$$\begin{array}{lll}
S; F; (\mathsf{i32.const}\ d)\ val\ (\mathsf{i32.const}\ n)\ \mathsf{memory.fill} & \hookrightarrow^{act} & S; F; \mathsf{trap} \\
\qquad (\mathrm{if}\ d + n > l) & & \\
S; F; (\mathsf{i32.const}\ d)\ val\ (\mathsf{i32.const}\ 0)\ \mathsf{memory.fill} & \hookrightarrow^{act} & S; F; \epsilon \\
\qquad (\mathrm{otherwise}) & &
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ d)\ val\ (\mathsf{i32.const}\ n+1)\ \mathsf{memory.fill} \quad \hookrightarrow^{act} \\
\qquad S; F; (\mathsf{i32.const}\ d)\ val\ (\mathsf{i32.store8}\ \{\mathsf{offset}\ 0, \mathsf{align}\ 0\}) \\
\qquad\qquad (\mathsf{i32.const}\ d+1)\ val\ (\mathsf{i32.const}\ n)\ \mathsf{memory.fill} \\
\qquad (\mathrm{otherwise})
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ a = F.\mathsf{module.memaddrs}[0] \\
\qquad \wedge\ mem = S.\mathsf{mems}[a] \\
\qquad \wedge\ ((mem.\mathsf{type} = limits\ \mathsf{unshared} \wedge l = |mem.\mathsf{data}| \wedge act = \epsilon) \vee (mem.\mathsf{type} = limits\ \mathsf{shared} \wedge act = (\mathsf{rd}\ a.\mathsf{len}\ l)))
\end{array}$$

memory.copy

1. Let F be the current frame.
2. Assert: due to validation, F.module.memaddrs$[0]$ exists.
3. Let ma be the memory address F.module.memaddrs$[0]$.
4. Assert: due to validation, S.mems$[ma]$ exists.
5. Let mem be the memory instance S.mems$[ma]$.
6. Assert: due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value i32.const n from the stack.
8. Assert: due to validation, a value of value type i32 is on the top of the stack.
9. Pop the value i32.const s from the stack.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value i32.const d from the stack.
12. If mem.type is $limits$ unshared, then:
    a. If $s + n$ is larger than the length of mem.data or $d + n$ is larger than the length of mem.data, then:
        i. Trap.
13. Else:
    a. Perform the action (rd a.len l) to read the length l of the shared memory instance at memory address a.
    a. If $s + n$ is larger than l or $d + n$ is larger than l, then:
        i. Trap.

14. If $n = 0$, then:
    a. Return.
15. If $d \leq s$, then:
    a. Push the value $\mathsf{i32.const}~d$ to the stack.
    b. Push the value $\mathsf{i32.const}~s$ to the stack.
    c. Execute the instruction $\mathsf{i32.load8\_u}~\{\mathsf{offset}~0, \mathsf{align}~0\}$.
    d. Execute the instruction $\mathsf{i32.store8}~\{\mathsf{offset}~0, \mathsf{align}~0\}$.
    e. Assert: due to the earlier check against the memory size, $d + 1 < 2^{32}$.
    f. Push the value $\mathsf{i32.const}~(d + 1)$ to the stack.
    g. Assert: due to the earlier check against the memory size, $s + 1 < 2^{32}$.
    h. Push the value $\mathsf{i32.const}~(s + 1)$ to the stack.
16. Else:
    a. Assert: due to the earlier check against the memory size, $d + n - 1 < 2^{32}$.
    b. Push the value $\mathsf{i32.const}~(d + n - 1)$ to the stack.
    c. Assert: due to the earlier check against the memory size, $s + n - 1 < 2^{32}$.
    d. Push the value $\mathsf{i32.const}~(s + n - 1)$ to the stack.
    e. Execute the instruction $\mathsf{i32.load8\_u}~\{\mathsf{offset}~0, \mathsf{align}~0\}$.
    f. Execute the instruction $\mathsf{i32.store8}~\{\mathsf{offset}~0, \mathsf{align}~0\}$.
    g. Push the value $\mathsf{i32.const}~d$ to the stack.
    h. Push the value $\mathsf{i32.const}~s$ to the stack.
17. Push the value $\mathsf{i32.const}~(n - 1)$ to the stack.
18. Execute the instruction $\mathsf{memory.copy}$.

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n)~\mathsf{memory.copy} & \hookrightarrow^{act} & S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~(s + n > l & & \\
\qquad\quad \vee~d + n > l)) & & \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~0)~\mathsf{memory.copy} & \hookrightarrow & S; F; \epsilon \\
\qquad (\mathrm{otherwise}) & & \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n+1)~\mathsf{memory.copy} & \hookrightarrow^{act} & \\
\qquad S; F; (\mathsf{i32.const}~d) & & \\
\qquad\qquad (\mathsf{i32.const}~s)~(\mathsf{i32.load8\_u}~\{\mathsf{offset}~0, \mathsf{align}~0\}) & & \\
\qquad\qquad (\mathsf{i32.store8}~\{\mathsf{offset}~0, \mathsf{align}~0\}) & & \\
\qquad\qquad (\mathsf{i32.const}~d+1)~(\mathsf{i32.const}~s+1)~(\mathsf{i32.const}~n)~\mathsf{memory.copy} & & \\
\qquad (\mathrm{otherwise}) & & \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n+1)~\mathsf{memory.copy} & \hookrightarrow^{act} & \\
\qquad S; F; (\mathsf{i32.const}~d+n) & & \\
\qquad\qquad (\mathsf{i32.const}~s+n)~(\mathsf{i32.load8\_u}~\{\mathsf{offset}~0, \mathsf{align}~0\}) & & \\
\qquad\qquad (\mathsf{i32.store8}~\{\mathsf{offset}~0, \mathsf{align}~0\}) & & \\
\qquad\qquad (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n)~\mathsf{memory.copy} & & \\
\qquad (\mathrm{otherwise}) & &
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{where}~a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge~\mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge~((\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \wedge l = |\mathit{mem}.\mathsf{data}| \wedge \mathit{act} = \epsilon) \vee (\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{shared} \wedge \mathit{act} = (\mathsf{rd}~a.\mathsf{len}~l)))
\end{array}
$$

memory.init x

1. Let F be the current frame.
2. Assert: due to validation, F.module.memaddrs$[0]$ exists.
3. Let ma be the memory address F.module.memaddrs$[0]$.
4. Assert: due to validation, S.mems$[ma]$ exists.
5. Let mem be the memory instance S.mems$[ma]$.
6. Assert: due to validation, F.module.dataaddrs$[x]$ exists.
7. Let da be the data address F.module.dataaddrs$[x]$.
8. Assert: due to validation, S.datas$[da]$ exists.
9. Let $data$ be the data instance S.datas$[da]$.
10. Assert: due to validation, a value of value type i32 is on the top of the stack.
11. Pop the value i32.const n from the stack.
12. Assert: due to validation, a value of value type i32 is on the top of the stack.
13. Pop the value i32.const s from the stack.
14. Assert: due to validation, a value of value type i32 is on the top of the stack.
15. Pop the value i32.const d from the stack.
16. If mem.type is *limits* unshared, then:
    a. If $s + n$ is larger than the length of $data$.data or $d + n$ is larger than the length of mem.data, then:
        i. Trap.
17. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ l)$ to read the length l of the shared memory instance at memory address a.
    a. If $s + n$ is larger than the length of $data$.data or $d + n$ is larger than l, then:
        i. Trap.
18. If $n = 0$, then:
    a. Return.
19. Let b be the byte $data$.data$[s]$.
20. Push the value i32.const d to the stack.
21. Push the value i32.const b to the stack.
22. Execute the instruction i32.store8 $\{\mathsf{offset}\ 0, \mathsf{align}\ 0\}$.
23. Assert: due to the earlier check against the memory size, $d + 1 < 2^{32}$.
24. Push the value i32.const $(d + 1)$ to the stack.
25. Assert: due to the earlier check against the memory size, $s + 1 < 2^{32}$.
26. Push the value i32.const $(s + 1)$ to the stack.
27. Push the value i32.const $(n - 1)$ to the stack.
28. Execute the instruction memory.init x.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n)~(\mathsf{memory.init}~x) \quad \hookrightarrow^{act} \quad S; F; \mathsf{trap} \\
\qquad (\text{if } (s + n > |S.\mathsf{datas}[F.\mathsf{module}.\mathsf{dataaddrs}[x]].\mathsf{data}| \\
\qquad\ \vee\ d + n > l)) \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~0)~(\mathsf{memory.init}~x) \quad \hookrightarrow^{act} \quad S; F; \epsilon \\
\qquad (\text{otherwise}) \\
S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~s)~(\mathsf{i32.const}~n+1)~(\mathsf{memory.init}~x) \quad \hookrightarrow^{act} \\
\qquad S; F; (\mathsf{i32.const}~d)~(\mathsf{i32.const}~b)~(\mathsf{i32.store8}~\{\mathsf{offset}~0, \mathsf{align}~0\}) \\
\qquad\qquad (\mathsf{i32.const}~d+1)~(\mathsf{i32.const}~s+1)~(\mathsf{i32.const}~n)~(\mathsf{memory.init}~x) \\
\qquad (\text{otherwise, if } b = S.\mathsf{datas}[F.\mathsf{module}.\mathsf{dataaddrs}[x]].\mathsf{data}[s])
\end{array}
$$

$$
\begin{array}{l}
(\text{where } a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge\ mem = S.\mathsf{mems}[a] \\
\qquad \wedge\ ((mem.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \wedge l = |mem.\mathsf{data}| \wedge act = \epsilon) \vee (mem.\mathsf{type} = \mathit{limits}~\mathsf{shared} \wedge act = (\mathsf{rd}~a.\mathsf{len}~l)))
\end{array}
$$

$\mathsf{data.drop}~x$

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{dataaddrs}[x]$ exists.
3. Let a be the data address $F.\mathsf{module}.\mathsf{dataaddrs}[x]$.
4. Assert: due to validation, $S.\mathsf{datas}[a]$ exists.
5. Replace $S.\mathsf{datas}[a]$ with the data instance $\{\mathsf{data}~\epsilon\}$.

$$
\begin{array}{l}
S; F; (\mathsf{data.drop}~x) \quad \hookrightarrow \quad S'; F; \epsilon \\
\qquad (\text{if } S' = S~\mathsf{with}~\mathsf{datas}[F.\mathsf{module}.\mathsf{dataaddrs}[x]] = \{\mathsf{data}~\epsilon\})
\end{array}
$$

4.4.8 Atomic Memory Instructions

$t.\mathsf{atomic.load}(N\_\mathsf{u})^?~\mathit{memarg}$

The rules are identical to non-atomic loads, except that $\mathit{ord} = \mathsf{seqcst}$.

$t.\mathsf{atomic.store}N^?~\mathit{memarg}$

The rules are identical to non-atomic stores, except that $\mathit{ord} = \mathsf{seqcst}$.

$t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathit{atop}~\mathit{memarg}$

1. If N is not part of the instruction, then:
   a. Let N be the bit width $|t|$ of value type t.
3. Assert: due to validation, a value of value type t is on the top of the stack.
4. Pop the value $t.\mathsf{const}~c_2$ from the stack.
5. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
6. Pop the value $\mathsf{i32.const}~i$ from the stack.
7. Let ea be $i + \mathit{memarg}.\mathsf{offset}$.
8. If ea modulo $N/8$ is not equal to 0, then:
   a. Trap.

9. Let F be the current frame.
10. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
11. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
12. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
13. Let mem be the memory instance $S.\mathsf{mems}[a]$.
14. If $mem.\mathsf{type}$ is limits $\mathsf{unshared}$, then:
    a. If $ea + N/8$ is larger than the length of $mem.\mathsf{data}$, then:
        i. Trap.
    b. Let b_{r}^* be the byte sequence $mem.\mathsf{data}[ea : N/8]$.
13. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $ea + N/8$ is larger than n, then:
        i. Trap.
    c. Let b_{r}^* be chosen to represent $N/8$ bytes of memory at location ea of mem.
14. Let c_{r} be the integer for which $\mathrm{bytes}_{iN}(n) = b_{\mathrm{r}}^*$.
15. Let c_1 be the result of computing $\mathrm{extend}^{\mathsf{u}}_{N,|t|}(c_{\mathrm{r}})$.
16. Let c be the result of computing $\mathit{atop}_t(c_1, c_2)$.
17. Let c_{w} be the result of computing $\mathrm{wrap}_{|t|,N}(c)$.
18. Let b_{w}^* be the byte sequence $\mathrm{bytes}_{iN}(c_{\mathrm{w}})$.
19. If $mem.\mathsf{type}$ is limits $\mathsf{unshared}$, then:
    a. Replace the bytes $mem.\mathsf{data}[ea : N/8]$ with b_{w}^*.
10. Else:
    a. Perform the atomic action $(\mathsf{rmw}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b_{\mathrm{r}}^*\ b_{\mathrm{w}}^*)$ to read $N/8$ bytes b_{r}^* from data offset ea of the shared memory instance at memory address a and replace them with bytes b_{w}^*.
20. Push the value $t.\mathsf{const}\ c$ to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}~i)~(t\mathsf{.const}~c_2)~(t\mathsf{.atomic.rmw}(N\mathsf{\_u})^?.\mathit{atop}~\mathit{memarg}) \quad\hookrightarrow\quad S'; F; (t\mathsf{.const}~c_1) \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \\
\qquad\ \wedge~\mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}| \\
\qquad\ \wedge~\mathit{ea} \bmod N/8 = 0 \\
\qquad\ \wedge~c_1 = \mathrm{extend}^{\mathsf{u}}_{N,|t|}(\mathrm{bytes}_{iN}^{-1}(\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8])) \\
\qquad\ \wedge~S' = S~\mathsf{with}~\mathsf{mems}[a].\mathsf{data}[\mathit{ea} : N/8] = \mathrm{bytes}_{iN}(\mathrm{wrap}_{|t|,N}(\mathit{atop}_t(c_1, c_2)))) \\
S; F; (\mathsf{i32.const}~k)~(t\mathsf{.const}~c)~(t\mathsf{.atomic.rmw}(N\mathsf{\_u})^?.\mathit{atop}~\mathit{memarg}) \quad\hookrightarrow\quad S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \\
\qquad\ \wedge~\mathit{ea} + N/8 > |\mathit{mem}.\mathsf{data}| \vee \mathit{ea} \bmod N/8 \neq 0)
\end{array}
$$

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}~i)~(t\mathsf{.const}~c_2)~(t\mathsf{.atomic.rmw}(N\mathsf{\_u})^?.\mathit{atop}~\mathit{memarg}) \quad\hookrightarrow^{(\mathsf{rd}~a.\mathsf{len}~n)~(\mathsf{rmw}~a.\mathsf{data}[\mathit{ea}]~b_{\mathrm{r}}^*~b_{\mathrm{w}}^*)}\quad S; F; (t\mathsf{.const}~c_1) \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{shared} \\
\qquad\ \wedge~\mathit{ea} + N/8 \leq n \\
\qquad\ \wedge~\mathit{ea} \bmod N/8 = 0 \\
\qquad\ \wedge~c_1 = \mathrm{extend}^{\mathsf{u}}_{N,|t|}(\mathrm{bytes}_{iN}^{-1}(b_{\mathrm{r}}^*)) \\
\qquad\ \wedge~b_{\mathrm{w}}^* = \mathrm{bytes}_{iN}(\mathrm{wrap}_{|t|,N}(\mathit{atop}_t(c_1, c_2)))) \\
S; F; (\mathsf{i32.const}~k)~(t\mathsf{.const}~c)~(t\mathsf{.atomic.rmw}(N\mathsf{\_u})^?.\mathit{atop}~\mathit{memarg}) \quad\hookrightarrow^{(\mathsf{rd}~a.\mathsf{len}~n)}\quad S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{shared} \\
\qquad\ \wedge~\mathit{ea} + N/8 > n \vee \mathit{ea} \bmod N/8 \neq 0)
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{where}~N = |t|~\mathrm{if}~N~\mathrm{not~present} \\
\qquad \wedge~a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge~\mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge~\mathit{ea} = i + \mathit{memarg}.\mathsf{offset})
\end{array}
$$

$t\mathsf{.atomic.rmw}(N\mathsf{\_u})^?\mathsf{.cmpxchg}~\mathit{memarg}$

1. If N is not part of the instruction, then:
    a. Let N be the bit width $|t|$ of value type t.
3. Assert: due to validation, two values of value type t are on the top of the stack.
4. Pop the value $t\mathsf{.const}~c_3$ from the stack.
5. Pop the value $t\mathsf{.const}~c_2$ from the stack.
6. Assert: due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $\mathsf{i32.const}~i$ from the stack.
8. Let ea be $i + \mathit{memarg}.\mathsf{offset}$.
9. If ea modulo $N/8$ is not equal to 0, then:
    a. Trap.
10. Let F be the current frame.
11. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
12. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
13. Let mem be the memory instance $S.\mathsf{mems}[a]$.
14. If $\mathit{mem}.\mathsf{type}$ is $\mathit{limits}~\mathsf{unshared}$, then:
    a. If $\mathit{ea} + N/8$ is larger than the length of $\mathit{mem}.\mathsf{data}$, then:
        i. Trap.
    b. Let b_{r}^* be the byte sequence $\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8]$.
15. Else:

   a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.

   b. If $ea + N/8$ is larger than n, then:

      i. Trap.

   c. Let b_{r}^* be chosen to represent $N/8$ bytes of memory at location ea of mem.

16. Let c_{r} be the integer for which $\mathrm{bytes}_{iN}(n) = b_{\mathrm{r}}^*$.

17. Let c_{ex} be the result of computing $\mathrm{wrap}_{|t|,N}(c_2)$.

18. If c_{r} equals c_{ex}, then:

    a. Let c_{w} be the result of computing $\mathrm{wrap}_{|t|,N}(c_3)$.

    b. Let b_{w}^* be the byte sequence $\mathrm{bytes}_{iN}(c_{\mathrm{w}})$.

    c. If $mem.\mathsf{type}$ is $\mathit{limits}\ \mathsf{unshared}$, then:

       i. Replace the bytes $mem.\mathsf{data}[ea : N/8]$ with b_{w}^*.

    d. Else:

       i. Perform the atomic action $(\mathsf{rmw}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b_{\mathrm{r}}^*\ b_{\mathrm{w}}^*)$ to read $N/8$ bytes b_{r}^* from data offset ea of the shared memory instance at memory address a and replace them with bytes b_{w}^*.

19. Else:

    a. If $mem.\mathsf{type}$ is $\mathit{limits}\ \mathsf{shared}$, then:

       i. Perform the action $(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b_{\mathrm{r}}^*)$ to read $N/8$ bytes b_{r}^* from data offset ea of the shared memory instance at memory address a.

20. Let c_1 be the result of computing $\mathrm{extend}^{\mathsf{u}}_{N,|t|}(c_{\mathrm{r}})$.

21. Push the value $t.\mathsf{const}\ c_1$ to the stack.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c_2)\ (t.\mathsf{const}\ c_3)\ (t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathsf{cmpxchg}\ \mathit{memarg}) \quad \hookrightarrow \quad S'; F; (t.\mathsf{const}\ (\mathrm{extend}^{\mathsf{u}}_{N,|t|}(c_{\mathrm{r}})))\\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared}\\
\qquad \wedge\ \mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}|\\
\qquad \wedge\ \mathit{ea} \bmod N/8 = 0\\
\qquad \wedge\ c_{\mathrm{r}} = \mathrm{bytes}_{\mathrm{i}N}^{-1}(\mathit{mem}.\mathsf{data}[\mathit{ea} : N/8])\\
\qquad \wedge\ c_{\mathrm{ex}} = \mathrm{wrap}_{|t|,N}(c_2)\\
\qquad \wedge\ ((c_{\mathrm{r}} = c_{\mathrm{ex}} \wedge c = \mathrm{wrap}_{|t|,N}(c_3)) \vee (c_{\mathrm{r}} \neq c_{\mathrm{ex}} \wedge c = c_{\mathrm{r}}))\\
\qquad \wedge\ S' = S\ \mathsf{with}\ \mathsf{mems}[a].\mathsf{data}[\mathit{ea} : N/8] = \mathrm{bytes}_{\mathrm{i}N}(c))
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c_2)\ (t.\mathsf{const}\ c_3)\ (t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathsf{cmpxchg}\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; \mathsf{trap}\\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{unshared}\\
\qquad \wedge\ \mathit{ea} + N/8 > |\mathit{mem}.\mathsf{data}| \vee \mathit{ea} \bmod N/8 \neq 0)
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c_2)\ (t.\mathsf{const}\ c_3)\ (t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathsf{cmpxchg}\ \mathit{memarg}) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rmw}\ a.\mathsf{data}[\mathit{ea}]\ b_{\mathrm{r}}^*\ b_{\mathrm{w}}^*)} \quad S;\\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared}\\
\qquad \wedge\ \mathit{ea} + N/8 \leq n\\
\qquad \wedge\ \mathit{ea} \bmod N/8 = 0\\
\qquad \wedge\ c_{\mathrm{r}} = \mathrm{bytes}_{\mathrm{i}N}^{-1}(b_{\mathrm{r}}^*)\\
\qquad \wedge\ c_{\mathrm{ex}} = \mathrm{wrap}_{|t|,N}(c_2)\\
\qquad \wedge\ c_{\mathrm{r}} = c_{\mathrm{ex}}\\
\qquad \wedge\ b_{\mathrm{w}}^* = \mathrm{bytes}_{\mathrm{i}N}(\mathrm{wrap}_{|t|,N}(c_3)))
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c_2)\ (t.\mathsf{const}\ c_3)\ (t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathsf{cmpxchg}\ \mathit{memarg}) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{data}[\mathit{ea}]\ b_{\mathrm{r}}^*)} \quad S; F\\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared}\\
\qquad \wedge\ \mathit{ea} + N/8 \leq n\\
\qquad \wedge\ \mathit{ea} \bmod N/8 = 0\\
\qquad \wedge\ c_{\mathrm{r}} = \mathrm{bytes}_{\mathrm{i}N}^{-1}(b_{\mathrm{r}}^*)\\
\qquad \wedge\ c_{\mathrm{ex}} = \mathrm{wrap}_{|t|,N}(c_2)\\
\qquad \wedge\ c_{\mathrm{r}} \neq c_{\mathrm{ex}}
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c_2)\ (t.\mathsf{const}\ c_3)\ (t.\mathsf{atomic.rmw}(N\_\mathsf{u})^?.\mathsf{cmpxchg}\ \mathit{memarg}) \quad \hookrightarrow^{(\mathsf{rd}\ a.\mathsf{len}\ n)} \quad S; F; \mathsf{trap}\\
\qquad (\mathrm{if}\ \mathit{mem}.\mathsf{type} = \mathit{limits}\ \mathsf{shared}\\
\qquad \wedge\ \mathit{ea} + N/8 > n \vee \mathit{ea} \bmod N/8 \neq 0)
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ N = |t|\ \mathrm{if}\ N\ \mathrm{not\ present}\\
\qquad \wedge\ a = F.\mathsf{module}.\mathsf{memaddrs}[0]\\
\qquad \wedge\ \mathit{mem} = S.\mathsf{mems}[a]\\
\qquad \wedge\ \mathit{ea} = i + \mathit{memarg}.\mathsf{offset})
\end{array}$$

memory.atomic.notify *memarg*

9. Let N be 32.

1. Assert: due to validation, a value of value type i32 is on the top of the stack.

2. Pop the value i32.const k from the stack.

6. Assert: due to validation, a value of value type i32 is on the top of the stack.

7. Pop the value i32.const i from the stack.

8. Let ea be $i + \mathit{memarg}.\mathsf{offset}$.

9. If ea modulo $N/8$ is not equal to 0, then:

   a. Trap.

10. Let F be the current frame.

11. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.

12. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.

13. Let mem be the memory instance $S.\mathsf{mems}[a]$.

14. If $\mathit{mem}.\mathsf{type}$ is limits $\mathsf{unshared}$, then:
    a. If $\mathit{ea} + N/8$ is larger than the length of $\mathit{mem}.\mathsf{data}$, then:
        i. Trap.
    b. Else:
        i. Push the value $t.\mathsf{const}\ 0$ to the stack.
15. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $\mathit{ea} + N/8$ is larger than n, then:
        i. Trap.
    c. Else:
        i. Perform the action $(\mathsf{notify}\ a.\mathsf{data}[\mathit{ea}]\ n\ k)$ to notify n threads (up to k) waiting at data offset ea of the shared memory instance at memory address a.
        ii. Push the value $t.\mathsf{const}\ n$ to the stack.

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}~i)~(\mathsf{i32.const}~k)~\mathsf{memory.atomic.notify}~\mathit{memarg} & \hookrightarrow & S; F; (\mathsf{i32.const}~0) \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \\
\qquad \wedge~\mathit{ea} + N/8 \leq |\mathit{mem}.\mathsf{data}| \\
\qquad \wedge~\mathit{ea} \bmod N/8 = 0)
\end{array}
$$

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}~i)~(\mathsf{i32.const}~k)~\mathsf{memory.atomic.notify}~\mathit{memarg} & \hookrightarrow & S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{unshared} \\
\qquad \wedge~(\mathit{ea} + N/8 > |\mathit{mem}.\mathsf{data}|~\vee~\mathit{ea} \bmod N/8 \neq 0))
\end{array}
$$

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}~i)~(\mathsf{i32.const}~k)~\mathsf{memory.atomic.notify}~\mathit{memarg} \\
\qquad \hookrightarrow^{(\mathsf{rd}~a.\mathsf{len}~n)~(\mathsf{notify}~a.\mathsf{data}[\mathit{ea}]~j~k)} \quad S; F; (\mathsf{i32.const}~j) \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{shared} \\
\qquad \wedge~\mathit{ea} + N/8 \leq n \\
\qquad \wedge~j \leq k \\
\qquad \wedge~\mathit{ea} \bmod N/8 = 0)
\end{array}
$$

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}~i)~(\mathsf{i32.const}~k)~\mathsf{memory.atomic.notify}~\mathit{memarg} & \hookrightarrow^{(\mathsf{rd}~a.\mathsf{len}~n)} & S; F; \mathsf{trap} \\
\qquad (\mathrm{if}~\mathit{mem}.\mathsf{type} = \mathit{limits}~\mathsf{shared} \\
\qquad \wedge~(\mathit{ea} + N/8 > n~\vee~\mathit{ea} \bmod N/8 \neq 0))
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{where}~a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\qquad \wedge~\mathit{mem} = S.\mathsf{mems}[a] \\
\qquad \wedge~\mathit{ea} = i + \mathit{memarg}.\mathsf{offset} \\
\qquad \wedge~N = 32)
\end{array}
$$

$\mathsf{memory.atomic.wait}N\ \mathit{memarg}$

1. Assert: due to validation, a value of value type $\mathsf{i64}$ is on the top of the stack.
2. Pop the value $\mathsf{i64}.\mathsf{const}\ k$ from the stack.
1. Assert: due to validation, a value of value type $\mathsf{i}N$ is on the top of the stack.
5. Pop the value $\mathsf{i}N.\mathsf{const}\ c$ from the stack.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32}.\mathsf{const}\ i$ from the stack.
8. Let ea be $i + \mathit{memarg}.\mathsf{offset}$.
9. If ea modulo $N/8$ is not equal to 0, then:

   a. Trap.

10. Let F be the current frame.
11. Assert: due to validation, $F.\mathsf{module}.\mathsf{memaddrs}[0]$ exists.
12. Let a be the memory address $F.\mathsf{module}.\mathsf{memaddrs}[0]$.
13. Let mem be the memory instance $S.\mathsf{mems}[a]$.
14. If $mem.\mathsf{type}$ is $limits$ $\mathsf{unshared}$, then:
    a. Trap.
15. Else:
    a. Perform the action $(\mathsf{rd}\ a.\mathsf{len}\ n)$ to read the length n of the shared memory instance at memory address a.
    b. If $ea + N/8$ is larger than n, then:
       i. Trap.
    c. Perform the action $(\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b^*)$ to read $N/8$ bytes b^* from data offset ea of the shared memory instance at memory address a.
    d. If $\mathrm{bytes}_{iN}(c)$ is equal to b^* then:
       i. Let t be $\mathrm{signed}_N(k)$.
       ii. Perform the action $(\mathsf{wait}\ a.\mathsf{data}[ea]\ t)$ to register the current thread as waiting for a signal at data offset ea of the shared memory instance at memory address a.
       iii. Execute the instruction $\mathsf{wait}'\ a.\mathsf{data}[ea]\ k$.
    e. Else:
       i. Push the value $t.\mathsf{const}\ 1$ to the stack.

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (iN.\mathsf{const}\ c)\ (\mathsf{i64.const}\ k)\ \mathsf{memory.atomic.wait}N\ memarg \\
\qquad \hookrightarrow \quad S; F; \mathsf{trap} \\
\quad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{unshared})
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (iN.\mathsf{const}\ c)\ (\mathsf{i64.const}\ k)\ \mathsf{memory.atomic.wait}N\ memarg \\
\qquad \hookrightarrow_{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b^*)\ (\mathsf{wait}\ a.\mathsf{data}[ea]\ t)} \quad S; F; (\mathsf{wait}'\ a.\mathsf{data}[ea]\ k) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared} \\
\quad \wedge\ ea + N/8 \leq n \\
\quad \wedge\ ea \bmod N/8 = 0 \\
\quad \wedge\ b^* = \mathrm{bytes}_{iN}(c)) \\
\quad \wedge\ t = \mathrm{signed}_N(k)
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (iN.\mathsf{const}\ c)\ (\mathsf{i64.const}\ k)\ \mathsf{memory.atomic.wait}N\ memarg \\
\qquad \hookrightarrow_{(\mathsf{rd}\ a.\mathsf{len}\ n)\ (\mathsf{rd}_{\mathsf{seqcst}}\ a.\mathsf{data}[ea]\ b^*)} \quad S; F; (\mathsf{i32.const}\ 1) \\
\quad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared} \\
\quad \wedge\ ea + N/8 \leq n \\
\quad \wedge\ ea \bmod N/8 = 0 \\
\quad \wedge\ b^* \neq \mathrm{bytes}_{iN}(c))
\end{array}$$

$$\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (iN.\mathsf{const}\ c)\ (\mathsf{i64.const}\ k)\ \mathsf{memory.atomic.wait}N\ memarg \\
\qquad \hookrightarrow_{(\mathsf{rd}\ a.\mathsf{len}\ n)} \quad S; F; \mathsf{trap} \\
\quad (\mathrm{if}\ mem.\mathsf{type} = limits\ \mathsf{shared} \\
\quad \wedge\ (ea + N/8 > n\ \vee\ ea \bmod N/8 \neq 0))
\end{array}$$

$$\begin{array}{l}
(\mathrm{where}\ a = F.\mathsf{module}.\mathsf{memaddrs}[0] \\
\quad \wedge\ mem = S.\mathsf{mems}[a] \\
\quad \wedge\ ea = i + memarg.\mathsf{offset})
\end{array}$$

$\mathsf{wait}'\ loc\ n$

1. Either the thread has been notified by an $(\mathsf{notify}\ loc)$ action performed in another thread:
   a. Perform the action $(\mathsf{woken}\ loc)$.
   b. Push the value $t.\mathsf{const}\ 0$ to the stack.
2. Or:
   a. If n is less than 0:
      i. The thread remains suspended.
   b. Else:
      i. Either the thread's suspension times out:
         a. Perform the action $(\mathsf{timeout}\ loc)$.
         b. Push the value $t.\mathsf{const}\ 2$ to the stack.
      ii. Or the thread remains suspended.

$$
\begin{array}{lcl}
\mathsf{wait}'\ loc\ n & \hookrightarrow^{(\mathsf{woken}\ loc)} & (\mathsf{i32}.\mathsf{const}\ 0) \\
\mathsf{wait}'\ loc\ n & \hookrightarrow^{(\mathsf{timeout}\ loc)} & (\mathsf{i32}.\mathsf{const}\ 2) \\
\quad (\mathrm{if}\ n \geq 0) & & \\
\mathsf{wait}'\ loc\ n & \hookrightarrow & \mathsf{wait}'\ loc\ n
\end{array}
$$

$\mathsf{atomic.fence}$

1. Perform the action $(\mathsf{fence}_{\mathsf{seqcst}})$.

$$
\mathsf{atomic.fence} \quad \hookrightarrow^{(\mathsf{fence}_{\mathsf{seqcst}})} \quad \epsilon
$$

4.4.9 Control Instructions

nop

1. Do nothing.

$$
\mathsf{nop} \quad \hookrightarrow \quad \epsilon
$$

$\mathsf{unreachable}$

1. Trap.

$$
\mathsf{unreachable} \quad \hookrightarrow \quad \mathsf{trap}
$$

$\mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end}$

1. Let F be the current frame.
2. Assert: due to validation, $\mathrm{expand}_F(\mathit{blocktype})$ is defined.
3. Let $[t_1^m] \rightarrow [t_2^n]$ be the function type $\mathrm{expand}_F(\mathit{blocktype})$.
4. Let L be the label whose arity is n and whose continuation is the end of the block.
5. Assert: due to validation, there are at least m values on the top of the stack.
6. Pop the values val^m from the stack.
7. Enter the block $\mathit{val}^m\ \mathit{instr}^*$ with label L.

$$\begin{array}{lcl} F; \mathit{val}^m\ \mathsf{block}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end} & \hookrightarrow & F; \mathsf{label}_n\{\epsilon\}\ \mathit{val}^m\ \mathit{instr}^*\ \mathsf{end} \\ & & \quad (\mathrm{if}\ \mathrm{expand}_F(\mathit{bt}) = [t_1^m] \rightarrow [t_2^n]) \end{array}$$

$\mathsf{loop}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end}$

1. Let F be the current frame.
2. Assert: due to validation, $\mathrm{expand}_F(\mathit{blocktype})$ is defined.
3. Let $[t_1^m] \rightarrow [t_2^n]$ be the function type $\mathrm{expand}_F(\mathit{blocktype})$.
4. Let L be the label whose arity is m and whose continuation is the start of the loop.
5. Assert: due to validation, there are at least m values on the top of the stack.
6. Pop the values val^m from the stack.
7. Enter the block $\mathit{val}^m\ \mathit{instr}^*$ with label L.

$$\begin{array}{lcl} F; \mathit{val}^m\ \mathsf{loop}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end} & \hookrightarrow & F; \mathsf{label}_m\{\mathsf{loop}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end}\}\ \mathit{val}^m\ \mathit{instr}^*\ \mathsf{end} \\ & & \quad (\mathrm{if}\ \mathrm{expand}_F(\mathit{bt}) = [t_1^m] \rightarrow [t_2^n]) \end{array}$$

$\mathsf{if}\ \mathit{blocktype}\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*\ \mathsf{end}$

1. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $\mathsf{i32.const}\ c$ from the stack.
3. If c is non-zero, then:
    a. Execute the block instruction $\mathsf{block}\ \mathit{blocktype}\ \mathit{instr}_1^*\ \mathsf{end}$.
4. Else:
    a. Execute the block instruction $\mathsf{block}\ \mathit{blocktype}\ \mathit{instr}_2^*\ \mathsf{end}$.

$$\begin{array}{lcl} (\mathsf{i32.const}\ c)\ \mathsf{if}\ \mathit{bt}\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ \mathit{bt}\ \mathit{instr}_1^*\ \mathsf{end} \\ & & \quad (\mathrm{if}\ c \neq 0) \\ (\mathsf{i32.const}\ c)\ \mathsf{if}\ \mathit{bt}\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ \mathit{bt}\ \mathit{instr}_2^*\ \mathsf{end} \\ & & \quad (\mathrm{if}\ c = 0) \end{array}$$

br l

1. Assert: due to validation, the stack contains at least $l+1$ labels.
2. Let L be the l-th label appearing on the stack, starting from the top and counting from zero.
3. Let n be the arity of L.
4. Assert: due to validation, there are at least n values on the top of the stack.
5. Pop the values val^n from the stack.
6. Repeat $l+1$ times:
   a. While the top of the stack is a value, do:
      i. Pop the value from the stack.
   b. Assert: due to validation, the top of the stack now is a label.
   c. Pop the label from the stack.
7. Push the values val^n to the stack.
8. Jump to the continuation of L.

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

br_if l

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. If c is non-zero, then:
   a. Execute the instruction br l.
4. Else:
   a. Do nothing.

$$\begin{array}{lcll}(\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & (\mathrm{if}\ c \neq 0)\\ (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & (\mathrm{if}\ c = 0)\end{array}$$

br_table l^* l_N

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const i from the stack.
3. If i is smaller than the length of l^*, then:
   a. Let l_i be the label $l^*[i]$.
   b. Execute the instruction br l_i.
4. Else:
   a. Execute the instruction br l_N.

$$\begin{array}{lcll}(\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_i) & (\mathrm{if}\ l^*[i] = l_i)\\ (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_N) & (\mathrm{if}\ |l^*| \leq i)\end{array}$$

return

1. Let F be the current frame.
2. Let n be the arity of F.
3. Assert: due to validation, there are at least n values on the top of the stack.
4. Pop the results val^n from the stack.
5. Assert: due to validation, the stack contains at least one frame.
6. While the top of the stack is not a frame, do:
   a. Pop the top element from the stack.
7. Assert: the top of the stack is the frame F.
8. Pop the frame from the stack.
9. Push val^n to the stack.
10. Jump to the instruction after the original call that pushed the frame.

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

call x

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{funcaddrs}[x]$ exists.
3. Let a be the function address $F.\mathsf{module}.\mathsf{funcaddrs}[x]$.
4. Invoke the function instance at address a.

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

call_indirect x y

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{tableaddrs}[x]$ exists.
3. Let ta be the table address $F.\mathsf{module}.\mathsf{tableaddrs}[x]$.
4. Assert: due to validation, $S.\mathsf{tables}[ta]$ exists.
5. Let tab be the table instance $S.\mathsf{tables}[ta]$.
6. Assert: due to validation, $F.\mathsf{module}.\mathsf{types}[y]$ exists.
7. Let ft_{expect} be the function type $F.\mathsf{module}.\mathsf{types}[y]$.
8. Assert: due to validation, a value with value type i32 is on the top of the stack.
9. Pop the value $\mathsf{i32.const}\ i$ from the stack.
10. If i is not smaller than the length of $tab.\mathsf{elem}$, then:
    a. Trap.
11. Let r be the reference $tab.\mathsf{elem}[i]$.
12. If r is $\mathsf{ref.null}\ t$, then:
    a. Trap.
13. Assert: due to validation of table mutation, r is a function reference.

14. Let $\mathsf{ref}\ a$ be the function reference r.
15. Assert: due to validation of table mutation, $S.\mathsf{funcs}[a]$ exists.
16. Let f be the function instance $S.\mathsf{funcs}[a]$.
17. Let $\mathit{ft}_{\mathrm{actual}}$ be the function type $f.\mathsf{type}$.
18. If $\mathit{ft}_{\mathrm{actual}}$ and $\mathit{ft}_{\mathrm{expect}}$ differ, then:
    a. Trap.
19. Invoke the function instance at address a.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call\_indirect}\ x\ y) & \hookrightarrow & S; F; (\mathsf{invoke}\ a) \\
\qquad (\mathrm{if}\ S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}[i] = \mathsf{ref}\ a & & \\
\qquad\ \wedge\ S.\mathsf{funcs}[a] = f & & \\
\qquad\ \wedge\ F.\mathsf{module}.\mathsf{types}[y] = f.\mathsf{type}) & & \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call\_indirect}\ x\ y) & \hookrightarrow & S; F; \mathsf{trap} \\
\qquad (\mathrm{otherwise}) & &
\end{array}$$

4.4.10 Blocks

The following auxiliary rules define the semantics of executing an instruction sequence that forms a block.

Entering instr^* with label L

1. Push L to the stack.
2. Jump to the start of the instruction sequence instr^*.

Note: No formal reduction rule is needed for entering an instruction sequence, because the label L is embedded in the administrative instruction that structured control instructions reduce to directly.

Exiting instr^* with label L

When the end of a block is reached without a jump or trap aborting it, then the following steps are performed.

1. Pop all values val^* from the top of the stack.
2. Assert: due to validation, the label L is now on the top of the stack.
3. Pop the label from the stack.
4. Push val^* back to the stack.
5. Jump to the position after the end of the structured control instruction associated with the label L.

$$\mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{val}^*\ \mathsf{end} \quad \hookrightarrow \quad \mathit{val}^*$$

Note: This semantics also applies to the instruction sequence contained in a loop instruction. Therefore, execution of a loop falls off the end, unless a backwards branch is performed explicitly.

4.4.11 Function Calls

The following auxiliary rules define the semantics of invoking a function instance through one of the call instructions and returning from it.

Invocation of function address a

1. Assert: due to validation, $S.\mathsf{funcs}[a]$ exists.
2. Let f be the function instance, $S.\mathsf{funcs}[a]$.
3. Let $[t_1^n] \rightarrow [t_2^m]$ be the function type $f.\mathsf{type}$.
4. Let t^* be the list of value types $f.\mathsf{code}.\mathsf{locals}$.
5. Let $\mathit{instr}^*\ \mathsf{end}$ be the expression $f.\mathsf{code}.\mathsf{body}$.
6. Assert: due to validation, n values are on the top of the stack.
7. Pop the values val^n from the stack.
8. Let F be the frame $\{\mathsf{module}\ f.\mathsf{module}, \mathsf{locals}\ \mathit{val}^n\ (\mathrm{default}_t)^*\}$.
9. Push the activation of F with arity m to the stack.
10. Let L be the label whose arity is m and whose continuation is the end of the function.
11. Enter the instruction sequence instr^* with label L.

$$\begin{array}{l}S; \mathit{val}^n\ (\mathsf{invoke}\ a) \quad \hookrightarrow \quad S; \mathsf{frame}_m\{F\}\ \mathsf{label}_m\{\}\ \mathit{instr}^*\ \mathsf{end}\ \mathsf{end} \\ \qquad (\mathrm{if}\ S.\mathsf{funcs}[a] = f \\ \qquad \wedge\ f.\mathsf{type} = [t_1^n] \rightarrow [t_2^m] \\ \qquad \wedge\ f.\mathsf{code} = \{\mathsf{type}\ x, \mathsf{locals}\ t^k, \mathsf{body}\ \mathit{instr}^*\ \mathsf{end}\} \\ \qquad \wedge\ F = \{\mathsf{module}\ f.\mathsf{module},\ \mathsf{locals}\ \mathit{val}^n\ (\mathrm{default}_t)^k\})\end{array}$$

Returning from a function

When the end of a function is reached without a jump (i.e., return) or trap aborting it, then the following steps are performed.

1. Let F be the current frame.
2. Let n be the arity of the activation of F.
3. Assert: due to validation, there are n values on the top of the stack.
4. Pop the results val^n from the stack.
5. Assert: due to validation, the frame F is now on the top of the stack.
6. Pop the frame from the stack.
7. Push val^n back to the stack.
8. Jump to the instruction after the original call.

$$\mathsf{frame}_n\{F\}\ \mathit{val}^n\ \mathsf{end} \quad \hookrightarrow \quad \mathit{val}^n$$

Host Functions

Invoking a host function has non-deterministic behavior. It may either terminate with a trap or return regularly. However, in the latter case, it must consume and produce the right number and types of WebAssembly values on the stack, according to its function type.

A host function may also modify the store. However, all store modifications must result in an extension of the original store, i.e., they must only modify mutable contents and must not have instances removed. Furthermore, the resulting store must be valid, i.e., all data and code in it is well-typed.

$$\begin{array}{lcl} S; \mathit{val}^n~(\mathsf{invoke}~a) & \hookrightarrow & S; \mathsf{host}~[t_2^m] \\ \quad (\mathrm{if}~S.\mathsf{funcs}[a] = \{\mathsf{type}~[t_1^n] \rightarrow [t_2^m], \mathsf{hostcode}~\mathit{hf}\}) \end{array}$$

During its execution, a host function call may do any of the following.

- Instantiate an arbitrary WebAssembly module.
- Allocate a new host function.
- Continue execution, possibly spawning a new thread, with no other observable effects.
- Terminate with a list of values that respects the host function's type annotation.
- Terminate with a trap.

$$\begin{array}{lcl} S; F; (\mathsf{host}~[t^*]) & \hookrightarrow & S'; F; (\mathsf{frame}_0\{F_0\}~e^*~\mathsf{end})~(\mathsf{host}~[t^*]) \\ \quad (\mathrm{if}~\mathrm{instantiate}(S, \mathit{module}, \mathit{externval}^k) = S'; F_0; e^*) \\ S; F; (\mathsf{host}~[t^*]) & \hookrightarrow & S'; F; (\mathsf{host}~[t^*]) \\ \quad (\mathrm{if}~\mathrm{allochostfunc}(S, \mathit{functype}, \mathit{hostfunc}) = S', \mathit{funcaddr}) \\ S; F; (\mathsf{host}~[t^*]) & \hookrightarrow^{\mathsf{spawn}^?} & S; F; (\mathsf{host}~[t^*]) \\ S; F; (\mathsf{host}~[t^*]) & \hookrightarrow & S; F; \mathit{val}^* \\ \quad (\mathrm{if}~S \vdash \mathit{val}^* : [t^*]) \\ S; F; (\mathsf{host}~[t^*]) & \hookrightarrow & S; F; \mathsf{trap} \end{array}$$

Note: A host function can call back into WebAssembly by invoking a function exported from a module. However, the effects of any such call are subsumed by the non-deterministic behavior allowed for the host function.

4.4.12 Expressions

An expression is *evaluated* relative to a current frame pointing to its containing module instance.

1. Jump to the start of the instruction sequence instr^* of the expression.
2. Execute the instruction sequence.
3. Assert: due to validation, the top of the stack contains a value.
4. Pop the value val from the stack.

The value val is the result of the evaluation.

$$S; F; \mathit{instr}^* \hookrightarrow^{\mathit{act}^*} S'; F'; \mathit{instr}'^* \qquad (\mathrm{if}~S; F; \mathit{instr}^*~\mathsf{end} \hookrightarrow^{\mathit{act}^*} S'; F'; \mathit{instr}'^*~\mathsf{end})$$

Note: Evaluation iterates this reduction rule until reaching a value. Expressions constituting function bodies are executed during function invocation.

4.5 Modules

For modules, the execution semantics primarily defines instantiation, which allocates instances for a module and its contained definitions, initializes tables and memories from contained element and data segments, and invokes the start function if present. It also includes invocation of exported functions.

Instantiation depends on a number of auxiliary notions for type-checking imports and allocating instances.

4.5.1 External Typing

For the purpose of checking external values against imports, such values are classified by external types. The following auxiliary typing rules specify this typing relation relative to a store S in which the referenced instances live.

$\mathsf{func}~a$

- The store entry $S.\mathsf{funcs}[a]$ must exist.
- Then $\mathsf{func}~a$ is valid with external type $\mathsf{func}~S.\mathsf{funcs}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{func}~a : \mathsf{func}~S.\mathsf{funcs}[a].\mathsf{type}}$$

$\mathsf{table}~a$

- The store entry $S.\mathsf{tables}[a]$ must exist.
- Then $\mathsf{table}~a$ is valid with external type $\mathsf{table}~S.\mathsf{tables}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{table}~a : \mathsf{table}~S.\mathsf{tables}[a].\mathsf{type}}$$

$\mathsf{mem}~a$

- The store entry $S.\mathsf{mems}[a]$ must exist.
- If $S.\mathsf{mems}[a].\mathsf{type}$ is $\mathit{limits}~\mathsf{unshared}$, then:
  - $\mathsf{mem}~a$ is valid with external type $\mathsf{mem}~(\mathit{limits}~\mathsf{unshared})$.
- Else:
  - Assert: $S.\mathsf{mems}[a].\mathsf{type}$ is $\mathit{limits}~\mathsf{shared}$.
  - Let $\{\mathsf{min}~x, \mathsf{max}~y^?\}~\mathsf{shared}$ be the memory type $S.\mathsf{mems}[a].\mathsf{type}$.
  - Assert: the current length of $S.\mathsf{mems}[a]$ in bytes, l, is given by the action $(\mathsf{rd}_{\mathsf{seqcst}}~a.\mathsf{len}~l)$.
  - $\mathsf{mem}~a$ is valid with external type $\mathsf{mem}~(\{\mathsf{min}~l, \mathsf{max}~y^?\}~\mathsf{shared})$.

$$\frac{S.\mathsf{mems}[a].\mathsf{type} = \mathit{limits}~\mathsf{unshared}}{S \vdash \mathsf{mem}~a : \mathsf{mem}~(\mathit{limits}~\mathsf{unshared})}$$

$$\frac{S.\mathsf{mems}[a].\mathsf{type} = \{\mathsf{min}~x, \mathsf{max}~y^?\}~\mathsf{shared}}{S; (\mathsf{rd}_{\mathsf{seqcst}}~a.\mathsf{len}~l) \vdash \mathsf{mem}~a : \mathsf{mem}~(\{\mathsf{min}~l, \mathsf{max}~y^?\}~\mathsf{shared})}$$

Note: In the shared case, the associated action is performed as part of instantiation, in order to record in the Relaxed Memory Model that a sequentially consistent bounds check on the shared memory has occurred.

global a

- The store entry $S.\mathsf{globals}[a]$ must exist.
- Then global a is valid with external type global $S.\mathsf{globals}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{global}\ a : \mathsf{global}\ S.\mathsf{globals}[a].\mathsf{type}}$$

4.5.2 Value Typing

For the purpose of checking argument values against the parameter types of exported functions, values are classified by value types. The following auxiliary typing rules specify this typing relation relative to a store S in which possibly referenced addresses live.

Numeric Values $t.\mathsf{const}\ c$

- The value is valid with number type t.

$$\frac{}{S \vdash t.\mathsf{const}\ c : t}$$

Null References $\mathsf{ref.null}\ t$

- The value is valid with reference type t.

$$\frac{}{S \vdash \mathsf{ref.null}\ t : t}$$

Function References $\mathsf{ref}\ a$

- The external value func a must be valid.
- Then the value is valid with reference type funcref.

$$\frac{S \vdash \mathsf{func}\ a : \mathsf{func}\ \mathit{functype}}{S \vdash \mathsf{ref}\ a : \mathsf{funcref}}$$

External References $\mathsf{ref.extern}\ a$

- The value is valid with reference type externref.

$$\frac{}{S \vdash \mathsf{ref.extern}\ a : \mathsf{externref}}$$

4.5.3 Allocation

New instances of functions, tables, memories, and globals are *allocated* in a store S, as defined by the following auxiliary functions.

Functions

1. Let func be the function to allocate and $\mathit{moduleinst}$ its module instance.
2. Let a be the first free function address in S.
3. Let $\mathit{functype}$ be the function type $\mathit{moduleinst}.\mathsf{types}[\mathit{func}.\mathsf{type}]$.
4. Let $\mathit{funcinst}$ be the function instance $\{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\}$.
5. Append $\mathit{funcinst}$ to the funcs of S.
6. Return a.

$$\begin{array}{rcl}
\mathrm{allocfunc}(S, \mathit{func}, \mathit{moduleinst}) &=& S', \mathit{funcaddr} \\
\mbox{where:} \hfill \\
\mathit{funcaddr} &=& |S.\mathsf{funcs}| \\
\mathit{functype} &=& \mathit{moduleinst}.\mathsf{types}[\mathit{func}.\mathsf{type}] \\
\mathit{funcinst} &=& \{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\} \\
S' &=& S \oplus \{\mathsf{funcs}~\mathit{funcinst}\}
\end{array}$$

Host Functions

1. Let $\mathit{hostfunc}$ be the host function to allocate and $\mathit{functype}$ its function type.
2. Let a be the first free function address in S.
3. Let $\mathit{funcinst}$ be the function instance $\{\mathsf{type}~\mathit{functype}, \mathsf{hostcode}~\mathit{hostfunc}\}$.
4. Append $\mathit{funcinst}$ to the funcs of S.
5. Return a.

$$\begin{array}{rcl}
\mathrm{allochostfunc}(S, \mathit{functype}, \mathit{hostfunc}) &=& S', \mathit{funcaddr} \\
\mbox{where:} \hfill \\
\mathit{funcaddr} &=& |S.\mathsf{funcs}| \\
\mathit{funcinst} &=& \{\mathsf{type}~\mathit{functype}, \mathsf{hostcode}~\mathit{hostfunc}\} \\
S' &=& S \oplus \{\mathsf{funcs}~\mathit{funcinst}\}
\end{array}$$

Note: Host functions are never allocated by the WebAssembly semantics itself, but may be allocated by the embedder.

Tables

1. Let $\mathit{tabletype}$ be the table type to allocate and ref the initialization value.
2. Let $(\{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathit{reftype})$ be the structure of table type $\mathit{tabletype}$.
3. Let a be the first free table address in S.
4. Let $\mathit{tableinst}$ be the table instance $\{\mathsf{type}~\mathit{tabletype}, \mathsf{elem}~\mathit{ref}^n\}$ with n elements set to ref.
5. Append $\mathit{tableinst}$ to the tables of S.
6. Return a.

$$\begin{array}{rcl}
\mathrm{alloctable}(S, \mathit{tabletype}, \mathit{ref}) &=& S', \mathit{tableaddr} \\
\text{where:} && \\
\mathit{tabletype} &=& \{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathit{reftype} \\
\mathit{tableaddr} &=& |S.\mathsf{tables}| \\
\mathit{tableinst} &=& \{\mathsf{type}~\mathit{tabletype}, \mathsf{elem}~\mathit{ref}^n\} \\
S' &=& S \oplus \{\mathsf{tables}~\mathit{tableinst}\}
\end{array}$$

Memories

1. Let $\mathit{memtype}$ be the memory type to allocate.
2. Let $\{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathit{share}$ be the structure of memory type $\mathit{memtype}$.
3. Let a be the first free memory address in S.
4. If share is unshared:
   a. Let $\mathit{meminst}$ be the memory instance $\{\mathsf{type}~\mathit{memtype}, \mathsf{data}~(\mathtt{0x00})^{n\cdot 64\,\mathrm{Ki}}\}$ that contains n pages of zeroed bytes.
   b. Let act^* be ϵ.
5. Else:
   a. Let $\mathit{meminst}$ be the memory instance $\{\mathsf{type}~\mathit{memtype}\}$.
   b. Let act^* be $(\mathsf{wr}_{\mathsf{seqcst}}~a.\mathsf{len}~n)(\mathsf{wr}_{\mathsf{init}}~a.\mathsf{data}[0]~(0)^{n\cdot 64})$.
5. Append $\mathit{meminst}$ to the mems of S.
6. Return a, act^*.

$$\begin{array}{rcl}
\mathrm{allocmem}(S, \mathit{memtype}) &=& S', \mathit{memaddr}, \epsilon \\
\text{if} && \\
\mathit{memtype} &=& \{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathsf{unshared} \\
\mathit{memaddr} &=& |S.\mathsf{mems}| \\
\mathit{meminst} &=& \{\mathsf{type}~\mathit{memtype}, \mathsf{data}~(\mathtt{0x00})^{n\cdot 64\,\mathrm{Ki}}\} \\
S' &=& S \oplus \{\mathsf{mems}~\mathit{meminst}\}
\end{array}$$

$$\begin{array}{rcl}
\mathrm{allocmem}(S, \mathit{memtype}) &=& S', a, (\mathsf{wr}_{\mathsf{seqcst}}~a.\mathsf{len}~n)(\mathsf{wr}_{\mathsf{init}}~a.\mathsf{data}[0]~(0)^{n\cdot 64}) \\
\text{if} && \\
\mathit{memtype} &=& \{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathsf{shared} \\
a &=& |S.\mathsf{mems}| \\
\mathit{meminst} &=& \{\mathsf{type}~\mathit{memtype}\} \\
S' &=& S \oplus \{\mathsf{mems}~\mathit{meminst}\}
\end{array}$$

Globals

1. Let $\mathit{globaltype}$ be the global type to allocate and val the value to initialize the global with.
2. Let a be the first free global address in S.
3. Let $\mathit{globalinst}$ be the global instance $\{\mathsf{type}~\mathit{globaltype}, \mathsf{value}~\mathit{val}\}$.
4. Append $\mathit{globalinst}$ to the globals of S.
5. Return a.

$$\begin{array}{rcl}
\mathrm{allocglobal}(S, \mathit{globaltype}, \mathit{val}) &=& S', \mathit{globaladdr} \\
\text{where:} && \\
\mathit{globaladdr} &=& |S.\mathsf{globals}| \\
\mathit{globalinst} &=& \{\mathsf{type}~\mathit{globaltype}, \mathsf{value}~\mathit{val}\} \\
S' &=& S \oplus \{\mathsf{globals}~\mathit{globalinst}\}
\end{array}$$

Element segments

1. Let $\mathit{reftype}$ be the elements' type and ref^* the vector of references to allocate.
2. Let a be the first free element address in S.
3. Let $\mathit{eleminst}$ be the element instance $\{\mathsf{type}~\mathit{reftype}, \mathsf{elem}~\mathit{ref}^*\}$.
4. Append $\mathit{eleminst}$ to the elems of S.
5. Return a.

$$\begin{array}{rcl}
\mathrm{allocelem}(S, \mathit{reftype}, \mathit{ref}^*) &=& S', \mathit{elemaddr} \\[1ex]
\mbox{where:} \\
\mathit{elemaddr} &=& |S.\mathsf{elems}| \\
\mathit{eleminst} &=& \{\mathsf{type}~\mathit{reftype}, \mathsf{elem}~\mathit{ref}^*\} \\
S' &=& S \oplus \{\mathsf{elems}~\mathit{eleminst}\}
\end{array}$$

Data segments

1. Let b^* be the vector of bytes to allocate.
2. Let a be the first free data address in S.
3. Let $\mathit{datainst}$ be the data instance $\{\mathsf{data}~b^*\}$.
4. Append $\mathit{datainst}$ to the datas of S.
5. Return a.

$$\begin{array}{rcl}
\mathrm{allocdata}(S, b^*) &=& S', \mathit{dataaddr} \\[1ex]
\mbox{where:} \\
\mathit{dataaddr} &=& |S.\mathsf{datas}| \\
\mathit{datainst} &=& \{\mathsf{data}~b^*\} \\
S' &=& S \oplus \{\mathsf{datas}~\mathit{datainst}\}
\end{array}$$

Growing tables

1. Let $\mathit{tableinst}$ be the table instance to grow, n the number of elements by which to grow it, and ref the initialization value.
2. Let len be n added to the length of $\mathit{tableinst}.\mathsf{elem}$.
3. If len is larger than or equal to 2^{32}, then fail.
4. Let $\mathit{limits}~t$ be the structure of table type $\mathit{tableinst}.\mathsf{type}$.
5. Let limits' be limits with min updated to len.
6. If limits' is not valid, then fail.
7. Append ref^n to $\mathit{tableinst}.\mathsf{elem}$.
8. Set $\mathit{tableinst}.\mathsf{type}$ to the table type $\mathit{limits}'~t$.

$$\begin{array}{rcl}
\mathrm{growtable}(\mathit{tableinst}, n, \mathit{ref}) &=& \mathit{tableinst}~\mathsf{with}~\mathsf{type} = \mathit{limits}'~t~\mathsf{with}~\mathsf{elem} = \mathit{tableinst}.\mathsf{elem}~\mathit{ref}^n \\
&& (\mbox{if}~\mathit{len} = n + |\mathit{tableinst}.\mathsf{elem}| \\
&& \wedge~\mathit{len} < 2^{32} \\
&& \wedge~\mathit{limits}~t = \mathit{tableinst}.\mathsf{type} \\
&& \wedge~\mathit{limits}' = \mathit{limits}~\mathsf{with}~\mathsf{min} = \mathit{len} \\
&& \wedge \vdash \mathit{limits}'~\mbox{ok})
\end{array}$$

Growing memories

1. Let $\mathit{meminst}$ be the memory instance to grow and n the number of pages by which to grow it.
2. Assert: The length of $\mathit{meminst}.\mathsf{data}$ is divisible by the page size $64\,\mathrm{Ki}$.
3. Let len be n added to the length of $\mathit{meminst}.\mathsf{data}$ divided by the page size $64\,\mathrm{Ki}$.
4. If len is larger than 2^{16}, then fail.
5. Let limits be the structure of memory type $\mathit{meminst}.\mathsf{type}$.
6. Let limits' be limits with min updated to len.
7. If limits' is not valid, then fail.
8. Append n times $64\,\mathrm{Ki}$ bytes with value $\mathtt{0x00}$ to $\mathit{meminst}.\mathsf{data}$.
9. Set $\mathit{meminst}.\mathsf{type}$ to the memory type limits'.

$$\begin{array}{lcl}\mathrm{growmem}(\mathit{meminst}, n) & = & \mathit{meminst}~\mathsf{with}~\mathsf{type} = \mathit{limits}'~\mathsf{with}~\mathsf{data} = \mathit{meminst}.\mathsf{data}~(\mathtt{0x00})^{n\cdot 64\,\mathrm{Ki}} \\ && (\mathrm{if}~\mathit{len} = n + |\mathit{meminst}.\mathsf{data}|/64\,\mathrm{Ki} \\ && \wedge~\mathit{len} \leq 2^{16} \\ && \wedge~\mathit{limits}~\mathsf{unshared} = \mathit{meminst}.\mathsf{type} \\ && \wedge~\mathit{limits}' = \mathit{limits}~\mathsf{with}~\mathsf{min} = \mathit{len} \\ && \wedge \vdash \mathit{limits}'~\mathrm{ok})\end{array}$$

Modules

The allocation function for modules requires a suitable list of external values that are assumed to match the import vector of the module, a list of initialization values for the module's globals, and list of reference vectors for the module's element segments.

1. Let module be the module to allocate and $\mathit{externval}^*_{\mathrm{im}}$ the vector of external values providing the module's imports, val^* the initialization values of the module's globals, and $(\mathit{ref}^*)^*$ the reference vectors of the module's element segments.
2. For each function func_i in $\mathit{module}.\mathsf{funcs}$, do:
    a. Let $\mathit{funcaddr}_i$ be the function address resulting from allocating func_i for the module instance $\mathit{moduleinst}$ defined below.
3. For each table table_i in $\mathit{module}.\mathsf{tables}$, do:
    a. Let $\mathit{limits}_i~t_i$ be the table type $\mathit{table}_i.\mathsf{type}$.
    b. Let $\mathit{tableaddr}_i$ be the table address resulting from allocating $\mathit{table}_i.\mathsf{type}$ with initialization value $\mathsf{ref.null}~t_i$.
4. For each memory mem_i in $\mathit{module}.\mathsf{mems}$, do:
    a. Let $\mathit{memaddr}_i$, act_i^* be the memory address resulting from allocating $\mathit{mem}_i.\mathsf{type}$.
5. For each global global_i in $\mathit{module}.\mathsf{globals}$, do:
    a. Let $\mathit{globaladdr}_i$ be the global address resulting from allocating $\mathit{global}_i.\mathsf{type}$ with initializer value $\mathit{val}^*[i]$.
6. For each element segment elem_i in $\mathit{module}.\mathsf{elems}$, do:
    a. Let $\mathit{elemaddr}_i$ be the element address resulting from allocating an element instance of reference type $\mathit{elem}_i.\mathsf{type}$ with contents $(\mathit{ref}^*)^*[i]$.
7. For each data segment data_i in $\mathit{module}.\mathsf{datas}$, do:
    a. Let $\mathit{dataaddr}_i$ be the data address resulting from allocating a data instance with contents $\mathit{data}_i.\mathsf{init}$.
8. Let $\mathit{funcaddr}^*$ be the concatenation of the function addresses $\mathit{funcaddr}_i$ in index order.

9. Let $\mathit{tableaddr}^*$ be the concatenation of the table addresses $\mathit{tableaddr}_i$ in index order.
10. Let $\mathit{memaddr}^*$ be the concatenation of the memory addresses $\mathit{memaddr}_i$ in index order.
11. Let act^* be the concatenation of the actions act_i^* in index order.
11. Let $\mathit{globaladdr}^*$ be the concatenation of the global addresses $\mathit{globaladdr}_i$ in index order.
12. Let $\mathit{elemaddr}^*$ be the concatenation of the element addresses $\mathit{elemaddr}_i$ in index order.
13. Let $\mathit{dataaddr}^*$ be the concatenation of the data addresses $\mathit{dataaddr}_i$ in index order.
14. Let $\mathit{funcaddr}^*_{\mathrm{mod}}$ be the list of function addresses extracted from $\mathit{externval}^*_{\mathrm{im}}$, concatenated with $\mathit{funcaddr}^*$.
15. Let $\mathit{tableaddr}^*_{\mathrm{mod}}$ be the list of table addresses extracted from $\mathit{externval}^*_{\mathrm{im}}$, concatenated with $\mathit{tableaddr}^*$.
16. Let $\mathit{memaddr}^*_{\mathrm{mod}}$ be the list of memory addresses extracted from $\mathit{externval}^*_{\mathrm{im}}$, concatenated with $\mathit{memaddr}^*$.
17. Let $\mathit{globaladdr}^*_{\mathrm{mod}}$ be the list of global addresses extracted from $\mathit{externval}^*_{\mathrm{im}}$, concatenated with $\mathit{globaladdr}^*$.
18. For each export export_i in $\mathit{module}.\mathsf{exports}$, do:
    a. If export_i is a function export for function index x, then let $\mathit{externval}_i$ be the external value $\mathsf{func}\ (\mathit{funcaddr}^*_{\mathrm{mod}}[x])$.
    b. Else, if export_i is a table export for table index x, then let $\mathit{externval}_i$ be the external value $\mathsf{table}\ (\mathit{tableaddr}^*_{\mathrm{mod}}[x])$.
    c. Else, if export_i is a memory export for memory index x, then let $\mathit{externval}_i$ be the external value $\mathsf{mem}\ (\mathit{memaddr}^*_{\mathrm{mod}}[x])$.
    d. Else, if export_i is a global export for global index x, then let $\mathit{externval}_i$ be the external value $\mathsf{global}\ (\mathit{globaladdr}^*_{\mathrm{mod}}[x])$.
    e. Let $\mathit{exportinst}_i$ be the export instance $\{\mathsf{name}\ (\mathit{export}_i.\mathsf{name}), \mathsf{value}\ \mathit{externval}_i\}$.
19. Let $\mathit{exportinst}^*$ be the concatenation of the export instances $\mathit{exportinst}_i$ in index order.
20. Let $\mathit{moduleinst}$ be the module instance $\{\mathsf{types}\ (\mathit{module}.\mathsf{types}), \mathsf{funcaddrs}\ \mathit{funcaddr}^*_{\mathrm{mod}}, \mathsf{tableaddrs}\ \mathit{tableaddr}^*_{\mathrm{mod}}, \mathsf{memaddrs}\ \mathit{memaddr}^*_{\mathrm{mod}}, \mathsf{globaladdrs}\ \mathit{globaladdr}^*_{\mathrm{mod}}, \mathsf{exports}\ \mathit{exportinst}^*\}$.
21. Return $\mathit{moduleinst}, \mathit{act}^*$.

$$\mathrm{allocmodule}(S, \mathit{module}, \mathit{externval}^*_{\mathrm{im}}, \mathit{val}^*, (\mathit{ref}^*)^*) \quad = \quad S', \mathit{moduleinst}, \mathit{act}^*$$

where:

$$
\begin{array}{rcl}
\mathit{table}^* &=& \mathit{module}.\mathsf{tables} \\
\mathit{mem}^* &=& \mathit{module}.\mathsf{mems} \\
\mathit{global}^* &=& \mathit{module}.\mathsf{globals} \\
\mathit{elem}^* &=& \mathit{module}.\mathsf{elems} \\
\mathit{data}^* &=& \mathit{module}.\mathsf{datas} \\
\mathit{export}^* &=& \mathit{module}.\mathsf{exports} \\[1ex]
\mathit{moduleinst} &=& \{~\mathsf{types}~\mathit{module}.\mathsf{types}, \\
&& \quad \mathsf{funcaddrs}~\mathrm{funcs}(\mathit{externval}^*_{\mathrm{im}})~\mathit{funcaddr}^*, \\
&& \quad \mathsf{tableaddrs}~\mathrm{tables}(\mathit{externval}^*_{\mathrm{im}})~\mathit{tableaddr}^*, \\
&& \quad \mathsf{memaddrs}~\mathrm{mems}(\mathit{externval}^*_{\mathrm{im}})~\mathit{memaddr}^*, \\
&& \quad \mathsf{globaladdrs}~\mathrm{globals}(\mathit{externval}^*_{\mathrm{im}})~\mathit{globaladdr}^*, \\
&& \quad \mathsf{elemaddrs}~\mathit{elemaddr}^*, \\
&& \quad \mathsf{dataaddrs}~\mathit{dataaddr}^*, \\
&& \quad \mathsf{exports}~\mathit{exportinst}^*~\} \\
S_1, \mathit{funcaddr}^* &=& \mathrm{allocfunc}^*(S, \mathit{module}.\mathsf{funcs}, \mathit{moduleinst}) \\
S_2, \mathit{tableaddr}^* &=& \mathrm{alloctable}^*(S_1, (\mathit{table}.\mathsf{type})^*, (\mathsf{ref.null}~t)^*) \quad (\text{where } (\mathit{table}.\mathsf{type})^* = (\mathit{limits}~t)^*) \\
S_3, \mathit{memaddr}^*, \mathit{act}^* &=& \mathrm{allocmem}^*(S_2, (\mathit{mem}.\mathsf{type})^*) \\
S_4, \mathit{globaladdr}^* &=& \mathrm{allocglobal}^*(S_3, (\mathit{global}.\mathsf{type})^*, \mathit{val}^*) \\
S_5, \mathit{elemaddr}^* &=& \mathrm{allocelem}^*(S_4, (\mathit{elem}.\mathsf{type})^*, (\mathit{ref}^*)^*) \\
S', \mathit{dataaddr}^* &=& \mathrm{allocdata}^*(S_5, (\mathit{data}.\mathsf{init})^*) \\
\mathit{exportinst}^* &=& \{\mathsf{name}~(\mathit{export}.\mathsf{name}), \mathsf{value}~\mathit{externval}_{\mathrm{ex}}\}^* \\[1ex]
\mathrm{funcs}(\mathit{externval}^*_{\mathrm{ex}}) &=& (\mathit{moduleinst}.\mathsf{funcaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{funcs}(\mathit{export}^*)) \\
\mathrm{tables}(\mathit{externval}^*_{\mathrm{ex}}) &=& (\mathit{moduleinst}.\mathsf{tableaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{tables}(\mathit{export}^*)) \\
\mathrm{mems}(\mathit{externval}^*_{\mathrm{ex}}) &=& (\mathit{moduleinst}.\mathsf{memaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{mems}(\mathit{export}^*)) \\
\mathrm{globals}(\mathit{externval}^*_{\mathrm{ex}}) &=& (\mathit{moduleinst}.\mathsf{globaladdrs}[x])^* \qquad (\text{where } x^* = \mathrm{globals}(\mathit{export}^*))
\end{array}
$$

Here, the notation allocx^* is shorthand for multiple allocations of object kind X, defined as follows:

$$
\begin{array}{rcl}
\mathrm{allocx}^*(S_0, X^n, \dots) &=& S_n, a^n \\
\multicolumn{3}{l}{\text{where for all } i < n\text{:}} \\
S_{i+1}, a^n[i] &=& \mathrm{allocx}(S_i, X^n[i], \dots)
\end{array}
$$

Moreover, if the dots $\dots$ are a sequence A^n (as for globals or tables), then the elements of this sequence are passed to the allocation function pointwise.

Note: The definition of module allocation is mutually recursive with the allocation of its associated functions, because the resulting module instance *moduleinst* is passed to the function allocator as an argument, in order to form the necessary closures. In an implementation, this recursion is easily unraveled by mutating one or the other in a secondary step.

4.5.4 Instantiation

Given a store S, a module *module* is instantiated with a list of external values $\mathit{externval}^n$ supplying the required imports as follows.

Instantiation checks that the module is valid and the provided imports match the declared types, and may *fail* with an error otherwise. Instantiation can also result in a trap from initializing a table or memory from an active segment or from executing the start function. It is up to the embedder to define how such conditions are reported.

1. If *module* is not valid, then:

    a. Fail.

2. Assert: *module* is valid with external types $\mathit{externtype}^m_{\mathrm{im}}$ classifying its imports.

3. If the number m of imports is not equal to the number n of provided external values, then:

   a. Fail.

4. For each external value $\mathit{externval}_i$ in $\mathit{externval}^n$ and external type $\mathit{externtype}'_i$ in $\mathit{externtype}^n_{\mathrm{im}}$, do:
   a. If $\mathit{externval}_i$ is valid with an external type $\mathit{externtype}_i$ in store S, then:
      i. Assert: in checking the validity of $\mathit{externval}_i$, the shared memory actions $\mathit{act}^?_i$ are performed.
   b. Else:
      i. Fail.
   c. If $\mathit{externtype}_i$ does not match $\mathit{externtype}'_i$, then:
      i. Fail.
5. Let $\mathit{moduleinst}_{\mathrm{init}}$ be the auxiliary module instance $\{\mathsf{globaladdrs}~\mathrm{globals}(\mathit{externval}^n), \mathsf{funcaddrs}~\mathit{moduleinst}.\mathsf{funcaddrs}\}$ that only consists of the imported globals and the imported and allocated functions from the final module instance $\mathit{moduleinst}$, defined below.
6. Let F_{init} be the auxiliary frame $\{\mathsf{module}~\mathit{moduleinst}_{\mathrm{init}}, \mathsf{locals}~\epsilon\}$.
7. Push the frame F_{init} to the stack.
8. Let val^* be the vector of global initialization values determined by module and $\mathit{externval}^n$. These may be calculated as follows.
   a. For each global global_i in $\mathit{module}.\mathsf{globals}$, do:
      i. Let val_i be the result of evaluating the initializer expression $\mathit{global}_i.\mathsf{init}$.
   b. Assert: due to validation, the frame F_{init} is now on the top of the stack.
   c. Let val^* be the concatenation of val_i in index order.
9. Let $(\mathit{ref}^*)^*$ be the list of reference vectors determined by the element segments in module. These may be calculated as follows.
   a. For each element segment elem_i in $\mathit{module}.\mathsf{elems}$, and for each element expression expr_{ij} in $\mathit{elem}_i.\mathsf{init}$, do:
      i. Let ref_{ij} be the result of evaluating the initializer expression expr_{ij}.
   b. Let ref^*_i be the concatenation of function elements ref_{ij} in order of index j.
   c. Let $(\mathit{ref}^*)^*$ be the concatenation of function element vectors ref^*_i in order of index i.
10. Pop the frame F_{init} from the stack.
11. Let $\mathit{moduleinst}$ be a new module instance allocated from module in store S with imports $\mathit{externval}^n$, global initializer values val^*, and element segment contents $(\mathit{ref}^*)^*$, and let S' be the extended store produced by module allocation.
12. Assert: in allocating $\mathit{moduleinst}$, the shared memory actions $\mathit{act}_{\mathrm{m}}{}^*$ are performed.
13. Let F be the auxiliary frame $\{\mathsf{module}~\mathit{moduleinst}, \mathsf{locals}~\epsilon\}$.
14. Push the frame F to the stack.
15. For each element segment elem_i in $\mathit{module}.\mathsf{elems}$ whose mode is of the form $\mathsf{active}~\{\mathsf{table}~\mathit{tableidx}_i, \mathsf{offset}~\mathit{einstr}^*_i~\mathsf{end}\}$, do:
   a. Let n be the length of the vector $\mathit{elem}_i.\mathsf{init}$.
   b. Execute the instruction sequence einstr^*_i.
   c. Execute the instruction $\mathsf{i32.const}~0$.
   d. Execute the instruction $\mathsf{i32.const}~n$.
   e. Execute the instruction $\mathsf{table.init}~\mathit{tableidx}_i~i$.
   f. Execute the instruction $\mathsf{elem.drop}~i$.
16. For each element segment elem_i in $\mathit{module}.\mathsf{elems}$ whose mode is of the form $\mathsf{declarative}$, do:

   a. Execute the instruction $\mathsf{elem.drop}\ i$.

17. For each data segment data_i in $\mathit{module}.\mathsf{datas}$ whose mode is of the form $\mathsf{active}\ \{\mathsf{memory}\ \mathit{memidx}_i, \mathsf{offset}\ \mathit{dinstr}_i^*\ \mathsf{end}\}$, do:
   a. Assert: memidx_i is 0.
   b. Let n be the length of the vector $\mathit{data}_i.\mathsf{init}$.
   c. Execute the instruction sequence dinstr_i^*.
   d. Execute the instruction $\mathsf{i32.const}\ 0$.
   e. Execute the instruction $\mathsf{i32.const}\ n$.
   f. Execute the instruction $\mathsf{memory.init}\ i$.
   g. Execute the instruction $\mathsf{data.drop}\ i$.
18. If the start function $\mathit{module}.\mathsf{start}$ is not empty, then:
   a. Let start be the start function $\mathit{module}.\mathsf{start}$.
   b. Execute the instruction $\mathsf{call}\ \mathit{start}.\mathsf{func}$.
19. Assert: due to validation, the frame F is now on the top of the stack.
20. Pop the frame F from the stack.

$$\begin{array}{lcl}
\mathrm{instantiate}(S, \mathit{module}, \mathit{externval}^k) &=& S'; F; (\mathsf{perform}\ (\mathit{act}^?))^k \\
&& \quad (\mathsf{perform}\ \mathit{act}_{\mathrm{m}}{}^*) \\
&& \quad \mathrm{runelem}_0(\mathit{elem}^n[0])\ \dots\ \mathrm{runelem}_{n-1}(\mathit{elem}^n[n-1]) \\
&& \quad \mathrm{rundata}_0(\mathit{data}^m[0])\ \dots\ \mathrm{rundata}_{m-1}(\mathit{data}^m[m-1]) \\
&& \quad (\mathsf{call}\ \mathit{start}.\mathsf{func})^? \\
&(\mathrm{if} & \vdash \mathit{module} : \mathit{externtype}_{\mathrm{im}}^k \to \mathit{externtype}_{\mathrm{ex}}^* \\
&\wedge & (S; \mathit{act}^? \vdash \mathit{externval} : \mathit{externtype})^k \\
&\wedge & (\vdash \mathit{externtype} \leq \mathit{externtype}_{\mathrm{im}})^k \\[1ex]
&\wedge & \mathit{module}.\mathsf{globals} = \mathit{global}^* \\
&\wedge & \mathit{module}.\mathsf{elems} = \mathit{elem}^n \\
&\wedge & \mathit{module}.\mathsf{datas} = \mathit{data}^m \\
&\wedge & \mathit{module}.\mathsf{start} = \mathit{start}^? \\
&\wedge & (\mathit{expr}_{\mathrm{g}} = \mathit{global}.\mathsf{init})^* \\
&\wedge & (\mathit{expr}_{\mathrm{e}}^* = \mathit{elem}.\mathsf{init})^n \\[1ex]
&\wedge & S', \mathit{moduleinst}, \mathit{act}_{\mathrm{m}}{}^* = \mathrm{allocmodule}(S, \mathit{module}, \mathit{externval}^k, \mathit{val}^*, (\mathit{ref}^*)^n) \\
&\wedge & F = \{\mathsf{module}\ \mathit{moduleinst}, \mathsf{locals}\ \epsilon\} \\[1ex]
&\wedge & (S'; F; \mathit{expr}_{\mathrm{g}} \hookrightarrow^* S'; F; \mathit{val}\ \mathsf{end})^* \\
&\wedge & ((S'; F; \mathit{expr}_{\mathrm{e}} \hookrightarrow^* S'; F; \mathit{ref}\ \mathsf{end})^*)^n) \\
\mathrm{instantiate}(S, \mathit{module}, \mathit{externval}^k) &=& S; \epsilon; \mathsf{trap} \quad (\mathrm{otherwise})
\end{array}$$

$$S; F; (\mathsf{perform}\ \mathit{act}^*) \quad \hookrightarrow^{\mathit{act}^*} \quad S; F; \epsilon$$

where:

$$\begin{array}{l}
\mathrm{runelem}_i(\{\mathsf{type}\ \mathit{et}, \mathsf{init}\ \mathit{expr}^n, \mathsf{mode}\ \mathsf{passive}\}) \quad = \quad \epsilon \\
\mathrm{runelem}_i(\{\mathsf{type}\ \mathit{et}, \mathsf{init}\ \mathit{expr}^n, \mathsf{mode}\ \mathsf{active}\{\mathsf{table}\ x, \mathsf{offset}\ \mathit{instr}^*\ \mathsf{end}\}\}) \quad = \\
\qquad \mathit{instr}^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ i)\ (\mathsf{elem.drop}\ i) \\
\mathrm{runelem}_i(\{\mathsf{type}\ \mathit{et}, \mathsf{init}\ \mathit{expr}^n, \mathsf{mode}\ \mathsf{declarative}\}) \quad = \\
\qquad (\mathsf{elem.drop}\ i)
\end{array}$$

$$\begin{array}{l}
\mathrm{rundata}_i(\{\mathsf{init}\ b^n, \mathsf{mode}\ \mathsf{passive}\}) \quad = \quad \epsilon \\
\mathrm{rundata}_i(\{\mathsf{init}\ b^n, \mathsf{mode}\ \mathsf{active}\{\mathsf{memory}\ 0, \mathsf{offset}\ \mathit{instr}^*\ \mathsf{end}\}\}) \quad = \\
\qquad \mathit{instr}^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ i)\ (\mathsf{data.drop}\ i)
\end{array}$$

Note: Module allocation and the evaluation of global initializers and element segments are mutually recursive because the global initialization values val^* and element segment contents $(ref^*)^*$ are passed to the module allocator while depending on the module instance $moduleinst$ and store S' returned by allocation. However, this recursion is just a specification device. In practice, the initialization values can be determined beforehand by staging module allocation such that first, the module's own function instances are pre-allocated in the store, then the initializer expressions are evaluated, then the rest of the module instance is allocated, and finally the new function instances' module fields are set to that module instance. This is possible because validation ensures that initialization expressions cannot actually call a function, only take their reference.

All failure conditions are checked before any observable mutation of the store takes place. Store mutation is not atomic; it happens in individual steps that may be interleaved with other threads.

Evaluation of constant expressions does not affect the store.

4.5.5 Invocation

Once a module has been instantiated, any exported function can be *invoked* externally via its function address $funcaddr$ in the store S and an appropriate list val^* of argument values.

Invocation may *fail* with an error if the arguments do not fit the function type. Invocation can also result in a trap. It is up to the embedder to define how such conditions are reported.

Note: If the embedder API performs type checks itself, either statically or dynamically, before performing an invocation, then no failure other than traps can occur.

The following steps are performed:

1. Assert: $S.\mathsf{funcs}[funcaddr]$ exists.
2. Let $funcinst$ be the function instance $S.\mathsf{funcs}[funcaddr]$.
3. Let $[t_1^n] \rightarrow [t_2^m]$ be the function type $funcinst.\mathsf{type}$.
4. If the length $|val^*|$ of the provided argument values is different from the number n of expected arguments, then:
    a. Fail.
5. For each value type t_i in t_1^n and corresponding value val_i in val^*, do:
    a. If val_i is not valid with value type t_i, then:
        i. Fail.
6. Let F be the dummy frame $\{\mathsf{module}~\{\}, \mathsf{locals}~\epsilon\}$.
7. Push the frame F to the stack.
8. Push the values val^* to the stack.
9. Invoke the function instance at address $funcaddr$.

Once the function has returned, the following steps are executed:

1. Assert: due to validation, m values are on the top of the stack.
2. Pop val_{res}^m from the stack.

The values $\mathit{val}_{\mathrm{res}}^m$ are returned as the results of the invocation.

$$\begin{array}{lcl} \mathrm{invoke}(S, \mathit{funcaddr}, \mathit{val}^n) &=& S; F; \mathit{val}^n~(\mathsf{invoke}~\mathit{funcaddr}) \\ &(\mathrm{if}& S.\mathsf{funcs}[\mathit{funcaddr}].\mathsf{type} = [t_1^n] \to [t_2^m] \\ &\wedge& (S \vdash \mathit{val} : t_1)^n \\ &\wedge& F = \{\mathsf{module}~\{\}, \mathsf{locals}~\epsilon\}) \end{array}$$

4.6 Relaxed Memory Model

The execution of a WebAssembly program gives rise to a trace of events. WebAssembly's relaxed memory model constrains the observable behaviours of the program's execution by defining a consistency condition on the trace of events.

Note: A relaxed memory model is necessary to describe the behaviour of programs exhibiting *shared memory concurrency*. WebAssembly's relaxed memory model is heavily based on those of C/C++11 and JavaScript. The relaxed memory model described here is derived from the following article:[30].

---

[30] The semantics of the relaxed memory model is derived from the following article: Conrad Watt, Andreas Rossberg, Jean Pichon-Pharabod. Weakening WebAssembly[31]. Proceedings of the ACM on Programming Languages (OOPSLA 2019). ACM 2019.

[31] https://dl.acm.org/citation.cfm?id=3360559

4.6.1 Preliminary Definitions

$$
\begin{array}{rcl}
\mathrm{time}(act^* ~ \mathsf{at} ~ time_p ~ time) & = & time \\
\mathrm{time}_p(act^* ~ \mathsf{at} ~ time_p ~ time) & = & time_p \\
\\
\mathrm{loc}(\mathsf{rd}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & loc \\
\mathrm{loc}(\mathsf{wr}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & loc \\
\mathrm{loc}(\mathsf{rmw} ~ loc ~ byte_1{}^* ~ byte_2{}^*) & = & loc \\
\mathrm{loc}(\mathsf{wait} ~ loc ~ s64) & = & loc \\
\mathrm{loc}(\mathsf{woken} ~ loc) & = & loc \\
\mathrm{loc}(\mathsf{timeout} ~ loc) & = & loc \\
\mathrm{loc}(\mathsf{notify} ~ loc ~ u32 ~ u32) & = & loc \\
\\
\mathrm{ord}(\mathsf{rd}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & ord \\
\mathrm{ord}(\mathsf{wr}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & ord \\
\mathrm{ord}(\mathsf{rmw} ~ loc ~ byte_1{}^* ~ byte_2{}^*) & = & \mathsf{seqcst} \\
\\
\mathrm{overlap}(act_1, act_2) & = & (\mathrm{range}(act_1) \cup \mathrm{range}(act_2) \neq \epsilon) \\
\mathrm{same}(act_1, act_2) & = & (\mathrm{range}(act_1) = \mathrm{range}(act_2)) \\
\\
\mathrm{reading}(act) & = & (\mathrm{read}(act) \neq \epsilon) \\
\mathrm{writing}(act) & = & (\mathrm{write}(act) \neq \epsilon) \\
\mathrm{susp}(u32, \mathsf{wait} ~ reg[u32] ~ s64) & = & \mathsf{wait} ~ reg[u32] ~ s64 \\
\mathrm{susp}(u32, \mathsf{woken} ~ reg[u32]) & = & \mathsf{woken} ~ reg[u32] \\
\mathrm{susp}(u32, \mathsf{timeout} ~ reg[u32]) & = & \mathsf{timeout} ~ reg[u32] \\
\mathrm{susp}(u32, \mathsf{notify} ~ reg[u32] ~ u32' ~ u32'') & = & \mathsf{notify} ~ reg[u32] ~ u32' ~ u32'' \\
\mathrm{susp}(u32, act) & = & \epsilon \qquad (\mathrm{otherwise}) \\
\\
\mathrm{read}(\mathsf{rd}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & byte^* \\
\mathrm{read}(\mathsf{rmw} ~ loc ~ byte_1{}^* ~ byte_2{}^*) & = & byte_1{}^* \\
\mathrm{read}(act) & = & \epsilon \qquad (\mathrm{otherwise}) \\
\\
\mathrm{write}(\mathsf{wr}_{ord} ~ loc ~ byte^* ~ \mathsf{notears}^?) & = & byte^* \\
\mathrm{write}(\mathsf{rmw} ~ loc ~ byte_1{}^* ~ byte_2{}^*) & = & byte_2{}^* \\
\mathrm{write}(act) & = & \epsilon \qquad (\mathrm{otherwise}) \\
\\
\mathrm{offset}(act) & = & u32 \qquad (\mathrm{if} ~ \mathrm{loc}(act) = reg[u32]) \\
\\
\mathrm{sync}(act_1, act_2) & = & (\mathrm{same}(act_1, act_2) \wedge \\
 & & \qquad \mathrm{ord}(act_1) = \mathrm{ord}(act_2) = \mathsf{seqcst}) \\
\mathrm{range}(act) & = & [u32 \dots u32 + n - 1] \\
 & & (\mathrm{if} ~ \mathrm{loc}(act) = reg[u32] \wedge \\
 & & \quad n = \max(|\mathrm{read}(act)|, |\mathrm{write}(act)|)) \\
\\
\mathrm{tearfree}(\mathsf{rd}_{ord} ~ loc ~ byte^*) & = & \bot \qquad (\mathrm{if} ~ ord = \mathsf{unord} \vee ord = \mathsf{init}) \\
\mathrm{tearfree}(\mathsf{wr}_{ord} ~ loc ~ byte^*) & = & \bot \qquad (\mathrm{if} ~ ord = \mathsf{unord} \vee ord = \mathsf{init}) \\
\mathrm{tearfree}(act) & = & \top \qquad (\mathrm{otherwise}) \\
\\
\mathrm{id}(act) & = & act
\end{array}
$$

The above operations on actions are raised to operations on events, indexed by region.

$$
\begin{array}{rcl}
func_{reg}(act_1^* ~ act ~ act_2^* ~ \mathsf{at} ~ time_p ~ time) & = & func(act) \\
 & & (\mathrm{if} ~ \mathrm{loc}(act) = reg[u32]) \\
func_{reg}(act_1^* ~ act ~ act_2^* ~ \mathsf{at} ~ time_p ~ time, & & \\
act_3^* ~ act' ~ act_4^* ~ \mathsf{at} ~ time_p' ~ time') & = & func(act, act') \\
 & & (\mathrm{if} ~ \mathrm{loc}(act) = \mathrm{loc}(act') = reg[u32])
\end{array}
$$

4.6.2 Traces

Todo: novel notation here?

A trace is a coinductive list of events. A trace is considered to be a *pre-execution* of a given global configuration if it represents the events emitted by the coinductive closure of the global reduction relation on that configuration, such that all of the trace's consituent events have unique time stamps that are totally ordered according to the reduction order.

$$\begin{array}{c}
\mathit{config} \hookrightarrow^{\mathit{evt}} \mathit{config}' \qquad \vdash \mathit{config}' : \mathit{tr} \qquad \mathrm{time}(\mathit{evt}) \notin \mathrm{time}^*(\mathit{tr}) \\
\forall \mathit{evt}' \in \mathit{tr}, \mathrm{time}(\mathit{evt}') \prec_{\mathsf{tot}} \mathrm{time}(\mathit{evt}) \qquad \mathrm{time}_p(\mathit{evt}) \notin \mathrm{time}^*_p(\mathit{tr}) \\
\hline\hline
\vdash \mathit{config} : \mathit{evt}\ \mathit{tr}
\end{array}$$

When a WebAssembly program is executed, all behaviours observed during that execution must correspond to a single consistent pre-execution of that execution's starting configuration.

4.6.3 Consistency

$$\frac{\forall \mathit{reg}, \vdash_{\mathit{reg}} \mathit{tr}\ \ \text{consistent-with}}{\vdash \mathit{tr}\ \ \text{consistent}}$$

$$\frac{\begin{array}{c}
\forall i, \vdash^{i}_{\mathit{reg}} \mathit{tr}\ \ \text{suspensions-consistent} \\
\forall \mathit{evt}_R \in \mathrm{reading}_{\mathit{reg}}(\mathit{tr}), \exists \mathit{evt}^*_W, \mathit{tr} \vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{reads-each-from}\ \ \mathit{evt}^*_W \\
\forall \mathit{evt}_I, \mathit{evt} \in \mathit{tr}, \mathrm{ord}_{\mathit{reg}}(\mathit{evt}_I) = \mathsf{init} \land \mathit{evt}_I \neq \mathit{evt} \land \mathrm{overlap}(\mathit{evt}_I, \mathit{evt}) \Rightarrow \mathit{evt}_I \prec_{\mathsf{hb}} \mathit{evt}
\end{array}}{\vdash_{\mathit{reg}} \mathit{tr}\ \ \text{consistent-with}}$$

$$\frac{\begin{array}{c}
|\mathit{evt}^*_W| = |\mathrm{read}_{\mathit{reg}}(\mathit{evt}_R)| \\
\forall i < |\mathit{evt}^*_W|, \mathit{tr} \vdash^{i}_{\mathit{reg}} \mathit{evt}_R\ \ \text{reads-from}\ \ (\mathit{evt}^*_W[i]) \\
\vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{no-tear}\ \ \mathit{evt}^*_W
\end{array}}{\mathit{tr} \vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{reads-each-from}\ \ \mathit{evt}^*_W}$$

$$\frac{\begin{array}{c}
\mathit{evt}_R \neq \mathit{evt}_W \\
\mathit{evt}_W \in \mathrm{writing}_{\mathit{reg}}(\mathit{tr}) \\
\mathit{tr} \vdash^{i,k}_{\mathit{reg}} \mathit{evt}_R\ \ \text{value-consistent}\ \ \mathit{evt}_W \\
\mathit{tr} \vdash^{k}_{\mathit{reg}} \mathit{evt}_R\ \ \text{hb-consistent}\ \ \mathit{evt}_W \\
\mathit{tr} \vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{sc-last-visible}\ \ \mathit{evt}^*_W
\end{array}}{\mathit{tr} \vdash^{i}_{\mathit{reg}} \mathit{evt}_R\ \ \text{reads-from}\ \ \mathit{evt}_W}$$

$$\frac{\begin{array}{c}
\mathrm{read}_{\mathit{reg}}(\mathit{evt}_R)[i] = \mathrm{write}_{\mathit{reg}}(\mathit{evt}_W)[j] \\
k = \mathrm{offset}_{\mathit{reg}}(\mathit{evt}_R) + i = \mathrm{offset}_{\mathit{reg}}(\mathit{evt}_W) + j
\end{array}}{\mathit{tr} \vdash^{i,k}_{\mathit{reg}} \mathit{evt}_R\ \ \text{value-consistent}\ \ \mathit{evt}_W}$$

$$\frac{\begin{array}{c}
\neg(\mathit{evt}_R \prec_{\mathsf{hb}} \mathit{evt}_W) \\
\mathrm{sync}_{\mathit{reg}}(\mathit{evt}_W, \mathit{evt}_R) \Rightarrow \mathit{evt}_W \prec_{\mathsf{hb}} \mathit{evt}_R \\
\forall \mathit{evt}'_W \in \mathrm{writing}_{\mathit{reg}}(\mathit{tr}), \mathit{evt}_W \prec_{\mathsf{hb}} \mathit{evt}'_W \prec_{\mathsf{hb}} \mathit{evt}_R \Rightarrow k \notin \mathrm{range}_{\mathit{reg}}(\mathit{evt}'_W)
\end{array}}{\mathit{tr} \vdash^{k}_{\mathit{reg}} \mathit{evt}_R\ \ \text{hb-consistent}\ \ \mathit{evt}_W}$$

$$\frac{\begin{array}{l}
\forall \mathit{evt}'_W \in \mathrm{writing}_{\mathit{reg}}(\mathit{tr}), \mathit{evt}_W \prec_{\mathsf{hb}} \mathit{evt}_R \Rightarrow \\
\quad \mathit{evt}_W \prec_{\mathsf{tot}} \mathit{evt}'_W \prec_{\mathsf{tot}} \mathit{evt}_R \land \mathrm{sync}_{\mathit{reg}}(\mathit{evt}_W, \mathit{evt}_R) \Rightarrow \neg\mathrm{sync}_{\mathit{reg}}(\mathit{evt}'_W, \mathit{evt}_R) \\
\quad \mathit{evt}_W \prec_{\mathsf{hb}} \mathit{evt}'_W \prec_{\mathsf{tot}} \mathit{evt}_R \Rightarrow \neg\mathrm{sync}_{\mathit{reg}}(\mathit{evt}'_W, \mathit{evt}_R) \\
\quad \mathit{evt}_W \prec_{\mathsf{tot}} \mathit{evt}'_W \prec_{\mathsf{hb}} \mathit{evt}_R \Rightarrow \neg\mathrm{sync}_{\mathit{reg}}(\mathit{evt}_W, \mathit{evt}'_W)
\end{array}}{\mathit{tr} \vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{sc-last-visible}\ \ \mathit{evt}_W}$$

$$\frac{\begin{array}{l}
\mathrm{tearfree}_{\mathit{reg}}(\mathit{evt}_R) \Rightarrow \\
\quad |\{\mathit{evt}_W \in \mathit{evt}^*_W \mid \mathrm{same}_{\mathit{reg}}(\mathit{evt}_R, \mathit{evt}_W) \land \mathrm{tearfree}_{\mathit{reg}}(\mathit{evt}_W)\}| \leq 1
\end{array}}{\vdash_{\mathit{reg}} \mathit{evt}_R\ \ \text{no-tear}\ \ \mathit{evt}^*_W}$$

$$\frac{\begin{array}{c}\mathrm{susp}^*_{reg}(i, tr) = tr' \qquad \vdash^i_{reg}\ tr'\ \ \text{suspensions-consistent-with}\ (\epsilon) \\ \forall evt, evt' \in tr',\ evt \prec_{\mathsf{tot}} evt' \Longrightarrow evt \prec_{\mathsf{hb}} evt'\end{array}}{\vdash^i_{reg}\ tr\ \ \text{suspensions-consistent}}$$

$$\frac{}{\vdash^i_{reg}\ \epsilon\ \ \text{suspensions-consistent-with}\ (time^*)}$$

$$\frac{\mathrm{id}_{reg}(evt) = (\mathsf{wait}\ reg[i]\ s64) \qquad \vdash^i_{reg}\ tr\ \ \text{suspensions-consistent-with}\ (\mathrm{time}(evt)\ time^*)}{\vdash^i_{reg}\ evt\ tr\ \ \text{suspensions-consistent-with}\ (time^*)}$$

$$\frac{\mathrm{id}_{reg}(evt) = (\mathsf{timeout}\ reg[i]) \qquad \vdash^i_{reg}\ tr\ \ \text{suspensions-consistent-with}\ (time^*\ time'^*)}{\vdash^i_{reg}\ evt\ tr\ \ \text{suspensions-consistent-with}\ (time^*\ \mathrm{time}_p(evt)\ time'^*)}$$

$$\frac{\begin{array}{c}\mathrm{id}^n_{reg}(evt^n) = (\mathsf{woken}\ reg[i]) \qquad \mathrm{id}_{reg}(evt_N) = (\mathsf{notify}\ reg[i]\ n\ k) \\ n < k \Longrightarrow m = 0 \qquad \vdash^i_{reg}\ tr\ \ \text{suspensions-consistent-with}\ (time^m)\end{array}}{\vdash^i_{reg}\ evt_N\ evt^n\ tr\ \ \text{suspensions-consistent-with}\ (time^m\ \mathrm{time}^n_p(evt^n))}$$

Note: The following is a non-normative and non-exhaustive explanation of WebAssembly's relaxed memory model in plain English. Note that the definition of Consistency is the sole normative definition of the relaxed memory model.

When a WebAssembly operation reads from shared mutable state, the WebAssembly relaxed memory model determines the value that this read access *observes*, in terms of the write access to the same location(s) that have occurred in the execution.

The WebAssembly memory model is built around the concept of a *happens-before* transitive partial order between accesses of shared mutable state, $\prec_{\mathsf{hb}}$, which captures a strong notion of causality. All sequential accesses in the same thread are related by $\prec_{\mathsf{hb}}$ according to execution order. Certain operations also establish a $\prec_{\mathsf{hb}}$ relation between operations of different threads (see *atomic* accesses below). A read access may never take its value from a write access that comes later in $\prec_{\mathsf{hb}}$. Moreover, if two write accesses ordered by $\prec_{\mathsf{hb}}$ come before a read access in $\prec_{\mathsf{hb}}$, the read access must take its value from the later of the two write accesses according to $\prec_{\mathsf{hb}}$. In the case that $\prec_{\mathsf{hb}}$ does not uniquely determine a write access that a given read access *must* take its value from, the read access may non-deterministically take its value from any permitted write.

In the case that a read operation is a multi-byte memory access, the value of each byte may in certain circumstances be determined by a different write event. If this happens, we describe the read operation as *tearing*. In general, naturally aligned multi-byte reads are not allowed to tear, unless they race with a partially overlapping write or are greater than four bytes in width.

Most WebAssembly accesses of shared mutable state are classified as *non-atomic*. However a number of operations are classified as performing *atomic* accesses. Atomic accesses must always be naturally aligned. If an atomic read takes its value from an atomic write of the same width, the write access is fixed as coming before the read access in $\prec_{\mathsf{hb}}$. This is the main mechanism by which a $\prec_{\mathsf{hb}}$ relation is established between threads.

WebAssembly's atomic operations are also required to be *sequentially consistent*. The relaxed memory model defines a toal order on all events of the execution, $\prec_{\mathsf{tot}}$, and sequentially consistent operations to identical ranges must respect this ordering - i.e. sequentially consistent reads cannot read from any sequentially consistent write of idential range other than the most recent preceding one according to $\prec_{\mathsf{tot}}$.

Some operations such as memory accesses must perform a bounds check in addition to accessing data. The relaxed memory model treats these accesses as additionally accessing a distinguished *length* location, with the observed value respecting the constraints of the relaxed memory model. Most bounds checks are non-atomic, but bounds checks peformed during instantiation are atomic, and changes to the length (e.g. memory.grow) are modelled as atomic read-modify-write accesses.

In some circumstances, two accesses to overlapping locations may occur in an execution without any relation in $\prec_{\mathsf{hb}}$. This situation is known as a *race*. If at least one of these accesses is a non-atomic write, we describe this situation as a *data race*. Unlike some other relaxed memory models, WebAssembly does not declare data races to be undefined behaviour. However, the allowed execution behaviours may still be highly non-deterministic as the lack of $\prec_{\mathsf{hb}}$ relations means that reads participating in or overlapping with the location of the data race may non-deterministically observe a number of different values.

The relaxed memory model also describes the concurrent behaviour of WebAssembly's wait (memory.atomic.wait) and notify (memory.atomic.notify) operations. Each memory location is associated with a queue of waiting threads. A thread suspending as the result of a wait operation enters the queue, and a notify operation to that location will attempt to wake up as many threads as possible from the head of the associated queue, up to the maximum specified by the arguments of the notify operation. All operations on the same location which change the state of that location's wait queue are sequentially consistent and totally ordered by $\prec_{\mathsf{hb}}$.

CHAPTER 5

Binary Format

5.1 Conventions

The binary format for WebAssembly modules is a dense linear *encoding* of their abstract syntax.[33]

The format is defined by an *attribute grammar* whose only terminal symbols are bytes. A byte sequence is a well-formed encoding of a module if and only if it is generated by the grammar.

Each production of this grammar has exactly one synthesized attribute: the abstract syntax that the respective byte sequence encodes. Thus, the attribute grammar implicitly defines a *decoding* function (i.e., a parsing function for the binary format).

Except for a few exceptions, the binary grammar closely mirrors the grammar of the abstract syntax.

Note: Some phrases of abstract syntax have multiple possible encodings in the binary format. For example, numbers may be encoded as if they had optional leading zeros. Implementations of decoders must support all possible alternatives; implementations of encoders can pick any allowed encoding.

The recommended extension for files containing WebAssembly modules in binary format is "`.wasm`" and the recommended Media Type[32] is "`application/wasm`".

5.1.1 Grammar

The following conventions are adopted in defining grammar rules for the binary format. They mirror the conventions used for abstract syntax. In order to distinguish symbols of the binary syntax from symbols of the abstract syntax, `typewriter` font is adopted for the former.

- Terminal symbols are bytes expressed in hexadecimal notation: `0x0F`.
- Nonterminal symbols are written in typewriter font: `valtype`, `instr`.
- B^n is a sequence of $n \geq 0$ iterations of B.
- B^* is a possibly empty sequence of iterations of B. (This is a shorthand for B^n used where n is not relevant.)

---

[33] Additional encoding layers – for example, introducing compression – may be defined on top of the basic representation defined here. However, such layers are outside the scope of the current specification.

[32] https://www.iana.org/assignments/media-types/media-types.xhtml

- $B^?$ is an optional occurrence of B. (This is a shorthand for B^n where $n \leq 1$.)
- $x{:}B$ denotes the same language as the nonterminal B, but also binds the variable x to the attribute synthesized for B. A pattern may also be used instead of a variable, e.g., $7{:}B$.
- Productions are written $\mathtt{sym} ::= B_1 \Rightarrow A_1 \mid \dots \mid B_n \Rightarrow A_n$, where each A_i is the attribute that is synthesized for $\mathtt{sym}$ in the given case, usually from attribute variables bound in B_i.
- Some productions are augmented by side conditions in parentheses, which restrict the applicability of the production. They provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production (in the syntax or in an attribute), then all those occurrences must have the same instantiation. (This is a shorthand for a side condition requiring multiple different variables to be equal.)

Note: For example, the binary grammar for number types is given as follows:

$$\begin{array}{llcll} \mathtt{numtype} & ::= & \mathtt{0x7F} & \Rightarrow & \mathsf{i32} \\ & | & \mathtt{0x7E} & \Rightarrow & \mathsf{i64} \\ & | & \mathtt{0x7D} & \Rightarrow & \mathsf{f32} \\ & | & \mathtt{0x7C} & \Rightarrow & \mathsf{f64} \end{array}$$

Consequently, the byte `0x7F` encodes the type i32, `0x7E` encodes the type i64, and so forth. No other byte value is allowed as the encoding of a number type.

The binary grammar for limits is defined as follows:

$$\begin{array}{llcll} \mathtt{limits} & ::= & \mathtt{0x00}~~n{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~\epsilon\} \\ & | & \mathtt{0x01}~~n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~m\} \end{array}$$

That is, a limits pair is encoded as either the byte `0x00` followed by the encoding of a $u32$ value, or the byte `0x01` followed by two such encodings. The variables n and m name the attributes of the respective u32 nonterminals, which in this case are the actual unsigned integers those decode into. The attribute of the complete production then is the abstract syntax for the limit, expressed in terms of the former values.

5.1.2 Auxiliary Notation

When dealing with binary encodings the following notation is also used:

- ϵ denotes the empty byte sequence.
- $||B||$ is the length of the byte sequence generated from the production B in a derivation.

5.1.3 Vectors

Vectors are encoded with their u32 length followed by the encoding of their element sequence.

$$\mathtt{vec}(\mathtt{B}) ::= n{:}\mathtt{u32}~~(x{:}\mathtt{B})^n \Rightarrow x^n$$

5.2 Values

5.2.1 Bytes

Bytes encode themselves.

$$\begin{array}{llcll} \mathtt{byte} & ::= & \mathtt{0x00} & \Rightarrow & \mathtt{0x00} \\ & | & & \dots & \\ & | & \mathtt{0xFF} & \Rightarrow & \mathtt{0xFF} \end{array}$$

5.2.2 Integers

All integers are encoded using the LEB128[34] variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in unsigned LEB128[35] format. As an additional constraint, the total number of bytes encoding a value of type uN must not exceed $\mathrm{ceil}(N/7)$ bytes.

$$\begin{array}{llclll} \mathtt{u}N & ::= & n{:}\mathtt{byte} & \Rightarrow & n & (\mathrm{if}\ n < 2^7 \wedge n < 2^N) \\ & | & n{:}\mathtt{byte}\ \ m{:}\mathtt{u}(N{-}7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (\mathrm{if}\ n \geq 2^7 \wedge N > 7) \end{array}$$

Signed integers are encoded in signed LEB128[36] format, which uses a two's complement representation. As an additional constraint, the total number of bytes encoding a value of type sN must not exceed $\mathrm{ceil}(N/7)$ bytes.

$$\begin{array}{llclll} \mathtt{s}N & ::= & n{:}\mathtt{byte} & \Rightarrow & n & (\mathrm{if}\ n < 2^6 \wedge n < 2^{N-1}) \\ & | & n{:}\mathtt{byte} & \Rightarrow & n - 2^7 & (\mathrm{if}\ 2^6 \leq n < 2^7 \wedge n \geq 2^7 - 2^{N-1}) \\ & | & n{:}\mathtt{byte}\ \ m{:}\mathtt{s}(N{-}7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (\mathrm{if}\ n \geq 2^7 \wedge N > 7) \end{array}$$

Uninterpreted integers are encoded as signed integers.

$$\begin{array}{llclll} \mathtt{i}N & ::= & n{:}\mathtt{s}N & \Rightarrow & i & (\mathrm{if}\ n = \mathrm{signed}_N(i)) \end{array}$$

Note: The side conditions $N > 7$ in the productions for non-terminal bytes of the u and s encodings restrict the encoding's length. However, "trailing zeros" are still allowed within these bounds. For example, `0x03` and `0x83 0x00` are both well-formed encodings for the value 3 as a *u8*. Similarly, either of `0x7e` and `0xFE 0x7F` and `0xFE 0xFF 0x7F` are well-formed encodings of the value -2 as a *s16*.

The side conditions on the value n of terminal bytes further enforce that any unused bits in these bytes must be 0 for positive values and 1 for negative ones. For example, `0x83 0x10` is malformed as a *u8* encoding. Similarly, both `0x83 0x3E` and `0xFF 0x7B` are malformed as *s8* encodings.

5.2.3 Floating-Point

Floating-point values are encoded directly by their IEEE 754[37] (Section 3.4) bit pattern in little endian[38] byte order:

$$\begin{array}{llcll} \mathtt{f}N & ::= & b^*{:}\,\mathtt{byte}^{N/8} & \Rightarrow & \mathrm{bytes}_{fN}^{-1}(b^*) \end{array}$$

---

[34] https://en.wikipedia.org/wiki/LEB128

[35] https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128

[36] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

[37] https://ieeexplore.ieee.org/document/8766229

[38] https://en.wikipedia.org/wiki/Endianness#Little-endian

5.2.4 Names

Names are encoded as a vector of bytes containing the Unicode[39] (Section 3.9) UTF-8 encoding of the name's character sequence.

$$\mathtt{name} \ ::= \ b^*{:}\mathtt{vec}(\mathtt{byte}) \ \Rightarrow \ \mathit{name} \qquad (\text{if } \mathrm{utf8}(\mathit{name}) = b^*)$$

The auxiliary utf8 function expressing this encoding is defined as follows:

$$\begin{array}{lcll}
\mathrm{utf8}(c^*) & = & (\mathrm{utf8}(c))^* & \\
\mathrm{utf8}(c) & = & b & (\text{if } c < \mathrm{U{+}80} \\
 & & & \quad \wedge\ c = b) \\
\mathrm{utf8}(c) & = & b_1\ b_2 & (\text{if } \mathrm{U{+}80} \leq c < \mathrm{U{+}800} \\
 & & & \quad \wedge\ c = 2^6(b_1 - \mathtt{0xC0}) + (b_2 - \mathtt{0x80})) \\
\mathrm{utf8}(c) & = & b_1\ b_2\ b_3 & (\text{if } \mathrm{U{+}800} \leq c < \mathrm{U{+}D800} \vee \mathrm{U{+}E000} \leq c < \mathrm{U{+}10000} \\
 & & & \quad \wedge\ c = 2^{12}(b_1 - \mathtt{0xE0}) + 2^6(b_2 - \mathtt{0x80}) + (b_3 - \mathtt{0x80})) \\
\mathrm{utf8}(c) & = & b_1\ b_2\ b_3\ b_4 & (\text{if } \mathrm{U{+}10000} \leq c < \mathrm{U{+}110000} \\
 & & & \quad \wedge\ c = 2^{18}(b_1 - \mathtt{0xF0}) + 2^{12}(b_2 - \mathtt{0x80}) + 2^6(b_3 - \mathtt{0x80}) + (b_4 - \mathtt{0x80})) \\
\end{array}$$

where $b_2, b_3, b_4 < \mathtt{0xC0}$

Note: Unlike in some other formats, name strings are not 0-terminated.

5.3 Types

Note: In some places, possible types include both type constructors or types denoted by type indices. Thus, the binary format for type constructors corresponds to the encodings of small negative sN values, such that they can unambiguously occur in the same place as (positive) type indices.

5.3.1 Number Types

Number types are encoded by a single byte.

$$\begin{array}{llcll}
\mathtt{numtype} & ::= & \mathtt{0x7F} & \Rightarrow & \mathsf{i32} \\
 & | & \mathtt{0x7E} & \Rightarrow & \mathsf{i64} \\
 & | & \mathtt{0x7D} & \Rightarrow & \mathsf{f32} \\
 & | & \mathtt{0x7C} & \Rightarrow & \mathsf{f64} \\
\end{array}$$

5.3.2 Vector Types

Vector types are also encoded by a single byte.

$$\mathtt{vectype} \ ::= \ \mathtt{0x7B} \ \Rightarrow \ \mathsf{v128}$$

---

[39] https://www.unicode.org/versions/latest/

5.3.3 Reference Types

Reference types are also encoded by a single byte.

$$\begin{array}{llcll} \mathtt{reftype} & ::= & \mathtt{0x70} & \Rightarrow & \mathsf{funcref} \\ & | & \mathtt{0x6F} & \Rightarrow & \mathsf{externref} \end{array}$$

5.3.4 Value Types

Value types are encoded with their respective encoding as a number type, vector type, or reference type.

$$\begin{array}{llcll} \mathtt{valtype} & ::= & t{:}\mathtt{numtype} & \Rightarrow & t \\ & | & t{:}\mathtt{vectype} & \Rightarrow & t \\ & | & t{:}\mathtt{reftype} & \Rightarrow & t \end{array}$$

Note: Value types can occur in contexts where type indices are also allowed, such as in the case of block types. Thus, the binary format for types corresponds to the signed LEB128[40] encoding of small negative sN values, so that they can coexist with (positive) type indices in the future.

5.3.5 Result Types

Result types are encoded by the respective vectors of value types.

$$\begin{array}{llcll} \mathtt{resulttype} & ::= & t^*{:}\,\mathtt{vec}(\mathtt{valtype}) & \Rightarrow & [t^*] \end{array}$$

5.3.6 Function Types

Function types are encoded by the byte `0x60` followed by the respective vectors of parameter and result types.

$$\begin{array}{llcll} \mathtt{functype} & ::= & \mathtt{0x60}~~rt_1{:}\,\mathtt{resulttype}~~rt_2{:}\,\mathtt{resulttype} & \Rightarrow & rt_1 \rightarrow rt_2 \end{array}$$

5.3.7 Limits

Limits are encoded with a preceding flag indicating whether a maximum is present.

$$\begin{array}{llcll} \mathtt{limits} & ::= & \mathtt{0x00}~~n{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~\epsilon\} \\ & | & \mathtt{0x01}~~n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~m\} \end{array}$$

5.3.8 Memory Types

Memory types are encoded with their limits.

$$\begin{array}{llcll} \mathtt{memtype} & ::= & \mathit{lim}{:}\mathtt{limits} & \Rightarrow & \mathit{lim} \end{array}$$

---

[40] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

5.3.9 Table Types

Table types are encoded with their limits and the encoding of their element reference type.

$$\begin{array}{llllll} \mathtt{tabletype} & ::= & et{:}\mathtt{reftype}~~lim{:}\mathtt{limits} & \Rightarrow & lim~et \end{array}$$

5.3.10 Global Types

Global types are encoded by their value type and a flag for their mutability.

$$\begin{array}{lllll} \mathtt{globaltype} & ::= & t{:}\mathtt{valtype}~~m{:}\mathtt{mut} & \Rightarrow & m~t \\ \mathtt{mut} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{const} \\ & | & \mathtt{0x01} & \Rightarrow & \mathsf{var} \end{array}$$

5.4 Instructions

Instructions are encoded by *opcodes*. Each opcode is represented by a single byte, and is followed by the instruction's immediate arguments, where present. The only exception are structured control instructions, which consist of several opcodes bracketing their nested instruction sequences.

Note: Gaps in the byte code ranges for encoding instructions are reserved for future extensions.

5.4.1 Control Instructions

Control instructions have varying encodings. For structured instructions, the instruction sequences forming nested blocks are terminated with explicit opcodes for end and else.

Block types are encoded in special compressed form, by either the byte `0x40` indicating the empty type, as a single value type, or as a type index encoded as a positive signed integer.

$$\begin{array}{llllll} \mathtt{blocktype} & ::= & \mathtt{0x40} & \Rightarrow & \epsilon \\ & | & t{:}\mathtt{valtype} & \Rightarrow & t \\ & | & x{:}\mathtt{s33} & \Rightarrow & x & (\mathrm{if}~x \geq 0) \\ \mathtt{instr} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{unreachable} \\ & | & \mathtt{0x01} & \Rightarrow & \mathsf{nop} \\ & | & \mathtt{0x02}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{block}~bt~in^*~\mathsf{end} \\ & | & \mathtt{0x03}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{loop}~bt~in^*~\mathsf{end} \\ & | & \mathtt{0x04}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{if}~bt~in^*~\mathsf{else}~\epsilon~\mathsf{end} \\ & | & \mathtt{0x04}~~bt{:}\mathtt{blocktype}~~(in_1{:}\mathtt{instr})^*~~\mathtt{0x05}~~(in_2{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{if}~bt~in_1^*~\mathsf{else}~in_2^*~\mathsf{end} \\ & | & \mathtt{0x0C}~~l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br}~l \\ & | & \mathtt{0x0D}~~l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br\_if}~l \\ & | & \mathtt{0x0E}~~l^*{:}\mathtt{vec}(\mathtt{labelidx})~~l_N{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br\_table}~l^*~l_N \\ & | & \mathtt{0x0F} & \Rightarrow & \mathsf{return} \\ & | & \mathtt{0x10}~~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{call}~x \\ & | & \mathtt{0x11}~~y{:}\mathtt{typeidx}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{call\_indirect}~x~y \end{array}$$

Note: The else opcode `0x05` in the encoding of an if instruction can be omitted if the following instruction sequence is empty.

Unlike any other occurrence, the type index in a block type is encoded as a positive signed integer, so that its signed LEB128 bit pattern cannot collide with the encoding of value types or the special code `0x40`, which correspond to the LEB128 encoding of negative integers. To avoid any loss in the range of allowed indices, it is treated as a 33 bit signed integer.

5.4.2 Reference Instructions

Reference instructions are represented by single byte codes.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xD0}~~t{:}\mathtt{reftype} & \Rightarrow & \mathsf{ref.null}~t \\
& | & \mathtt{0xD1} & \Rightarrow & \mathsf{ref.is\_null} \\
& | & \mathtt{0xD2}~~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{ref.func}~x \\
\end{array}
$$

5.4.3 Parametric Instructions

Parametric instructions are represented by single byte codes, possibly followed by a type annotation.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x1A} & \Rightarrow & \mathsf{drop} \\
& | & \mathtt{0x1B} & \Rightarrow & \mathsf{select} \\
& | & \mathtt{0x1C}~~t^*{:}\mathtt{vec}(\mathtt{valtype}) & \Rightarrow & \mathsf{select}~t^* \\
\end{array}
$$

5.4.4 Variable Instructions

Variable instructions are represented by byte codes followed by the encoding of the respective index.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x20}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{local.get}~x \\
& | & \mathtt{0x21}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{local.set}~x \\
& | & \mathtt{0x22}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{local.tee}~x \\
& | & \mathtt{0x23}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{global.get}~x \\
& | & \mathtt{0x24}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{global.set}~x \\
\end{array}
$$

5.4.5 Table Instructions

Table instructions are represented either by a single byte or a one byte prefix followed by a variable-length unsigned integer.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x25}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.get}~x \\
& | & \mathtt{0x26}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.set}~x \\
& | & \mathtt{0xFC}~~12{:}\mathtt{u32}~~y{:}\mathtt{elemidx}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.init}~x~y \\
& | & \mathtt{0xFC}~~13{:}\mathtt{u32}~~x{:}\mathtt{elemidx} & \Rightarrow & \mathsf{elem.drop}~x \\
& | & \mathtt{0xFC}~~14{:}\mathtt{u32}~~x{:}\mathtt{tableidx}~~y{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.copy}~x~y \\
& | & \mathtt{0xFC}~~15{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.grow}~x \\
& | & \mathtt{0xFC}~~16{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.size}~x \\
& | & \mathtt{0xFC}~~17{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.fill}~x \\
\end{array}
$$

5.4.6 Memory Instructions

Each variant of memory instruction is encoded with a different byte code. Loads and stores are followed by the encoding of their *memarg* immediate.

$$
\begin{array}{llclll}
\mathtt{memarg} & ::= & a{:}\mathtt{u32}~~o{:}\mathtt{u32} & \Rightarrow & \{\mathsf{align}~a,~\mathsf{offset}~o\} \\
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x28}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load}~m \\
& | & \mathtt{0x29}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load}~m \\
& | & \mathtt{0x2A}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{f32.load}~m \\
& | & \mathtt{0x2B}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{f64.load}~m \\
& | & \mathtt{0x2C}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load8\_s}~m \\
& | & \mathtt{0x2D}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load8\_u}~m \\
& | & \mathtt{0x2E}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load16\_s}~m \\
& | & \mathtt{0x2F}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load16\_u}~m \\
& | & \mathtt{0x30}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load8\_s}~m \\
& | & \mathtt{0x31}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load8\_u}~m \\
& | & \mathtt{0x32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load16\_s}~m \\
& | & \mathtt{0x33}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load16\_u}~m \\
& | & \mathtt{0x34}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load32\_s}~m \\
& | & \mathtt{0x35}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load32\_u}~m \\
& | & \mathtt{0x36}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store}~m \\
& | & \mathtt{0x37}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store}~m \\
& | & \mathtt{0x38}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{f32.store}~m \\
& | & \mathtt{0x39}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{f64.store}~m \\
& | & \mathtt{0x3A}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store8}~m \\
& | & \mathtt{0x3B}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store16}~m \\
& | & \mathtt{0x3C}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store8}~m \\
& | & \mathtt{0x3D}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store16}~m \\
& | & \mathtt{0x3E}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store32}~m \\
& | & \mathtt{0x3F}~~\mathtt{0x00} & \Rightarrow & \mathsf{memory.size} \\
& | & \mathtt{0x40}~~\mathtt{0x00} & \Rightarrow & \mathsf{memory.grow} \\
& | & \mathtt{0xFC}~~8{:}\mathtt{u32}~~x{:}\mathtt{dataidx}~\mathtt{0x00} & \Rightarrow & \mathsf{memory.init}~x \\
& | & \mathtt{0xFC}~~9{:}\mathtt{u32}~~x{:}\mathtt{dataidx} & \Rightarrow & \mathsf{data.drop}~x \\
& | & \mathtt{0xFC}~~10{:}\mathtt{u32}~~\mathtt{0x00}~~\mathtt{0x00} & \Rightarrow & \mathsf{memory.copy} \\
& | & \mathtt{0xFC}~~11{:}\mathtt{u32}~~\mathtt{0x00} & \Rightarrow & \mathsf{memory.fill}
\end{array}
$$

Note: In future versions of WebAssembly, the additional zero bytes occurring in the encoding of the memory.size, memory.grow, memory.copy, and memory.fill instructions may be used to index additional memories.

5.4.7 Atomic Memory Instructions

Each variant of atomic memory instruction is encoded with a different byte code. Loads, stores and RMW instructions are followed by the encoding of their *memarg* immediate.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFE}~~00{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{memory.atomic.notify}~m \\
& | & \mathtt{0xFE}~~01{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{memory.atomic.wait32}~m \\
& | & \mathtt{0xFE}~~02{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{memory.atomic.wait64}~m \\
& | & \mathtt{0xFE}~~03{:}\mathtt{u32}~~\mathtt{0x00} & \Rightarrow & \mathsf{atomic.fence}
\end{array}
$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~16{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.load}~m \\
| & \mathtt{0xFE}~~17{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.load}~m \\
| & \mathtt{0xFE}~~18{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.load8\_u}~m \\
| & \mathtt{0xFE}~~19{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.load16\_u}~m \\
| & \mathtt{0xFE}~~20{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.load8\_u}~m \\
| & \mathtt{0xFE}~~21{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.load16\_u}~m \\
| & \mathtt{0xFE}~~22{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.load32\_u}~m \\
| & \mathtt{0xFE}~~23{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.store}~m \\
| & \mathtt{0xFE}~~24{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.store}~m \\
| & \mathtt{0xFE}~~25{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.store8}~m \\
| & \mathtt{0xFE}~~26{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.store16}~m \\
| & \mathtt{0xFE}~~27{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.store8}~m \\
| & \mathtt{0xFE}~~28{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.store16}~m \\
| & \mathtt{0xFE}~~29{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.store32}~m
\end{array}$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~30{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.add}~m \\
| & \mathtt{0xFE}~~31{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.add}~m \\
| & \mathtt{0xFE}~~32{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.add\_u}~m \\
| & \mathtt{0xFE}~~33{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.add\_u}~m \\
| & \mathtt{0xFE}~~34{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.add\_u}~m \\
| & \mathtt{0xFE}~~35{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.add\_u}~m \\
| & \mathtt{0xFE}~~36{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.add\_u}~m
\end{array}$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~37{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.sub}~m \\
| & \mathtt{0xFE}~~38{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.sub}~m \\
| & \mathtt{0xFE}~~39{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.sub\_u}~m \\
| & \mathtt{0xFE}~~40{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.sub\_u}~m \\
| & \mathtt{0xFE}~~41{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.sub\_u}~m \\
| & \mathtt{0xFE}~~42{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.sub\_u}~m \\
| & \mathtt{0xFE}~~43{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.sub\_u}~m
\end{array}$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~44{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.and}~m \\
| & \mathtt{0xFE}~~45{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.and}~m \\
| & \mathtt{0xFE}~~46{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.and\_u}~m \\
| & \mathtt{0xFE}~~47{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.and\_u}~m \\
| & \mathtt{0xFE}~~48{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.and\_u}~m \\
| & \mathtt{0xFE}~~49{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.and\_u}~m \\
| & \mathtt{0xFE}~~50{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.and\_u}~m
\end{array}$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~51{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.or}~m \\
| & \mathtt{0xFE}~~52{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.or}~m \\
| & \mathtt{0xFE}~~53{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.or\_u}~m \\
| & \mathtt{0xFE}~~54{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.or\_u}~m \\
| & \mathtt{0xFE}~~55{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.or\_u}~m \\
| & \mathtt{0xFE}~~56{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.or\_u}~m \\
| & \mathtt{0xFE}~~57{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.or\_u}~m
\end{array}$$

$$\begin{array}{lllcl}
| & \mathtt{0xFE}~~58{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.xor}~m \\
| & \mathtt{0xFE}~~59{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.xor}~m \\
| & \mathtt{0xFE}~~60{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.xor\_u}~m \\
| & \mathtt{0xFE}~~61{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.xor\_u}~m \\
| & \mathtt{0xFE}~~62{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.xor\_u}~m \\
| & \mathtt{0xFE}~~63{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.xor\_u}~m \\
| & \mathtt{0xFE}~~64{:}\mathtt{u32} & m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.xor\_u}~m
\end{array}$$

$$
\begin{array}{llcl}
| & \mathtt{0xFE}~~65{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.xchg}~m \\
| & \mathtt{0xFE}~~66{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.xchg}~m \\
| & \mathtt{0xFE}~~67{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.xchg\_u}~m \\
| & \mathtt{0xFE}~~68{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.xchg\_u}~m \\
| & \mathtt{0xFE}~~69{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.xchg\_u}~m \\
| & \mathtt{0xFE}~~70{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.xchg\_u}~m \\
| & \mathtt{0xFE}~~71{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.xchg\_u}~m
\end{array}
$$

$$
\begin{array}{llcl}
| & \mathtt{0xFE}~~72{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw.cmpxchg}~m \\
| & \mathtt{0xFE}~~73{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw.cmpxchg}~m \\
| & \mathtt{0xFE}~~74{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw8.cmpxchg\_u}~m \\
| & \mathtt{0xFE}~~75{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.atomic.rmw16.cmpxchg\_u}~m \\
| & \mathtt{0xFE}~~76{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw8.cmpxchg\_u}~m \\
| & \mathtt{0xFE}~~77{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw16.cmpxchg\_u}~m \\
| & \mathtt{0xFE}~~78{:}\mathtt{u32}~~m{:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.atomic.rmw32.cmpxchg\_u}~m
\end{array}
$$

5.4.8 Numeric Instructions

All variants of numeric instructions are represented by separate byte codes.

The const instructions are followed by the respective literal.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
 & | & \mathtt{0x41}~~n{:}\mathtt{i32} & \Rightarrow & \mathsf{i32.const}~n \\
 & | & \mathtt{0x42}~~n{:}\mathtt{i64} & \Rightarrow & \mathsf{i64.const}~n \\
 & | & \mathtt{0x43}~~z{:}\mathtt{f32} & \Rightarrow & \mathsf{f32.const}~z \\
 & | & \mathtt{0x44}~~z{:}\mathtt{f64} & \Rightarrow & \mathsf{f64.const}~z
\end{array}
$$

All other numeric instructions are plain opcodes without any immediates.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
 & | & \mathtt{0x45} & \Rightarrow & \mathsf{i32.eqz} \\
 & | & \mathtt{0x46} & \Rightarrow & \mathsf{i32.eq} \\
 & | & \mathtt{0x47} & \Rightarrow & \mathsf{i32.ne} \\
 & | & \mathtt{0x48} & \Rightarrow & \mathsf{i32.lt\_s} \\
 & | & \mathtt{0x49} & \Rightarrow & \mathsf{i32.lt\_u} \\
 & | & \mathtt{0x4A} & \Rightarrow & \mathsf{i32.gt\_s} \\
 & | & \mathtt{0x4B} & \Rightarrow & \mathsf{i32.gt\_u} \\
 & | & \mathtt{0x4C} & \Rightarrow & \mathsf{i32.le\_s} \\
 & | & \mathtt{0x4D} & \Rightarrow & \mathsf{i32.le\_u} \\
 & | & \mathtt{0x4E} & \Rightarrow & \mathsf{i32.ge\_s} \\
 & | & \mathtt{0x4F} & \Rightarrow & \mathsf{i32.ge\_u} \\
 \\
 & | & \mathtt{0x50} & \Rightarrow & \mathsf{i64.eqz} \\
 & | & \mathtt{0x51} & \Rightarrow & \mathsf{i64.eq} \\
 & | & \mathtt{0x52} & \Rightarrow & \mathsf{i64.ne} \\
 & | & \mathtt{0x53} & \Rightarrow & \mathsf{i64.lt\_s} \\
 & | & \mathtt{0x54} & \Rightarrow & \mathsf{i64.lt\_u} \\
 & | & \mathtt{0x55} & \Rightarrow & \mathsf{i64.gt\_s} \\
 & | & \mathtt{0x56} & \Rightarrow & \mathsf{i64.gt\_u} \\
 & | & \mathtt{0x57} & \Rightarrow & \mathsf{i64.le\_s} \\
 & | & \mathtt{0x58} & \Rightarrow & \mathsf{i64.le\_u} \\
 & | & \mathtt{0x59} & \Rightarrow & \mathsf{i64.ge\_s} \\
 & | & \mathtt{0x5A} & \Rightarrow & \mathsf{i64.ge\_u}
\end{array}
$$

$$
\begin{array}{llll}
| & \mathtt{0x5B} & \Rightarrow & \mathsf{f32.eq} \\
| & \mathtt{0x5C} & \Rightarrow & \mathsf{f32.ne} \\
| & \mathtt{0x5D} & \Rightarrow & \mathsf{f32.lt} \\
| & \mathtt{0x5E} & \Rightarrow & \mathsf{f32.gt} \\
| & \mathtt{0x5F} & \Rightarrow & \mathsf{f32.le} \\
| & \mathtt{0x60} & \Rightarrow & \mathsf{f32.ge}
\end{array}
$$

$$
\begin{array}{llll}
| & \mathtt{0x61} & \Rightarrow & \mathsf{f64.eq} \\
| & \mathtt{0x62} & \Rightarrow & \mathsf{f64.ne} \\
| & \mathtt{0x63} & \Rightarrow & \mathsf{f64.lt} \\
| & \mathtt{0x64} & \Rightarrow & \mathsf{f64.gt} \\
| & \mathtt{0x65} & \Rightarrow & \mathsf{f64.le} \\
| & \mathtt{0x66} & \Rightarrow & \mathsf{f64.ge}
\end{array}
$$

$$
\begin{array}{llll}
| & \mathtt{0x67} & \Rightarrow & \mathsf{i32.clz} \\
| & \mathtt{0x68} & \Rightarrow & \mathsf{i32.ctz} \\
| & \mathtt{0x69} & \Rightarrow & \mathsf{i32.popcnt} \\
| & \mathtt{0x6A} & \Rightarrow & \mathsf{i32.add} \\
| & \mathtt{0x6B} & \Rightarrow & \mathsf{i32.sub} \\
| & \mathtt{0x6C} & \Rightarrow & \mathsf{i32.mul} \\
| & \mathtt{0x6D} & \Rightarrow & \mathsf{i32.div\_s} \\
| & \mathtt{0x6E} & \Rightarrow & \mathsf{i32.div\_u} \\
| & \mathtt{0x6F} & \Rightarrow & \mathsf{i32.rem\_s} \\
| & \mathtt{0x70} & \Rightarrow & \mathsf{i32.rem\_u} \\
| & \mathtt{0x71} & \Rightarrow & \mathsf{i32.and} \\
| & \mathtt{0x72} & \Rightarrow & \mathsf{i32.or} \\
| & \mathtt{0x73} & \Rightarrow & \mathsf{i32.xor} \\
| & \mathtt{0x74} & \Rightarrow & \mathsf{i32.shl} \\
| & \mathtt{0x75} & \Rightarrow & \mathsf{i32.shr\_s} \\
| & \mathtt{0x76} & \Rightarrow & \mathsf{i32.shr\_u} \\
| & \mathtt{0x77} & \Rightarrow & \mathsf{i32.rotl} \\
| & \mathtt{0x78} & \Rightarrow & \mathsf{i32.rotr}
\end{array}
$$

$$
\begin{array}{llll}
| & \mathtt{0x79} & \Rightarrow & \mathsf{i64.clz} \\
| & \mathtt{0x7A} & \Rightarrow & \mathsf{i64.ctz} \\
| & \mathtt{0x7B} & \Rightarrow & \mathsf{i64.popcnt} \\
| & \mathtt{0x7C} & \Rightarrow & \mathsf{i64.add} \\
| & \mathtt{0x7D} & \Rightarrow & \mathsf{i64.sub} \\
| & \mathtt{0x7E} & \Rightarrow & \mathsf{i64.mul} \\
| & \mathtt{0x7F} & \Rightarrow & \mathsf{i64.div\_s} \\
| & \mathtt{0x80} & \Rightarrow & \mathsf{i64.div\_u} \\
| & \mathtt{0x81} & \Rightarrow & \mathsf{i64.rem\_s} \\
| & \mathtt{0x82} & \Rightarrow & \mathsf{i64.rem\_u} \\
| & \mathtt{0x83} & \Rightarrow & \mathsf{i64.and} \\
| & \mathtt{0x84} & \Rightarrow & \mathsf{i64.or} \\
| & \mathtt{0x85} & \Rightarrow & \mathsf{i64.xor} \\
| & \mathtt{0x86} & \Rightarrow & \mathsf{i64.shl} \\
| & \mathtt{0x87} & \Rightarrow & \mathsf{i64.shr\_s} \\
| & \mathtt{0x88} & \Rightarrow & \mathsf{i64.shr\_u} \\
| & \mathtt{0x89} & \Rightarrow & \mathsf{i64.rotl} \\
| & \mathtt{0x8A} & \Rightarrow & \mathsf{i64.rotr}
\end{array}
$$

$$\begin{array}{llll}
\mid & \texttt{0x8B} & \Rightarrow & \mathsf{f32.abs} \\
\mid & \texttt{0x8C} & \Rightarrow & \mathsf{f32.neg} \\
\mid & \texttt{0x8D} & \Rightarrow & \mathsf{f32.ceil} \\
\mid & \texttt{0x8E} & \Rightarrow & \mathsf{f32.floor} \\
\mid & \texttt{0x8F} & \Rightarrow & \mathsf{f32.trunc} \\
\mid & \texttt{0x90} & \Rightarrow & \mathsf{f32.nearest} \\
\mid & \texttt{0x91} & \Rightarrow & \mathsf{f32.sqrt} \\
\mid & \texttt{0x92} & \Rightarrow & \mathsf{f32.add} \\
\mid & \texttt{0x93} & \Rightarrow & \mathsf{f32.sub} \\
\mid & \texttt{0x94} & \Rightarrow & \mathsf{f32.mul} \\
\mid & \texttt{0x95} & \Rightarrow & \mathsf{f32.div} \\
\mid & \texttt{0x96} & \Rightarrow & \mathsf{f32.min} \\
\mid & \texttt{0x97} & \Rightarrow & \mathsf{f32.max} \\
\mid & \texttt{0x98} & \Rightarrow & \mathsf{f32.copysign}
\end{array}$$

$$\begin{array}{llll}
\mid & \texttt{0x99} & \Rightarrow & \mathsf{f64.abs} \\
\mid & \texttt{0x9A} & \Rightarrow & \mathsf{f64.neg} \\
\mid & \texttt{0x9B} & \Rightarrow & \mathsf{f64.ceil} \\
\mid & \texttt{0x9C} & \Rightarrow & \mathsf{f64.floor} \\
\mid & \texttt{0x9D} & \Rightarrow & \mathsf{f64.trunc} \\
\mid & \texttt{0x9E} & \Rightarrow & \mathsf{f64.nearest} \\
\mid & \texttt{0x9F} & \Rightarrow & \mathsf{f64.sqrt} \\
\mid & \texttt{0xA0} & \Rightarrow & \mathsf{f64.add} \\
\mid & \texttt{0xA1} & \Rightarrow & \mathsf{f64.sub} \\
\mid & \texttt{0xA2} & \Rightarrow & \mathsf{f64.mul} \\
\mid & \texttt{0xA3} & \Rightarrow & \mathsf{f64.div} \\
\mid & \texttt{0xA4} & \Rightarrow & \mathsf{f64.min} \\
\mid & \texttt{0xA5} & \Rightarrow & \mathsf{f64.max} \\
\mid & \texttt{0xA6} & \Rightarrow & \mathsf{f64.copysign}
\end{array}$$

$$\begin{array}{llll}
\mid & \texttt{0xA7} & \Rightarrow & \mathsf{i32.wrap\_i64} \\
\mid & \texttt{0xA8} & \Rightarrow & \mathsf{i32.trunc\_f32\_s} \\
\mid & \texttt{0xA9} & \Rightarrow & \mathsf{i32.trunc\_f32\_u} \\
\mid & \texttt{0xAA} & \Rightarrow & \mathsf{i32.trunc\_f64\_s} \\
\mid & \texttt{0xAB} & \Rightarrow & \mathsf{i32.trunc\_f64\_u} \\
\mid & \texttt{0xAC} & \Rightarrow & \mathsf{i64.extend\_i32\_s} \\
\mid & \texttt{0xAD} & \Rightarrow & \mathsf{i64.extend\_i32\_u} \\
\mid & \texttt{0xAE} & \Rightarrow & \mathsf{i64.trunc\_f32\_s} \\
\mid & \texttt{0xAF} & \Rightarrow & \mathsf{i64.trunc\_f32\_u} \\
\mid & \texttt{0xB0} & \Rightarrow & \mathsf{i64.trunc\_f64\_s} \\
\mid & \texttt{0xB1} & \Rightarrow & \mathsf{i64.trunc\_f64\_u} \\
\mid & \texttt{0xB2} & \Rightarrow & \mathsf{f32.convert\_i32\_s} \\
\mid & \texttt{0xB3} & \Rightarrow & \mathsf{f32.convert\_i32\_u} \\
\mid & \texttt{0xB4} & \Rightarrow & \mathsf{f32.convert\_i64\_s} \\
\mid & \texttt{0xB5} & \Rightarrow & \mathsf{f32.convert\_i64\_u} \\
\mid & \texttt{0xB6} & \Rightarrow & \mathsf{f32.demote\_f64} \\
\mid & \texttt{0xB7} & \Rightarrow & \mathsf{f64.convert\_i32\_s} \\
\mid & \texttt{0xB8} & \Rightarrow & \mathsf{f64.convert\_i32\_u} \\
\mid & \texttt{0xB9} & \Rightarrow & \mathsf{f64.convert\_i64\_s} \\
\mid & \texttt{0xBA} & \Rightarrow & \mathsf{f64.convert\_i64\_u} \\
\mid & \texttt{0xBB} & \Rightarrow & \mathsf{f64.promote\_f32} \\
\mid & \texttt{0xBC} & \Rightarrow & \mathsf{i32.reinterpret\_f32} \\
\mid & \texttt{0xBD} & \Rightarrow & \mathsf{i64.reinterpret\_f64} \\
\mid & \texttt{0xBE} & \Rightarrow & \mathsf{f32.reinterpret\_i32} \\
\mid & \texttt{0xBF} & \Rightarrow & \mathsf{f64.reinterpret\_i64}
\end{array}$$

$$\begin{array}{llll}
| & \mathtt{0xC0} & \Rightarrow & \mathsf{i32.extend8\_s} \\
| & \mathtt{0xC1} & \Rightarrow & \mathsf{i32.extend16\_s} \\
| & \mathtt{0xC2} & \Rightarrow & \mathsf{i64.extend8\_s} \\
| & \mathtt{0xC3} & \Rightarrow & \mathsf{i64.extend16\_s} \\
| & \mathtt{0xC4} & \Rightarrow & \mathsf{i64.extend32\_s}
\end{array}$$

The saturating truncation instructions all have a one byte prefix, whereas the actual opcode is encoded by a variable-length unsigned integer.

$$\begin{array}{llllll}
\mathtt{instr} & ::= & \dots & & & \\
& | & \mathtt{0xFC}\ \ 0{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.trunc\_sat\_f32\_s} \\
& | & \mathtt{0xFC}\ \ 1{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.trunc\_sat\_f32\_u} \\
& | & \mathtt{0xFC}\ \ 2{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.trunc\_sat\_f64\_s} \\
& | & \mathtt{0xFC}\ \ 3{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.trunc\_sat\_f64\_u} \\
& | & \mathtt{0xFC}\ \ 4{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.trunc\_sat\_f32\_s} \\
& | & \mathtt{0xFC}\ \ 5{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.trunc\_sat\_f32\_u} \\
& | & \mathtt{0xFC}\ \ 6{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.trunc\_sat\_f64\_s} \\
& | & \mathtt{0xFC}\ \ 7{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.trunc\_sat\_f64\_u}
\end{array}$$

5.4.9 Vector Instructions

All variants of vector instructions are represented by separate byte codes. They all have a one byte prefix, whereas the actual opcode is encoded by a variable-length unsigned integer.

Vector loads and stores are followed by the encoding of their *memarg* immediate.

$$\begin{array}{lllll}
\mathtt{laneidx} & ::= & l{:}\mathtt{byte} & \Rightarrow & l \\
\mathtt{instr} & ::= & \dots & & \\
& | & \mathtt{0xFD}\ \ 0{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load}\ m \\
& | & \mathtt{0xFD}\ \ 1{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8x8\_s}\ m \\
& | & \mathtt{0xFD}\ \ 2{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8x8\_u}\ m \\
& | & \mathtt{0xFD}\ \ 3{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16x4\_s}\ m \\
& | & \mathtt{0xFD}\ \ 4{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16x4\_u}\ m \\
& | & \mathtt{0xFD}\ \ 5{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32x2\_s}\ m \\
& | & \mathtt{0xFD}\ \ 6{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32x2\_u}\ m \\
& | & \mathtt{0xFD}\ \ 7{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8\_splat}\ m \\
& | & \mathtt{0xFD}\ \ 8{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16\_splat}\ m \\
& | & \mathtt{0xFD}\ \ 9{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32\_splat}\ m \\
& | & \mathtt{0xFD}\ \ 10{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load64\_splat}\ m \\
& | & \mathtt{0xFD}\ \ 92{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32\_zero}\ m \\
& | & \mathtt{0xFD}\ \ 93{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load64\_zero}\ m \\
& | & \mathtt{0xFD}\ \ 11{:}\mathtt{u32}\ \ m{:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.store}\ m \\
& | & \mathtt{0xFD}\ \ 84{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load8\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 85{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load16\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 86{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load32\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 87{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load64\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 88{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store8\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 89{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store16\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 90{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store32\_lane}\ m\ l \\
& | & \mathtt{0xFD}\ \ 91{:}\mathtt{u32}\ \ m{:}\mathtt{memarg}\ l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store64\_lane}\ m\ l
\end{array}$$

The const instruction is followed by 16 immediate bytes, which are converted into a *i128* in littleendian byte order:

$$\begin{array}{lllll}
\mathtt{instr} & ::= & \dots & & \\
& | & \mathtt{0xFD}\ \ 12{:}\mathtt{u32}\ \ (b{:}\mathtt{byte})^{16} & \Rightarrow & \mathsf{v128.const}\ \mathrm{bytes}_{\mathsf{i128}}^{-1}(b_0\ \dots\ b_{15})
\end{array}$$

The shuffle instruction is also followed by the encoding of 16 $\mathit{laneidx}$ immediates.

$$\begin{array}{rrlll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~13{:}\mathtt{u32}~~(l{:}\mathtt{laneidx})^{16} & \Rightarrow & \mathsf{i8x16.shuffle}~l^{16}
\end{array}$$

extract_lane and replace_lane instructions are followed by the encoding of a $\mathit{laneidx}$ immediate.

$$\begin{array}{rrlll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~21{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.extract\_lane\_s}~l \\
& | & \mathtt{0xFD}~~22{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.extract\_lane\_u}~l \\
& | & \mathtt{0xFD}~~23{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.replace\_lane}~l \\
& | & \mathtt{0xFD}~~24{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.extract\_lane\_s}~l \\
& | & \mathtt{0xFD}~~25{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.extract\_lane\_u}~l \\
& | & \mathtt{0xFD}~~26{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.replace\_lane}~l \\
& | & \mathtt{0xFD}~~27{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i32x4.extract\_lane}~l \\
& | & \mathtt{0xFD}~~28{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i32x4.replace\_lane}~l \\
& | & \mathtt{0xFD}~~29{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i64x2.extract\_lane}~l \\
& | & \mathtt{0xFD}~~30{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i64x2.replace\_lane}~l \\
& | & \mathtt{0xFD}~~31{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f32x4.extract\_lane}~l \\
& | & \mathtt{0xFD}~~32{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f32x4.replace\_lane}~l \\
& | & \mathtt{0xFD}~~33{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f64x2.extract\_lane}~l \\
& | & \mathtt{0xFD}~~34{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f64x2.replace\_lane}~l
\end{array}$$

All other vector instructions are plain opcodes without any immediates.

$$\begin{array}{rrlll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~14{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.swizzle} \\
& | & \mathtt{0xFD}~~15{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.splat} \\
& | & \mathtt{0xFD}~~16{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.splat} \\
& | & \mathtt{0xFD}~~17{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.splat} \\
& | & \mathtt{0xFD}~~18{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.splat} \\
& | & \mathtt{0xFD}~~19{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.splat} \\
& | & \mathtt{0xFD}~~20{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.splat}
\end{array}$$

$$\begin{array}{rrlll}
& | & \mathtt{0xFD}~~35{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.eq} \\
& | & \mathtt{0xFD}~~36{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ne} \\
& | & \mathtt{0xFD}~~37{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.lt\_s} \\
& | & \mathtt{0xFD}~~38{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.lt\_u} \\
& | & \mathtt{0xFD}~~39{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.gt\_s} \\
& | & \mathtt{0xFD}~~40{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.gt\_u} \\
& | & \mathtt{0xFD}~~41{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.le\_s} \\
& | & \mathtt{0xFD}~~42{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.le\_u} \\
& | & \mathtt{0xFD}~~43{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ge\_s} \\
& | & \mathtt{0xFD}~~44{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ge\_u}
\end{array}$$

$$\begin{array}{rrlll}
& | & \mathtt{0xFD}~~45{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.eq} \\
& | & \mathtt{0xFD}~~46{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ne} \\
& | & \mathtt{0xFD}~~47{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.lt\_s} \\
& | & \mathtt{0xFD}~~48{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.lt\_u} \\
& | & \mathtt{0xFD}~~49{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.gt\_s} \\
& | & \mathtt{0xFD}~~50{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.gt\_u} \\
& | & \mathtt{0xFD}~~51{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.le\_s} \\
& | & \mathtt{0xFD}~~52{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.le\_u} \\
& | & \mathtt{0xFD}~~53{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ge\_s} \\
& | & \mathtt{0xFD}~~54{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ge\_u}
\end{array}$$

$$\begin{array}{lllll}
| & \mathtt{0xFD} & 55{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.eq} \\
| & \mathtt{0xFD} & 56{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.ne} \\
| & \mathtt{0xFD} & 57{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.lt\_s} \\
| & \mathtt{0xFD} & 58{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.lt\_u} \\
| & \mathtt{0xFD} & 59{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.gt\_s} \\
| & \mathtt{0xFD} & 60{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.gt\_u} \\
| & \mathtt{0xFD} & 61{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.le\_s} \\
| & \mathtt{0xFD} & 62{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.le\_u} \\
| & \mathtt{0xFD} & 63{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.ge\_s} \\
| & \mathtt{0xFD} & 64{:}\mathrm{u32} & \Rightarrow & \mathrm{i32x4.ge\_u}
\end{array}$$

$$\begin{array}{lllll}
| & \mathtt{0xFD} & 214{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.eq} \\
| & \mathtt{0xFD} & 215{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.ne} \\
| & \mathtt{0xFD} & 216{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.lt\_s} \\
| & \mathtt{0xFD} & 217{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.gt\_s} \\
| & \mathtt{0xFD} & 218{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.le\_s} \\
| & \mathtt{0xFD} & 219{:}\mathrm{u32} & \Rightarrow & \mathrm{i64x2.ge\_s}
\end{array}$$

$$\begin{array}{lllll}
| & \mathtt{0xFD} & 65{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.eq} \\
| & \mathtt{0xFD} & 66{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.ne} \\
| & \mathtt{0xFD} & 67{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.lt} \\
| & \mathtt{0xFD} & 68{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.gt} \\
| & \mathtt{0xFD} & 69{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.le} \\
| & \mathtt{0xFD} & 70{:}\mathrm{u32} & \Rightarrow & \mathrm{f32x4.ge}
\end{array}$$

$$\begin{array}{lllll}
| & \mathtt{0xFD} & 71{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.eq} \\
| & \mathtt{0xFD} & 72{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.ne} \\
| & \mathtt{0xFD} & 73{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.lt} \\
| & \mathtt{0xFD} & 74{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.gt} \\
| & \mathtt{0xFD} & 75{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.le} \\
| & \mathtt{0xFD} & 76{:}\mathrm{u32} & \Rightarrow & \mathrm{f64x2.ge}
\end{array}$$

$$\begin{array}{lllll}
| & \mathtt{0xFD} & 77{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.not} \\
| & \mathtt{0xFD} & 78{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.and} \\
| & \mathtt{0xFD} & 79{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.andnot} \\
| & \mathtt{0xFD} & 80{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.or} \\
| & \mathtt{0xFD} & 81{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.xor} \\
| & \mathtt{0xFD} & 82{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.bitselect} \\
| & \mathtt{0xFD} & 83{:}\mathrm{u32} & \Rightarrow & \mathrm{v128.any\_true}
\end{array}$$

$$\begin{array}{llll}
| & \mathtt{0xFD}~~96{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.abs} \\
| & \mathtt{0xFD}~~97{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.neg} \\
| & \mathtt{0xFD}~~98{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.popcnt} \\
| & \mathtt{0xFD}~~99{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.all\_true} \\
| & \mathtt{0xFD}~~100{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.bitmask} \\
| & \mathtt{0xFD}~~101{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.narrow\_i16x8\_s} \\
| & \mathtt{0xFD}~~102{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.narrow\_i16x8\_u} \\
| & \mathtt{0xFD}~~107{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.shl} \\
| & \mathtt{0xFD}~~108{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.shr\_s} \\
| & \mathtt{0xFD}~~109{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.shr\_u} \\
| & \mathtt{0xFD}~~110{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.add} \\
| & \mathtt{0xFD}~~111{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.add\_sat\_s} \\
| & \mathtt{0xFD}~~112{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.add\_sat\_u} \\
| & \mathtt{0xFD}~~113{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.sub} \\
| & \mathtt{0xFD}~~114{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.sub\_sat\_s} \\
| & \mathtt{0xFD}~~115{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.sub\_sat\_u} \\
| & \mathtt{0xFD}~~118{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.min\_s} \\
| & \mathtt{0xFD}~~119{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.min\_u} \\
| & \mathtt{0xFD}~~120{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.max\_s} \\
| & \mathtt{0xFD}~~121{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.max\_u} \\
| & \mathtt{0xFD}~~123{:}\mathsf{u32} & \Rightarrow & \mathsf{i8x16.avgr\_u}
\end{array}$$

$$\begin{array}{llll}
| & \mathtt{0xFD}~~124{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extadd\_pairwise\_i8x16\_s} \\
| & \mathtt{0xFD}~~125{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extadd\_pairwise\_i8x16\_u} \\
| & \mathtt{0xFD}~~128{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.abs} \\
| & \mathtt{0xFD}~~129{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.neg} \\
| & \mathtt{0xFD}~~130{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.q15mulr\_sat\_s} \\
| & \mathtt{0xFD}~~131{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.all\_true} \\
| & \mathtt{0xFD}~~132{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.bitmask} \\
| & \mathtt{0xFD}~~133{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.narrow\_i32x4\_s} \\
| & \mathtt{0xFD}~~134{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.narrow\_i32x4\_u} \\
| & \mathtt{0xFD}~~135{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extend\_low\_i8x16\_s} \\
| & \mathtt{0xFD}~~136{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extend\_high\_i8x16\_s} \\
| & \mathtt{0xFD}~~137{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extend\_low\_i8x16\_u} \\
| & \mathtt{0xFD}~~138{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extend\_high\_i8x16\_u} \\
| & \mathtt{0xFD}~~139{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.shl} \\
| & \mathtt{0xFD}~~140{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.shr\_s} \\
| & \mathtt{0xFD}~~141{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.shr\_u} \\
| & \mathtt{0xFD}~~142{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.add} \\
| & \mathtt{0xFD}~~143{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.add\_sat\_s} \\
| & \mathtt{0xFD}~~144{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.add\_sat\_u} \\
| & \mathtt{0xFD}~~145{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.sub} \\
| & \mathtt{0xFD}~~146{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.sub\_sat\_s} \\
| & \mathtt{0xFD}~~147{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.sub\_sat\_u} \\
| & \mathtt{0xFD}~~149{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.mul} \\
| & \mathtt{0xFD}~~150{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.min\_s} \\
| & \mathtt{0xFD}~~151{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.min\_u} \\
| & \mathtt{0xFD}~~152{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.max\_s} \\
| & \mathtt{0xFD}~~153{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.max\_u} \\
| & \mathtt{0xFD}~~155{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.avgr\_u} \\
| & \mathtt{0xFD}~~156{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extmul\_low\_i8x16\_s} \\
| & \mathtt{0xFD}~~157{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extmul\_high\_i8x16\_s} \\
| & \mathtt{0xFD}~~158{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extmul\_low\_i8x16\_u} \\
| & \mathtt{0xFD}~~159{:}\mathsf{u32} & \Rightarrow & \mathsf{i16x8.extmul\_high\_i8x16\_u}
\end{array}$$

| 0xFD 126:u32 ⇒ i32x4.extadd_pairwise_i16x8_s
| 0xFD 127:u32 ⇒ i32x4.extadd_pairwise_i16x8_u
| 0xFD 160:u32 ⇒ i32x4.abs
| 0xFD 161:u32 ⇒ i32x4.neg
| 0xFD 163:u32 ⇒ i32x4.all_true
| 0xFD 164:u32 ⇒ i32x4.bitmask
| 0xFD 167:u32 ⇒ i32x4.extend_low_i16x8_s
| 0xFD 168:u32 ⇒ i32x4.extend_high_i16x8_s
| 0xFD 169:u32 ⇒ i32x4.extend_low_i16x8_u
| 0xFD 170:u32 ⇒ i32x4.extend_high_i16x8_u
| 0xFD 171:u32 ⇒ i32x4.shl
| 0xFD 172:u32 ⇒ i32x4.shr_s
| 0xFD 173:u32 ⇒ i32x4.shr_u
| 0xFD 174:u32 ⇒ i32x4.add
| 0xFD 177:u32 ⇒ i32x4.sub
| 0xFD 181:u32 ⇒ i32x4.mul
| 0xFD 182:u32 ⇒ i32x4.min_s
| 0xFD 183:u32 ⇒ i32x4.min_u
| 0xFD 184:u32 ⇒ i32x4.max_s
| 0xFD 185:u32 ⇒ i32x4.max_u
| 0xFD 186:u32 ⇒ i32x4.dot_i16x8_s
| 0xFD 188:u32 ⇒ i32x4.extmul_low_i16x8_s
| 0xFD 189:u32 ⇒ i32x4.extmul_high_i16x8_s
| 0xFD 190:u32 ⇒ i32x4.extmul_low_i16x8_u
| 0xFD 191:u32 ⇒ i32x4.extmul_high_i16x8_u

| 0xFD 192:u32 ⇒ i64x2.abs
| 0xFD 193:u32 ⇒ i64x2.neg
| 0xFD 195:u32 ⇒ i64x2.all_true
| 0xFD 196:u32 ⇒ i64x2.bitmask
| 0xFD 199:u32 ⇒ i64x2.extend_low_i32x4_s
| 0xFD 200:u32 ⇒ i64x2.extend_high_i32x4_s
| 0xFD 201:u32 ⇒ i64x2.extend_low_i32x4_u
| 0xFD 202:u32 ⇒ i64x2.extend_high_i32x4_u
| 0xFD 203:u32 ⇒ i64x2.shl
| 0xFD 204:u32 ⇒ i64x2.shr_s
| 0xFD 205:u32 ⇒ i64x2.shr_u
| 0xFD 206:u32 ⇒ i64x2.add
| 0xFD 209:u32 ⇒ i64x2.sub
| 0xFD 213:u32 ⇒ i64x2.mul
| 0xFD 220:u32 ⇒ i64x2.extmul_low_i32x4_s
| 0xFD 221:u32 ⇒ i64x2.extmul_high_i32x4_s
| 0xFD 222:u32 ⇒ i64x2.extmul_low_i32x4_u
| 0xFD 223:u32 ⇒ i64x2.extmul_high_i32x4_u

$$
\begin{array}{llcl}
| & \mathtt{0xFD}~~103{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.ceil} \\
| & \mathtt{0xFD}~~104{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.floor} \\
| & \mathtt{0xFD}~~105{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.trunc} \\
| & \mathtt{0xFD}~~106{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.nearest} \\
| & \mathtt{0xFD}~~224{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.abs} \\
| & \mathtt{0xFD}~~225{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.neg} \\
| & \mathtt{0xFD}~~227{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.sqrt} \\
| & \mathtt{0xFD}~~228{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.add} \\
| & \mathtt{0xFD}~~229{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.sub} \\
| & \mathtt{0xFD}~~230{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.mul} \\
| & \mathtt{0xFD}~~231{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.div} \\
| & \mathtt{0xFD}~~232{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.min} \\
| & \mathtt{0xFD}~~233{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.max} \\
| & \mathtt{0xFD}~~234{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.pmin} \\
| & \mathtt{0xFD}~~235{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.pmax}
\end{array}
$$

$$
\begin{array}{llcl}
| & \mathtt{0xFD}~~116{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.ceil} \\
| & \mathtt{0xFD}~~117{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.floor} \\
| & \mathtt{0xFD}~~122{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.trunc} \\
| & \mathtt{0xFD}~~148{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.nearest} \\
| & \mathtt{0xFD}~~236{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.abs} \\
| & \mathtt{0xFD}~~237{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.neg} \\
| & \mathtt{0xFD}~~239{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.sqrt} \\
| & \mathtt{0xFD}~~240{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.add} \\
| & \mathtt{0xFD}~~241{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.sub} \\
| & \mathtt{0xFD}~~242{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.mul} \\
| & \mathtt{0xFD}~~243{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.div} \\
| & \mathtt{0xFD}~~244{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.min} \\
| & \mathtt{0xFD}~~245{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.max} \\
| & \mathtt{0xFD}~~246{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.pmin} \\
| & \mathtt{0xFD}~~247{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.pmax}
\end{array}
$$

$$
\begin{array}{llcl}
| & \mathtt{0xFD}~~248{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_s} \\
| & \mathtt{0xFD}~~249{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_u} \\
| & \mathtt{0xFD}~~250{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_s} \\
| & \mathtt{0xFD}~~251{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_u} \\
| & \mathtt{0xFD}~~252{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f64x2\_s\_zero} \\
| & \mathtt{0xFD}~~253{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f64x2\_u\_zero} \\
| & \mathtt{0xFD}~~254{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.convert\_low\_i32x4\_s} \\
| & \mathtt{0xFD}~~255{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.convert\_low\_i32x4\_u} \\
| & \mathtt{0xFD}~~94{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.demote\_f64x2\_zero} \\
| & \mathtt{0xFD}~~95{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.promote\_low\_f32x4}
\end{array}
$$

5.4.10 Expressions

Expressions are encoded by their instruction sequence terminated with an explicit `0x0B` opcode for end.

$$
\begin{array}{llclll}
\mathtt{expr} & ::= & (\mathit{in}{:}\mathtt{instr})^* & \mathtt{0x0B} & \Rightarrow & \mathit{in}^*~\mathsf{end}
\end{array}
$$

5.5 Modules

The binary encoding of modules is organized into *sections*. Most sections correspond to one component of a module record, except that function definitions are split into two sections, separating their type declarations in the function section from their bodies in the code section.

Note: This separation enables *parallel* and *streaming* compilation of the functions in a module.

5.5.1 Indices

All indices are encoded with their respective value.

$$\begin{array}{llcll}
\mathtt{typeidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{funcidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{tableidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{memidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{globalidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{elemidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{dataidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{localidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{labelidx} & ::= & l{:}\mathtt{u32} & \Rightarrow & l
\end{array}$$

5.5.2 Sections

Each section consists of

- a one-byte section *id*,
- the $\mathit{u32}$ *size* of the contents, in bytes,
- the actual *contents*, whose structure is dependent on the section id.

Every section is optional; an omitted section is equivalent to the section being present with empty contents.

The following parameterized grammar rule defines the generic structure of a section with id N and contents described by the grammar $\mathtt{B}$.

$$\begin{array}{llclll}
\mathtt{section}_N(\mathtt{B}) & ::= & N{:}\mathtt{byte}~~\mathit{size}{:}\mathtt{u32}~~\mathit{cont}{:}\mathtt{B} & \Rightarrow & \mathit{cont} & (\text{if } \mathit{size} = ||\mathtt{B}||) \\
 & | & \epsilon & \Rightarrow & \epsilon &
\end{array}$$

For most sections, the contents $\mathtt{B}$ encodes a vector. In these cases, the empty result ϵ is interpreted as the empty vector.

Note: Other than for unknown custom sections, the *size* is not required for decoding, but can be used to skip sections when navigating through a binary. The module is malformed if the size does not match the length of the binary contents $\mathtt{B}$.

The following section ids are used:

Id	Section
0	custom section
1	type section
2	import section
3	function section
4	table section
5	memory section
6	global section
7	export section
8	start section
9	element section
10	code section
11	data section
12	data count section

5.5.3 Custom Section

Custom sections have the id 0. They are intended to be used for debugging information or third-party extensions, and are ignored by the WebAssembly semantics. Their contents consist of a name further identifying the custom section, followed by an uninterpreted sequence of bytes for custom use.

$$\begin{array}{lll} \mathtt{customsec} & ::= & \mathtt{section}_0(\mathtt{custom}) \\ \mathtt{custom} & ::= & \mathtt{name}~\mathtt{byte}^* \end{array}$$

Note: If an implementation interprets the data of a custom section, then errors in that data, or the placement of the section, must not invalidate the module.

5.5.4 Type Section

The *type section* has the id 1. It decodes into a vector of function types that represent the types component of a module.

$$\begin{array}{llll} \mathtt{typesec} & ::= & ft^*{:}\,\mathtt{section}_1(\mathtt{vec}(\mathtt{functype})) & \Rightarrow \quad ft^* \end{array}$$

5.5.5 Import Section

The *import section* has the id 2. It decodes into a vector of imports that represent the imports component of a module.

$$\begin{array}{llclll} \mathtt{importsec} & ::= & im^*{:}\mathtt{section}_2(\mathtt{vec}(\mathtt{import})) & \Rightarrow & im^* \\ \mathtt{import} & ::= & mod{:}\mathtt{name}~~nm{:}\mathtt{name}~~d{:}\mathtt{importdesc} & \Rightarrow & \{\mathsf{module}~mod, \mathsf{name}~nm, \mathsf{desc}~d\} \\ \mathtt{importdesc} & ::= & \mathtt{0x00}~~x{:}\mathtt{typeidx} & \Rightarrow & \mathsf{func}~x \\ & | & \mathtt{0x01}~~tt{:}\mathtt{tabletype} & \Rightarrow & \mathsf{table}~tt \\ & | & \mathtt{0x02}~~mt{:}\mathtt{memtype} & \Rightarrow & \mathsf{mem}~mt \\ & | & \mathtt{0x03}~~gt{:}\mathtt{globaltype} & \Rightarrow & \mathsf{global}~gt \end{array}$$

5.5.6 Function Section

The *function section* has the id 3. It decodes into a vector of type indices that represent the type fields of the functions in the funcs component of a module. The locals and body fields of the respective functions are encoded separately in the code section.

$$\begin{array}{llll} \mathtt{funcsec} & ::= & x^*{:}\mathtt{section}_3(\mathtt{vec}(\mathtt{typeidx})) & \Rightarrow & x^* \end{array}$$

5.5.7 Table Section

The *table section* has the id 4. It decodes into a vector of tables that represent the tables component of a module.

$$\begin{array}{lllll} \mathtt{tablesec} & ::= & \mathit{tab}^*{:}\mathtt{section}_4(\mathtt{vec}(\mathtt{table})) & \Rightarrow & \mathit{tab}^* \\ \mathtt{table} & ::= & \mathit{tt}{:}\mathtt{tabletype} & \Rightarrow & \{\mathsf{type}~\mathit{tt}\} \end{array}$$

5.5.8 Memory Section

The *memory section* has the id 5. It decodes into a vector of memories that represent the mems component of a module.

$$\begin{array}{lllll} \mathtt{memsec} & ::= & \mathit{mem}^*{:}\mathtt{section}_5(\mathtt{vec}(\mathtt{mem})) & \Rightarrow & \mathit{mem}^* \\ \mathtt{mem} & ::= & \mathit{mt}{:}\mathtt{memtype} & \Rightarrow & \{\mathsf{type}~\mathit{mt}\} \end{array}$$

5.5.9 Global Section

The *global section* has the id 6. It decodes into a vector of globals that represent the globals component of a module.

$$\begin{array}{lllll} \mathtt{globalsec} & ::= & \mathit{glob}^*{:}\mathtt{section}_6(\mathtt{vec}(\mathtt{global})) & \Rightarrow & \mathit{glob}^* \\ \mathtt{global} & ::= & \mathit{gt}{:}\mathtt{globaltype}~~e{:}\mathtt{expr} & \Rightarrow & \{\mathsf{type}~\mathit{gt}, \mathsf{init}~e\} \end{array}$$

5.5.10 Export Section

The *export section* has the id 7. It decodes into a vector of exports that represent the exports component of a module.

$$\begin{array}{lllll} \mathtt{exportsec} & ::= & \mathit{ex}^*{:}\mathtt{section}_7(\mathtt{vec}(\mathtt{export})) & \Rightarrow & \mathit{ex}^* \\ \mathtt{export} & ::= & \mathit{nm}{:}\mathtt{name}~~d{:}\mathtt{exportdesc} & \Rightarrow & \{\mathsf{name}~\mathit{nm}, \mathsf{desc}~d\} \\ \mathtt{exportdesc} & ::= & \mathtt{0x00}~~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{func}~x \\ & | & \mathtt{0x01}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table}~x \\ & | & \mathtt{0x02}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{mem}~x \\ & | & \mathtt{0x03}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{global}~x \end{array}$$

5.5.11 Start Section

The *start section* has the id 8. It decodes into an optional start function that represents the start component of a module.

$$\begin{array}{llcll}
\mathtt{startsec} & ::= & st^?{:}\mathtt{section}_8(\mathtt{start}) & \Rightarrow & st^? \\
\mathtt{start} & ::= & x{:}\mathtt{funcidx} & \Rightarrow & \{\mathsf{func}~x\}
\end{array}$$

5.5.12 Element Section

The *element section* has the id 9. It decodes into a vector of element segments that represent the elems component of a module.

$$\begin{array}{llll}
\mathtt{elemsec} & ::= & seg^*{:}\mathtt{section}_9(\mathtt{vec}(\mathtt{elem})) & \Rightarrow & seg^* \\
\mathtt{elem} & ::= & 0{:}\mathtt{u32}~~e{:}\mathtt{expr}~~y^*{:}\mathtt{vec}(\mathtt{funcidx}) & \Rightarrow \\
& & \quad \{\mathsf{type}~\mathsf{funcref}, \mathsf{init}~((\mathsf{ref.func}~y)~\mathsf{end})^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~0, \mathsf{offset}~e\}\} \\
& | & 1{:}\mathtt{u32}~~et:\mathtt{elemkind}~~y^*{:}\mathtt{vec}(\mathtt{funcidx}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~((\mathsf{ref.func}~y)~\mathsf{end})^*, \mathsf{mode}~\mathsf{passive}\} \\
& | & 2{:}\mathtt{u32}~~x{:}\mathtt{tableidx}~~e{:}\mathtt{expr}~~et:\mathtt{elemkind}~~y^*{:}\mathtt{vec}(\mathtt{funcidx}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~((\mathsf{ref.func}~y)~\mathsf{end})^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~e\}\} \\
& | & 3{:}\mathtt{u32}~~et:\mathtt{elemkind}~~y^*{:}\mathtt{vec}(\mathtt{funcidx}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~((\mathsf{ref.func}~y)~\mathsf{end})^*, \mathsf{mode}~\mathsf{declarative}\} \\
& | & 4{:}\mathtt{u32}~~e{:}\mathtt{expr}~~el^*{:}\mathtt{vec}(\mathtt{expr}) & \Rightarrow \\
& & \quad \{\mathsf{type}~\mathsf{funcref}, \mathsf{init}~el^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~0, \mathsf{offset}~e\}\} \\
& | & 5{:}\mathtt{u32}~~et:\mathtt{reftype}~~el^*{:}\mathtt{vec}(\mathtt{expr}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~el^*, \mathsf{mode}~\mathsf{passive}\} \\
& | & 6{:}\mathtt{u32}~~x{:}\mathtt{tableidx}~~e{:}\mathtt{expr}~~et:\mathtt{reftype}~~el^*{:}\mathtt{vec}(\mathtt{expr}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~el^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~e\}\} \\
& | & 7{:}\mathtt{u32}~~et:\mathtt{reftype}~~el^*{:}\mathtt{vec}(\mathtt{expr}) & \Rightarrow \\
& & \quad \{\mathsf{type}~et, \mathsf{init}~el^*, \mathsf{mode}~\mathsf{declarative}\} \\
\mathtt{elemkind} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{funcref}
\end{array}$$

Note: The initial integer can be interpreted as a bitfield. Bit 0 indicates a passive or declarative segment, bit 1 indicates the presence of an explicit table index for an active segment and otherwise distinguishes passive from declarative segments, bit 2 indicates the use of element type and element expressions instead of element kind and element indices.

Additional element kinds may be added in future versions of WebAssembly.

5.5.13 Code Section

The *code section* has the id 10. It decodes into a vector of *code* entries that are pairs of value type vectors and expressions. They represent the locals and body field of the functions in the funcs component of a module. The type fields of the respective functions are encoded separately in the function section.

The encoding of each code entry consists of

- the *u32* *size* of the function code in bytes,
- the actual *function code*, which in turn consists of
  - the declaration of *locals*,
  - the function *body* as an expression.

Local declarations are compressed into a vector whose entries consist of

- a *u32* *count*,

- a value type,

denoting *count* locals of the same value type.

$$\begin{array}{llclll}
\mathtt{codesec} & ::= & \mathit{code}^*{:}\mathtt{section}_{10}(\mathtt{vec}(\mathtt{code})) & \Rightarrow & \mathit{code}^* & \\
\mathtt{code} & ::= & \mathit{size}{:}\mathtt{u32}~~\mathit{code}{:}\mathtt{func} & \Rightarrow & \mathit{code} & (\mathrm{if}~\mathit{size} = ||\mathtt{func}||) \\
\mathtt{func} & ::= & (t^*)^*{:}\mathtt{vec}(\mathtt{locals})~~e{:}\mathtt{expr} & \Rightarrow & \mathrm{concat}((t^*)^*), e & (\mathrm{if}~|\mathrm{concat}((t^*)^*)| < 2^{32}) \\
\mathtt{locals} & ::= & n{:}\mathtt{u32}~~t{:}\mathtt{valtype} & \Rightarrow & t^n &
\end{array}$$

Here, *code* ranges over pairs $(\mathit{valtype}^*, \mathit{expr})$. The meta function $\mathrm{concat}((t^*)^*)$ concatenates all sequences t_i^* in $(t^*)^*$. Any code for which the length of the resulting sequence is out of bounds of the maximum size of a vector is malformed.

Note: Like with sections, the code *size* is not needed for decoding, but can be used to skip functions when navigating through a binary. The module is malformed if a size does not match the length of the respective function code.

5.5.14 Data Section

The *data section* has the id 11. It decodes into a vector of data segments that represent the datas component of a module.

$$\begin{array}{llcll}
\mathtt{datasec} & ::= & \mathit{seg}^*{:}\mathtt{section}_{11}(\mathtt{vec}(\mathtt{data})) & \Rightarrow & \mathit{seg}^* \\
\mathtt{data} & ::= & 0{:}\mathtt{u32}~~e{:}\mathtt{expr}~~b^*{:}\mathtt{vec}(\mathtt{byte}) & \Rightarrow & \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{memory}~0, \mathsf{offset}~e\}\} \\
 & | & 1{:}\mathtt{u32}~~b^*{:}\mathtt{vec}(\mathtt{byte}) & \Rightarrow & \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{passive}\} \\
 & | & 2{:}\mathtt{u32}~~x{:}\mathtt{memidx}~~e{:}\mathtt{expr}~~b^*{:}\mathtt{vec}(\mathtt{byte}) & \Rightarrow & \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{memory}~x, \mathsf{offset}~e\}\}
\end{array}$$

Note: The initial integer can be interpreted as a bitfield. Bit 0 indicates a passive segment, bit 1 indicates the presence of an explicit memory index for an active segment.

In the current version of WebAssembly, at most one memory may be defined or imported in a single module, so all valid active data segments have a memory value of 0.

5.5.15 Data Count Section

The *data count section* has the id 12. It decodes into an optional u32 that represents the number of data segments in the data section. If this count does not match the length of the data segment vector, the module is malformed.

$$\mathtt{datacountsec} ~~::=~~ n^?{:}\mathtt{section}_{12}(\mathtt{u32}) ~~\Rightarrow~~ n^?$$

Note: The data count section is used to simplify single-pass validation. Since the data section occurs after the code section, the memory.init and data.drop instructions would not be able to check whether the data segment index is valid until the data section is read. The data count section occurs before the code section, so a single-pass validator can use this count instead of deferring validation.

5.5.16 Modules

The encoding of a module starts with a preamble containing a 4-byte magic number (the string '`\0asm`') and a version field. The current version of the WebAssembly binary format is 1.

The preamble is followed by a sequence of sections. Custom sections may be inserted at any place in this sequence, while other sections must occur at most once and in the prescribed order. All sections can be empty.

The lengths of vectors produced by the (possibly empty) function and code section must match up.

Similarly, the optional data count must match the length of the data segment vector. Furthermore, it must be present if any data index occurs in the code section.

$$
\begin{array}{llll}
\mathtt{magic} & ::= & \mathtt{0x00}\ \mathtt{0x61}\ \mathtt{0x73}\ \mathtt{0x6D} \\
\mathtt{version} & ::= & \mathtt{0x01}\ \mathtt{0x00}\ \mathtt{0x00}\ \mathtt{0x00} \\
\mathtt{module} & ::= & \mathtt{magic} \\
&& \mathtt{version} \\
&& \mathtt{customsec}^* \\
&& \mathit{functype}^*{:}\,\mathtt{typesec} \\
&& \mathtt{customsec}^* \\
&& \mathit{import}^*{:}\,\mathtt{importsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{typeidx}^n{:}\,\mathtt{funcsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{table}^*{:}\,\mathtt{tablesec} \\
&& \mathtt{customsec}^* \\
&& \mathit{mem}^*{:}\,\mathtt{memsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{global}^*{:}\,\mathtt{globalsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{export}^*{:}\,\mathtt{exportsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{start}^?{:}\,\mathtt{startsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{elem}^*{:}\,\mathtt{elemsec} \\
&& \mathtt{customsec}^* \\
&& m^?{:}\,\mathtt{datacountsec} \\
&& \mathtt{customsec}^* \\
&& \mathit{code}^n{:}\,\mathtt{codesec} \\
&& \mathtt{customsec}^* \\
&& \mathit{data}^m{:}\,\mathtt{datasec} \\
&& \mathtt{customsec}^* \quad \Rightarrow & \{\, \mathsf{types}\ \mathit{functype}^*, \\
&&& \ \ \mathsf{funcs}\ \mathit{func}^n, \\
&&& \ \ \mathsf{tables}\ \mathit{table}^*, \\
&&& \ \ \mathsf{mems}\ \mathit{mem}^*, \\
&&& \ \ \mathsf{globals}\ \mathit{global}^*, \\
&&& \ \ \mathsf{elems}\ \mathit{elem}^*, \\
&&& \ \ \mathsf{datas}\ \mathit{data}^m, \\
&&& \ \ \mathsf{start}\ \mathit{start}^?, \\
&&& \ \ \mathsf{imports}\ \mathit{import}^*, \\
&&& \ \ \mathsf{exports}\ \mathit{export}^*\ \} \\
&& (\text{if } m^? \neq \epsilon \vee \mathrm{dataidx}(\mathit{code}^n) = \emptyset) &
\end{array}
$$

where for each t_i^*, e_i in code^n,

$$\mathit{func}^n[i] = \{\mathsf{type}\ \mathit{typeidx}^n[i], \mathsf{locals}\ t_i^*, \mathsf{body}\ e_i\}$$

Note: The version of the WebAssembly binary format may increase in the future if backward-incompatible changes have to be made to the format. However, such changes are expected to occur very infrequently, if ever. The binary format is intended to be forward-compatible, such that future extensions can be made without incrementing its version.

CHAPTER 6

Text Format

6.1 Conventions

The textual format for WebAssembly modules is a rendering of their abstract syntax into S-expressions[41].

Like the binary format, the text format is defined by an *attribute grammar*. A text string is a well-formed description of a module if and only if it is generated by the grammar. Each production of this grammar has at most one synthesized attribute: the abstract syntax that the respective character sequence expresses. Thus, the attribute grammar implicitly defines a *parsing* function. Some productions also take a context as an inherited attribute that records bound identifiers.

Except for a few exceptions, the core of the text grammar closely mirrors the grammar of the abstract syntax. However, it also defines a number of *abbreviations* that are "syntactic sugar" over the core syntax.

The recommended extension for files containing WebAssembly modules in text format is "`.wat`". Files with this extension are assumed to be encoded in UTF-8, as per Unicode[42] (Section 2.5).

6.1.1 Grammar

The following conventions are adopted in defining grammar rules of the text format. They mirror the conventions used for abstract syntax and for the binary format. In order to distinguish symbols of the textual syntax from symbols of the abstract syntax, `typewriter` font is adopted for the former.

- Terminal symbols are either literal strings of characters enclosed in quotes or expressed as Unicode[43] scalar values: '`module`', U+0A. (All characters written literally are unambiguously drawn from the 7-bit ASCII[44] subset of Unicode.)
- Nonterminal symbols are written in typewriter font: `valtype`, `instr`.
- T^n is a sequence of $n \geq 0$ iterations of T.
- T^* is a possibly empty sequence of iterations of T. (This is a shorthand for T^n used where n is not relevant.)
- T^+ is a sequence of one or more iterations of T. (This is a shorthand for T^n where $n \geq 1$.)
- $T^?$ is an optional occurrence of T. (This is a shorthand for T^n where $n \leq 1$.)

---

[41] https://en.wikipedia.org/wiki/S-expression

[42] https://www.unicode.org/versions/latest/

[43] https://www.unicode.org/versions/latest/

[44] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

- $x{:}T$ denotes the same language as the nonterminal T, but also binds the variable x to the attribute synthesized for T. A pattern may also be used instead of a variable, e.g., $(x, y){:}T$.
- Productions are written $\mathtt{sym} ::= T_1 \Rightarrow A_1 \mid \dots \mid T_n \Rightarrow A_n$, where each A_i is the attribute that is synthesized for `sym` in the given case, usually from attribute variables bound in T_i.
- Some productions are augmented by side conditions in parentheses, which restrict the applicability of the production. They provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production (in the syntax or in an attribute), then all those occurrences must have the same instantiation.
- A distinction is made between *lexical* and *syntactic* productions. For the latter, arbitrary white space is allowed in any place where the grammar contains spaces. The productions defining lexical syntax and the syntax of values are considered lexical, all others are syntactic.

Note: For example, the textual grammar for number types is given as follows:

$$\begin{array}{llcll}
\mathtt{numtype} & ::= & \text{`}\mathtt{i32}\text{'} & \Rightarrow & \mathsf{i32} \\
& | & \text{`}\mathtt{i64}\text{'} & \Rightarrow & \mathsf{i64} \\
& | & \text{`}\mathtt{f32}\text{'} & \Rightarrow & \mathsf{f32} \\
& | & \text{`}\mathtt{f64}\text{'} & \Rightarrow & \mathsf{f64}
\end{array}$$

The textual grammar for limits is defined as follows:

$$\begin{array}{llcll}
\mathtt{limits} & ::= & n{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~\epsilon\} \\
& | & n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}~n, \mathsf{max}~m\}
\end{array}$$

The variables n and m name the attributes of the respective `u32` nonterminals, which in this case are the actual unsigned integers those parse into. The attribute of the complete production then is the abstract syntax for the limit, expressed in terms of the former values.

6.1.2 Abbreviations

In addition to the core grammar, which corresponds directly to the abstract syntax, the textual syntax also defines a number of *abbreviations* that can be used for convenience and readability.

Abbreviations are defined by *rewrite rules* specifying their expansion into the core syntax:

$$\mathit{abbreviation~syntax} \quad \equiv \quad \mathit{expanded~syntax}$$

These expansions are assumed to be applied, recursively and in order of appearance, before applying the core grammar rules to construct the abstract syntax.

6.1.3 Contexts

The text format allows the use of symbolic identifiers in place of indices. To resolve these identifiers into concrete indices, some grammar productions are indexed by an *identifier context* I as a synthesized attribute that records the declared identifiers in each index space. In addition, the context records the types defined in the module, so that parameter indices can be computed for functions.

It is convenient to define identifier contexts as records I with abstract syntax as follows:

$$\begin{array}{llll} I & ::= & \{ & \mathsf{types} & (\mathtt{id}^?)^*, \\ &&& \mathsf{funcs} & (\mathtt{id}^?)^*, \\ &&& \mathsf{tables} & (\mathtt{id}^?)^*, \\ &&& \mathsf{mems} & (\mathtt{id}^?)^*, \\ &&& \mathsf{globals} & (\mathtt{id}^?)^*, \\ &&& \mathsf{elem} & (\mathtt{id}^?)^*, \\ &&& \mathsf{data} & (\mathtt{id}^?)^*, \\ &&& \mathsf{locals} & (\mathtt{id}^?)^*, \\ &&& \mathsf{labels} & (\mathtt{id}^?)^*, \\ &&& \mathsf{typedefs} & \mathit{functype}^* ~\} \end{array}$$

For each index space, such a context contains the list of identifiers assigned to the defined indices. Unnamed indices are associated with empty (ϵ) entries in these lists.

An identifier context is *well-formed* if no index space contains duplicate identifiers.

Conventions

To avoid unnecessary clutter, empty components are omitted when writing out identifier contexts. For example, the record $\{\}$ is shorthand for an identifier context whose components are all empty.

6.1.4 Vectors

Vectors are written as plain sequences, but with a restriction on the length of these sequence.

$$\mathtt{vec}(\mathtt{A}) \quad ::= \quad (x{:}\mathtt{A})^n \quad \Rightarrow \quad x^n \qquad (\text{if } n < 2^{32})$$

6.2 Lexical Format

6.2.1 Characters

The text format assigns meaning to *source text*, which consists of a sequence of *characters*. Characters are assumed to be represented as valid Unicode[45] (Section 2.4) *scalar values*.

$$\begin{array}{lll} \mathtt{source} & ::= & \mathtt{char}^* \\ \mathtt{char} & ::= & \mathrm{U{+}00} ~|~ \dots ~|~ \mathrm{U{+}D7FF} ~|~ \mathrm{U{+}E000} ~|~ \dots ~|~ \mathrm{U{+}10FFFF} \end{array}$$

Note: While source text may contain any Unicode character in comments or string literals, the rest of the grammar is formed exclusively from the characters supported by the 7-bit ASCII[46] subset of Unicode.

---

[45] https://www.unicode.org/versions/latest/

[46] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

6.2.2 Tokens

The character stream in the source text is divided, from left to right, into a sequence of *tokens*, as defined by the following grammar.

$$
\begin{array}{llll}
\mathtt{token} & ::= & \mathtt{keyword} \mid \mathtt{u}N \mid \mathtt{s}N \mid \mathtt{f}N \mid \mathtt{string} \mid \mathtt{id} \mid \text{‘(’} \mid \text{‘)’} \mid \mathtt{reserved} & \\
\mathtt{keyword} & ::= & (\text{‘}\mathtt{a}\text{’} \mid \dots \mid \text{‘}\mathtt{z}\text{’})~\mathtt{idchar}^* & (\text{if occurring as a literal terminal in the grammar}) \\
\mathtt{reserved} & ::= & (\mathtt{idchar} \mid \mathtt{string})^+ &
\end{array}
$$

Tokens are formed from the input character stream according to the *longest match* rule. That is, the next token always consists of the longest possible sequence of characters that is recognized by the above lexical grammar. Tokens can be separated by white space, but except for strings, they cannot themselves contain whitespace.

Keyword tokens are defined either implicitly by an occurrence of a terminal symbol in literal form, such as ‘`keyword`’, in a syntactic production of this chapter, or explicitly where they arise in this chapter.

Any token that does not fall into any of the other categories is considered *reserved*, and cannot occur in source text.

Note: The effect of defining the set of reserved tokens is that all tokens must be separated by either parentheses, white space, or comments. For example, ‘`0$x`’ is a single reserved token, as is ‘`"a""b"`’. Consequently, they are not recognized as two separate tokens ‘`0`’ and ‘`$x`’, or ”a” and ”b”, respectively, but instead disallowed. This property of tokenization is not affected by the fact that the definition of reserved tokens overlaps with other token classes.

6.2.3 White Space

White space is any sequence of literal space characters, formatting characters, or comments. The allowed formatting characters correspond to a subset of the ASCII[47] *format effectors*, namely, *horizontal tabulation* (U+09), *line feed* (U+0A), and *carriage return* (U+0D).

$$
\begin{array}{lll}
\mathtt{space} & ::= & (\text{‘ ’} \mid \mathtt{format} \mid \mathtt{comment})^* \\
\mathtt{format} & ::= & \mathrm{U{+}09} \mid \mathrm{U{+}0A} \mid \mathrm{U{+}0D}
\end{array}
$$

The only relevance of white space is to separate tokens. It is otherwise ignored.

6.2.4 Comments

A *comment* can either be a *line comment*, started with a double semicolon ‘;;’ and extending to the end of the line, or a *block comment*, enclosed in delimiters ‘(;’ … ‘;)’. Block comments can be nested.

$$
\begin{array}{llll}
\mathtt{comment} & ::= & \mathtt{linecomment} \mid \mathtt{blockcomment} & \\
\mathtt{linecomment} & ::= & \text{‘;;’}~\mathtt{linechar}^*~(\mathrm{U{+}0A} \mid \mathtt{eof}) & \\
\mathtt{linechar} & ::= & c{:}\mathtt{char} & (\text{if } c \neq \mathrm{U{+}0A}) \\
\mathtt{blockcomment} & ::= & \text{‘(;’}~\mathtt{blockchar}^*~\text{‘;)’} & \\
\mathtt{blockchar} & ::= & c{:}\mathtt{char} & (\text{if } c \neq \text{‘;’} \wedge c \neq \text{‘(’}) \\
& \mid & \text{‘;’} & (\text{if the next character is not ‘)’}) \\
& \mid & \text{‘(’} & (\text{if the next character is not ‘;’}) \\
& \mid & \mathtt{blockcomment} &
\end{array}
$$

Here, the pseudo token `eof` indicates the end of the input. The *look-ahead* restrictions on the productions for `blockchar` disambiguate the grammar such that only well-bracketed uses of block comment delimiters are allowed.

Note: Any formatting and control characters are allowed inside comments.

---

[47] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

6.3 Values

The grammar productions in this section define *lexical syntax*, hence no white space is allowed.

6.3.1 Integers

All integers can be written in either decimal or hexadecimal notation. In both cases, digits can optionally be separated by underscores.

$$
\begin{array}{llll}
\mathtt{sign} & ::= & \epsilon \Rightarrow + ~|~ \text{`}\mathtt{+}\text{'} \Rightarrow + ~|~ \text{`}\mathtt{-}\text{'} \Rightarrow - \\
\mathtt{digit} & ::= & \text{`}\mathtt{0}\text{'} \Rightarrow 0 ~|~ \dots ~|~ \text{`}\mathtt{9}\text{'} \Rightarrow 9 \\
\mathtt{hexdigit} & ::= & d{:}\mathtt{digit} \Rightarrow d \\
& | & \text{`}\mathtt{A}\text{'} \Rightarrow 10 ~|~ \dots ~|~ \text{`}\mathtt{F}\text{'} \Rightarrow 15 \\
& | & \text{`}\mathtt{a}\text{'} \Rightarrow 10 ~|~ \dots ~|~ \text{`}\mathtt{f}\text{'} \Rightarrow 15
\end{array}
$$

$$
\begin{array}{llllll}
\mathtt{num} & ::= & d{:}\mathtt{digit} & \Rightarrow & d \\
& | & n{:}\mathtt{num}~\text{`}\mathtt{\_}\text{'}^?~d{:}\mathtt{digit} & \Rightarrow & 10 \cdot n + d \\
\mathtt{hexnum} & ::= & h{:}\mathtt{hexdigit} & \Rightarrow & h \\
& | & n{:}\mathtt{hexnum}~\text{`}\mathtt{\_}\text{'}^?~h{:}\mathtt{hexdigit} & \Rightarrow & 16 \cdot n + h
\end{array}
$$

The allowed syntax for integer literals depends on size and signedness. Moreover, their value must lie within the range of the respective type.

$$
\begin{array}{llllll}
\mathtt{u}N & ::= & n{:}\mathtt{num} & \Rightarrow & n & (\text{if } n < 2^N) \\
& | & \text{`}\mathtt{0x}\text{'}~n{:}\mathtt{hexnum} & \Rightarrow & n & (\text{if } n < 2^N) \\
\mathtt{s}N & ::= & \pm{:}\mathtt{sign}~n{:}\mathtt{num} & \Rightarrow & \pm n & (\text{if } -2^{N-1} \leq \pm n < 2^{N-1}) \\
& | & \pm{:}\mathtt{sign}~\text{`}\mathtt{0x}\text{'}~n{:}\mathtt{hexnum} & \Rightarrow & \pm n & (\text{if } -2^{N-1} \leq \pm n < 2^{N-1})
\end{array}
$$

Uninterpreted integers can be written as either signed or unsigned, and are normalized to unsigned in the abstract syntax.

$$
\begin{array}{llllll}
\mathtt{i}N & ::= & n{:}\mathtt{u}N & \Rightarrow & n \\
& | & i{:}\mathtt{s}N & \Rightarrow & n & (\text{if } i = \mathrm{signed}(n))
\end{array}
$$

6.3.2 Floating-Point

Floating-point values can be represented in either decimal or hexadecimal notation.

$$
\begin{array}{llllll}
\mathtt{frac} & ::= & d{:}\mathtt{digit} & \Rightarrow & d/10 \\
& | & d{:}\mathtt{digit}~\text{`}\mathtt{\_}\text{'}^?~p{:}\mathtt{frac} & \Rightarrow & (d + p/10)/10 \\
\mathtt{hexfrac} & ::= & h{:}\mathtt{hexdigit} & \Rightarrow & h/16 \\
& | & h{:}\mathtt{hexdigit}~\text{`}\mathtt{\_}\text{'}^?~p{:}\mathtt{hexfrac} & \Rightarrow & (h + p/16)/16 \\
\mathtt{float} & ::= & p{:}\mathtt{num}~\text{`}\mathtt{.}\text{'}^? & \Rightarrow & p \\
& | & p{:}\mathtt{num}~\text{`}\mathtt{.}\text{'}~q{:}\mathtt{frac} & \Rightarrow & p + q \\
& | & p{:}\mathtt{num}~\text{`}\mathtt{.}\text{'}^?~(\text{`}\mathtt{E}\text{'} ~|~ \text{`}\mathtt{e}\text{'})~\pm{:}\mathtt{sign}~e{:}\mathtt{num} & \Rightarrow & p \cdot 10^{\pm e} \\
& | & p{:}\mathtt{num}~\text{`}\mathtt{.}\text{'}~q{:}\mathtt{frac}~(\text{`}\mathtt{E}\text{'} ~|~ \text{`}\mathtt{e}\text{'})~\pm{:}\mathtt{sign}~e{:}\mathtt{num} & \Rightarrow & (p + q) \cdot 10^{\pm e} \\
\mathtt{hexfloat} & ::= & \text{`}\mathtt{0x}\text{'}~p{:}\mathtt{hexnum}~\text{`}\mathtt{.}\text{'}^? & \Rightarrow & p \\
& | & \text{`}\mathtt{0x}\text{'}~p{:}\mathtt{hexnum}~\text{`}\mathtt{.}\text{'}~q{:}\mathtt{hexfrac} & \Rightarrow & p + q \\
& | & \text{`}\mathtt{0x}\text{'}~p{:}\mathtt{hexnum}~\text{`}\mathtt{.}\text{'}^?~(\text{`}\mathtt{P}\text{'} ~|~ \text{`}\mathtt{p}\text{'})~\pm{:}\mathtt{sign}~e{:}\mathtt{num} & \Rightarrow & p \cdot 2^{\pm e} \\
& | & \text{`}\mathtt{0x}\text{'}~p{:}\mathtt{hexnum}~\text{`}\mathtt{.}\text{'}~q{:}\mathtt{hexfrac}~(\text{`}\mathtt{P}\text{'} ~|~ \text{`}\mathtt{p}\text{'})~\pm{:}\mathtt{sign}~e{:}\mathtt{num} & \Rightarrow & (p + q) \cdot 2^{\pm e}
\end{array}
$$

The value of a literal must not lie outside the representable range of the corresponding IEEE 754[48] type (that is, a numeric value must not overflow to ±infinity), but it may be rounded to the nearest representable value.

---

[48] https://ieeexplore.ieee.org/document/8766229

Note: Rounding can be prevented by using hexadecimal notation with no more significant bits than supported by the required type.

Floating-point values may also be written as constants for *infinity* or *canonical NaN* (*not a number*). Furthermore, arbitrary NaN values may be expressed by providing an explicit payload value.

$$\begin{array}{llllll}
\mathtt{f}N & ::= & \pm{:}\mathtt{sign}~z{:}\mathtt{f}N\mathtt{mag} & \Rightarrow & \pm z & \\
\mathtt{f}N\mathtt{mag} & ::= & z{:}\mathtt{float} & \Rightarrow & \mathrm{float}_N(z) & (\mathrm{if}~\mathrm{float}_N(z) \neq \pm\infty) \\
 & | & z{:}\mathtt{hexfloat} & \Rightarrow & \mathrm{float}_N(z) & (\mathrm{if}~\mathrm{float}_N(z) \neq \pm\infty) \\
 & | & \text{‘}\mathtt{inf}\text{’} & \Rightarrow & \infty & \\
 & | & \text{‘}\mathtt{nan}\text{’} & \Rightarrow & \mathsf{nan}(\mathrm{canon}_N) & \\
 & | & \text{‘}\mathtt{nan{:}0x}\text{’}~n{:}\mathtt{hexnum} & \Rightarrow & \mathsf{nan}(n) & (\mathrm{if}~1 \leq n < 2^{\mathrm{signif}(N)})
\end{array}$$

6.3.3 Strings

Strings denote sequences of bytes that can represent both textual and binary data. They are enclosed in quotation marks and may contain any character other than ASCII[49] control characters, quotation marks (‘"’), or backslash (‘\’), except when expressed with an *escape sequence*.

$$\begin{array}{llllll}
\mathtt{string} & ::= & \text{‘"’}~(b^*{:}\mathtt{stringelem})^*~\text{‘"’} & \Rightarrow & \mathrm{concat}((b^*)^*) & (\mathrm{if}~|\mathrm{concat}((b^*)^*)| < 2^{32}) \\
\mathtt{stringelem} & ::= & c{:}\mathtt{stringchar} & \Rightarrow & \mathrm{utf8}(c) & \\
 & | & \text{‘}\backslash\text{’}~n{:}\mathtt{hexdigit}~m{:}\mathtt{hexdigit} & \Rightarrow & 16 \cdot n + m &
\end{array}$$

Each character in a string literal represents the byte sequence corresponding to its UTF-8 Unicode[50] (Section 2.5) encoding, except for hexadecimal escape sequences ‘$\backslash hh$’, which represent raw bytes of the respective value.

$$\begin{array}{llllll}
\mathtt{stringchar} & ::= & c{:}\mathtt{char} & \Rightarrow & c & (\mathrm{if}~c \geq \mathrm{U{+}20} \wedge c \neq \mathrm{U{+}7F} \wedge c \neq \text{‘"’} \wedge c \neq \text{‘}\backslash\text{’}) \\
 & | & \text{‘}\backslash\mathtt{t}\text{’} & \Rightarrow & \mathrm{U{+}09} & \\
 & | & \text{‘}\backslash\mathtt{n}\text{’} & \Rightarrow & \mathrm{U{+}0A} & \\
 & | & \text{‘}\backslash\mathtt{r}\text{’} & \Rightarrow & \mathrm{U{+}0D} & \\
 & | & \text{‘}\backslash\text{"’} & \Rightarrow & \mathrm{U{+}22} & \\
 & | & \text{‘}\backslash\text{'’} & \Rightarrow & \mathrm{U{+}27} & \\
 & | & \text{‘}\backslash\backslash\text{’} & \Rightarrow & \mathrm{U{+}5C} & \\
 & | & \text{‘}\backslash\mathtt{u}\{\text{’}~n{:}\mathtt{hexnum}~\text{‘}\}\text{’} & \Rightarrow & \mathrm{U{+}(n)} & (\mathrm{if}~n < \mathtt{0xD800} \vee \mathtt{0xE000} \leq n < \mathtt{0x110000})
\end{array}$$

6.3.4 Names

Names are strings denoting a literal character sequence. A name string must form a valid UTF-8 encoding as defined by Unicode[51] (Section 2.5) and is interpreted as a string of Unicode scalar values.

$$\mathtt{name} \quad ::= \quad b^*{:}\mathtt{string} \quad \Rightarrow \quad c^* \qquad (\mathrm{if}~b^* = \mathrm{utf8}(c^*))$$

Note: Presuming the source text is itself encoded correctly, strings that do not contain any uses of hexadecimal byte escapes are always valid names.

---

[49] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

[50] https://www.unicode.org/versions/latest/

[51] https://www.unicode.org/versions/latest/

6.3.5 Identifiers

Indices can be given in both numeric and symbolic form. Symbolic *identifiers* that stand in lieu of indices start with '$', followed by any sequence of printable ASCII[52] characters that does not contain a space, quotation mark, comma, semicolon, or bracket.

$$\begin{array}{llll}
\mathtt{id} & ::= & \text{`\$'}\ \mathtt{idchar}^+ \\
\mathtt{idchar} & ::= & \text{`}\mathtt{0}\text{'} ~|~ \dots ~|~ \text{`}\mathtt{9}\text{'} \\
& | & \text{`}\mathtt{A}\text{'} ~|~ \dots ~|~ \text{`}\mathtt{Z}\text{'} \\
& | & \text{`}\mathtt{a}\text{'} ~|~ \dots ~|~ \text{`}\mathtt{z}\text{'} \\
& | & \text{`!'} ~|~ \text{`\#'} ~|~ \text{`\$'} ~|~ \text{`\%'} ~|~ \text{`\&'} ~|~ \text{`}'\text{'} ~|~ \text{`}{*}\text{'} ~|~ \text{`}{+}\text{'} ~|~ \text{`}{-}\text{'} ~|~ \text{`.'} ~|~ \text{`/'} \\
& | & \text{`:'} ~|~ \text{`}{<}\text{'} ~|~ \text{`}{=}\text{'} ~|~ \text{`}{>}\text{'} ~|~ \text{`?'} ~|~ \text{`@'} ~|~ \text{`}\backslash\text{'} ~|~ \text{`\^{}'} ~|~ \text{`\_'} ~|~ \text{`\`{}'} ~|~ \text{`}|\text{'} ~|~ \text{`\~{}'}
\end{array}$$

Conventions

The expansion rules of some abbreviations require insertion of a *fresh* identifier. That may be any syntactically valid identifier that does not already occur in the given source text.

6.4 Types

6.4.1 Number Types

$$\begin{array}{llcll}
\mathtt{numtype} & ::= & \text{`}\mathtt{i32}\text{'} & \Rightarrow & \mathsf{i32} \\
& | & \text{`}\mathtt{i64}\text{'} & \Rightarrow & \mathsf{i64} \\
& | & \text{`}\mathtt{f32}\text{'} & \Rightarrow & \mathsf{f32} \\
& | & \text{`}\mathtt{f64}\text{'} & \Rightarrow & \mathsf{f64}
\end{array}$$

6.4.2 Vector Types

$$\begin{array}{llcll}
\mathtt{vectype} & ::= & \text{`}\mathtt{v128}\text{'} & \Rightarrow & \mathsf{v128}
\end{array}$$

6.4.3 Reference Types

$$\begin{array}{llcll}
\mathtt{reftype} & ::= & \text{`}\mathtt{funcref}\text{'} & \Rightarrow & \mathsf{funcref} \\
& | & \text{`}\mathtt{externref}\text{'} & \Rightarrow & \mathsf{externref} \\
\mathtt{heaptype} & ::= & \text{`}\mathtt{func}\text{'} & \Rightarrow & \mathsf{funcref} \\
& | & \text{`}\mathtt{extern}\text{'} & \Rightarrow & \mathsf{externref}
\end{array}$$

6.4.4 Value Types

$$\begin{array}{llcll}
\mathtt{valtype} & ::= & t{:}\mathtt{numtype} & \Rightarrow & t \\
& | & t{:}\mathtt{vectype} & \Rightarrow & t \\
& | & t{:}\mathtt{reftype} & \Rightarrow & t
\end{array}$$

---

[52] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

6.4.5 Function Types

$$\begin{array}{llcll}
\mathtt{functype} & ::= & \text{‘(’ ‘}\mathtt{func}\text{’ } t_1^*{:}\mathtt{vec}(\mathtt{param})\ t_2^*{:}\mathtt{vec}(\mathtt{result}) \text{ ‘)’} & \Rightarrow & [t_1^*] \rightarrow [t_2^*] \\
\mathtt{param} & ::= & \text{‘(’ ‘}\mathtt{param}\text{’ } \mathtt{id}^?\ t{:}\mathtt{valtype} \text{ ‘)’} & \Rightarrow & t \\
\mathtt{result} & ::= & \text{‘(’ ‘}\mathtt{result}\text{’ } t{:}\mathtt{valtype} \text{ ‘)’} & \Rightarrow & t
\end{array}$$

Note: The optional identifier names for parameters in a function type only have documentation purpose. They cannot be referenced from anywhere.

Abbreviations

Multiple anonymous parameters or results may be combined into a single declaration:

$$\begin{array}{lcl}
\text{‘(’ ‘}\mathtt{param}\text{’ } \mathtt{valtype}^* \text{ ‘)’} & \equiv & (\text{‘(’ ‘}\mathtt{param}\text{’ } \mathtt{valtype} \text{ ‘)’})^* \\
\text{‘(’ ‘}\mathtt{result}\text{’ } \mathtt{valtype}^* \text{ ‘)’} & \equiv & (\text{‘(’ ‘}\mathtt{result}\text{’ } \mathtt{valtype} \text{ ‘)’})^*
\end{array}$$

6.4.6 Limits

$$\begin{array}{llcll}
\mathtt{limits} & ::= & n{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}\ n, \mathsf{max}\ \epsilon\} \\
& | & n{:}\mathtt{u32}\ m{:}\mathtt{u32} & \Rightarrow & \{\mathsf{min}\ n, \mathsf{max}\ m\}
\end{array}$$

6.4.7 Memory Types

$$\begin{array}{llcll}
\mathtt{memtype} & ::= & \mathit{lim}{:}\mathtt{limits} & \Rightarrow & \mathit{lim}
\end{array}$$

6.4.8 Table Types

$$\begin{array}{llcll}
\mathtt{tabletype} & ::= & \mathit{lim}{:}\mathtt{limits}\ \mathit{et}{:}\mathtt{reftype} & \Rightarrow & \mathit{lim}\ \mathit{et}
\end{array}$$

6.4.9 Global Types

$$\begin{array}{llcll}
\mathtt{globaltype} & ::= & t{:}\mathtt{valtype} & \Rightarrow & \mathsf{const}\ t \\
& | & \text{‘(’ ‘}\mathtt{mut}\text{’ } t{:}\mathtt{valtype} \text{ ‘)’} & \Rightarrow & \mathsf{var}\ t
\end{array}$$

6.5 Instructions

Instructions are syntactically distinguished into *plain* and *structured* instructions.

$$\begin{array}{llcll}
\mathtt{instr}_I & ::= & \mathit{in}{:}\mathtt{plaininstr}_I & \Rightarrow & \mathit{in} \\
& | & \mathit{in}{:}\mathtt{blockinstr}_I & \Rightarrow & \mathit{in}
\end{array}$$

In addition, as a syntactic abbreviation, instructions can be written as S-expressions in folded form, to group them visually.

6.5.1 Labels

Structured control instructions can be annotated with a symbolic label identifier. They are the only symbolic identifiers that can be bound locally in an instruction sequence. The following grammar handles the corresponding update to the identifier context by composing the context with an additional label entry.

$$
\begin{array}{llcllll}
\mathtt{label}_I & ::= & v{:}\mathtt{id} & \Rightarrow & \{\mathsf{labels}~v\} \oplus I & & (\mathrm{if}~v \notin I.\mathsf{labels}) \\
& | & v{:}\mathtt{id} & \Rightarrow & \{\mathsf{labels}~v\} \oplus (I~\mathrm{with}~\mathsf{labels}[i] = \epsilon) & & (\mathrm{if}~I.\mathsf{labels}[i] = v) \\
& | & \epsilon & \Rightarrow & \{\mathsf{labels}~(\epsilon)\} \oplus I & &
\end{array}
$$

Note: The new label entry is inserted at the *beginning* of the label list in the identifier context. This effectively shifts all existing labels up by one, mirroring the fact that control instructions are indexed relatively not absolutely.

If a label with the same name already exists, then it is shadowed and the earlier label becomes inaccessible.

6.5.2 Control Instructions

Structured control instructions can bind an optional symbolic label identifier. The same label identifier may optionally be repeated after the corresponding `end` and `else` pseudo instructions, to indicate the matching delimiters.

Their block type is given as a type use, analogous to the type of functions. However, the special case of a type use that is syntactically empty or consists of only a single result is not regarded as an abbreviation for an inline function type, but is parsed directly into an optional value type.

$$
\begin{array}{llll}
\mathtt{blocktype}_I & ::= & (t{:}\mathtt{result})^? & \Rightarrow \quad t^? \\
& | & x, I'{:}\mathtt{typeuse}_I & \Rightarrow \quad x \qquad (\mathrm{if}~I' = \{\mathsf{locals}~(\epsilon)^\ast\}) \\
\mathtt{blockinstr}_I & ::= & \text{`}\mathtt{block}\text{'}~~I'{:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in{:}\mathtt{instr}_{I'})^\ast~~\text{`}\mathtt{end}\text{'}~~\mathtt{id}^? & \\
& & \qquad \Rightarrow \quad \mathsf{block}~bt~in^\ast~\mathsf{end} \qquad (\mathrm{if}~\mathtt{id}^? = \epsilon \vee \mathtt{id}^? = \mathtt{label}) & \\
& | & \text{`}\mathtt{loop}\text{'}~~I'{:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in{:}\mathtt{instr}_{I'})^\ast~~\text{`}\mathtt{end}\text{'}~~\mathtt{id}^? & \\
& & \qquad \Rightarrow \quad \mathsf{loop}~bt~in^\ast~\mathsf{end} \qquad (\mathrm{if}~\mathtt{id}^? = \epsilon \vee \mathtt{id}^? = \mathtt{label}) & \\
& | & \text{`}\mathtt{if}\text{'}~~I'{:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in_1{:}\mathtt{instr}_{I'})^\ast~~\text{`}\mathtt{else}\text{'}~~\mathtt{id}_1^?~~(in_2{:}\mathtt{instr}_{I'})^\ast~~\text{`}\mathtt{end}\text{'}~~\mathtt{id}_2^? & \\
& & \qquad \Rightarrow \quad \mathsf{if}~bt~in_1^\ast~\mathsf{else}~in_2^\ast~\mathsf{end} \qquad (\mathrm{if}~\mathtt{id}_1^? = \epsilon \vee \mathtt{id}_1^? = \mathtt{label}, \mathtt{id}_2^? = \epsilon \vee \mathtt{id}_2^? = \mathtt{label}) &
\end{array}
$$

Note: The side condition stating that the identifier context I' must only contain unnamed entries in the rule for `typeuse` block types enforces that no identifier can be bound in any `param` declaration for a block type.

All other control instruction are represented verbatim.

$$
\begin{array}{llll}
\mathtt{plaininstr}_I & ::= & \text{`}\mathtt{unreachable}\text{'} & \Rightarrow \quad \mathsf{unreachable} \\
& | & \text{`}\mathtt{nop}\text{'} & \Rightarrow \quad \mathsf{nop} \\
& | & \text{`}\mathtt{br}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br}~l \\
& | & \text{`}\mathtt{br\_if}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br\_if}~l \\
& | & \text{`}\mathtt{br\_table}\text{'}~~l^\ast{:}\mathtt{vec}(\mathtt{labelidx}_I)~~l_N{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br\_table}~l^\ast~l_N \\
& | & \text{`}\mathtt{return}\text{'} & \Rightarrow \quad \mathsf{return} \\
& | & \text{`}\mathtt{call}\text{'}~~x{:}\mathtt{funcidx}_I & \Rightarrow \quad \mathsf{call}~x \\
& | & \text{`}\mathtt{call\_indirect}\text{'}~~x{:}\mathtt{tableidx}~~y, I'{:}\mathtt{typeuse}_I & \Rightarrow \quad \mathsf{call\_indirect}~x~y \quad (\mathrm{if}~I' = \{\mathsf{locals}~(\epsilon)^\ast\})
\end{array}
$$

Note: The side condition stating that the identifier context I' must only contain unnamed entries in the rule for call_indirect enforces that no identifier can be bound in any `param` declaration appearing in the type annotation.

Abbreviations

The ‘`else`’ keyword of an ‘`if`’ instruction can be omitted if the following instruction sequence is empty.

$$\text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘}\mathtt{end}\text{’} \quad\equiv\quad \text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘}\mathtt{else}\text{’}~\text{‘}\mathtt{end}\text{’}$$

Also, for backwards compatibility, the table index to ‘`call_indirect`’ can be omitted, defaulting to 0.

$$\text{‘}\mathtt{call\_indirect}\text{’}~\mathtt{typeuse} \quad\equiv\quad \text{‘}\mathtt{call\_indirect}\text{’}~0~\mathtt{typeuse}$$

6.5.3 Reference Instructions

$$\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{‘}\mathtt{ref.null}\text{’}~t{:}\mathtt{heaptype} & \Rightarrow & \mathsf{ref.null}~t \\
& | & \text{‘}\mathtt{ref.is\_null}\text{’} & \Rightarrow & \mathsf{ref.is\_null} \\
& | & \text{‘}\mathtt{ref.func}\text{’}~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{ref.func}~x
\end{array}$$

6.5.4 Parametric Instructions

$$\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{‘}\mathtt{drop}\text{’} & \Rightarrow & \mathsf{drop} \\
& | & \text{‘}\mathtt{select}\text{’}~((t{:}\mathtt{result})^*)^? & \Rightarrow & \mathsf{select}~(t^*)^?
\end{array}$$

6.5.5 Variable Instructions

$$\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{‘}\mathtt{local.get}\text{’}~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{local.get}~x \\
& | & \text{‘}\mathtt{local.set}\text{’}~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{local.set}~x \\
& | & \text{‘}\mathtt{local.tee}\text{’}~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{local.tee}~x \\
& | & \text{‘}\mathtt{global.get}\text{’}~x{:}\mathtt{globalidx}_I & \Rightarrow & \mathsf{global.get}~x \\
& | & \text{‘}\mathtt{global.set}\text{’}~x{:}\mathtt{globalidx}_I & \Rightarrow & \mathsf{global.set}~x
\end{array}$$

6.5.6 Table Instructions

$$\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{‘}\mathtt{table.get}\text{’}~x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.get}~x \\
& | & \text{‘}\mathtt{table.set}\text{’}~x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.set}~x \\
& | & \text{‘}\mathtt{table.size}\text{’}~x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.size}~x \\
& | & \text{‘}\mathtt{table.grow}\text{’}~x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.grow}~x \\
& | & \text{‘}\mathtt{table.fill}\text{’}~x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.fill}~x \\
& | & \text{‘}\mathtt{table.copy}\text{’}~x{:}\mathtt{tableidx}_I~~y{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.copy}~x~y \\
& | & \text{‘}\mathtt{table.init}\text{’}~x{:}\mathtt{tableidx}_I~~y{:}\mathtt{elemidx}_I & \Rightarrow & \mathsf{table.init}~x~y \\
& | & \text{‘}\mathtt{elem.drop}\text{’}~x{:}\mathtt{elemidx}_I & \Rightarrow & \mathsf{elem.drop}~x
\end{array}$$

Abbreviations

For backwards compatibility, all table indices may be omitted from table instructions, defaulting to 0.

$$\begin{array}{lcl}
\text{‘}\mathtt{table.get}\text{’} & \equiv & \text{‘}\mathtt{table.get}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.set}\text{’} & \equiv & \text{‘}\mathtt{table.set}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.size}\text{’} & \equiv & \text{‘}\mathtt{table.size}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.grow}\text{’} & \equiv & \text{‘}\mathtt{table.grow}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.fill}\text{’} & \equiv & \text{‘}\mathtt{table.fill}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.copy}\text{’} & \equiv & \text{‘}\mathtt{table.copy}\text{’}~\text{‘}\mathtt{0}\text{’}~\text{‘}\mathtt{0}\text{’} \\
\text{‘}\mathtt{table.init}\text{’}~x{:}\mathtt{elemidx}_I & \equiv & \text{‘}\mathtt{table.init}\text{’}~\text{‘}\mathtt{0}\text{’}~x{:}\mathtt{elemidx}_I
\end{array}$$

6.5.7 Memory Instructions

The offset and alignment immediates to memory instructions are optional. The offset defaults to $\mathtt{0}$, the alignment to the storage size of the respective memory access, which is its *natural alignment*. Lexically, an `offset` or `align` phrase is considered a single keyword token, so no white space is allowed around the ‘$=$’.

$$\begin{array}{llclll}
\mathtt{memarg}_N & ::= & o{:}\mathtt{offset}~~a{:}\mathtt{align}_N & \Rightarrow & \{\mathsf{align}~n,~\mathsf{offset}~o\} & (\text{if } a = 2^n) \\
\mathtt{offset} & ::= & \text{‘}\mathtt{offset{=}}\text{’}o{:}\mathtt{u32} & \Rightarrow & o \\
 & | & \epsilon & \Rightarrow & 0 \\
\mathtt{align}_N & ::= & \text{‘}\mathtt{align{=}}\text{’}a{:}\mathtt{u32} & \Rightarrow & a \\
 & | & \epsilon & \Rightarrow & N \\
\mathtt{plaininstr}_I & ::= & \dots \\
 & | & \text{‘}\mathtt{i32.load}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i32.load}~m \\
 & | & \text{‘}\mathtt{i64.load}\text{’}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{i64.load}~m \\
 & | & \text{‘}\mathtt{f32.load}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{f32.load}~m \\
 & | & \text{‘}\mathtt{f64.load}\text{’}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{f64.load}~m \\
 & | & \text{‘}\mathtt{i32.load8\_s}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.load8\_s}~m \\
 & | & \text{‘}\mathtt{i32.load8\_u}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.load8\_u}~m \\
 & | & \text{‘}\mathtt{i32.load16\_s}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.load16\_s}~m \\
 & | & \text{‘}\mathtt{i32.load16\_u}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.load16\_u}~m \\
 & | & \text{‘}\mathtt{i64.load8\_s}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.load8\_s}~m \\
 & | & \text{‘}\mathtt{i64.load8\_u}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.load8\_u}~m \\
 & | & \text{‘}\mathtt{i64.load16\_s}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.load16\_s}~m \\
 & | & \text{‘}\mathtt{i64.load16\_u}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.load16\_u}~m \\
 & | & \text{‘}\mathtt{i64.load32\_s}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.load32\_s}~m \\
 & | & \text{‘}\mathtt{i64.load32\_u}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.load32\_u}~m \\
 & | & \text{‘}\mathtt{i32.store}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i32.store}~m \\
 & | & \text{‘}\mathtt{i64.store}\text{’}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{i64.store}~m \\
 & | & \text{‘}\mathtt{f32.store}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{f32.store}~m \\
 & | & \text{‘}\mathtt{f64.store}\text{’}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{f64.store}~m \\
 & | & \text{‘}\mathtt{i32.store8}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.store8}~m \\
 & | & \text{‘}\mathtt{i32.store16}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.store16}~m \\
 & | & \text{‘}\mathtt{i64.store8}\text{’}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.store8}~m \\
 & | & \text{‘}\mathtt{i64.store16}\text{’}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.store16}~m \\
 & | & \text{‘}\mathtt{i64.store32}\text{’}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.store32}~m \\
 & | & \text{‘}\mathtt{memory.size}\text{’} & \Rightarrow & \mathsf{memory.size} \\
 & | & \text{‘}\mathtt{memory.grow}\text{’} & \Rightarrow & \mathsf{memory.grow} \\
 & | & \text{‘}\mathtt{memory.fill}\text{’} & \Rightarrow & \mathsf{memory.fill} \\
 & | & \text{‘}\mathtt{memory.copy}\text{’} & \Rightarrow & \mathsf{memory.copy} \\
 & | & \text{‘}\mathtt{memory.init}\text{’}~~x{:}\mathtt{dataidx}_I & \Rightarrow & \mathsf{memory.init}~x \\
 & | & \text{‘}\mathtt{data.drop}\text{’}~~x{:}\mathtt{dataidx}_I & \Rightarrow & \mathsf{data.drop}~x
\end{array}$$

6.5.8 Atomic Memory Instructions

The offset immediate to atomic memory instructions is optional, and defaults to 0.

$$
\begin{array}{llll}
\mathtt{plaininstr}_I & ::= & \dots & \\
& | & \text{`}\mathtt{memory.atomic.notify}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{memory.atomic.notify}~m \\
& | & \text{`}\mathtt{memory.atomic.wait32}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{memory.atomic.wait32}~m \\
& | & \text{`}\mathtt{memory.atomic.wait64}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{memory.atomic.wait64}~m \\
& | & \text{`}\mathtt{atomic.fence}\text{'} & \Rightarrow~~ \mathsf{atomic.fence} \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.load}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.load}~m \\
| & \text{`}\mathtt{i64.atomic.load}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.load}~m \\
| & \text{`}\mathtt{i32.atomic.load8\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.load8\_u}~m \\
| & \text{`}\mathtt{i32.atomic.load16\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.load16\_u}~m \\
| & \text{`}\mathtt{i64.atomic.load8\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.load8\_u}~m \\
| & \text{`}\mathtt{i64.atomic.load16\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.load16\_u}~m \\
| & \text{`}\mathtt{i64.atomic.load32\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.load32\_u}~m \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.store}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.store}~m \\
| & \text{`}\mathtt{i64.atomic.store}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.store}~m \\
| & \text{`}\mathtt{i32.atomic.store8}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.store8}~m \\
| & \text{`}\mathtt{i32.atomic.store16}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.store16}~m \\
| & \text{`}\mathtt{i64.atomic.store8}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.store8}~m \\
| & \text{`}\mathtt{i64.atomic.store16}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.store16}~m \\
| & \text{`}\mathtt{i64.atomic.store32}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.store32}~m \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.rmw.add}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.rmw.add}~m \\
| & \text{`}\mathtt{i64.atomic.rmw.add}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.rmw.add}~m \\
| & \text{`}\mathtt{i32.atomic.rmw8.add\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.rmw8.add\_u}~m \\
| & \text{`}\mathtt{i32.atomic.rmw16.add\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.rmw16.add\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw8.add\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.rmw8.add\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw16.add\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.rmw16.add\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw32.add\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.rmw32.add\_u}~m \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.rmw.sub}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.rmw.sub}~m \\
| & \text{`}\mathtt{i64.atomic.rmw.sub}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.rmw.sub}~m \\
| & \text{`}\mathtt{i32.atomic.rmw8.sub\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.rmw8.sub\_u}~m \\
| & \text{`}\mathtt{i32.atomic.rmw16.sub\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.rmw16.sub\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw8.sub\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.rmw8.sub\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw16.sub\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.rmw16.sub\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw32.sub\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.rmw32.sub\_u}~m \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.rmw.and}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.rmw.and}~m \\
| & \text{`}\mathtt{i64.atomic.rmw.and}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.rmw.and}~m \\
| & \text{`}\mathtt{i32.atomic.rmw8.and\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.rmw8.and\_u}~m \\
| & \text{`}\mathtt{i32.atomic.rmw16.and\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.rmw16.and\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw8.and\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.rmw8.and\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw16.and\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.rmw16.and\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw32.and\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.rmw32.and\_u}~m \\
\end{array}
$$

$$
\begin{array}{lll}
| & \text{`}\mathtt{i32.atomic.rmw.or}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i32.atomic.rmw.or}~m \\
| & \text{`}\mathtt{i64.atomic.rmw.or}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow~~ \mathsf{i64.atomic.rmw.or}~m \\
| & \text{`}\mathtt{i32.atomic.rmw8.or\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i32.atomic.rmw8.or\_u}~m \\
| & \text{`}\mathtt{i32.atomic.rmw16.or\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i32.atomic.rmw16.or\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw8.or\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow~~ \mathsf{i64.atomic.rmw8.or\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw16.or\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow~~ \mathsf{i64.atomic.rmw16.or\_u}~m \\
| & \text{`}\mathtt{i64.atomic.rmw32.or\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow~~ \mathsf{i64.atomic.rmw32.or\_u}~m \\
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\texttt{i32.atomic.rmw.xor}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i32.atomic.rmw.xor}~m \\
| & \text{`}\texttt{i64.atomic.rmw.xor}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{i64.atomic.rmw.xor}~m \\
| & \text{`}\texttt{i32.atomic.rmw8.xor\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.atomic.rmw8.xor\_u}~m \\
| & \text{`}\texttt{i32.atomic.rmw16.xor\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.atomic.rmw16.xor\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw8.xor\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.atomic.rmw8.xor\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw16.xor\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.atomic.rmw16.xor\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw32.xor\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.atomic.rmw32.xor\_u}~m
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\texttt{i32.atomic.rmw.xchg}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i32.atomic.rmw.xchg}~m \\
| & \text{`}\texttt{i64.atomic.rmw.xchg}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{i64.atomic.rmw.xchg}~m \\
| & \text{`}\texttt{i32.atomic.rmw8.xchg\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.atomic.rmw8.xchg\_u}~m \\
| & \text{`}\texttt{i32.atomic.rmw16.xchg\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.atomic.rmw16.xchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw8.xchg\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.atomic.rmw8.xchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw16.xchg\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.atomic.rmw16.xchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw32.xchg\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.atomic.rmw32.xchg\_u}~m
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\texttt{i32.atomic.rmw.cmpxchg}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i32.atomic.rmw.cmpxchg}~m \\
| & \text{`}\texttt{i64.atomic.rmw.cmpxchg}\text{'}~~m{:}\mathtt{memarg}_8 & \Rightarrow & \mathsf{i64.atomic.rmw.cmpxchg}~m \\
| & \text{`}\texttt{i32.atomic.rmw8.cmpxchg\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i32.atomic.rmw8.cmpxchg\_u}~m \\
| & \text{`}\texttt{i32.atomic.rmw16.cmpxchg\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i32.atomic.rmw16.cmpxchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw8.cmpxchg\_u}\text{'}~~m{:}\mathtt{memarg}_1 & \Rightarrow & \mathsf{i64.atomic.rmw8.cmpxchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw16.cmpxchg\_u}\text{'}~~m{:}\mathtt{memarg}_2 & \Rightarrow & \mathsf{i64.atomic.rmw16.cmpxchg\_u}~m \\
| & \text{`}\texttt{i64.atomic.rmw32.cmpxchg\_u}\text{'}~~m{:}\mathtt{memarg}_4 & \Rightarrow & \mathsf{i64.atomic.rmw32.cmpxchg\_u}~m
\end{array}
$$

6.5.9 Numeric Instructions

$$
\begin{array}{llclll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{`}\texttt{i32.const}\text{'}~~n{:}\mathtt{i32} & \Rightarrow & \mathsf{i32.const}~n \\
& | & \text{`}\texttt{i64.const}\text{'}~~n{:}\mathtt{i64} & \Rightarrow & \mathsf{i64.const}~n \\
& | & \text{`}\texttt{f32.const}\text{'}~~z{:}\mathtt{f32} & \Rightarrow & \mathsf{f32.const}~z \\
& | & \text{`}\texttt{f64.const}\text{'}~~z{:}\mathtt{f64} & \Rightarrow & \mathsf{f64.const}~z
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\texttt{i32.clz}\text{'} & \Rightarrow & \mathsf{i32.clz} \\
| & \text{`}\texttt{i32.ctz}\text{'} & \Rightarrow & \mathsf{i32.ctz} \\
| & \text{`}\texttt{i32.popcnt}\text{'} & \Rightarrow & \mathsf{i32.popcnt} \\
| & \text{`}\texttt{i32.add}\text{'} & \Rightarrow & \mathsf{i32.add} \\
| & \text{`}\texttt{i32.sub}\text{'} & \Rightarrow & \mathsf{i32.sub} \\
| & \text{`}\texttt{i32.mul}\text{'} & \Rightarrow & \mathsf{i32.mul} \\
| & \text{`}\texttt{i32.div\_s}\text{'} & \Rightarrow & \mathsf{i32.div\_s} \\
| & \text{`}\texttt{i32.div\_u}\text{'} & \Rightarrow & \mathsf{i32.div\_u} \\
| & \text{`}\texttt{i32.rem\_s}\text{'} & \Rightarrow & \mathsf{i32.rem\_s} \\
| & \text{`}\texttt{i32.rem\_u}\text{'} & \Rightarrow & \mathsf{i32.rem\_u} \\
| & \text{`}\texttt{i32.and}\text{'} & \Rightarrow & \mathsf{i32.and} \\
| & \text{`}\texttt{i32.or}\text{'} & \Rightarrow & \mathsf{i32.or} \\
| & \text{`}\texttt{i32.xor}\text{'} & \Rightarrow & \mathsf{i32.xor} \\
| & \text{`}\texttt{i32.shl}\text{'} & \Rightarrow & \mathsf{i32.shl} \\
| & \text{`}\texttt{i32.shr\_s}\text{'} & \Rightarrow & \mathsf{i32.shr\_s} \\
| & \text{`}\texttt{i32.shr\_u}\text{'} & \Rightarrow & \mathsf{i32.shr\_u} \\
| & \text{`}\texttt{i32.rotl}\text{'} & \Rightarrow & \mathsf{i32.rotl} \\
| & \text{`}\texttt{i32.rotr}\text{'} & \Rightarrow & \mathsf{i32.rotr}
\end{array}
$$

$$\begin{array}{llll}
| & \texttt{‘i64.clz’} & \Rightarrow & \mathsf{i64.clz} \\
| & \texttt{‘i64.ctz’} & \Rightarrow & \mathsf{i64.ctz} \\
| & \texttt{‘i64.popcnt’} & \Rightarrow & \mathsf{i64.popcnt} \\
| & \texttt{‘i64.add’} & \Rightarrow & \mathsf{i64.add} \\
| & \texttt{‘i64.sub’} & \Rightarrow & \mathsf{i64.sub} \\
| & \texttt{‘i64.mul’} & \Rightarrow & \mathsf{i64.mul} \\
| & \texttt{‘i64.div\_s’} & \Rightarrow & \mathsf{i64.div\_s} \\
| & \texttt{‘i64.div\_u’} & \Rightarrow & \mathsf{i64.div\_u} \\
| & \texttt{‘i64.rem\_s’} & \Rightarrow & \mathsf{i64.rem\_s} \\
| & \texttt{‘i64.rem\_u’} & \Rightarrow & \mathsf{i64.rem\_u} \\
| & \texttt{‘i64.and’} & \Rightarrow & \mathsf{i64.and} \\
| & \texttt{‘i64.or’} & \Rightarrow & \mathsf{i64.or} \\
| & \texttt{‘i64.xor’} & \Rightarrow & \mathsf{i64.xor} \\
| & \texttt{‘i64.shl’} & \Rightarrow & \mathsf{i64.shl} \\
| & \texttt{‘i64.shr\_s’} & \Rightarrow & \mathsf{i64.shr\_s} \\
| & \texttt{‘i64.shr\_u’} & \Rightarrow & \mathsf{i64.shr\_u} \\
| & \texttt{‘i64.rotl’} & \Rightarrow & \mathsf{i64.rotl} \\
| & \texttt{‘i64.rotr’} & \Rightarrow & \mathsf{i64.rotr}
\end{array}$$

$$\begin{array}{llll}
| & \texttt{‘f32.abs’} & \Rightarrow & \mathsf{f32.abs} \\
| & \texttt{‘f32.neg’} & \Rightarrow & \mathsf{f32.neg} \\
| & \texttt{‘f32.ceil’} & \Rightarrow & \mathsf{f32.ceil} \\
| & \texttt{‘f32.floor’} & \Rightarrow & \mathsf{f32.floor} \\
| & \texttt{‘f32.trunc’} & \Rightarrow & \mathsf{f32.trunc} \\
| & \texttt{‘f32.nearest’} & \Rightarrow & \mathsf{f32.nearest} \\
| & \texttt{‘f32.sqrt’} & \Rightarrow & \mathsf{f32.sqrt} \\
| & \texttt{‘f32.add’} & \Rightarrow & \mathsf{f32.add} \\
| & \texttt{‘f32.sub’} & \Rightarrow & \mathsf{f32.sub} \\
| & \texttt{‘f32.mul’} & \Rightarrow & \mathsf{f32.mul} \\
| & \texttt{‘f32.div’} & \Rightarrow & \mathsf{f32.div} \\
| & \texttt{‘f32.min’} & \Rightarrow & \mathsf{f32.min} \\
| & \texttt{‘f32.max’} & \Rightarrow & \mathsf{f32.max} \\
| & \texttt{‘f32.copysign’} & \Rightarrow & \mathsf{f32.copysign}
\end{array}$$

$$\begin{array}{llll}
| & \texttt{‘f64.abs’} & \Rightarrow & \mathsf{f64.abs} \\
| & \texttt{‘f64.neg’} & \Rightarrow & \mathsf{f64.neg} \\
| & \texttt{‘f64.ceil’} & \Rightarrow & \mathsf{f64.ceil} \\
| & \texttt{‘f64.floor’} & \Rightarrow & \mathsf{f64.floor} \\
| & \texttt{‘f64.trunc’} & \Rightarrow & \mathsf{f64.trunc} \\
| & \texttt{‘f64.nearest’} & \Rightarrow & \mathsf{f64.nearest} \\
| & \texttt{‘f64.sqrt’} & \Rightarrow & \mathsf{f64.sqrt} \\
| & \texttt{‘f64.add’} & \Rightarrow & \mathsf{f64.add} \\
| & \texttt{‘f64.sub’} & \Rightarrow & \mathsf{f64.sub} \\
| & \texttt{‘f64.mul’} & \Rightarrow & \mathsf{f64.mul} \\
| & \texttt{‘f64.div’} & \Rightarrow & \mathsf{f64.div} \\
| & \texttt{‘f64.min’} & \Rightarrow & \mathsf{f64.min} \\
| & \texttt{‘f64.max’} & \Rightarrow & \mathsf{f64.max} \\
| & \texttt{‘f64.copysign’} & \Rightarrow & \mathsf{f64.copysign}
\end{array}$$

| ‘i32.eqz’ ⇒ i32.eqz
| ‘i32.eq’ ⇒ i32.eq
| ‘i32.ne’ ⇒ i32.ne
| ‘i32.lt_s’ ⇒ i32.lt_s
| ‘i32.lt_u’ ⇒ i32.lt_u
| ‘i32.gt_s’ ⇒ i32.gt_s
| ‘i32.gt_u’ ⇒ i32.gt_u
| ‘i32.le_s’ ⇒ i32.le_s
| ‘i32.le_u’ ⇒ i32.le_u
| ‘i32.ge_s’ ⇒ i32.ge_s
| ‘i32.ge_u’ ⇒ i32.ge_u

| ‘i64.eqz’ ⇒ i64.eqz
| ‘i64.eq’ ⇒ i64.eq
| ‘i64.ne’ ⇒ i64.ne
| ‘i64.lt_s’ ⇒ i64.lt_s
| ‘i64.lt_u’ ⇒ i64.lt_u
| ‘i64.gt_s’ ⇒ i64.gt_s
| ‘i64.gt_u’ ⇒ i64.gt_u
| ‘i64.le_s’ ⇒ i64.le_s
| ‘i64.le_u’ ⇒ i64.le_u
| ‘i64.ge_s’ ⇒ i64.ge_s
| ‘i64.ge_u’ ⇒ i64.ge_u

| ‘f32.eq’ ⇒ f32.eq
| ‘f32.ne’ ⇒ f32.ne
| ‘f32.lt’ ⇒ f32.lt
| ‘f32.gt’ ⇒ f32.gt
| ‘f32.le’ ⇒ f32.le
| ‘f32.ge’ ⇒ f32.ge

| ‘f64.eq’ ⇒ f64.eq
| ‘f64.ne’ ⇒ f64.ne
| ‘f64.lt’ ⇒ f64.lt
| ‘f64.gt’ ⇒ f64.gt
| ‘f64.le’ ⇒ f64.le
| ‘f64.ge’ ⇒ f64.ge

| ‘i32.wrap_i64’ ⇒ i32.wrap_i64
| ‘i32.trunc_f32_s’ ⇒ i32.trunc_f32_s
| ‘i32.trunc_f32_u’ ⇒ i32.trunc_f32_u
| ‘i32.trunc_f64_s’ ⇒ i32.trunc_f64_s
| ‘i32.trunc_f64_u’ ⇒ i32.trunc_f64_u
| ‘i32.trunc_sat_f32_s’ ⇒ i32.trunc_sat_f32_s
| ‘i32.trunc_sat_f32_u’ ⇒ i32.trunc_sat_f32_u
| ‘i32.trunc_sat_f64_s’ ⇒ i32.trunc_sat_f64_s
| ‘i32.trunc_sat_f64_u’ ⇒ i32.trunc_sat_f64_u
| ‘i64.extend_i32_s’ ⇒ i64.extend_i32_s
| ‘i64.extend_i32_u’ ⇒ i64.extend_i32_u
| ‘i64.trunc_f32_s’ ⇒ i64.trunc_f32_s
| ‘i64.trunc_f32_u’ ⇒ i64.trunc_f32_u
| ‘i64.trunc_f64_s’ ⇒ i64.trunc_f64_s
| ‘i64.trunc_f64_u’ ⇒ i64.trunc_f64_u
| ‘i64.trunc_sat_f32_s’ ⇒ i64.trunc_sat_f32_s
| ‘i64.trunc_sat_f32_u’ ⇒ i64.trunc_sat_f32_u
| ‘i64.trunc_sat_f64_s’ ⇒ i64.trunc_sat_f64_s
| ‘i64.trunc_sat_f64_u’ ⇒ i64.trunc_sat_f64_u
| ‘f32.convert_i32_s’ ⇒ f32.convert_i32_s
| ‘f32.convert_i32_u’ ⇒ f32.convert_i32_u
| ‘f32.convert_i64_s’ ⇒ f32.convert_i64_s
| ‘f32.convert_i64_u’ ⇒ f32.convert_i64_u
| ‘f32.demote_f64’ ⇒ f32.demote_f64
| ‘f64.convert_i32_s’ ⇒ f64.convert_i32_s
| ‘f64.convert_i32_u’ ⇒ f64.convert_i32_u
| ‘f64.convert_i64_s’ ⇒ f64.convert_i64_s
| ‘f64.convert_i64_u’ ⇒ f64.convert_i64_u
| ‘f64.promote_f32’ ⇒ f64.promote_f32
| ‘i32.reinterpret_f32’ ⇒ i32.reinterpret_f32
| ‘i64.reinterpret_f64’ ⇒ i64.reinterpret_f64
| ‘f32.reinterpret_i32’ ⇒ f32.reinterpret_i32
| ‘f64.reinterpret_i64’ ⇒ f64.reinterpret_i64

| ‘i32.extend8_s’ ⇒ i32.extend8_s
| ‘i32.extend16_s’ ⇒ i32.extend16_s
| ‘i64.extend8_s’ ⇒ i64.extend8_s
| ‘i64.extend16_s’ ⇒ i64.extend16_s
| ‘i64.extend32_s’ ⇒ i64.extend32_s

6.5.10 Vector Instructions

Vector memory instructions have optional offset and alignment immediates, like the memory instructions.

$$
\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots & & \\
& | & \text{‘}\mathtt{v128.load}\text{’}~~m{:}\mathtt{memarg}_{16} & \Rightarrow & \mathsf{v128.load}~m \\
& | & \text{‘}\mathtt{v128.load8x8\_s}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load8x8\_s}~m \\
& | & \text{‘}\mathtt{v128.load8x8\_u}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load8x8\_u}~m \\
& | & \text{‘}\mathtt{v128.load16x4\_s}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load16x4\_s}~m \\
& | & \text{‘}\mathtt{v128.load16x4\_u}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load16x4\_u}~m \\
& | & \text{‘}\mathtt{v128.load32x2\_s}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load32x2\_s}~m \\
& | & \text{‘}\mathtt{v128.load32x2\_u}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load32x2\_u}~m \\
& | & \text{‘}\mathtt{v128.load8\_splat}\text{’}~~m{:}\mathtt{memarg}_{1} & \Rightarrow & \mathsf{v128.load8\_splat}~m \\
& | & \text{‘}\mathtt{v128.load16\_splat}\text{’}~~m{:}\mathtt{memarg}_{2} & \Rightarrow & \mathsf{v128.load16\_splat}~m \\
& | & \text{‘}\mathtt{v128.load32\_splat}\text{’}~~m{:}\mathtt{memarg}_{4} & \Rightarrow & \mathsf{v128.load32\_splat}~m \\
& | & \text{‘}\mathtt{v128.load64\_splat}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load64\_splat}~m \\
& | & \text{‘}\mathtt{v128.load32\_zero}\text{’}~~m{:}\mathtt{memarg}_{4} & \Rightarrow & \mathsf{v128.load32\_zero}~m \\
& | & \text{‘}\mathtt{v128.load64\_zero}\text{’}~~m{:}\mathtt{memarg}_{8} & \Rightarrow & \mathsf{v128.load64\_zero}~m \\
& | & \text{‘}\mathtt{v128.store}\text{’}~~m{:}\mathtt{memarg}_{16} & \Rightarrow & \mathsf{v128.store}~m \\
& | & \text{‘}\mathtt{v128.load8\_lane}\text{’}~~m{:}\mathtt{memarg}_{1}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.load8\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.load16\_lane}\text{’}~~m{:}\mathtt{memarg}_{2}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.load16\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.load32\_lane}\text{’}~~m{:}\mathtt{memarg}_{4}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.load32\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.load64\_lane}\text{’}~~m{:}\mathtt{memarg}_{8}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.load64\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.store8\_lane}\text{’}~~m{:}\mathtt{memarg}_{1}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.store8\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.store16\_lane}\text{’}~~m{:}\mathtt{memarg}_{2}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.store16\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.store32\_lane}\text{’}~~m{:}\mathtt{memarg}_{4}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.store32\_lane}~m~\mathit{laneidx} \\
& | & \text{‘}\mathtt{v128.store64\_lane}\text{’}~~m{:}\mathtt{memarg}_{8}~~\mathit{laneidx}{:}\mathtt{u8} & \Rightarrow & \mathsf{v128.store64\_lane}~m~\mathit{laneidx}
\end{array}
$$

Vector constant instructions have a mandatory shape descriptor, which determines how the following values are parsed.

$$
\begin{array}{llcl}
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{i8x16}\text{’}~~(n{:}\mathtt{i8})^{16} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i8}(n)^{16}) \\
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{i16x8}\text{’}~~(n{:}\mathtt{i16})^{8} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i16}(n)^{8}) \\
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{i32x4}\text{’}~~(n{:}\mathtt{i32})^{4} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i32}(n)^{4}) \\
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{i64x2}\text{’}~~(n{:}\mathtt{i64})^{2} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i64}(n)^{2}) \\
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{f32x4}\text{’}~~(z{:}\mathtt{f32})^{4} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{f32}(z)^{4}) \\
| & \text{‘}\mathtt{v128.const}\text{’}~~\text{‘}\mathtt{f64x2}\text{’}~~(z{:}\mathtt{f64})^{2} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{f64}(z)^{2}) \\
& & & \\
| & \text{‘}\mathtt{i8x16.shuffle}\text{’}~~(\mathit{laneidx}{:}\mathtt{u8})^{16} & \Rightarrow & \mathsf{i8x16.shuffle}~\mathit{laneidx}^{16} \\
| & \text{‘}\mathtt{i8x16.swizzle}\text{’} & \Rightarrow & \mathsf{i8x16.swizzle} \\
& & & \\
| & \text{‘}\mathtt{i8x16.splat}\text{’} & \Rightarrow & \mathsf{i8x16.splat} \\
| & \text{‘}\mathtt{i16x8.splat}\text{’} & \Rightarrow & \mathsf{i16x8.splat} \\
| & \text{‘}\mathtt{i32x4.splat}\text{’} & \Rightarrow & \mathsf{i32x4.splat} \\
| & \text{‘}\mathtt{i64x2.splat}\text{’} & \Rightarrow & \mathsf{i64x2.splat} \\
| & \text{‘}\mathtt{f32x4.splat}\text{’} & \Rightarrow & \mathsf{f32x4.splat} \\
| & \text{‘}\mathtt{f64x2.splat}\text{’} & \Rightarrow & \mathsf{f64x2.splat}
\end{array}
$$

| ‘i8x16.extract_lane_s’ *laneidx*:u8 ⇒ i8x16.extract_lane_s *laneidx*
| ‘i8x16.extract_lane_u’ *laneidx*:u8 ⇒ i8x16.extract_lane_u *laneidx*
| ‘i8x16.replace_lane’ *laneidx*:u8 ⇒ i8x16.replace_lane *laneidx*
| ‘i16x8.extract_lane_s’ *laneidx*:u8 ⇒ i16x8.extract_lane_s *laneidx*
| ‘i16x8.extract_lane_u’ *laneidx*:u8 ⇒ i16x8.extract_lane_u *laneidx*
| ‘i16x8.replace_lane’ *laneidx*:u8 ⇒ i16x8.replace_lane *laneidx*
| ‘i32x4.extract_lane’ *laneidx*:u8 ⇒ i32x4.extract_lane *laneidx*
| ‘i32x4.replace_lane’ *laneidx*:u8 ⇒ i32x4.replace_lane *laneidx*
| ‘i64x2.extract_lane’ *laneidx*:u8 ⇒ i64x2.extract_lane *laneidx*
| ‘i64x2.replace_lane’ *laneidx*:u8 ⇒ i64x2.replace_lane *laneidx*
| ‘f32x4.extract_lane’ *laneidx*:u8 ⇒ f32x4.extract_lane *laneidx*
| ‘f32x4.replace_lane’ *laneidx*:u8 ⇒ f32x4.replace_lane *laneidx*
| ‘f64x2.extract_lane’ *laneidx*:u8 ⇒ f64x2.extract_lane *laneidx*
| ‘f64x2.replace_lane’ *laneidx*:u8 ⇒ f64x2.replace_lane *laneidx*

| ‘i8x16.eq’ ⇒ i8x16.eq
| ‘i8x16.ne’ ⇒ i8x16.ne
| ‘i8x16.lt_s’ ⇒ i8x16.lt_s
| ‘i8x16.lt_u’ ⇒ i8x16.lt_u
| ‘i8x16.gt_s’ ⇒ i8x16.gt_s
| ‘i8x16.gt_u’ ⇒ i8x16.gt_u
| ‘i8x16.le_s’ ⇒ i8x16.le_s
| ‘i8x16.le_u’ ⇒ i8x16.le_u
| ‘i8x16.ge_s’ ⇒ i8x16.ge_s
| ‘i8x16.ge_u’ ⇒ i8x16.ge_u

| ‘i16x8.eq’ ⇒ i16x8.eq
| ‘i16x8.ne’ ⇒ i16x8.ne
| ‘i16x8.lt_s’ ⇒ i16x8.lt_s
| ‘i16x8.lt_u’ ⇒ i16x8.lt_u
| ‘i16x8.gt_s’ ⇒ i16x8.gt_s
| ‘i16x8.gt_u’ ⇒ i16x8.gt_u
| ‘i16x8.le_s’ ⇒ i16x8.le_s
| ‘i16x8.le_u’ ⇒ i16x8.le_u
| ‘i16x8.ge_s’ ⇒ i16x8.ge_s
| ‘i16x8.ge_u’ ⇒ i16x8.ge_u

| ‘i32x4.eq’ ⇒ i32x4.eq
| ‘i32x4.ne’ ⇒ i32x4.ne
| ‘i32x4.lt_s’ ⇒ i32x4.lt_s
| ‘i32x4.lt_u’ ⇒ i32x4.lt_u
| ‘i32x4.gt_s’ ⇒ i32x4.gt_s
| ‘i32x4.gt_u’ ⇒ i32x4.gt_u
| ‘i32x4.le_s’ ⇒ i32x4.le_s
| ‘i32x4.le_u’ ⇒ i32x4.le_u
| ‘i32x4.ge_s’ ⇒ i32x4.ge_s
| ‘i32x4.ge_u’ ⇒ i32x4.ge_u

| ‘i64x2.eq’ ⇒ i64x2.eq
| ‘i64x2.ne’ ⇒ i64x2.ne
| ‘i64x2.lt_s’ ⇒ i64x2.lt_s
| ‘i64x2.gt_s’ ⇒ i64x2.gt_s
| ‘i64x2.le_s’ ⇒ i64x2.le_s
| ‘i64x2.ge_s’ ⇒ i64x2.ge_s
|

| ‘f32x4.eq’ ⇒ f32x4.eq
| ‘f32x4.ne’ ⇒ f32x4.ne
| ‘f32x4.lt’ ⇒ f32x4.lt
| ‘f32x4.gt’ ⇒ f32x4.gt
| ‘f32x4.le’ ⇒ f32x4.le
| ‘f32x4.ge’ ⇒ f32x4.ge

| ‘f64x2.eq’ ⇒ f64x2.eq
| ‘f64x2.ne’ ⇒ f64x2.ne
| ‘f64x2.lt’ ⇒ f64x2.lt
| ‘f64x2.gt’ ⇒ f64x2.gt
| ‘f64x2.le’ ⇒ f64x2.le
| ‘f64x2.ge’ ⇒ f64x2.ge

| ‘v128.not’ ⇒ v128.not
| ‘v128.and’ ⇒ v128.and
| ‘v128.andnot’ ⇒ v128.andnot
| ‘v128.or’ ⇒ v128.or
| ‘v128.xor’ ⇒ v128.xor
| ‘v128.bitselect’ ⇒ v128.bitselect
| ‘v128.any_true’ ⇒ v128.any_true

| ‘i8x16.abs’ ⇒ i8x16.abs
| ‘i8x16.neg’ ⇒ i8x16.neg
| ‘i8x16.all_true’ ⇒ i8x16.all_true
| ‘i8x16.bitmask’ ⇒ i8x16.bitmask
| ‘i8x16.narrow_i16x8_s’ ⇒ i8x16.narrow_i16x8_s
| ‘i8x16.narrow_i16x8_u’ ⇒ i8x16.narrow_i16x8_u
| ‘i8x16.shl’ ⇒ i8x16.shl
| ‘i8x16.shr_s’ ⇒ i8x16.shr_s
| ‘i8x16.shr_u’ ⇒ i8x16.shr_u
| ‘i8x16.add’ ⇒ i8x16.add
| ‘i8x16.add_sat_s’ ⇒ i8x16.add_sat_s
| ‘i8x16.add_sat_u’ ⇒ i8x16.add_sat_u
| ‘i8x16.sub’ ⇒ i8x16.sub
| ‘i8x16.sub_sat_s’ ⇒ i8x16.sub_sat_s
| ‘i8x16.sub_sat_u’ ⇒ i8x16.sub_sat_u
| ‘i8x16.min_s’ ⇒ i8x16.min_s
| ‘i8x16.min_u’ ⇒ i8x16.min_u
| ‘i8x16.max_s’ ⇒ i8x16.max_s
| ‘i8x16.max_u’ ⇒ i8x16.max_u
| ‘i8x16.avgr_u’ ⇒ i8x16.avgr_u
| ‘i8x16.popcnt’ ⇒ i8x16.popcnt

| ‘i16x8.abs’ ⇒ i16x8.abs
| ‘i16x8.neg’ ⇒ i16x8.neg
| ‘i16x8.all_true’ ⇒ i16x8.all_true
| ‘i16x8.bitmask’ ⇒ i16x8.bitmask
| ‘i16x8.narrow_i32x4_s’ ⇒ i16x8.narrow_i32x4_s
| ‘i16x8.narrow_i32x4_u’ ⇒ i16x8.narrow_i32x4_u
| ‘i16x8.extend_low_i8x16_s’ ⇒ i16x8.extend_low_i8x16_s
| ‘i16x8.extend_high_i8x16_s’ ⇒ i16x8.extend_high_i8x16_s
| ‘i16x8.extend_low_i8x16_u’ ⇒ i16x8.extend_low_i8x16_u
| ‘i16x8.extend_high_i8x16_u’ ⇒ i16x8.extend_high_i8x16_u
| ‘i16x8.shl’ ⇒ i16x8.shl
| ‘i16x8.shr_s’ ⇒ i16x8.shr_s
| ‘i16x8.shr_u’ ⇒ i16x8.shr_u
| ‘i16x8.add’ ⇒ i16x8.add
| ‘i16x8.add_sat_s’ ⇒ i16x8.add_sat_s
| ‘i16x8.add_sat_u’ ⇒ i16x8.add_sat_u
| ‘i16x8.sub’ ⇒ i16x8.sub
| ‘i16x8.sub_sat_s’ ⇒ i16x8.sub_sat_s
| ‘i16x8.sub_sat_u’ ⇒ i16x8.sub_sat_u
| ‘i16x8.mul’ ⇒ i16x8.mul
| ‘i16x8.min_s’ ⇒ i16x8.min_s
| ‘i16x8.min_u’ ⇒ i16x8.min_u
| ‘i16x8.max_s’ ⇒ i16x8.max_s
| ‘i16x8.max_u’ ⇒ i16x8.max_u
| ‘i16x8.avgr_u’ ⇒ i16x8.avgr_u
| ‘i16x8.q15mulr_sat_s’ ⇒ i16x8.q15mulr_sat_s
| ‘i16x8.extmul_low_i8x16_s’ ⇒ i16x8.extmul_low_i8x16_s
| ‘i16x8.extmul_high_i8x16_s’ ⇒ i16x8.extmul_high_i8x16_s
| ‘i16x8.extmul_low_i8x16_u’ ⇒ i16x8.extmul_low_i8x16_u
| ‘i16x8.extmul_high_i8x16_u’ ⇒ i16x8.extmul_high_i8x16_u
| ‘i16x8.extadd_pairwise_i8x16_s’ ⇒ i16x8.extadd_pairwise_i8x16_s
| ‘i16x8.extadd_pairwise_i8x16_u’ ⇒ i16x8.extadd_pairwise_i8x16_u

| ‘i32x4.abs’ ⇒ i32x4.abs
| ‘i32x4.neg’ ⇒ i32x4.neg
| ‘i32x4.all_true’ ⇒ i32x4.all_true
| ‘i32x4.bitmask’ ⇒ i32x4.bitmask
| ‘i32x4.extadd_pairwise_i16x8_s’ ⇒ i32x4.extadd_pairwise_i16x8_s
| ‘i32x4.extend_low_i16x8_s’ ⇒ i32x4.extend_low_i16x8_s
| ‘i32x4.extend_high_i16x8_s’ ⇒ i32x4.extend_high_i16x8_s
| ‘i32x4.extend_low_i16x8_u’ ⇒ i32x4.extend_low_i16x8_u
| ‘i32x4.extend_high_i16x8_u’ ⇒ i32x4.extend_high_i16x8_u
| ‘i32x4.shl’ ⇒ i32x4.shl
| ‘i32x4.shr_s’ ⇒ i32x4.shr_s
| ‘i32x4.shr_u’ ⇒ i32x4.shr_u
| ‘i32x4.add’ ⇒ i32x4.add
| ‘i32x4.sub’ ⇒ i32x4.sub
| ‘i32x4.mul’ ⇒ i32x4.mul
| ‘i32x4.min_s’ ⇒ i32x4.min_s
| ‘i32x4.min_u’ ⇒ i32x4.min_u
| ‘i32x4.max_s’ ⇒ i32x4.max_s
| ‘i32x4.max_u’ ⇒ i32x4.max_u
| ‘i32x4.dot_i16x8_s’ ⇒ i32x4.dot_i16x8_s
| ‘i32x4.extmul_low_i16x8_s’ ⇒ i32x4.extmul_low_i16x8_s
| ‘i32x4.extmul_high_i16x8_s’ ⇒ i32x4.extmul_high_i16x8_s
| ‘i32x4.extmul_low_i16x8_u’ ⇒ i32x4.extmul_low_i16x8_u
| ‘i32x4.extmul_high_i16x8_u’ ⇒ i32x4.extmul_high_i16x8_u

| ‘i64x2.abs’ ⇒ i64x2.abs
| ‘i64x2.neg’ ⇒ i64x2.neg
| ‘i64x2.all_true’ ⇒ i64x2.all_true
| ‘i64x2.bitmask’ ⇒ i64x2.bitmask
| ‘i64x2.extend_low_i32x4_s’ ⇒ i64x2.extend_low_i32x4_s
| ‘i64x2.extend_high_i32x4_s’ ⇒ i64x2.extend_high_i32x4_s
| ‘i64x2.extend_low_i32x4_u’ ⇒ i64x2.extend_low_i32x4_u
| ‘i64x2.extend_high_i32x4_u’ ⇒ i64x2.extend_high_i32x4_u
| ‘i64x2.shl’ ⇒ i64x2.shl
| ‘i64x2.shr_s’ ⇒ i64x2.shr_s
| ‘i64x2.shr_u’ ⇒ i64x2.shr_u
| ‘i64x2.add’ ⇒ i64x2.add
| ‘i64x2.sub’ ⇒ i64x2.sub
| ‘i64x2.mul’ ⇒ i64x2.mul
| ‘i64x2.extmul_low_i32x4_s’ ⇒ i64x2.extmul_low_i32x4_s
| ‘i64x2.extmul_high_i32x4_s’ ⇒ i64x2.extmul_high_i32x4_s
| ‘i64x2.extmul_low_i32x4_u’ ⇒ i64x2.extmul_low_i32x4_u
| ‘i64x2.extmul_high_i32x4_u’ ⇒ i64x2.extmul_high_i32x4_u

| ‘f32x4.abs’ ⇒ f32x4.abs
| ‘f32x4.neg’ ⇒ f32x4.neg
| ‘f32x4.sqrt’ ⇒ f32x4.sqrt
| ‘f32x4.add’ ⇒ f32x4.add
| ‘f32x4.sub’ ⇒ f32x4.sub
| ‘f32x4.mul’ ⇒ f32x4.mul
| ‘f32x4.div’ ⇒ f32x4.div
| ‘f32x4.min’ ⇒ f32x4.min
| ‘f32x4.max’ ⇒ f32x4.max
| ‘f32x4.pmin’ ⇒ f32x4.pmin
| ‘f32x4.pmax’ ⇒ f32x4.pmax

| ‘f64x2.abs’ ⇒ f64x2.abs
| ‘f64x2.neg’ ⇒ f64x2.neg
| ‘f64x2.sqrt’ ⇒ f64x2.sqrt
| ‘f64x2.add’ ⇒ f64x2.add
| ‘f64x2.sub’ ⇒ f64x2.sub
| ‘f64x2.mul’ ⇒ f64x2.mul
| ‘f64x2.div’ ⇒ f64x2.div
| ‘f64x2.min’ ⇒ f64x2.min
| ‘f64x2.max’ ⇒ f64x2.max
| ‘f64x2.pmin’ ⇒ f64x2.pmin
| ‘f64x2.pmax’ ⇒ f64x2.pmax

| ‘i32x4.trunc_sat_f32x4_s’ ⇒ i32x4.trunc_sat_f32x4_s
| ‘i32x4.trunc_sat_f32x4_u’ ⇒ i32x4.trunc_sat_f32x4_u
| ‘i32x4.trunc_sat_f64x2_s_zero’ ⇒ i32x4.trunc_sat_f64x2_s_zero
| ‘i32x4.trunc_sat_f64x2_u_zero’ ⇒ i32x4.trunc_sat_f64x2_u_zero
| ‘f32x4.convert_i32x4_s’ ⇒ f32x4.convert_i32x4_s
| ‘f32x4.convert_i32x4_u’ ⇒ f32x4.convert_i32x4_u
| ‘f64x2.convert_low_i32x4_s’ ⇒ f64x2.convert_low_i32x4_s
| ‘f64x2.convert_low_i32x4_u’ ⇒ f64x2.convert_low_i32x4_u
| ‘f32x4.demote_f64x2_zero’ ⇒ f32x4.demote_f64x2_zero
| ‘f64x2.promote_low_f32x4’ ⇒ f64x2.promote_low_f32x4

6.5.11 Folded Instructions

Instructions can be written as S-expressions by grouping them into *folded* form. In that notation, an instruction is wrapped in parentheses and optionally includes nested folded instructions to indicate its operands.

In the case of block instructions, the folded form omits the '`end`' delimiter. For if instructions, both branches have to be wrapped into nested S-expressions, headed by the keywords '`then`' and '`else`'.

The set of all phrases defined by the following abbreviations recursively forms the auxiliary syntactic class `foldedinstr`. Such a folded instruction can appear anywhere a regular instruction can.

$$\begin{array}{lll}
\text{`('}\ \mathtt{plaininstr}\ \mathtt{foldedinstr}^*\ \text{`)'} & \equiv & \mathtt{foldedinstr}^*\ \mathtt{plaininstr} \\
\text{`('}\ \text{`}\mathtt{block}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{instr}^*\ \text{`)'} & \equiv & \text{`}\mathtt{block}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{instr}^*\ \text{`}\mathtt{end}\text{'} \\
\text{`('}\ \text{`}\mathtt{loop}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{instr}^*\ \text{`)'} & \equiv & \text{`}\mathtt{loop}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{instr}^*\ \text{`}\mathtt{end}\text{'} \\
\multicolumn{3}{l}{\text{`('}\ \text{`}\mathtt{if}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{foldedinstr}^*\ \text{`('}\ \text{`}\mathtt{then}\text{'}\ \mathtt{instr}_1^*\ \text{`)'}\ (\text{`('}\ \text{`}\mathtt{else}\text{'}\ \mathtt{instr}_2^*\ \text{`)'})^?\ \text{`)'}\quad \equiv} \\
\multicolumn{3}{l}{\quad \mathtt{foldedinstr}^*\ \text{`}\mathtt{if}\text{'}\ \mathtt{label}\ \mathtt{blocktype}\ \mathtt{instr}_1^*\ \text{`}\mathtt{else}\text{'}\ (\mathtt{instr}_2^*)^?\ \text{`}\mathtt{end}\text{'}}
\end{array}$$

Note: For example, the instruction sequence

(`local.get $x`) (`i32.const 2`) `i32.add` (`i32.const 3`) `i32.mul`

can be folded into

(`i32.mul` (`i32.add` (`local.get $x`) (`i32.const 2`)) (`i32.const 3`))

Folded instructions are solely syntactic sugar, no additional syntactic or type-based checking is implied.

6.5.12 Expressions

Expressions are written as instruction sequences. No explicit '`end`' keyword is included, since they only occur in bracketed positions.

$$\mathtt{expr}_I \quad ::= \quad (in{:}\mathtt{instr}_I)^* \quad \Rightarrow \quad in^*\ \mathsf{end}$$

6.6 Modules

6.6.1 Indices

Indices can be given either in raw numeric form or as symbolic identifiers when bound by a respective construct. Such identifiers are looked up in the suitable space of the identifier context I.

$$\begin{array}{llcll}
\mathtt{typeidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{types}[x] = v) \\
\mathtt{funcidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{funcs}[x] = v) \\
\mathtt{tableidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{tables}[x] = v) \\
\mathtt{memidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{mems}[x] = v) \\
\mathtt{globalidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{globals}[x] = v) \\
\mathtt{elemidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{elem}[x] = v) \\
\mathtt{dataidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{data}[x] = v) \\
\mathtt{localidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
 & | & v{:}\mathtt{id} & \Rightarrow & x \quad (\text{if } I.\mathsf{locals}[x] = v) \\
\mathtt{labelidx}_I & ::= & l{:}\mathtt{u32} & \Rightarrow & l \\
 & | & v{:}\mathtt{id} & \Rightarrow & l \quad (\text{if } I.\mathsf{labels}[l] = v)
\end{array}$$

6.6.2 Types

Type definitions can bind a symbolic type identifier.

$$\mathtt{type} \quad ::= \quad \text{‘('}~\text{‘}\mathtt{type}\text{'}~\mathtt{id}^?~ft{:}\mathtt{functype}~\text{‘)'} \quad \Rightarrow \quad ft$$

6.6.3 Type Uses

A *type use* is a reference to a type definition. It may optionally be augmented by explicit inlined parameter and result declarations. That allows binding symbolic identifiers to name the local indices of parameters. If inline declarations are given, then their types must match the referenced function type.

$$\begin{array}{llll}
\mathtt{typeuse}_I & ::= & \text{‘('}~\text{‘}\mathtt{type}\text{'}~x{:}\mathtt{typeidx}_I~\text{‘)'} & \Rightarrow \quad x, I' \\
 & & \quad (\text{if } I.\mathsf{typedefs}[x] = [t_1^n] \to [t_2^*] \wedge I' = \{\mathsf{locals}~(\epsilon)^n\}) & \\
 & | & \text{‘('}~\text{‘}\mathtt{type}\text{'}~x{:}\mathtt{typeidx}_I~\text{‘)'}~(t_1{:}\mathtt{param})^*~(t_2{:}\mathtt{result})^* & \Rightarrow \quad x, I' \\
 & & \quad (\text{if } I.\mathsf{typedefs}[x] = [t_1^*] \to [t_2^*] \wedge I' = \{\mathsf{locals}~\mathrm{id}(\mathtt{param})^*\}~\text{well-formed}) &
\end{array}$$

The synthesized attribute of a `typeuse` is a pair consisting of both the used type index and the local identifier context containing possible parameter identifiers. The following auxiliary function extracts optional identifiers from parameters:

$$\mathrm{id}(\text{‘('}~\text{‘}\mathtt{param}\text{'}~\mathtt{id}^?~\ldots~\text{‘)'}) \quad = \quad \mathtt{id}^?$$

Note: Both productions overlap for the case that the function type is $[] \to []$. However, in that case, they also produce the same results, so that the choice is immaterial.

The well-formedness condition on I' ensures that the parameters do not contain duplicate identifiers.

Abbreviations

A `typeuse` may also be replaced entirely by inline parameter and result declarations. In that case, a type index is automatically inserted:

$$(t_1{:}\mathtt{param})^*~~(t_2{:}\mathtt{result})^* \quad\equiv\quad \text{‘(’}~\text{‘}\mathtt{type}\text{’}~x~\text{‘)’}~~\mathtt{param}^*~~\mathtt{result}^*$$

where x is the smallest existing type index whose definition in the current module is the function type $[t_1^*] \rightarrow [t_2^*]$. If no such index exists, then a new type definition of the form

$$\text{‘(’}~\text{‘}\mathtt{type}\text{’}~~\text{‘(’}~\text{‘}\mathtt{func}\text{’}~~\mathtt{param}^*~~\mathtt{result}^*~\text{‘)’}~\text{‘)’}$$

is inserted at the end of the module.

Abbreviations are expanded in the order they appear, such that previously inserted type definitions are reused by consecutive expansions.

6.6.4 Imports

The descriptors in imports can bind a symbolic function, table, memory, or global identifier.

$$\begin{array}{llll}
\mathtt{import}_I & ::= & \text{‘(’}~\text{‘}\mathtt{import}\text{’}~~mod{:}\mathtt{name}~~nm{:}\mathtt{name}~~d{:}\mathtt{importdesc}_I~\text{‘)’} & \\
 & & \qquad\Rightarrow\quad \{\mathsf{module}~mod, \mathsf{name}~nm, \mathsf{desc}~d\} & \\
\mathtt{importdesc}_I & ::= & \text{‘(’}~\text{‘}\mathtt{func}\text{’}~~\mathtt{id}^?~~x, I'{:}\mathtt{typeuse}_I~\text{‘)’} & \Rightarrow\quad \mathsf{func}~x \\
 & | & \text{‘(’}~\text{‘}\mathtt{table}\text{’}~~\mathtt{id}^?~~tt{:}\mathtt{tabletype}~\text{‘)’} & \Rightarrow\quad \mathsf{table}~tt \\
 & | & \text{‘(’}~\text{‘}\mathtt{memory}\text{’}~~\mathtt{id}^?~~mt{:}\mathtt{memtype}~\text{‘)’} & \Rightarrow\quad \mathsf{mem}~~mt \\
 & | & \text{‘(’}~\text{‘}\mathtt{global}\text{’}~~\mathtt{id}^?~~gt{:}\mathtt{globaltype}~\text{‘)’} & \Rightarrow\quad \mathsf{global}~gt
\end{array}$$

Abbreviations

As an abbreviation, imports may also be specified inline with function, table, memory, or global definitions; see the respective sections.

6.6.5 Functions

Function definitions can bind a symbolic function identifier, and local identifiers for its parameters and locals.

$$\begin{array}{lll}
\mathtt{func}_I & ::= & \text{‘(’}~\text{‘}\mathtt{func}\text{’}~~\mathtt{id}^?~~x, I'{:}\mathtt{typeuse}_I~~(t{:}\mathtt{local})^*~~(in{:}\mathtt{instr}_{I''})^*~\text{‘)’} \\
 & & \qquad\Rightarrow\quad \{\mathsf{type}~x, \mathsf{locals}~t^*, \mathsf{body}~in^*~\mathsf{end}\} \\
 & & \qquad\qquad\qquad (\text{if } I'' = I \oplus I' \oplus \{\mathsf{locals}~\mathrm{id}(\mathtt{local})^*\}~\text{well-formed}) \\
\mathtt{local} & ::= & \text{‘(’}~\text{‘}\mathtt{local}\text{’}~~\mathtt{id}^?~~t{:}\mathtt{valtype}~\text{‘)’} \quad\Rightarrow\quad t
\end{array}$$

The definition of the local identifier context I'' uses the following auxiliary function to extract optional identifiers from locals:

$$\mathrm{id}(\text{‘(’}~\text{‘}\mathtt{local}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}) \quad=\quad \mathtt{id}^?$$

Note: The well-formedness condition on I'' ensures that parameters and locals do not contain duplicate identifiers.

Abbreviations

Multiple anonymous locals may be combined into a single declaration:

$$\text{‘(’}~\text{‘}\mathtt{local}\text{’}~\mathtt{valtype}^*~\text{‘)’} \quad\equiv\quad (\text{‘(’}~\text{‘}\mathtt{local}\text{’}~\mathtt{valtype}~\text{‘)’})^*$$

Functions can be defined as imports or exports inline:

$$\begin{array}{l}
\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}^?~\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\mathtt{name}_1~\mathtt{name}_2~\text{‘)’}~\mathtt{typeuse}~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{import}\text{’}~\mathtt{name}_1~\mathtt{name}_2~\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}^?~\mathtt{typeuse}~\text{‘)’}~\text{‘)’} \\
\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}^?~\text{‘(’}~\text{‘}\mathtt{export}\text{’}~\mathtt{name}~\text{‘)’}~\dots~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{export}\text{’}~\mathtt{name}~\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}'~\text{‘)’}~\text{‘)’}~\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}'~\dots~\text{‘)’} \\
\qquad (\text{if}~\mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a function declaration can contain any number of exports, possibly followed by an import.

6.6.6 Tables

Table definitions can bind a symbolic table identifier.

$$\mathtt{table}_I \quad ::= \quad \text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~tt{:}\mathtt{tabletype}~\text{‘)’} \quad\Rightarrow\quad \{\mathrm{type}~tt\}$$

Abbreviations

An element segment can be given inline with a table definition, in which case its offset is 0 and the limits of the table type are inferred from the length of the given segment:

$$\begin{array}{l}
\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\mathtt{reftype}~\text{‘(’}~\text{‘}\mathtt{elem}\text{’}~\mathit{expr}^n{:}\mathtt{vec}(\mathtt{elemexpr})~\text{‘)’}~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~n~n~\mathtt{reftype}~\text{‘)’} \\
\quad \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~\text{‘)’}~\text{‘(’}~\text{‘}\mathtt{i32.const}\text{’}~\text{‘}\mathtt{0}\text{’}~\text{‘)’}~\mathtt{reftype}~\mathtt{vec}(\mathtt{elemexpr})~\text{‘)’} \\
\qquad (\text{if}~\mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

$$\begin{array}{l}
\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\mathtt{reftype}~\text{‘(’}~\text{‘}\mathtt{elem}\text{’}~x^n{:}\mathtt{vec}(\mathtt{funcidx})~\text{‘)’}~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~n~n~\mathtt{reftype}~\text{‘)’} \\
\quad \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~\text{‘)’}~\text{‘(’}~\text{‘}\mathtt{i32.const}\text{’}~\text{‘}\mathtt{0}\text{’}~\text{‘)’}~\text{‘}\mathtt{func}\text{’}~\mathtt{vec}(\mathtt{funcidx})~\text{‘)’} \\
\qquad (\text{if}~\mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

Tables can be defined as imports or exports inline:

$$\begin{array}{l}
\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\mathtt{name}_1~\mathtt{name}_2~\text{‘)’}~\mathtt{tabletype}~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{import}\text{’}~\mathtt{name}_1~\mathtt{name}_2~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\mathtt{tabletype}~\text{‘)’}~\text{‘)’} \\
\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\text{‘(’}~\text{‘}\mathtt{export}\text{’}~\mathtt{name}~\text{‘)’}~\dots~\text{‘)’} \quad\equiv \\
\quad \text{‘(’}~\text{‘}\mathtt{export}\text{’}~\mathtt{name}~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~\text{‘)’}~\text{‘)’}~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}'~\dots~\text{‘)’} \\
\qquad (\text{if}~\mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a table declaration can contain any number of exports, possibly followed by an import.

6.6.7 Memories

Memory definitions can bind a symbolic memory identifier.

$$\mathtt{mem}_I \quad ::= \quad \text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~mt{:}\mathtt{memtype}~\text{`)'} \quad \Rightarrow \quad \{\mathsf{type}~mt\}$$

Abbreviations

A data segment can be given inline with a memory definition, in which case its offset is 0 and the limits of the memory type are inferred from the length of the data, rounded up to page size:

$$\begin{array}{l}
\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\text{`('}~\text{`}\mathtt{data}\text{'}~b^n{:}\mathtt{datastring}~\text{`)'}~\text{`)'} \quad \equiv \\
\quad \text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}'~m~m~\text{`)'} \\
\quad \text{`('}~\text{`}\mathtt{data}\text{'}~\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}'~\text{`)'}~\text{`('}~\text{`}\mathtt{i32.const}\text{'}~\text{`}0\text{'}~\text{`)'}~\mathtt{datastring}~\text{`)'} \\
\qquad (\text{if } \mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh}, m = \mathrm{ceil}(n/64\,\mathrm{Ki}))
\end{array}$$

Memories can be defined as imports or exports inline:

$$\begin{array}{l}
\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\text{`('}~\text{`}\mathtt{import}\text{'}~\mathtt{name}_1~\mathtt{name}_2~\text{`)'}~\mathtt{memtype}~\text{`)'} \quad \equiv \\
\quad \text{`('}~\text{`}\mathtt{import}\text{'}~\mathtt{name}_1~\mathtt{name}_2~\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\mathtt{memtype}~\text{`)'}~\text{`)'} \\
\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\text{`('}~\text{`}\mathtt{export}\text{'}~\mathtt{name}~\text{`)'}~\dots~\text{`)'} \quad \equiv \\
\quad \text{`('}~\text{`}\mathtt{export}\text{'}~\mathtt{name}~\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}'~\text{`)'}~\text{`)'}~\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}'~\dots~\text{`)'} \\
\qquad (\text{if } \mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

Note: The latter abbreviation can be applied repeatedly, if "..." contains additional export clauses. Consequently, a memory declaration can contain any number of exports, possibly followed by an import.

6.6.8 Globals

Global definitions can bind a symbolic global identifier.

$$\mathtt{global}_I \quad ::= \quad \text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~gt{:}\mathtt{globaltype}~e{:}\mathtt{expr}_I~\text{`)'} \quad \Rightarrow \quad \{\mathsf{type}~gt, \mathsf{init}~e\}$$

Abbreviations

Globals can be defined as imports or exports inline:

$$\begin{array}{l}
\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~\text{`('}~\text{`}\mathtt{import}\text{'}~\mathtt{name}_1~\mathtt{name}_2~\text{`)'}~\mathtt{globaltype}~\text{`)'} \quad \equiv \\
\quad \text{`('}~\text{`}\mathtt{import}\text{'}~\mathtt{name}_1~\mathtt{name}_2~\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~\mathtt{globaltype}~\text{`)'}~\text{`)'} \\
\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~\text{`('}~\text{`}\mathtt{export}\text{'}~\mathtt{name}~\text{`)'}~\dots~\text{`)'} \quad \equiv \\
\quad \text{`('}~\text{`}\mathtt{export}\text{'}~\mathtt{name}~\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}'~\text{`)'}~\text{`)'}~\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}'~\dots~\text{`)'} \\
\qquad (\text{if } \mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}'~\text{fresh})
\end{array}$$

Note: The latter abbreviation can be applied repeatedly, if "..." contains additional export clauses. Consequently, a global declaration can contain any number of exports, possibly followed by an import.

6.6.9 Exports

The syntax for exports mirrors their abstract syntax directly.

$$\begin{array}{llclll}
\mathtt{export}_I & ::= & \text{‘(’}~\text{‘}\mathtt{export}\text{’}~~nm{:}\mathtt{name}~~d{:}\mathtt{exportdesc}_I~\text{‘)’} & \Rightarrow & \{\mathsf{name}~nm, \mathsf{desc}~d\} \\
\mathtt{exportdesc}_I & ::= & \text{‘(’}~\text{‘}\mathtt{func}\text{’}~~x{:}\mathtt{funcidx}_I~\text{‘)’} & \Rightarrow & \mathsf{func}~x \\
 & | & \text{‘(’}~\text{‘}\mathtt{table}\text{’}~~x{:}\mathtt{tableidx}_I~\text{‘)’} & \Rightarrow & \mathsf{table}~x \\
 & | & \text{‘(’}~\text{‘}\mathtt{memory}\text{’}~~x{:}\mathtt{memidx}_I~\text{‘)’} & \Rightarrow & \mathsf{mem}~x \\
 & | & \text{‘(’}~\text{‘}\mathtt{global}\text{’}~~x{:}\mathtt{globalidx}_I~\text{‘)’} & \Rightarrow & \mathsf{global}~x
\end{array}$$

Abbreviations

As an abbreviation, exports may also be specified inline with function, table, memory, or global definitions; see the respective sections.

6.6.10 Start Function

A start function is defined in terms of its index.

$$\begin{array}{llcll}
\mathtt{start}_I & ::= & \text{‘(’}~\text{‘}\mathtt{start}\text{’}~~x{:}\mathtt{funcidx}_I~\text{‘)’} & \Rightarrow & \{\mathsf{func}~x\}
\end{array}$$

Note: At most one start function may occur in a module, which is ensured by a suitable side condition on the `module` grammar.

6.6.11 Element Segments

Element segments allow for an optional table index to identify the table to initialize.

$$\begin{array}{llll}
\mathtt{elem}_I & ::= & \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~~\mathtt{id}^?~~(et, y^*){:}\mathtt{elemlist}_I~\text{‘)’} \\
 & & \qquad \Rightarrow \quad \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{passive}\} \\
 & | & \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~~\mathtt{id}^?~~x{:}\mathtt{tableuse}_I~~\text{‘(’}~\text{‘}\mathtt{offset}\text{’}~~e{:}\mathtt{expr}_I~\text{‘)’}~~(et, y^*){:}\mathtt{elemlist}_I~\text{‘)’} \\
 & & \qquad \Rightarrow \quad \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~e\}\} \\
 & & \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~~\mathtt{id}^?~~\text{‘}\mathtt{declare}\text{’}~~(et, y^*){:}\mathtt{elemlist}_I~\text{‘)’} \\
 & & \qquad \Rightarrow \quad \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{declarative}\} \\
\mathtt{elemlist}_I & ::= & t{:}\mathtt{reftype}~~y^*{:}\mathtt{vec}(\mathtt{elemexpr}_I) \quad \Rightarrow \quad (\mathsf{type}~t, \mathsf{init}~y^*) \\
\mathtt{elemexpr}_I & ::= & \text{‘(’}~\text{‘}\mathtt{item}\text{’}~~e{:}\mathtt{expr}_I~\text{‘)’} \quad \Rightarrow \quad e \\
\mathtt{tableuse}_I & ::= & \text{‘(’}~\text{‘}\mathtt{table}\text{’}~~x{:}\mathtt{tableidx}_I~\text{‘)’} \quad \Rightarrow \quad x
\end{array}$$

Abbreviations

As an abbreviation, a single instruction may occur in place of the offset of an active element segment or as an element expression:

$$\begin{array}{lll}
\text{‘(’}~\mathtt{instr}~\text{‘)’} & \equiv & \text{‘(’}~\text{‘}\mathtt{offset}\text{’}~~\mathtt{instr}~\text{‘)’} \\
\text{‘(’}~\mathtt{instr}~\text{‘)’} & \equiv & \text{‘(’}~\text{‘}\mathtt{item}\text{’}~~\mathtt{instr}~\text{‘)’}
\end{array}$$

Also, the element list may be written as just a sequence of function indices:

$$\text{‘}\mathtt{func}\text{’}~~\mathtt{vec}(\mathtt{funcidx}_I) \quad \equiv \quad \text{‘}\mathtt{funcref}\text{’}~~\mathtt{vec}(\text{‘(’}~\text{‘}\mathtt{ref.func}\text{’}~~\mathtt{funcidx}_I~\text{‘)’})$$

A table use can be omitted, defaulting to `0`. Furthermore, for backwards compatibility with earlier versions of WebAssembly, if the table use is omitted, the ‘`func`’ keyword can be omitted as well.

$$\begin{array}{lll}
\epsilon & \equiv & \text{‘(’}~\text{‘}\mathtt{table}\text{’}~~\text{‘}\mathtt{0}\text{’}~\text{‘)’} \\
\text{‘(’}~\text{‘}\mathtt{elem}\text{’}~~\mathtt{id}^?~~\text{‘(’}~\text{‘}\mathtt{offset}\text{’}~~\mathtt{expr}_I~\text{‘)’}~~\mathtt{vec}(\mathtt{funcidx}_I)~\text{‘)’} & \equiv & \text{‘(’}~\text{‘}\mathtt{elem}\text{’}~~\mathtt{id}^?~~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~~\text{‘}\mathtt{0}\text{’}~\text{‘)’}~~\text{‘(’}~\text{‘}\mathtt{offset}\text{’}~~\mathtt{expr}_I~\text{‘)’}
\end{array}$$

As another abbreviation, element segments may also be specified inline with table definitions; see the respective section.

6.6.12 Data Segments

Data segments allow for an optional memory index to identify the memory to initialize. The data is written as a string, which may be split up into a possibly empty sequence of individual string literals.

$$
\begin{array}{lcl}
\mathtt{data}_I & ::= & \text{`('}~\text{`}\mathtt{data}\text{'}~\mathtt{id}^?~b^*{:}\mathtt{datastring}~\text{`)'} \\
 & & \qquad\Rightarrow\quad \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{passive}\} \\
 & | & \text{`('}~\text{`}\mathtt{data}\text{'}~\mathtt{id}^?~x{:}\mathtt{memuse}_I~\text{`('}~\text{`}\mathtt{offset}\text{'}~e{:}\mathtt{expr}_I~\text{`)'}~b^*{:}\mathtt{datastring}~\text{`)'} \\
 & & \qquad\Rightarrow\quad \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{memory}~x', \mathsf{offset}~e\}\} \\
\mathtt{datastring} & ::= & (b^*{:}\mathtt{string})^* \quad\Rightarrow\quad \mathrm{concat}((b^*)^*) \\
\mathtt{memuse}_I & ::= & \text{`('}~\text{`}\mathtt{memory}\text{'}~x{:}\mathtt{memidx}_I~\text{`)'} \quad\Rightarrow\quad x
\end{array}
$$

Note: In the current version of WebAssembly, the only valid memory index is 0 or a symbolic memory identifier resolving to the same value.

Abbreviations

As an abbreviation, a single instruction may occur in place of the offset of an active data segment:

$$\text{`('}~\mathtt{instr}~\text{`)'} \quad\equiv\quad \text{`('}~\text{`}\mathtt{offset}\text{'}~\mathtt{instr}~\text{`)'}$$

Also, a memory use can be omitted, defaulting to 0.

$$\epsilon \quad\equiv\quad \text{`('}~\text{`}\mathtt{memory}\text{'}~\text{`}\mathtt{0}\text{'}~\text{`)'}$$

As another abbreviation, data segments may also be specified inline with memory definitions; see the respective section.

6.6.13 Modules

A module consists of a sequence of fields that can occur in any order. All definitions and their respective bound identifiers scope over the entire module, including the text preceding them.

A module may optionally bind an identifier that names the module. The name serves a documentary role only.

Note: Tools may include the module name in the name section of the binary format.

$$
\begin{array}{lclcl}
\mathtt{module} & ::= & \text{`('}~\text{`}\mathtt{module}\text{'}~\mathtt{id}^?~(m{:}\mathtt{modulefield}_I)^*~\text{`)'} & \Rightarrow & \bigoplus m^* \\
 & & \qquad (\text{if } I = \bigoplus \mathrm{idc}(\mathtt{modulefield})^* ~\text{well-formed}) & & \\
\mathtt{modulefield}_I & ::= & \mathit{ty}{:}\mathtt{type} & \Rightarrow & \{\mathsf{types}~\mathit{ty}\} \\
 & | & \mathit{im}{:}\mathtt{import}_I & \Rightarrow & \{\mathsf{imports}~\mathit{im}\} \\
 & | & \mathit{fn}{:}\mathtt{func}_I & \Rightarrow & \{\mathsf{funcs}~\mathit{fn}\} \\
 & | & \mathit{ta}{:}\mathtt{table}_I & \Rightarrow & \{\mathsf{tables}~\mathit{ta}\} \\
 & | & \mathit{me}{:}\mathtt{mem}_I & \Rightarrow & \{\mathsf{mems}~\mathit{me}\} \\
 & | & \mathit{gl}{:}\mathtt{global}_I & \Rightarrow & \{\mathsf{globals}~\mathit{gl}\} \\
 & | & \mathit{ex}{:}\mathtt{export}_I & \Rightarrow & \{\mathsf{exports}~\mathit{ex}\} \\
 & | & \mathit{st}{:}\mathtt{start}_I & \Rightarrow & \{\mathsf{start}~\mathit{st}\} \\
 & | & \mathit{el}{:}\mathtt{elem}_I & \Rightarrow & \{\mathsf{elems}~\mathit{el}\} \\
 & | & \mathit{da}{:}\mathtt{data}_I & \Rightarrow & \{\mathsf{datas}~\mathit{da}\}
\end{array}
$$

The following restrictions are imposed on the composition of modules: $m_1 \oplus m_2$ is defined if and only if

- $m_1.\mathsf{start} = \epsilon \vee m_2.\mathsf{start} = \epsilon$
- $m_1.\mathsf{funcs} = m_1.\mathsf{tables} = m_1.\mathsf{mems} = m_1.\mathsf{globals} = \epsilon \vee m_2.\mathsf{imports} = \epsilon$

Note: The first condition ensures that there is at most one start function. The second condition enforces that all imports must occur before any regular definition of a function, table, memory, or global, thereby maintaining the ordering of the respective index spaces.

The well-formedness condition on I in the grammar for `module` ensures that no namespace contains duplicate identifiers.

The definition of the initial identifier context I uses the following auxiliary definition which maps each relevant definition to a singular context with one (possibly empty) identifier:

$$\begin{array}{lcl}
\mathrm{idc}(\text{`('}~\text{`}\mathtt{type}\text{'}~\mathtt{id}^?~ft{:}\mathtt{functype}~\text{`)'}) & = & \{\mathsf{types}~(\mathtt{id}^?), \mathsf{typedefs}~ft\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{func}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{funcs}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{table}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{tables}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{mems}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{globals}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{elem}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{elem}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{data}\text{'}~\mathtt{id}^?~\dots~\text{`)'}) & = & \{\mathsf{data}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{func}\text{'}~\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) & = & \{\mathsf{funcs}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{table}\text{'}~\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) & = & \{\mathsf{tables}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{memory}\text{'}~\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) & = & \{\mathsf{mems}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{global}\text{'}~\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) & = & \{\mathsf{globals}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{`('}~\dots~\text{`)'}) & = & \{\}
\end{array}$$

Abbreviations

In a source file, the toplevel $(\mathtt{module}~\dots)$ surrounding the module body may be omitted.

$$\mathtt{modulefield}^* \quad \equiv \quad \text{`('}~\text{`}\mathtt{module}\text{'}~\mathtt{modulefield}^*~\text{`)'}$$

CHAPTER 7

Appendix

7.1 Embedding

A WebAssembly implementation will typically be *embedded* into a *host* environment. An *embedder* implements the connection between such a host environment and the WebAssembly semantics as defined in the main body of this specification. An embedder is expected to interact with the semantics in well-defined ways.

This section defines a suitable interface to the WebAssembly semantics in the form of entry points through which an embedder can access it. The interface is intended to be complete, in the sense that an embedder does not need to reference other functional parts of the WebAssembly specification directly.

Note: On the other hand, an embedder does not need to provide the host environment with access to all functionality defined in this interface. For example, an implementation may not support parsing of the text format.

7.1.1 Types

In the description of the embedder interface, syntactic classes from the abstract syntax and the runtime's abstract machine are used as names for variables that range over the possible objects from that class. Hence, these syntactic classes can also be interpreted as types.

For numeric parameters, notation like $n : u32$ is used to specify a symbolic name in addition to the respective value range.

7.1.2 Errors

Failure of an interface operation is indicated by an auxiliary syntactic class:

$$error ::= \mathsf{error}$$

In addition to the error conditions specified explicitly in this section, implementations may also return errors when specific implementation limitations are reached.

Note: Errors are abstract and unspecific with this definition. Implementations can refine it to carry suitable classifications and diagnostic messages.

7.1.3 Pre- and Post-Conditions

Some operations state *pre-conditions* about their arguments or *post-conditions* about their results. It is the embedder's responsibility to meet the pre-conditions. If it does, the post conditions are guaranteed by the semantics.

In addition to pre- and post-conditions explicitly stated with each operation, the specification adopts the following conventions for runtime objects ($store$, $moduleinst$, $externval$, addresses):

- Every runtime object passed as a parameter must be valid per an implicit pre-condition.
- Every runtime object returned as a result is valid per an implicit post-condition.

Note: As long as an embedder treats runtime objects as abstract and only creates and manipulates them through the interface defined here, all implicit pre-conditions are automatically met.

7.1.4 Store

$\mathrm{store\_init}() : store$

1. Return the empty store.

$$\mathrm{store\_init}() \quad = \quad \{\mathsf{funcs}~\epsilon,~\mathsf{mems}~\epsilon,~\mathsf{tables}~\epsilon,~\mathsf{globals}~\epsilon\}$$

7.1.5 Modules

$\mathrm{module\_decode}(byte^*) : module \mid error$

1. If there exists a derivation for the byte sequence $byte^*$ as a `module` according to the binary grammar for modules, yielding a module m, then return m.
2. Else, return error.

$$\begin{array}{lcll} \mathrm{module\_decode}(b^*) & = & m & (\mathrm{if}~\mathtt{module} \stackrel{*}{\Longrightarrow} m{:}b^*) \\ \mathrm{module\_decode}(b^*) & = & \mathsf{error} & (\mathrm{otherwise}) \end{array}$$

$\mathrm{module\_parse}(char^*) : module \mid error$

1. If there exists a derivation for the source $char^*$ as a `module` according to the text grammar for modules, yielding a module m, then return m.
2. Else, return error.

$$\begin{array}{lcll} \mathrm{module\_parse}(c^*) & = & m & (\mathrm{if}~\mathtt{module} \stackrel{*}{\Longrightarrow} m{:}c^*) \\ \mathrm{module\_parse}(c^*) & = & \mathsf{error} & (\mathrm{otherwise}) \end{array}$$

$\mathrm{module\_validate}(\mathit{module}) : \mathit{error}^?$

1. If module is valid, then return nothing.
2. Else, return error.

$$\begin{array}{lcll}
\mathrm{module\_validate}(m) & = & \epsilon & (\mathrm{if} \vdash m : \mathit{externtype}^* \to \mathit{externtype}'^*) \\
\mathrm{module\_validate}(m) & = & \mathsf{error} & (\mathrm{otherwise})
\end{array}$$

$\mathrm{module\_instantiate}(\mathit{store}, \mathit{module}, \mathit{externval}^*) : (\mathit{store}, \mathit{moduleinst} \mid \mathit{error})$

1. Try instantiating module in store with external values $\mathit{externval}^*$ as imports:
    a. If it succeeds with a module instance $\mathit{moduleinst}$, then let result be $\mathit{moduleinst}$.
    b. Else, let result be error.
2. Return the new store paired with result.

$$\begin{array}{lcll}
\mathrm{module\_instantiate}(S, m, ev^*) & = & (S', F.\mathsf{module}) & (\mathrm{if}\ \mathrm{instantiate}(S, m, ev^*) \hookrightarrow^* S'; F; \epsilon) \\
\mathrm{module\_instantiate}(S, m, ev^*) & = & (S', \mathsf{error}) & (\mathrm{if}\ \mathrm{instantiate}(S, m, ev^*) \hookrightarrow^* S'; F; \mathsf{trap})
\end{array}$$

Note: The store may be modified even in case of an error.

$\mathrm{module\_imports}(\mathit{module}) : (\mathit{name}, \mathit{name}, \mathit{externtype})^*$

1. Pre-condition: module is valid with external import types $\mathit{externtype}^*$ and external export types $\mathit{externtype}'^*$.
2. Let import^* be the imports $\mathit{module}.\mathsf{imports}$.
3. Assert: the length of import^* equals the length of $\mathit{externtype}^*$.
4. For each import_i in import^* and corresponding $\mathit{externtype}_i$ in $\mathit{externtype}^*$, do:
    a. Let result_i be the triple $(\mathit{import}_i.\mathsf{module}, \mathit{import}_i.\mathsf{name}, \mathit{externtype}_i)$.
5. Return the concatenation of all result_i, in index order.
6. Post-condition: each $\mathit{externtype}_i$ is valid.

$$\begin{array}{lcl}
\mathrm{module\_imports}(m) & = & (im.\mathsf{module}, im.\mathsf{name}, \mathit{externtype})^* \\
 & & \qquad (\mathrm{if}\ im^* = m.\mathsf{imports} \wedge \vdash m : \mathit{externtype}^* \to \mathit{externtype}'^*)
\end{array}$$

$\mathrm{module\_exports}(\mathit{module}) : (\mathit{name}, \mathit{externtype})^*$

1. Pre-condition: module is valid with external import types $\mathit{externtype}^*$ and external export types $\mathit{externtype}'^*$.
2. Let export^* be the exports $\mathit{module}.\mathsf{exports}$.
3. Assert: the length of export^* equals the length of $\mathit{externtype}'^*$.
4. For each export_i in export^* and corresponding $\mathit{externtype}'_i$ in $\mathit{externtype}'^*$, do:
    a. Let result_i be the pair $(\mathit{export}_i.\mathsf{name}, \mathit{externtype}'_i)$.
5. Return the concatenation of all result_i, in index order.
6. Post-condition: each $\mathit{externtype}'_i$ is valid.

$$\begin{array}{lcl} \mathrm{module\_exports}(m) & = & (ex.\mathsf{name}, \mathit{externtype}')^* \\ & & \quad (\mathrm{if}\ ex^* = m.\mathsf{exports} \wedge \vdash m : \mathit{externtype}^* \to \mathit{externtype}'^*) \end{array}$$

7.1.6 Module Instances

$\mathrm{instance\_export}(\mathit{moduleinst}, \mathit{name}) : \mathit{externval} \mid \mathit{error}$

1. Assert: due to validity of the module instance $\mathit{moduleinst}$, all its export names are different.
2. If there exists an $\mathit{exportinst}_i$ in $\mathit{moduleinst}.\mathsf{exports}$ such that name $\mathit{exportinst}_i.\mathsf{name}$ equals name, then:
   a. Return the external value $\mathit{exportinst}_i.\mathsf{value}$.
3. Else, return error.

$$\begin{array}{lcll} \mathrm{instance\_export}(m, \mathit{name}) & = & m.\mathsf{exports}[i].\mathsf{value} & (\mathrm{if}\ m.\mathsf{exports}[i].\mathsf{name} = \mathit{name}) \\ \mathrm{instance\_export}(m, \mathit{name}) & = & \mathsf{error} & (\mathrm{otherwise}) \end{array}$$

7.1.7 Functions

$\mathrm{func\_alloc}(\mathit{store}, \mathit{functype}, \mathit{hostfunc}) : (\mathit{store}, \mathit{funcaddr})$

1. Pre-condition: $\mathit{functype}$ is valid.
2. Let $\mathit{funcaddr}$ be the result of allocating a host function in store with function type $\mathit{functype}$ and host function code $\mathit{hostfunc}$.
3. Return the new store paired with $\mathit{funcaddr}$.

$$\mathrm{func\_alloc}(S, \mathit{ft}, \mathit{code}) \quad = \quad (S', a) \qquad (\mathrm{if}\ \mathrm{allochostfunc}(S, \mathit{ft}, \mathit{code}) = S', a)$$

Note: This operation assumes that $\mathit{hostfunc}$ satisfies the pre- and post-conditions required for a function instance with type $\mathit{functype}$.

Regular (non-host) function instances can only be created indirectly through module instantiation.

$\mathrm{func\_type}(\mathit{store}, \mathit{funcaddr}) : \mathit{functype}$

1. Return $S.\mathsf{funcs}[a].\mathsf{type}$.
2. Post-condition: the returned function type is valid.

$$\mathrm{func\_type}(S, a) \quad = \quad S.\mathsf{funcs}[a].\mathsf{type}$$

$\mathrm{func\_invoke}(\mathit{store}, \mathit{funcaddr}, \mathit{val}^*) : (\mathit{store}, \mathit{val}^* \mid \mathit{error})$

1. Try invoking the function $\mathit{funcaddr}$ in store with values val^* as arguments:
   a. If it succeeds with values val'^* as results, then let result be val'^*.
   b. Else it has trapped, hence let result be error.
2. Return the new store paired with result.

$$\begin{array}{lcll} \mathrm{func\_invoke}(S, a, v^*) & = & (S', v'^*) & (\text{if } \mathrm{invoke}(S, a, v^*) \hookrightarrow^* S'; F; v'^*) \\ \mathrm{func\_invoke}(S, a, v^*) & = & (S', \mathsf{error}) & (\text{if } \mathrm{invoke}(S, a, v^*) \hookrightarrow^* S'; F; \mathsf{trap}) \end{array}$$

Note: The store may be modified even in case of an error.

7.1.8 Tables

$\mathrm{table\_alloc}(\mathit{store}, \mathit{tabletype}, \mathit{ref}) : (\mathit{store}, \mathit{tableaddr})$

1. Pre-condition: $\mathit{tabletype}$ is valid.
2. Let $\mathit{tableaddr}$ be the result of allocating a table in store with table type $\mathit{tabletype}$ and initialization value ref.
3. Return the new store paired with $\mathit{tableaddr}$.

$$\mathrm{table\_alloc}(S, \mathit{tt}, r) \quad = \quad (S', a) \qquad (\text{if } \mathrm{alloctable}(S, \mathit{tt}, r) = S', a)$$

$\mathrm{table\_type}(\mathit{store}, \mathit{tableaddr}) : \mathit{tabletype}$

1. Return $S.\mathsf{tables}[a].\mathsf{type}$.
2. Post-condition: the returned table type is valid.

$$\mathrm{table\_type}(S, a) \quad = \quad S.\mathsf{tables}[a].\mathsf{type}$$

$\mathrm{table\_read}(\mathit{store}, \mathit{tableaddr}, i : \mathit{u32}) : \mathit{ref} \mid \mathit{error}$

1. Let ti be the table instance $\mathit{store}.\mathsf{tables}[\mathit{tableaddr}]$.
2. If i is larger than or equal to the length of $\mathit{ti}.\mathsf{elem}$, then return error.
3. Else, return the reference value $\mathit{ti}.\mathsf{elem}[i]$.

$$\begin{array}{lcll} \mathrm{table\_read}(S, a, i) & = & r & (\text{if } S.\mathsf{tables}[a].\mathsf{elem}[i] = r) \\ \mathrm{table\_read}(S, a, i) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{table\_write}(\mathit{store}, \mathit{tableaddr}, i : \mathit{u32}, \mathit{ref}) : \mathit{store} \mid \mathit{error}$

1. Let ti be the table instance $\mathit{store}.\mathsf{tables}[\mathit{tableaddr}]$.
2. If i is larger than or equal to the length of $\mathit{ti}.\mathsf{elem}$, then return error.
3. Replace $\mathit{ti}.\mathsf{elem}[i]$ with the reference value ref.
4. Return the updated store.

$$\begin{array}{lcll} \mathrm{table\_write}(S, a, i, r) & = & S' & (\text{if } S' = S \text{ with } \mathsf{tables}[a].\mathsf{elem}[i] = r) \\ \mathrm{table\_write}(S, a, i, r) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{table\_size}(\mathit{store}, \mathit{tableaddr}) : \mathit{u32}$

1. Return the length of $\mathit{store}.\mathsf{tables}[\mathit{tableaddr}].\mathsf{elem}$.

$$\mathrm{table\_size}(S, a) \quad = \quad n \qquad (\text{if } |S.\mathsf{tables}[a].\mathsf{elem}| = n)$$

$\mathrm{table\_grow}(\mathit{store}, \mathit{tableaddr}, n : \mathit{u32}, \mathit{ref}) : \mathit{store} \mid \mathit{error}$

1. Try growing the table instance $\mathit{store}.\mathsf{tables}[\mathit{tableaddr}]$ by n elements with initialization value ref:
    a. If it succeeds, return the updated store.
    b. Else, return error.

$$\begin{array}{lcll} \mathrm{table\_grow}(S, a, n, r) & = & S' & (\text{if } S' = S \mathrel{\mathsf{with}} \mathsf{tables}[a] = \mathrm{growtable}(S.\mathsf{tables}[a], n, r)) \\ \mathrm{table\_grow}(S, a, n, r) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

7.1.9 Memories

$\mathrm{mem\_alloc}(\mathit{store}, \mathit{memtype}) : (\mathit{store}, \mathit{memaddr})$

1. Pre-condition: $\mathit{memtype}$ is valid.
2. Let $\mathit{memaddr}$ be the result of allocating a memory in store with memory type $\mathit{memtype}$.
3. Return the new store paired with $\mathit{memaddr}$.

$$\mathrm{mem\_alloc}(S, \mathit{mt}) \quad = \quad (S', a) \qquad (\text{if } \mathrm{allocmem}(S, \mathit{mt}) = S', a)$$

$\mathrm{mem\_type}(\mathit{store}, \mathit{memaddr}) : \mathit{memtype}$

1. Return $S.\mathsf{mems}[a].\mathsf{type}$.
2. Post-condition: the returned memory type is valid.

$$\mathrm{mem\_type}(S, a) \quad = \quad S.\mathsf{mems}[a].\mathsf{type}$$

$\mathrm{mem\_read}(\mathit{store}, \mathit{memaddr}, i : \mathit{u32}) : \mathit{byte} \mid \mathit{error}$

1. Let mi be the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$.
2. If i is larger than or equal to the length of $\mathit{mi}.\mathsf{data}$, then return error.
3. Else, return the byte $\mathit{mi}.\mathsf{data}[i]$.

$$\begin{array}{lcll} \mathrm{mem\_read}(S, a, i) & = & b & (\text{if } S.\mathsf{mems}[a].\mathsf{data}[i] = b) \\ \mathrm{mem\_read}(S, a, i) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{mem\_write}(\mathit{store}, \mathit{memaddr}, \mathit{i} : \mathit{u32}, \mathit{byte}) : \mathit{store} \mid \mathit{error}$

1. Let mi be the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$.
2. If $\mathit{u32}$ is larger than or equal to the length of $\mathit{mi}.\mathsf{data}$, then return error.
3. Replace $\mathit{mi}.\mathsf{data}[i]$ with byte.
4. Return the updated store.

$$\begin{array}{lclll}
\mathrm{mem\_write}(S, a, i, b) &=& S' && (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{mems}[a].\mathsf{data}[i] = b) \\
\mathrm{mem\_write}(S, a, i, b) &=& \mathsf{error} && (\mathrm{otherwise})
\end{array}$$

$\mathrm{mem\_size}(\mathit{store}, \mathit{memaddr}) : \mathit{u32}$

1. Return the length of $\mathit{store}.\mathsf{mems}[\mathit{memaddr}].\mathsf{data}$ divided by the page size.

$$\mathrm{mem\_size}(S, a) \quad = \quad n \qquad (\mathrm{if}\ |S.\mathsf{mems}[a].\mathsf{data}| = n \cdot 64\,\mathrm{Ki})$$

$\mathrm{mem\_grow}(\mathit{store}, \mathit{memaddr}, n : \mathit{u32}) : \mathit{store} \mid \mathit{error}$

1. Try growing the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$ by n pages:
    a. If it succeeds, return the updated store.
    b. Else, return error.

$$\begin{array}{lclll}
\mathrm{mem\_grow}(S, a, n) &=& S' && (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{mems}[a] = \mathrm{growmem}(S.\mathsf{mems}[a], n)) \\
\mathrm{mem\_grow}(S, a, n) &=& \mathsf{error} && (\mathrm{otherwise})
\end{array}$$

7.1.10 Globals

$\mathrm{global\_alloc}(\mathit{store}, \mathit{globaltype}, \mathit{val}) : (\mathit{store}, \mathit{globaladdr})$

1. Pre-condition: $\mathit{globaltype}$ is valid.
2. Let $\mathit{globaladdr}$ be the result of allocating a global in store with global type $\mathit{globaltype}$ and initialization value val.
3. Return the new store paired with $\mathit{globaladdr}$.

$$\mathrm{global\_alloc}(S, \mathit{gt}, v) \quad = \quad (S', a) \qquad (\mathrm{if}\ \mathrm{allocglobal}(S, \mathit{gt}, v) = S', a)$$

$\mathrm{global\_type}(\mathit{store}, \mathit{globaladdr}) : \mathit{globaltype}$

1. Return $S.\mathsf{globals}[a].\mathsf{type}$.
2. Post-condition: the returned global type is valid.

$$\mathrm{global\_type}(S, a) \quad = \quad S.\mathsf{globals}[a].\mathsf{type}$$

$\mathrm{global\_read}(store, globaladdr) : val$

1. Let gi be the global instance $store.\mathsf{globals}[globaladdr]$.
2. Return the value $gi.\mathsf{value}$.

$$\mathrm{global\_read}(S, a) \quad = \quad v \qquad (\text{if } S.\mathsf{globals}[a].\mathsf{value} = v)$$

$\mathrm{global\_write}(store, globaladdr, val) : store \mid error$

1. Let gi be the global instance $store.\mathsf{globals}[globaladdr]$.
2. Let $mut\ t$ be the structure of the global type $gi.\mathsf{type}$.
3. If mut is not var, then return error.
4. Replace $gi.\mathsf{value}$ with the value val.
5. Return the updated store.

$$\begin{array}{lcll}
\mathrm{global\_write}(S, a, v) & = & S' & (\text{if } S.\mathsf{globals}[a].\mathsf{type} = \mathsf{var}\ t \wedge S' = S \mathrel{\mathsf{with}} \mathsf{globals}[a].\mathsf{value} = v) \\
\mathrm{global\_write}(S, a, v) & = & \mathsf{error} & (\text{otherwise})
\end{array}$$

7.2 Implementation Limitations

Implementations typically impose additional restrictions on a number of aspects of a WebAssembly module or execution. These may stem from:

- physical resource limits,
- constraints imposed by the embedder or its environment,
- limitations of selected implementation strategies.

This section lists allowed limitations. Where restrictions take the form of numeric limits, no minimum requirements are given, nor are the limits assumed to be concrete, fixed numbers. However, it is expected that all implementations have “reasonably” large limits to enable common applications.

Note: A conforming implementation is not allowed to leave out individual *features*. However, designated subsets of WebAssembly may be specified in the future.

7.2.1 Syntactic Limits

Structure

An implementation may impose restrictions on the following dimensions of a module:

- the number of types in a module
- the number of functions in a module, including imports
- the number of tables in a module, including imports
- the number of memories in a module, including imports
- the number of globals in a module, including imports
- the number of element segments in a module
- the number of data segments in a module

- the number of imports to a module
- the number of exports from a module
- the number of parameters in a function type
- the number of results in a function type
- the number of parameters in a block type
- the number of results in a block type
- the number of locals in a function
- the size of a function body
- the size of a structured control instruction
- the number of structured control instructions in a function
- the nesting depth of structured control instructions
- the number of label indices in a `br_table` instruction
- the length of an element segment
- the length of a data segment
- the length of a name
- the range of characters in a name

If the limits of an implementation are exceeded for a given module, then the implementation may reject the validation, compilation, or instantiation of that module with an embedder-specific error.

Note: The last item allows embedders that operate in limited environments without support for Unicode[53] to limit the names of imports and exports to common subsets like ASCII[54].

Binary Format

For a module given in binary format, additional limitations may be imposed on the following dimensions:

- the size of a module
- the size of any section
- the size of an individual function's code
- the number of sections

Text Format

For a module given in text format, additional limitations may be imposed on the following dimensions:

- the size of the source text
- the size of any syntactic element
- the size of an individual token
- the nesting depth of folded instructions
- the length of symbolic identifiers
- the range of literal characters allowed in the source text

---

[53] https://www.unicode.org/versions/latest/

[54] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

7.2.2 Validation

An implementation may defer validation of individual functions until they are first invoked.

If a function turns out to be invalid, then the invocation, and every consecutive call to the same function, results in a trap.

Note: This is to allow implementations to use interpretation or just-in-time compilation for functions. The function must still be fully validated before execution of its body begins.

7.2.3 Execution

Restrictions on the following dimensions may be imposed during execution of a WebAssembly program:

- the number of allocated module instances
- the number of allocated function instances
- the number of allocated table instances
- the number of allocated memory instances
- the number of allocated global instances
- the size of a table instance
- the size of a memory instance
- the number of threads in the global configuration
- the number of suspended threads in the global configuration
- the number of frames on the stack
- the number of labels on the stack
- the number of values on the stack

If the runtime limits of an implementation are exceeded during execution of a computation, then it may terminate that computation and report an embedder-specific error to the invoking code.

Some of the above limits may already be verified during instantiation, in which case an implementation may report exceedance in the same manner as for syntactic limits.

Note: Concrete limits are usually not fixed but may be dependent on specifics, interdependent, vary over time, or depend on other implementation- or embedder-specific situations or events.

7.3 Validation Algorithm

The specification of WebAssembly validation is purely *declarative*. It describes the constraints that must be met by a module or instruction sequence to be valid.

This section sketches the skeleton of a sound and complete *algorithm* for effectively validating code, i.e., sequences of instructions. (Other aspects of validation are straightforward to implement.)

In fact, the algorithm is expressed over the flat sequence of opcodes as occurring in the binary format, and performs only a single pass over it. Consequently, it can be integrated directly into a decoder.

The algorithm is expressed in typed pseudo code whose semantics is intended to be self-explanatory.

7.3.1 Data Structures

Types are representable as an enumeration.

```
type val_type = I32 | I64 | F32 | F64 | V128 | Funcref | Externref

func is_num(t : val_type | Unknown) : bool =
  return t = I32 || t = I64 || t = F32 || t = F64 || t = Unknown

func is_vec(t : val_type | Unknown) : bool =
  return t = V128 || t = Unknown

func is_ref(t : val_type | Unknown) : bool =
  return t = Funcref || t = Externref || t = Unknown
```

The algorithm uses two separate stacks: the *value stack* and the *control stack*. The former tracks the types of operand values on the stack, the latter surrounding structured control instructions and their associated blocks.

```
type val_stack = stack(val_type | Unknown)

type ctrl_stack = stack(ctrl_frame)
type ctrl_frame = {
  opcode : opcode
  start_types : list(val_type)
  end_types : list(val_type)
  height : nat
  unreachable : bool
}
```

For each value, the value stack records its value type, or `Unknown` when the type is not known.

For each entered block, the control stack records a *control frame* with the originating opcode, the types on the top of the operand stack at the start and end of the block (used to check its result as well as branches), the height of the operand stack at the start of the block (used to check that operands do not underflow the current block), and a flag recording whether the remainder of the block is unreachable (used to handle stack-polymorphic typing after branches).

For the purpose of presenting the algorithm, the operand and control stacks are simply maintained as global variables:

```
var vals : val_stack
var ctrls : ctrl_stack
```

However, these variables are not manipulated directly by the main checking function, but through a set of auxiliary functions:

```
func push_val(type : val_type | Unknown) =
  vals.push(type)

func pop_val() : val_type | Unknown =
  if (vals.size() = ctrls[0].height && ctrls[0].unreachable) return Unknown
  error_if(vals.size() = ctrls[0].height)
  return vals.pop()

func pop_val(expect : val_type | Unknown) : val_type | Unknown =
  let actual = pop_val()
  error_if(actual =/= expect && actual =/= Unknown && expect =/= Unknown)
  return actual

```

(continues on next page)

(continued from previous page)

```
func push_vals(types : list(val_type)) = foreach (t in types) push_val(t)
func pop_vals(types : list(val_type)) : list(val_type) =
  var popped := []
  foreach (t in reverse(types)) popped.prepend(pop_val(t))
  return popped
```

Pushing an operand value simply pushes the respective type to the value stack.

Popping an operand value checks that the value stack does not underflow the current block and then removes one type. But first, a special case is handled where the block contains no known values, but has been marked as unreachable. That can occur after an unconditional branch, when the stack is typed polymorphically. In that case, an unknown type is returned.

A second function for popping an operand value takes an expected type, which the actual operand type is checked against. The types may differ in case one of them is Unknown. The function returns the actual type popped from the stack.

Finally, there are accumulative functions for pushing or popping multiple operand types.

Note: The notation `stack[i]` is meant to index the stack from the top, so that, e.g., `ctrls[0]` accesses the element pushed last.

The control stack is likewise manipulated through auxiliary functions:

```
func push_ctrl(opcode : opcode, in : list(val_type), out : list(val_type)) =
  let frame = ctrl_frame(opcode, in, out, vals.size(), false)
  ctrls.push(frame)
  push_vals(in)

func pop_ctrl() : ctrl_frame =
  error_if(ctrls.is_empty())
  let frame = ctrls[0]
  pop_vals(frame.end_types)
  error_if(vals.size() =/= frame.height)
  ctrls.pop()
  return frame

func label_types(frame : ctrl_frame) : list(val_types) =
  return (if frame.opcode == loop then frame.start_types else frame.end_types)

func unreachable() =
  vals.resize(ctrls[0].height)
  ctrls[0].unreachable := true
```

Pushing a control frame takes the types of the label and result values. It allocates a new frame record recording them along with the current height of the operand stack and marks the block as reachable.

Popping a frame first checks that the control stack is not empty. It then verifies that the operand stack contains the right types of values expected at the end of the exited block and pops them off the operand stack. Afterwards, it checks that the stack has shrunk back to its initial height.

The type of the label associated with a control frame is either that of the stack at the start or the end of the frame, determined by the opcode that it originates from.

Finally, the current frame can be marked as unreachable. In that case, all existing operand types are purged from the value stack, in order to allow for the stack-polymorphism logic in `pop_val` to take effect. Because every function has an implicit outermost label that corresponds to an implicit block frame, it is an invariant of the validation algorithm that there always is at least one frame on the control stack when validating an instruction, and hence, *ctrls[0]* is always defined.

Note: Even with the unreachable flag set, consecutive operands are still pushed to and popped from the operand stack. That is necessary to detect invalid examples like (unreachable (i32.const) i64.add). However, a polymorphic stack cannot underflow, but instead generates `Unknown` types as needed.

7.3.2 Validation of Opcode Sequences

The following function shows the validation of a number of representative instructions that manipulate the stack. Other instructions are checked in a similar manner.

Note: Various instructions not shown here will additionally require the presence of a validation context for checking uses of indices. That is an easy addition and therefore omitted from this presentation.

```
func validate(opcode) =
  switch (opcode)
    case (i32.add)
      pop_val(I32)
      pop_val(I32)
      push_val(I32)

    case (drop)
      pop_val()

    case (select)
      pop_val(I32)
      let t1 = pop_val()
      let t2 = pop_val()
      error_if(not ((is_num(t1) && is_num(t2)) || (is_vec(t1) && is_vec(t2))))
      error_if(t1 =/= t2 && t1 =/= Unknown && t2 =/= Unknown)
      push_val(if (t1 = Unknown) t2 else t1)

    case (select t)
      pop_val(I32)
      pop_val(t)
      pop_val(t)
      push_val(t)

    case (unreachable)
      unreachable()

    case (block t1*->t2*)
      pop_vals([t1*])
      push_ctrl(block, [t1*], [t2*])

    case (loop t1*->t2*)
      pop_vals([t1*])
      push_ctrl(loop, [t1*], [t2*])

    case (if t1*->t2*)
      pop_val(I32)
      pop_vals([t1*])
      push_ctrl(if, [t1*], [t2*])

    case (end)
```

(continues on next page)

(continued from previous page)

```
    let frame = pop_ctrl()
    push_vals(frame.end_types)

  case (else)
    let frame = pop_ctrl()
    error_if(frame.opcode =/= if)
    push_ctrl(else, frame.start_types, frame.end_types)

  case (br n)
    error_if(ctrls.size() < n)
    pop_vals(label_types(ctrls[n]))
    unreachable()

  case (br_if n)
    error_if(ctrls.size() < n)
    pop_val(I32)
    pop_vals(label_types(ctrls[n]))
    push_vals(label_types(ctrls[n]))

  case (br_table n* m)
    pop_val(I32)
    error_if(ctrls.size() < m)
    let arity = label_types(ctrls[m]).size()
    foreach (n in n*)
      error_if(ctrls.size() < n)
      error_if(label_types(ctrls[n]).size() =/= arity)
      push_vals(pop_vals(label_types(ctrls[n])))
    pop_vals(label_types(ctrls[m]))
    unreachable()
```

Note: It is an invariant under the current WebAssembly instruction set that an operand of `Unknown` type is never duplicated on the stack. This would change if the language were extended with stack instructions like `dup`. Under such an extension, the above algorithm would need to be refined by replacing the `Unknown` type with proper *type variables* to ensure that all uses are consistent.

7.4 Custom Sections

This appendix defines dedicated custom sections for WebAssembly's binary format. Such sections do not contribute to, or otherwise affect, the WebAssembly semantics, and like any custom section they may be ignored by an implementation. However, they provide useful meta data that implementations can make use of to improve user experience or take compilation hints.

Currently, only one dedicated custom section is defined, the name section.

7.4.1 Name Section

The *name section* is a custom section whose name string is itself ‘`name`’. The name section should appear only once in a module, and only after the data section.

The purpose of this section is to attach printable names to definitions in a module, which e.g. can be used by a debugger or when parts of the module are to be rendered in text form.

Note: All names are represented in Unicode[55] encoded in UTF-8. Names need not be unique.

Subsections

The data of a name section consists of a sequence of *subsections*. Each subsection consists of a

- a one-byte subsection *id*,
- the *u32* *size* of the contents, in bytes,
- the actual *contents*, whose structure is dependent on the subsection id.

$$
\begin{array}{llll}
\mathtt{namesec} & ::= & \mathtt{section}_0(\mathtt{namedata}) & \\
\mathtt{namedata} & ::= & n{:}\mathtt{name} & (\text{if } n = \text{‘}\mathtt{name}\text{’}) \\
& & \mathtt{modulenamesubsec}^? & \\
& & \mathtt{funcnamesubsec}^? & \\
& & \mathtt{localnamesubsec}^? & \\
\mathtt{namesubsection}_N(\mathtt{B}) & ::= & N{:}\mathtt{byte}~~\mathit{size}{:}\mathtt{u32}~~\mathtt{B} & (\text{if } \mathit{size} = ||\mathtt{B}||)
\end{array}
$$

The following subsection ids are used:

Id	Subsection
0	module name
1	function names
2	local names

Each subsection may occur at most once, and in order of increasing id.

Name Maps

A *name map* assigns names to indices in a given index space. It consists of a vector of index/name pairs in order of increasing index value. Each index must be unique, but the assigned names need not be.

$$
\begin{array}{lll}
\mathtt{namemap} & ::= & \mathtt{vec}(\mathtt{nameassoc}) \\
\mathtt{nameassoc} & ::= & \mathtt{idx}~\mathtt{name}
\end{array}
$$

An *indirect name map* assigns names to a two-dimensional index space, where secondary indices are *grouped* by primary indices. It consists of a vector of primary index/name map pairs in order of increasing index value, where each name map in turn maps secondary indices to names. Each primary index must be unique, and likewise each secondary index per individual name map.

$$
\begin{array}{lll}
\mathtt{indirectnamemap} & ::= & \mathtt{vec}(\mathtt{indirectnameassoc}) \\
\mathtt{indirectnameassoc} & ::= & \mathtt{idx}~\mathtt{namemap}
\end{array}
$$

---

[55] https://www.unicode.org/versions/latest/

Module Names

The *module name subsection* has the id 0. It simply consists of a single name that is assigned to the module itself.

$$\mathtt{modulenamesubsec} ::= \mathtt{namesubsection}_0(\mathtt{name})$$

Function Names

The *function name subsection* has the id 1. It consists of a name map assigning function names to function indices.

$$\mathtt{funcnamesubsec} ::= \mathtt{namesubsection}_1(\mathtt{namemap})$$

Local Names

The *local name subsection* has the id 2. It consists of an indirect name map assigning local names to local indices grouped by function indices.

$$\mathtt{localnamesubsec} ::= \mathtt{namesubsection}_2(\mathtt{indirectnamemap})$$

7.5 Soundness

The type system of WebAssembly is *sound*, implying both *type safety* and *memory safety* with respect to the WebAssembly semantics. For example:

- All types declared and derived during validation are respected at run time; e.g., every local or global variable will only contain type-correct values, every instruction will only be applied to operands of the expected type, and every function invocation always evaluates to a result of the right type (if it does not trap or diverge).
- No memory location will be read or written except those explicitly defined by the program, i.e., as a local, a global, an element in a table, or a location within a linear memory.
- There is no undefined behavior, i.e., the execution rules cover all possible cases that can occur in a valid program, and the rules are mutually consistent.

Soundness also is instrumental in ensuring additional properties, most notably, *encapsulation* of function and module scopes: no locals can be accessed outside their own function and no module components can be accessed outside their own module unless they are explicitly exported or imported.

The typing rules defining WebAssembly validation only cover the *static* components of a WebAssembly program. In order to state and prove soundness precisely, the typing rules must be extended to the *dynamic* components of the abstract runtime, that is, the store, configurations, and administrative instructions.[56]

---

[56] The formalization and theorems are derived from the following article: Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman. Bringing the Web up to Speed with WebAssembly[57]. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM 2017.

[57] https://dl.acm.org/citation.cfm?doid=3062341.3062363

7.5.1 Results

Results can be classified by result types as follows.

Results val^*

- For each value val_i in val^*:
  - The value val_i is valid with some value type t_i.
- Let t^* be the concatenation of all t_i.
- Then the result is valid with result type $[t^*]$.

$$\frac{(S \vdash val : t)^*}{S \vdash val^* : [t^*]}$$

Results trap

- The result is valid with result type $[t^*]$, for any sequence t^* of value types.

$$\frac{}{S \vdash \mathsf{trap} : [t^*]}$$

7.5.2 Store Validity

The following typing rules specify when a runtime store S is *valid*. A valid store must consist of function, table, memory, global, and module instances that are themselves valid, relative to S.

To that end, each kind of instance is classified by a respective function, table, memory, or global type. Module instances are classified by *module contexts*, which are regular contexts repurposed as module types describing the index spaces defined by a module.

Store S

- Each function instance $funcinst_i$ in $S.\mathsf{funcs}$ must be valid with some function type $functype_i$.
- Each table instance $tableinst_i$ in $S.\mathsf{tables}$ must be valid with some table type $tabletype_i$.
- Each memory instance $meminst_i$ in $S.\mathsf{mems}$ must be valid with some memory type $memtype_i$.
- Each global instance $globalinst_i$ in $S.\mathsf{globals}$ must be valid with some global type $globaltype_i$.
- Each element instance $eleminst_i$ in $S.\mathsf{elems}$ must be valid with some reference type $reftype_i$.
- Each data instance $datainst_i$ in $S.\mathsf{datas}$ must be valid.
- Then the store is valid.

$$\frac{\begin{array}{c}(S \vdash funcinst : functype)^* \qquad (S \vdash tableinst : tabletype)^* \\ (S \vdash meminst : memtype)^* \qquad (S \vdash globalinst : globaltype)^* \\ (S \vdash eleminst : reftype)^* \qquad (S \vdash datainst~\mathrm{ok})^* \\ S = \{\mathsf{funcs}~funcinst^*, \mathsf{tables}~tableinst^*, \mathsf{mems}~meminst^*, \mathsf{globals}~globalinst^*, \mathsf{elems}~eleminst^*, \mathsf{datas}~datainst^*\}\end{array}}{\vdash S~\mathrm{ok}}$$

Function Instances $\{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\}$

- The function type $\mathit{functype}$ must be valid.
- The module instance $\mathit{moduleinst}$ must be valid with some context C.
- Under context C, the function func must be valid with function type $\mathit{functype}$.
- Then the function instance is valid with function type $\mathit{functype}$.

$$\frac{\vdash \mathit{functype}~\mathrm{ok} \qquad S \vdash \mathit{moduleinst} : C \qquad C \vdash \mathit{func} : \mathit{functype}}{S \vdash \{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\} : \mathit{functype}}$$

Host Function Instances $\{\mathsf{type}~\mathit{functype}, \mathsf{hostcode}~\mathit{hf}\}$

- The function type $\mathit{functype}$ must be valid.
- Then the function instance is valid with function type $\mathit{functype}$.

$$\frac{\vdash [t_1^*] \to [t_2^*]~\mathrm{ok}}{S \vdash \{\mathsf{type}~[t_1^*] \to [t_2^*], \mathsf{hostcode}~\mathit{hf}\} : [t_1^*] \to [t_2^*]}$$

Table Instances $\{\mathsf{type}~(\mathit{limits}~t), \mathsf{elem}~\mathit{ref}^*\}$

- The table type $\mathit{limits}~t$ must be valid.
- The length of ref^* must equal $\mathit{limits}.\mathsf{min}$.
- For each reference ref_i in the table's elements ref^n:
  - The reference ref_i must be valid with reference type t.
- Then the table instance is valid with table type $\mathit{limits}~t$.

$$\frac{\vdash \mathit{limits}~t~\mathrm{ok} \qquad n = \mathit{limits}.\mathsf{min} \qquad (S \vdash \mathit{ref} : t)^n}{S \vdash \{\mathsf{type}~(\mathit{limits}~t), \mathsf{elem}~\mathit{ref}^n\} : \mathit{limits}~t}$$

Memory Instances $\{\mathsf{type}~\mathit{limits}~\mathsf{unshared}, \mathsf{data}~b^*\}$

- The memory type $\mathit{limits}~\mathsf{unshared}$ must be valid.
- The length of b^* must equal $\mathit{limits}.\mathsf{min}$ multiplied by the page size $64\,\mathrm{Ki}$.
- Then the memory instance is valid with memory type $\mathit{limits}~\mathsf{unshared}$.

$$\frac{\vdash \mathit{limits}~\mathsf{unshared}~\mathrm{ok} \qquad n = \mathit{limits}.\mathsf{min} \cdot 64\,\mathrm{Ki}}{S \vdash \{\mathsf{type}~\mathit{limits}~\mathsf{unshared}, \mathsf{data}~b^n\} : \mathit{limits}~\mathsf{unshared}}$$

Memory Instances $\{\mathsf{type}~\mathit{limits}~\mathsf{shared}\}$

- The memory type $\mathit{limits}~\mathsf{shared}$ must be valid.
- Then the memory instance is valid with memory type $\mathit{limits}~\mathsf{shared}$.

$$\frac{\vdash \mathit{limits}~\mathsf{shared}~\mathrm{ok}}{S \vdash \{\mathsf{type}~\mathit{limits}~\mathsf{shared}\} : \mathit{limits}~\mathsf{shared}}$$

Global Instances $\{\mathsf{type}~(\mathit{mut}~t), \mathsf{value}~\mathit{val}\}$

- The global type $\mathit{mut}~t$ must be valid.
- The value val must be valid with value type t.
- Then the global instance is valid with global type $\mathit{mut}~t$.

$$\frac{\vdash \mathit{mut}~t~\mathrm{ok} \qquad S \vdash \mathit{val} : t}{S \vdash \{\mathsf{type}~(\mathit{mut}~t), \mathsf{value}~\mathit{val}\} : \mathit{mut}~t}$$

Element Instances $\{\mathsf{type}~t, \mathsf{elem}~\mathit{ref}^*\}$

- For each reference ref_i in the elements ref^n:
  - The reference ref_i must be valid with reference type t.
- Then the element instance is valid with reference type t.

$$\frac{(S \vdash \mathit{ref} : t)^*}{S \vdash \{\mathsf{type}~t, \mathsf{elem}~\mathit{ref}^*\} : t}$$

Data Instances $\{\mathsf{data}~b^*\}$

- The data instance is valid.

$$\frac{}{S \vdash \{\mathsf{data}~b^*\}~\mathrm{ok}}$$

Export Instances $\{\mathsf{name}~\mathit{name}, \mathsf{value}~\mathit{externval}\}$

- The external value $\mathit{externval}$ must be valid with some external type $\mathit{externtype}$.
- Then the export instance is valid.

$$\frac{S \vdash \mathit{externval} : \mathit{externtype}}{S \vdash \{\mathsf{name}~\mathit{name}, \mathsf{value}~\mathit{externval}\}~\mathrm{ok}}$$

Module Instances $\mathit{moduleinst}$

- Each function type $\mathit{functype}_i$ in $\mathit{moduleinst}.\mathsf{types}$ must be valid.
- For each function address $\mathit{funcaddr}_i$ in $\mathit{moduleinst}.\mathsf{funcaddrs}$, the external value $\mathsf{func}~\mathit{funcaddr}_i$ must be valid with some external type $\mathsf{func}~\mathit{functype}'_i$.
- For each table address $\mathit{tableaddr}_i$ in $\mathit{moduleinst}.\mathsf{tableaddrs}$, the external value $\mathsf{table}~\mathit{tableaddr}_i$ must be valid with some external type $\mathsf{table}~\mathit{tabletype}_i$.
- For each memory address $\mathit{memaddr}_i$ in $\mathit{moduleinst}.\mathsf{memaddrs}$, the external value $\mathsf{mem}~\mathit{memaddr}_i$ must be valid with some external type $\mathsf{mem}~\mathit{memtype}_i$.
- For each global address $\mathit{globaladdr}_i$ in $\mathit{moduleinst}.\mathsf{globaladdrs}$, the external value $\mathsf{global}~\mathit{globaladdr}_i$ must be valid with some external type $\mathsf{global}~\mathit{globaltype}_i$.
- For each element address $\mathit{elemaddr}_i$ in $\mathit{moduleinst}.\mathsf{elemaddrs}$, the element instance $S.\mathsf{elems}[\mathit{elemaddr}_i]$ must be valid with some reference type $\mathit{reftype}_i$.
- For each data address $\mathit{dataaddr}_i$ in $\mathit{moduleinst}.\mathsf{dataaddrs}$, the data instance $S.\mathsf{datas}[\mathit{dataaddr}_i]$ must be valid.

- Each export instance $\mathit{exportinst}_i$ in $\mathit{moduleinst}.\mathsf{exports}$ must be valid.
- For each export instance $\mathit{exportinst}_i$ in $\mathit{moduleinst}.\mathsf{exports}$, the name $\mathit{exportinst}_i.\mathsf{name}$ must be different from any other name occurring in $\mathit{moduleinst}.\mathsf{exports}$.
- Let $\mathit{functype}'^*$ be the concatenation of all $\mathit{functype}'_i$ in order.
- Let $\mathit{tabletype}^*$ be the concatenation of all $\mathit{tabletype}_i$ in order.
- Let $\mathit{memtype}^*$ be the concatenation of all $\mathit{memtype}_i$ in order.
- Let $\mathit{globaltype}^*$ be the concatenation of all $\mathit{globaltype}_i$ in order.
- Let $\mathit{reftype}^*$ be the concatenation of all $\mathit{reftype}_i$ in order.
- Let n be the length of $\mathit{moduleinst}.\mathsf{dataaddrs}$.
- Then the module instance is valid with context $\{\mathsf{types}~\mathit{functype}^*, \mathsf{funcs}~\mathit{functype}'^*, \mathsf{tables}~\mathit{tabletype}^*, \mathsf{mems}~\mathit{memtype}^*, \mathsf{globals}~\mathit{globaltype}^*, \mathsf{elems}~\mathit{reftype}^*, \mathsf{datas}~\mathrm{ok}^n\}$.

$$\frac{\begin{array}{c} (\vdash \mathit{functype}~\mathrm{ok})^* \\ (S \vdash \mathsf{func}~\mathit{funcaddr} : \mathsf{func}~\mathit{functype}')^* \qquad (S \vdash \mathsf{table}~\mathit{tableaddr} : \mathsf{table}~\mathit{tabletype})^* \\ (S; \mathit{act}^? \vdash \mathsf{mem}~\mathit{memaddr} : \mathsf{mem}~\mathit{memtype})^* \qquad (S \vdash \mathsf{global}~\mathit{globaladdr} : \mathsf{global}~\mathit{globaltype})^* \\ (S \vdash S.\mathsf{elems}[\mathit{elemaddr}] : \mathit{reftype})^* \qquad (S \vdash S.\mathsf{datas}[\mathit{dataaddr}]~\mathrm{ok})^n \\ (S \vdash \mathit{exportinst}~\mathrm{ok})^* \qquad (\mathit{exportinst}.\mathsf{name})^*~\mathrm{disjoint} \end{array}}{S \vdash \{\begin{array}[t]{ll} \mathsf{types} & \mathit{functype}^*, \\ \mathsf{funcaddrs} & \mathit{funcaddr}^*, \\ \mathsf{tableaddrs} & \mathit{tableaddr}^*, \\ \mathsf{memaddrs} & \mathit{memaddr}^*, \\ \mathsf{globaladdrs} & \mathit{globaladdr}^*, \\ \mathsf{elemaddrs} & \mathit{elemaddr}^*, \\ \mathsf{dataaddrs} & \mathit{dataaddr}^n, \\ \mathsf{exports} & \mathit{exportinst}^*~\} : \{\begin{array}[t]{ll} \mathsf{types} & \mathit{functype}^*, \\ \mathsf{funcs} & \mathit{functype}'^*, \\ \mathsf{tables} & \mathit{tabletype}^*, \\ \mathsf{mems} & \mathit{memtype}^*, \\ \mathsf{globals} & \mathit{globaltype}^*, \\ \mathsf{elems} & \mathit{reftype}^*, \\ \mathsf{datas} & \mathrm{ok}^n~\} \end{array} \end{array}}$$

7.5.3 Configuration Validity

To relate the WebAssembly type system to its execution semantics, the typing rules for instructions must be extended to configurations $S; P^*$, which relates the store to execution threads.

Threads are classified by their result type, and configurations are classified by a list of their threads' result types. In addition to the store S, threads are typed under a *return type* $\mathit{resulttype}^?$, which controls whether and with which type a return instruction is allowed. This type is absent (ϵ) except for instruction sequences inside an administrative frame instruction.

Finally, frames are classified with *frame contexts*, which extend the module contexts of a frame's associated module instance with the locals that the frame contains.

Configurations $S; P^*$

- The store S must be valid.
- Each thread P_i in P^*, under no allowed return type, must be valid with some result type $[t_i^*]$.
- Let $[t^*]^*$ be the concatenation of all $[t_i^*]$ in order.
- Then the configuration is valid with the result type $[t^*]^*$.

$$\frac{\vdash S\ \mathsf{ok} \qquad (S; \epsilon \vdash P : [t^*])^*}{\vdash S; P^* : [t^*]^*}$$

Threads $F; instr^*$

- Let $resulttype^?$ be the current allowed return type.
- The frame F must be valid with a context C.
- Let C' be the same context as C, but with return set to $resulttype^?$.
- Under context C', the instruction sequence $instr^*$ must be valid with some type $[] \to [t^*]$.
- Then the thread is valid with the result type $[t^*]$.

$$\frac{S \vdash F : C \qquad S; C, \mathsf{return}\ resulttype^? \vdash instr^* : [] \to [t^*]}{S; resulttype^? \vdash F; instr^* : [t^*]}$$

Frames $\{\mathsf{locals}\ val^*, \mathsf{module}\ moduleinst\}$

- The module instance $moduleinst$ must be valid with some module context C.
- Each value val_i in val^* must be valid with some value type t_i.
- Let t^* be the concatenation of all t_i in order.
- Let C' be the same context as C, but with the value types t^* prepended to the locals vector.
- Then the frame is valid with frame context C'.

$$\frac{S \vdash moduleinst : C \qquad (S \vdash val : t)^*}{S \vdash \{\mathsf{locals}\ val^*, \mathsf{module}\ moduleinst\} : (C, \mathsf{locals}\ t^*)}$$

7.5.4 Administrative Instructions

Typing rules for administrative instructions are specified as follows. In addition to the context C, typing of these instructions is defined under a given store S. To that end, all previous typing judgements $C \vdash prop$ are generalized to include the store, as in $S; C \vdash prop$, by implicitly adding S to all rules – S is never modified by the pre-existing rules, but it is accessed in the extra rules for administrative instructions given below.

trap

- The instruction is valid with type $[t_1^*] \to [t_2^*]$, for any sequences of value types t_1^* and t_2^*.

$$\frac{}{S; C \vdash \mathsf{trap} : [t_1^*] \to [t_2^*]}$$

$\mathsf{ref.extern}\ \mathit{externaddr}$

- The instruction is valid with type $[] \to [\mathsf{externref}]$.

$$\frac{}{S; C \vdash \mathsf{ref.extern}\ \mathit{externaddr} : [] \to [\mathsf{externref}]}$$

$\mathsf{ref}\ \mathit{funcaddr}$

- The external function value $\mathsf{func}\ \mathit{funcaddr}$ must be valid with external function type $\mathsf{func}\ \mathit{functype}$.
- Then the instruction is valid with type $[] \to [\mathsf{funcref}]$.

$$\frac{S \vdash \mathsf{func}\ \mathit{funcaddr} : \mathsf{func}\ \mathit{functype}}{S; C \vdash \mathsf{ref}\ \mathit{funcaddr} : [] \to [\mathsf{funcref}]}$$

$\mathsf{invoke}\ \mathit{funcaddr}$

- The external function value $\mathsf{func}\ \mathit{funcaddr}$ must be valid with external function type $\mathsf{func}([t_1^*] \to [t_2^*])$.
- Then the instruction is valid with type $[t_1^*] \to [t_2^*]$.

$$\frac{S \vdash \mathsf{func}\ \mathit{funcaddr} : \mathsf{func}\ [t_1^*] \to [t_2^*]}{S; C \vdash \mathsf{invoke}\ \mathit{funcaddr} : [t_1^*] \to [t_2^*]}$$

$\mathsf{label}_n\{\mathit{instr}_0^*\}\ \mathit{instr}^*\ \mathsf{end}$

- The instruction sequence instr_0^* must be valid with some type $[t_1^n] \to [t_2^*]$.
- Let C' be the same context as C, but with the result type $[t_1^n]$ prepended to the labels vector.
- Under context C', the instruction sequence instr^* must be valid with type $[] \to [t_2^*]$.
- Then the compound instruction is valid with type $[] \to [t_2^*]$.

$$\frac{S; C \vdash \mathit{instr}_0^* : [t_1^n] \to [t_2^*] \qquad S; C, \mathsf{labels}\,[t_1^n] \vdash \mathit{instr}^* : [] \to [t_2^*]}{S; C \vdash \mathsf{label}_n\{\mathit{instr}_0^*\}\ \mathit{instr}^*\ \mathsf{end} : [] \to [t_2^*]}$$

$\mathsf{frame}_n\{F\}\ \mathit{instr}^*\ \mathsf{end}$

- Under the return type $[t^n]$, the thread $F; \mathit{instr}^*$ must be valid with result type $[t^n]$.
- Then the compound instruction is valid with type $[] \to [t^n]$.

$$\frac{S; [t^n] \vdash F; \mathit{instr}^* : [t^n]}{S; C \vdash \mathsf{frame}_n\{F\}\ \mathit{instr}^*\ \mathsf{end} : [] \to [t^n]}$$

$\mathsf{wait}'~\mathit{memaddr}.\mathsf{data}[k]~n$

- The external memory value $\mathsf{mem}~\mathit{memaddr}$ must be valid with external memory type $\mathsf{mem}~(\mathit{limits}~\mathsf{shared})$.
- Then the instruction is valid with type $[] \to []$.

$$\frac{S; \mathit{act}^? \vdash \mathsf{mem}~\mathit{memaddr} : \mathsf{mem}~(\mathit{limits}~\mathsf{shared})}{S; C \vdash \mathsf{wait}'~\mathit{memaddr}.\mathsf{data}[k]~n : [] \to []}$$

$\mathsf{perform}~\mathit{act}^*$

- The instruction is valid with type $[] \to []$.

$$\frac{}{S; C \vdash \mathsf{perform}~\mathit{act}^* : [] \to []}$$

7.5.5 Store Extension

Programs can mutate the store and its contained instances. Any such modification must respect certain invariants, such as not removing allocated instances or changing immutable definitions. While these invariants are inherent to the execution semantics of WebAssembly instructions and modules, host functions do not automatically adhere to them. Consequently, the required invariants must be stated as explicit constraints on the invocation of host functions. Soundness only holds when the embedder ensures these constraints.

The necessary constraints are codified by the notion of store *extension*: a store state S' extends state S, written $S \preceq S'$, when the following rules hold.

Note: Extension does not imply that the new store is valid, which is defined separately above.

Store S

- The length of $S.\mathsf{funcs}$ must not shrink.
- The length of $S.\mathsf{tables}$ must not shrink.
- The length of $S.\mathsf{mems}$ must not shrink.
- The length of $S.\mathsf{globals}$ must not shrink.
- The length of $S.\mathsf{elems}$ must not shrink.
- The length of $S.\mathsf{datas}$ must not shrink.
- For each function instance $\mathit{funcinst}_i$ in the original $S.\mathsf{funcs}$, the new function instance must be an extension of the old.
- For each table instance $\mathit{tableinst}_i$ in the original $S.\mathsf{tables}$, the new table instance must be an extension of the old.
- For each memory instance $\mathit{meminst}_i$ in the original $S.\mathsf{mems}$, the new memory instance must be an extension of the old.
- For each global instance $\mathit{globalinst}_i$ in the original $S.\mathsf{globals}$, the new global instance must be an extension of the old.
- For each element instance $\mathit{eleminst}_i$ in the original $S.\mathsf{elems}$, the new global instance must be an extension of the old.
- For each data instance $\mathit{datainst}_i$ in the original $S.\mathsf{datas}$, the new global instance must be an extension of the old.

$$\frac{\begin{array}{ccc} S_1.\mathsf{funcs} = \mathit{funcinst}_1^* & S_2.\mathsf{funcs} = {\mathit{funcinst}'_1}^*\ \mathit{funcinst}_2^* & (\vdash \mathit{funcinst}_1 \preceq \mathit{funcinst}'_1)^* \\ S_1.\mathsf{tables} = \mathit{tableinst}_1^* & S_2.\mathsf{tables} = {\mathit{tableinst}'_1}^*\ \mathit{tableinst}_2^* & (\vdash \mathit{tableinst}_1 \preceq \mathit{tableinst}'_1)^* \\ S_1.\mathsf{mems} = \mathit{meminst}_1^* & S_2.\mathsf{mems} = {\mathit{meminst}'_1}^*\ \mathit{meminst}_2^* & (\vdash \mathit{meminst}_1 \preceq \mathit{meminst}'_1)^* \\ S_1.\mathsf{globals} = \mathit{globalinst}_1^* & S_2.\mathsf{globals} = {\mathit{globalinst}'_1}^*\ \mathit{globalinst}_2^* & (\vdash \mathit{globalinst}_1 \preceq \mathit{globalinst}'_1)^* \\ S_1.\mathsf{elems} = \mathit{eleminst}_1^* & S_2.\mathsf{elems} = {\mathit{eleminst}'_1}^*\ \mathit{eleminst}_2^* & (\vdash \mathit{eleminst}_1 \preceq \mathit{eleminst}'_1)^* \\ S_1.\mathsf{datas} = \mathit{datainst}_1^* & S_2.\mathsf{datas} = {\mathit{datainst}'_1}^*\ \mathit{datainst}_2^* & (\vdash \mathit{datainst}_1 \preceq \mathit{datainst}'_1)^* \end{array}}{\vdash S_1 \preceq S_2}$$

Function Instance $\mathit{funcinst}$

- A function instance must remain unchanged.

$$\frac{}{\vdash \mathit{funcinst} \preceq \mathit{funcinst}}$$

Table Instance $\mathit{tableinst}$

- The table type $\mathit{tableinst}.\mathsf{type}$ must remain unchanged.
- The length of $\mathit{tableinst}.\mathsf{elem}$ must not shrink.

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}~\mathit{tt}, \mathsf{elem}~(\mathit{fa}_1^?)^{n_1}\} \preceq \{\mathsf{type}~\mathit{tt}, \mathsf{elem}~(\mathit{fa}_2^?)^{n_2}\}}$$

Memory Instance $\mathit{meminst}$

- The memory type $\mathit{meminst}.\mathsf{type}$ must remain unchanged.
- The length of $\mathit{meminst}.\mathsf{data}$ must not shrink.

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}~\mathit{mt}, \mathsf{data}~b_1^{n_1}\} \preceq \{\mathsf{type}~\mathit{mt}, \mathsf{data}~b_2^{n_2}\}}$$

$$\frac{}{\vdash \{\mathsf{type}~\mathit{mt}\} \preceq \{\mathsf{type}~\mathit{mt}\}}$$

Global Instance $\mathit{globalinst}$

- The global type $\mathit{globalinst}.\mathsf{type}$ must remain unchanged.
- Let $\mathit{mut}~t$ be the structure of $\mathit{globalinst}.\mathsf{type}$.
- If mut is const, then the value $\mathit{globalinst}.\mathsf{value}$ must remain unchanged.

$$\frac{\mathit{mut} = \mathsf{var} \vee \mathit{val}_1 = \mathit{val}_2}{\vdash \{\mathsf{type}~(\mathit{mut}~t), \mathsf{value}~\mathit{val}_1\} \preceq \{\mathsf{type}~(\mathit{mut}~t), \mathsf{value}~\mathit{val}_2\}}$$

Element Instance $\mathit{eleminst}$

- The vector $\mathit{eleminst}.\mathsf{elem}$ must either remain unchanged or shrink to length 0.

$$\frac{\mathit{fa}_1^* = \mathit{fa}_2^* \vee \mathit{fa}_2^* = \epsilon}{\vdash \{\mathsf{elem}~\mathit{fa}_1^*\} \preceq \{\mathsf{elem}~\mathit{fa}_2^*\}}$$

Data Instance $\mathit{datainst}$

- The vector $\mathit{datainst}.\mathsf{data}$ must either remain unchanged or shrink to length 0.

$$\frac{b_1^* = b_2^* \vee b_2^* = \epsilon}{\vdash \{\mathsf{data}~b_1^*\} \preceq \{\mathsf{data}~b_2^*\}}$$

7.5.6 Theorems

Given the definition of valid configurations, the standard soundness theorems hold.[58][60]

Theorem (Preservation). If a configuration $S; P^*$ is valid with result type $[t^*]^*$ (i.e., $\vdash S; P^* : [t^*]^*$), and steps to $S'; P'^*$ (i.e., $S; P^* \hookrightarrow^{\mathit{evt}} S'; P'^*$), then $S'; P'^*$ is a valid configuration with the same result type (i.e., $\vdash S'; P'^* : [t^*]^*$). Furthermore, S' is an extension of S (i.e., $\vdash S \preceq S'$).

A *terminal* thread is one whose sequence of instructions is a result. A terminal configuration is a configuration whose threads are all terminal.

Theorem (Progress). If a configuration $S; P^*$ is valid (i.e., $\vdash S; P^* : [t^*]^*$ for some result type $[t^*]^*$), then either it is terminal, or it can step to some configuration $S'; P'^*$ (i.e., $S; P^* \hookrightarrow^{\mathit{evt}} S'; P'^*$).

From Preservation and Progress the soundness of the WebAssembly type system follows directly.

Corollary (Soundness). If a configuration $S; P^*$ is valid (i.e., $\vdash S; P^* : [t^*]^*$ for some result type $[t^*]^*$), then it either diverges or takes a finite number of steps to reach a terminal configuration $S'; P'^*$ (i.e., $S; P^* \hookrightarrow^{*\,(\mathit{evt}^*)} S'; P'^*$) that is valid with the same result type (i.e., $\vdash S'; P'^* : [t^*]^*$) and where S' is an extension of S (i.e., $\vdash S \preceq S'$).

In other words, every thread in a valid configuration either runs forever, traps, or terminates with a result that has the expected type. Consequently, given a valid store, no computation defined by instantiation or invocation of a valid module can "crash" or otherwise (mis)behave in ways not covered by the execution semantics given in this specification.

7.6 Change History

Since the original release 1.0 of the WebAssembly specification, a number of proposals for extensions have been integrated. The following sections provide an overview of what has changed.

---

[58] A machine-verified version of the formalization and soundness proof of the PLDI 2017 paper is described in the following article: Conrad Watt. Mechanising and Verifying the WebAssembly Specification[59]. Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2018). ACM 2018.

[59] https://dl.acm.org/citation.cfm?id=3167082

[60] Machine-verified formalizations and soundness proofs of the semantics from the official specification are described in the following article: Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, Philippa Gardner. Two Mechanisations of WebAssembly 1.0[61]. Proceedings of the 24th International Symposium on Formal Methods (FM 2021). Springer 2021.

[61] https://link.springer.com/chapter/10.1007/978-3-030-90870-6_4

7.6.1 Release XX

Threads and Atomics

Added a new concept of shared memory, denoting a memory which can be accessed by multiple host-created threads, and atomic instructions for synchronising shared memory concurrency[62].

- New memory type: shared
- New memory instructions: atomic.load, atomic.store, atomic.rmw, memory.atomic.notify, memory.atomic.wait, atomic.fence

Note: The proposal does not add a mechanism for creating threads from within WebAssembly. Thread creation must be handled by the host.

7.6.2 Release 2.0

Sign extension instructions

Added new numeric instructions for performing sign extension within integer representations[63].

- New numeric instructions: inn.extend$N$_s

Non-trapping float-to-int conversions

Added new conversion instructions that avoid trapping when converting a floating-point number to an integer[64].

- New numeric instructions: inn.trunc_sat_fmm_sx

Multiple values

Generalized the result type of blocks and functions to allow for multiple values; in addition, introduced the ability to have block parameters[65].

- Function types allow more than one result
- Block types can be arbitrary function types

Reference types

Added funcref and externref as new value types and respective instructions[66].

- New value types: reference types funcref and externref
- New reference instructions: ref.null, ref.func, ref.is_null
- Enrich parametric instruction: select with optional type immediate
- New declarative form of element segment

---

[62] https://github.com/WebAssembly/spec/tree/main/proposals/threads/
[63] https://github.com/WebAssembly/spec/tree/main/proposals/sign-extension-ops/
[64] https://github.com/WebAssembly/spec/tree/main/proposals/nontrapping-float-to-int-conversion/
[65] https://github.com/WebAssembly/spec/tree/main/proposals/multi-value/
[66] https://github.com/WebAssembly/spec/tree/main/proposals/reference-types/

Table instructions

Added instructions to directly access and modify tables[?].

- Table types allow any reference type as element type
- New table instructions: table.get, table.set, table.size, table.grow

Multiple tables

Added the ability to use multiple tables per module[?].

- Modules may define, import, and export multiple tables
- Table instructions take a table index immediate: table.get, table.set, table.size, table.grow, call_indirect
- Element segments take a table index

Bulk memory and table instructions

Added instructions that modify ranges of memory or table entries[?][67]

- New memory instructions: memory.fill, memory.init, memory.copy, data.drop
- New table instructions: table.fill, table.init, table.copy, elem.drop
- New passive form of data segment
- New passive form of element segment
- New data count section in binary format
- Active data and element segments boundaries are no longer checked at compile time but may trap instead

Vector instructions

Added vector type and instructions that manipulate multiple numeric values in parallel (also known as *SIMD*, single instruction multiple data)[68]

- New value type: v128
- New memory instructions: v128.load, v128.load$N$xM_sx, v128.load$N$_zero, v128.load$N$_splat, v128.load$N$_lane, v128.store, v128.store$N$_lane
- New constant vector instruction: v128.const
- New unary vector instructions: v128.not, iNxM.abs, iNxM.neg, i8x16.popcnt, fNxM.abs, fNxM.neg, fNxM.sqrt, fNxM.ceil, fNxM.floor, fNxM.trunc, fNxM.nearest
- New binary vector instructions: v128.and, v128.andnot, v128.or, v128.xor, iNxM.add, iNxM.sub, iNxM.mul, iNxM.add_sat_sx, iNxM.sub_sat_sx, iNxM.min_sx, iNxM.max_sx, iNxM.shl, iNxM.shr_sx, fNxM.add, iNxM.extmul_*half*_iN'xM'_sx, i16x8.q15mulr_sat_s, i32x4.dot_i16x8_s, i16x8.extadd_pairwise_i8x16_sx, i32x4.extadd_pairwise_i16x8_sx, i8x16.avgr_u, i16x8.avgr_u, fNxM.sub, fNxM.mul, fNxM.div, fNxM.min, fNxM.max, fNxM.pmin, fNxM.pmax
- New ternary vector instruction: v128.bitselect
- New test vector instructions: v128.any_true, iNxM.all_true
- New relational vector instructions: iNxM.eq, iNxM.ne, iNxM.lt_sx, iNxM.gt_sx, iNxM.le_sx, iNxM.ge_sx, fNxM.eq, fNxM.ne, fNxM.lt, fNxM.gt, fNxM.le, fNxM.ge

---

[67] https://github.com/WebAssembly/spec/tree/main/proposals/bulk-memory-operations/

[68] https://github.com/WebAssembly/spec/tree/main/proposals/simd/

- New conversion vector instructions:i32x4.trunc_sat_f32x4_sx, i32x4.trunc_sat_f64x2_sx_zero, f32x4.convert_i32x4_sx, f32x4.demote_f64x2_zero, f64x2.convert_low_i32x4_sx, f64x2.promote_low_f32x4
- New lane access vector instructions: iNxM.extract_lane_$sx^?$, iNxM.replace_lane, fNxM.extract_lane, fNxM.replace_lane
- New lane splitting/combining vector instructions: iNxM.extend_$half$_iN'xM'_sx, i8x16.narrow_i16x8_sx, i16x8.narrow_i32x4_sx
- New byte reordering vector instructions: i8x16.shuffle, i8x16.swizzle
- New injection/projection vector instructions: iNxM.splat, fNxM.splat, iNxM.bitmask

Index