WebAssembly Specification

Release 1.0

WebAssembly Community Group

Jul 27, 2017

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 can be employed in other environments as well.

WebAssembly is an open standard developed by a W3C Community Group[1] that includes representatives of all major browser vendors.

This document describes version 1.0 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 It is intended that it will be superseded by new incremental releases with additional features in the future.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.
  - **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.

[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.

- **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.

1.2 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 *value types*. These are integers and IEEE 754 floating-point[4] 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.

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* [5] 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 excecution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

[4] http://ieeexplore.ieee.org/document/4610935/

[5] 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.

Functions Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns a sequence of values as results. [6] 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.

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. 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 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.

[6] In the current version of WebAssembly, there may be at most one result value.

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: $\mathsf{i32}, \mathsf{end}$.
- 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 $\mathit{sym} ::= A_1 \mid \dots \mid A_n$.
- 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.

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.
- $s \ \mathsf{with} \ [i] = A$ denotes the same sequence as s, except that the i-the element is replaced with A.
- $s \ \mathsf{with} \ [i:n] = A^n$ 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 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" $pth ::= ([\dots] \mid .\mathsf{field})^+$:

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

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

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

The latter class 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 $u32$, $u64$, $s32$, $s64$, $i8$, $i16$, $i32$, $i64$. 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 consists of 32 or 64 bit values according to the IEEE 754[7] standard. 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 signifcant 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 $(1 + m \cdot 2^{-M})$, where M is the bit width of m; similarly for subnormals.

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.

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

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

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

[7] http://ieeexplore.ieee.org/document/4610935/

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.

Conventions

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

2.2.4 Names

Names are sequences of scalar Unicode[8] *code points*.

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

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

Convention

- Code points are sometimes used interchangeably with natural numbers $n < 1114112$.

2.3 Types

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

2.3.1 Value Types

Value types classify the individual values that WebAssembly code can compute with and the values that a variable accepts.

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

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

The types $\mathsf{f32}$ and $\mathsf{f64}$ classify 32 and 64 bit floating-point data, respectively. They correspond to *single* and *double* precision floating-point types as defined by the IEEE 754[9] standard

Conventions

- The meta variable t ranges over value types where clear from context.
- The notation $|t|$ denotes the *bit width* of a value type. That is, $|\mathsf{i32}| = |\mathsf{f32}| = 32$ and $|\mathsf{i64}| = |\mathsf{f64}| = 64$.

[8] http://www.unicode.org/versions/latest/

[9] http://ieeexplore.ieee.org/document/4610935/

2.3.2 Result Types

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

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

2.3.3 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*.

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

2.3.4 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.

2.3.5 Memory Types

Memory types classify linear *memories* and their size range.

$$\mathit{memtype} \quad ::= \quad \mathit{limits}$$

The limits constrain the minimum and optionally the maximum size of a memory. The limits are given in units of *page size*.

2.3.6 Table Types

Table types classify *tables* over elements of *element types* within a size range.

$$\begin{array}{lll} \mathit{tabletype} & ::= & \mathit{limits}\ \mathit{elemtype} \\ \mathit{elemtype} & ::= & \mathsf{anyfunc} \end{array}$$

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

The element type anyfunc is the infinite union of all *function types*. A table of that type thus contains references to functions of heterogeneous type.

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

2.3.7 Global Types

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

$$\begin{array}{lll} \mathit{globaltype} & ::= & \mathit{mut}^?\ \mathit{valtype} \\ \mathit{mut} & ::= & \mathsf{const} \mid \mathsf{var} \end{array}$$

2.3.8 External Types

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

$$\mathit{externtype} ::= \mathsf{func}\ \mathit{functype} \mid \mathsf{table}\ \mathit{tabletype} \mid \mathsf{mem}\ \mathit{memtype} \mid \mathsf{global}\ \mathit{globaltype}$$

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:

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

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 (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}{lcl}
\mathit{nn}, \mathit{mm} & ::= & \mathsf{32} \mid \mathsf{64} \\
\mathit{sx} & ::= & \mathsf{u} \mid \mathsf{s} \\
\mathit{instr} & ::= & \mathsf{i}\mathit{nn}\mathsf{.const}\ \mathit{i nn} \mid \mathsf{f}\mathit{nn}\mathsf{.const}\ \mathit{f nn} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{iunop} \mid \mathsf{f}\mathit{nn}\mathsf{.}\mathit{funop} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{ibinop} \mid \mathsf{f}\mathit{nn}\mathsf{.}\mathit{fbinop} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{itestop} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.}\mathit{irelop} \mid \mathsf{f}\mathit{nn}\mathsf{.}\mathit{frelop} \\
& | & \mathsf{i32.wrap/i64} \mid \mathsf{i64.extend\_}\mathit{sx}\mathsf{/i32} \mid \mathsf{i}\mathit{nn}\mathsf{.trunc\_}\mathit{sx}\mathsf{/f}\mathit{mm} \\
& | & \mathsf{f32.trunc/f64} \mid \mathsf{f64.promote/f32} \mid \mathsf{f}\mathit{nn}\mathsf{.convert\_}\mathit{sx}\mathsf{/i}\mathit{mm} \\
& | & \mathsf{i}\mathit{nn}\mathsf{.reinterpret/f}\mathit{nn} \mid \mathsf{f}\mathit{nn}\mathsf{.reinterpret/i}\mathit{nn} \\
\mathit{iunop} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\
\mathit{ibinop} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div\_}\mathit{sx} \mid \mathsf{rem\_}\mathit{sx} \\
& | & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}\mathit{sx} \mid \mathsf{rotl} \mid \mathsf{rotr} \\
\mathit{funop} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\
\mathit{fbinop} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \\
\mathit{itestop} & ::= & \mathsf{eqz} \\
\mathit{irelop} & ::= & \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{frelop} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge}
\end{array}
$$

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

- *Constants*: return a static constant.

- *Unary Operators*: consume one operand and produce one result of the respective type.
- *Binary Operators*: 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 flavours, 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:

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

2.4.2 Parametric Instructions

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

$$
\begin{array}{lll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{drop} \\
& | & \mathsf{select}
\end{array}
$$

The drop operator simply throws away a single operand.

The select operator selects one of its first two operands based on whether its third operand is zero or not.

2.4.3 Variable Instructions

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

$$
\begin{array}{lll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{get\_local}~\mathit{localidx} \\
& | & \mathsf{set\_local}~\mathit{localidx} \\
& | & \mathsf{tee\_local}~\mathit{localidx} \\
& | & \mathsf{get\_global}~\mathit{globalidx} \\
& | & \mathsf{set\_global}~\mathit{globalidx}
\end{array}
$$

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

2.4.4 Memory Instructions

Instructions in this group are concerned with linear *memory*.

$$
\begin{array}{lll}
\mathit{memarg} & ::= & \{\mathsf{offset}~\mathit{u32}, \mathsf{align}~\mathit{u32}\} \\
\mathit{instr} & ::= & \dots \\
& | & \mathsf{i}\mathit{nn}.\mathsf{load}~\mathit{memarg} ~|~ \mathsf{f}\mathit{nn}.\mathsf{load}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}.\mathsf{store}~\mathit{memarg} ~|~ \mathsf{f}\mathit{nn}.\mathsf{store}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}.\mathsf{load8\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}.\mathsf{load16\_}\mathit{sx}~\mathit{memarg} ~|~ \mathsf{i64}.\mathsf{load32\_}\mathit{sx}~\mathit{memarg} \\
& | & \mathsf{i}\mathit{nn}.\mathsf{store8}~\mathit{memarg} ~|~ \mathsf{i}\mathit{nn}.\mathsf{store16}~\mathit{memarg} ~|~ \mathsf{i64}.\mathsf{store32}~\mathit{memarg} \\
& | & \mathsf{current\_memory} \\
& | & \mathsf{grow\_memory}
\end{array}
$$

Memory is accessed with load and store instructions for the different *value types*. They all take a *memory immediate* *memarg* that contains an address *offset* and an *alignment* hint. 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.

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[10] 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 version of WebAssembly might provide memory instructions with 64 bit address ranges.

The current_memory instruction returns the current size of a memory. The grow_memory 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*.

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

2.4.5 Control Instructions

Instructions in this group affect the flow of control.

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

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.

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

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 $[] \to [\mathit{valtype}^?]$.

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, 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*. Since tables may contain function elements of heterogeneous type anyfunc, the callee is dynamically checked against the *function type* indexed by the instruction's immediate, and the call aborted with a *trap* if it does not match.

Note: In the current version of WebAssembly, call_indirect implicitly operates on *table index* 0. This restriction may be lifted in future versions.

2.4.6 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.

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

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 logic in the form of *data* and *element* segments or a *start function*.

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

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.

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

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.

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

Label indices reference *structured control instructions* inside an instruction sequence.

Conventions

- The meta variable l ranges over label indices.
- The meta variables x, y ranges over indices in any of the other index spaces.

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:

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

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.

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 table *element 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.

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

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):

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

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. The elem component of a module defines a vector of *element segments* that initialize a subrange of a table at a given offset from a static *vector* of elements.

$$\mathit{elem} \quad ::= \quad \{\mathsf{table}\ \mathit{tableidx}, \mathsf{offset}\ \mathit{expr}, \mathsf{init}\ \mathit{vec}(\mathit{funcidx})\}$$

The offset is given by a *constant expression*.

Note: In the current version of WebAssembly, at most one table is allowed in a module. Consequently, the only valid $\mathit{tableidx}$ is 0.

2.5.8 Data Segments

The initial contents of a *memory* are zero bytes. The data component of a module defines a vector of *data segments* that initialize a range of memory at a given offset with a static *vector* of *bytes*.

$$\mathit{data} \quad ::= \quad \{\mathsf{data}\ \mathit{memidx}, \mathsf{offset}\ \mathit{expr}, \mathsf{init}\ \mathit{vec}(\mathit{byte})\}$$

The offset is given by a *constant expression*.

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

$$\mathit{start} \quad ::= \quad \{\mathsf{func}\ \mathit{funcidx}\}$$

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*.

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

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

Note: In the current version of WebAssembly, only *immutable* globals may be exported.

Conventions

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

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

2.5.11 Imports

The $\mathsf{imports}$ component of a module defines a set of *imports* that are required for *instantiation*.

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

Each import is identified by a two-level *name* space, consisting of a module name and a unique 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: In the current version of WebAssembly, only *immutable* globals may be imported.

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.

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. 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.
- *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 a result type.

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.

It is convenient to define contexts as *records* C with abstract syntax:

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

Note: The fields of a context are not defined as *vectors*, since their lengths are not bounded by the maximum vector size.

In addition to field access $C.\mathsf{field}$ 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: This notation is defined to *prepend* not *append*. It is only used in situations where the original $C.\mathsf{field}$ is either empty or field is labels. In the latter case adding to the front is desired because the *label index* space is indexed relatively, that is, in reverse order of addition.

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. [11]

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.

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

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

$$\frac{premise_1 \qquad premise_2 \qquad \dots \qquad premise_n}{conclusion}$$

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:

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

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

An instruction like get_local can be typed as follows:

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

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:

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

A block instruction is only valid when the instruction sequence in its body is. Moreover, the result type must match the block's annotation *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.

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 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.

typeidx

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

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

$[valtype^?]$

- The block type is valid as *function type* $[] \to [valtype^?]$.

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

3.2.2 Limits

Limits must have menaingful bounds.

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

- If the maximum $m^?$ is not empty, then its value must not be smaller than n.
- Then the limit is valid.

$$\frac{(n \leq m)^?}{\vdash \{\mathsf{min}\ n, \mathsf{max}\ m^?\}\ \mathrm{ok}}$$

3.3 Instructions

Instructions are classified by *function types* $[t_1^*] \to [t_2^*]$ that describe how they manipulate the *operand stack*. The types describe the required input stack with argument values of types t_1^* that an instruction pops off and the provided output stack with result values of types t_2^* that it pushes back.

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

Typing extends to *instruction sequences* $instr^*$. Such a sequence has a *function types* $[t_1^*] \to [t_2^*]$ 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 therefor 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) *function type* $[t_1^*] \to [t_2^*]$ of the instruction is unconstrained. That is the case for all *control instructions* that perform an *unconditional control transfer*, such as $\mathsf{unreachable}$, br, $\mathsf{br\_table}$, 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 $[t\ t\ \mathsf{i32}] \to [t]$, for any possible *value 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 $\mathsf{i32}$ or $\mathsf{f64}$, respectively.

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

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

is valid by assuming type $[] \to [\mathsf{i32}~\mathsf{i32}]$ 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.

3.3.1 Numeric Instructions

$t.\mathsf{const}~c$

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

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

$t.unop$

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

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

$t.binop$

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

$$\frac{}{C \vdash t.binop : [t~t] \to [t]}$$

$t.testop$

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

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

$t.relop$

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

$$\frac{}{C \vdash t.relop : [t~t] \to [\mathsf{i32}]}$$

$t_2.cvtop/t_1$

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

$$\frac{}{C \vdash t_2.cvtop/t_1 : [t_1] \to [t_2]}$$

3.3.2 Parametric Instructions

drop

- The instruction is valid with type $[t] \to []$, for any *value type* t.

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

select

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

$$\frac{}{C \vdash \mathsf{select} : [t\ t\ \mathsf{i32}] \to [t]}$$

Note: Both drop and select are *value-polymorphic* instructions.

3.3.3 Variable Instructions

get_local 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 $[] \to [t]$.

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

set_local 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 $[t] \to []$.

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

tee_local 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 $[t] \to [t]$.

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

$\mathsf{get\_global}\ x$

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

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

$\mathsf{set\_global}\ x$

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

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

3.3.4 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 *width* of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}] \to [t]$.

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

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

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

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

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

- 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 *width* of t divided by 8.
- Then the instruction is valid with type $[\mathsf{i32}\ t] \to []$.

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

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

- 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 $[\mathsf{i32}\ t] \to []$.

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

current_memory

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

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

grow_memory

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

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

3.3.5 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 *value types* t_1^* and t_2^*.

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

Note: The unreachable instruction is *stack-polymorphic*.

$\mathsf{block}\ blocktype\ instr^*\ \mathsf{end}$

- 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 $instr^*$ must be *valid* with type $[t_1^*] \to [t_2^*]$.
- Then the compound instruction is valid with type $[t_1^*] \to [t_2^*]$.

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

loop $\mathit{blocktype}$ 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 empty *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 $[t_1^*] \to [t_2^*]$.

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

if $\mathit{blocktype}$ instr_1^* else instr_2^* 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 empty *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 $[t_1^*\ \mathsf{i32}] \to [t_2^*]$.

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

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 *value 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 br instruction is *stack-polymorphic*.

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 $[t^*\ \mathsf{i32}] \to [t^*]$.

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

br_table $l^* \ l_N$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t^*]$ be the *result type* $C.\mathsf{labels}[l_N]$.
- For all l_i in l^*, the label $C.\mathsf{labels}[l_i]$ must be defined in the context.
- For all l_i in l^*, $C.\mathsf{labels}[l_i]$ must be t^*.
- Then the instruction is valid with type $[t_1^* \ t^* \ \mathsf{i32}] \rightarrow [t_2^*]$, for any sequences of *value types* t_1^* and t_2^*.

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

Note: The br_table instruction is *stack-polymorphic*.

return

- The return type $C.\mathsf{return}$ must not be empty 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 *value 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 empty (ϵ) when validating an *expression* that is not a function body. This differs from it being set to the empty result type ($[]$), 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

- The table $C.\mathsf{tables}[0]$ must be defined in the context.
- Let *limits elemtype* be the *table type* $C.\mathsf{tables}[0]$.
- The *element type elemtype* must be anyfunc.
- The type $C.\mathsf{types}[x]$ must be defined in the context.
- Then the instruction is valid with type $C.\mathsf{types}[x]$.

$$\frac{C.\mathsf{tables}[0] = \mathit{limits}\ \mathsf{anyfunc} \qquad C.\mathsf{types}[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \mathsf{call\_indirect}\ x : [t_1^*] \rightarrow [t_2^*]}$$

3.3.6 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

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

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

Non-empty Instruction Sequence: $instr^*\ instr_N$

- The instruction sequence $instr^*$ must be valid with type $[t_1^*] \rightarrow [t_2^*]$, for some sequences of *value types* t_1^* and t_2^*.
- The instruction $instr_N$ must be valid with type $[t^*] \rightarrow [t_3^*]$, for some sequences of *value types* t^* and t_3^*.
- There must be a sequence of *value types* t_0^*, such that $t_2^* = t_0^*\ t^*$.
- Then the combined instruction sequence is valid with type $[t_1^*] \rightarrow [t_0^*\ t_3^*]$.

$$\frac{C \vdash instr^* : [t_1^*] \rightarrow [t_0^*\ t^*] \qquad C \vdash instr_N : [t^*] \rightarrow [t_3^*]}{C \vdash instr^*\ instr_N : [t_1^*] \rightarrow [t_0^*\ t_3^*]}$$

3.3.7 Expressions

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

$instr^*$ end

- The instruction sequence $instr^*$ must be *valid* with type $[] \rightarrow [t^*]$, for some *result type* $[t^*]$.
- Then the expression is valid with *result type* $[t^*]$.

$$\frac{C \vdash instr^* : [] \rightarrow [t^*]}{C \vdash instr^*\ \mathsf{end} : [t^*]}$$

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 $\mathsf{get\_global}\ x$, in which case $C.\mathsf{globals}[x]$ must be a *global type* of the form $\mathsf{const}\ t$.

$$\frac{(C \vdash instr\ \mathrm{const})^*}{C \vdash instr\ \mathsf{end}\ \mathrm{const}} \qquad \frac{}{C \vdash t.\mathsf{const}\ c\ \mathrm{const}} \qquad \frac{C.\mathsf{globals}[x] = \mathsf{const}\ t}{C \vdash \mathsf{get\_global}\ x\ \mathrm{const}}$$

Note: 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 $[t_1^*] \to [t_2^*]$.

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

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \to [t_2^*]$ 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 $[t_1^*] \to [t_2^*]$.

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

3.4.2 Tables

Tables *table* are classified by *table types* of the form $limits\ elemtype$.

$\{\mathsf{type}\ tabletype\}$

- Let $limits\ elemtype$ be the *table types* $tabletype$.
- The limits $limits$ must be *valid*.
- Then the table definition is valid with type $tabletype$.

$$\frac{\vdash limits\ \mathrm{ok}}{C \vdash \{\mathsf{type}\ limits\ elemtype\} : limits\ elemtype}$$

3.4.3 Memories

Memories *mem* are classified by *memory types* of the form $limits$.

$\{\mathsf{type}\ memtype\}$

- Let $limits$ be the *memory types* $memtype$.
- The limits $limits$ must be *valid*.
- Then the memory definition is valid with type $memtype$.

$$\frac{\vdash limits\ \mathrm{ok}}{C \vdash \{\mathsf{type}\ limits\} : limits\ elemtype}$$

3.4.4 Globals

Globals $global$ are classified by *global types* of the form $mut\ t$.

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

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

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

3.4.5 Element Segments

Element segments $elem$ are not classified by a type.

$\{\mathsf{table}\ x, \mathsf{offset}\ expr, \mathsf{init}\ y^*\}$

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $limits\ elemtype$ be the *table type* $C.\mathsf{tables}[x]$.
- The *element type* $elemtype$ must be $\mathsf{anyfunc}$.
- The expression $expr$ must be *valid* with *result type* $[\mathsf{i32}]$.
- The expression $expr$ must be *constant*.
- For each y_i in y^*, the function $C.\mathsf{funcs}[y]$ must be defined in the context.
- Then the element segment is valid.

$$\frac{C.\mathsf{tables}[x] = limits\ \mathsf{anyfunc} \qquad C \vdash expr : [\mathsf{i32}] \qquad C \vdash expr\ \mathrm{const} \qquad (C.\mathsf{funcs}[y] = functype)^*}{C \vdash \{\mathsf{table}\ x, \mathsf{offset}\ expr, \mathsf{init}\ y^*\}\ \mathrm{ok}}$$

3.4.6 Data Segments

Data segments $data$ are not classified by any type.

$\{\mathsf{data}\ x, \mathsf{offset}\ \mathit{expr}, \mathsf{init}\ b^*\}$

- 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 segment is valid.

$$\frac{C.\mathsf{mems}[x] = \mathit{limits} \qquad C \vdash \mathit{expr} : [\mathsf{i32}] \qquad C \vdash \mathit{expr}\ \mathrm{const}}{C \vdash \{\mathsf{data}\ x, \mathsf{offset}\ \mathit{expr}, \mathsf{init}\ b^*\}\ \mathrm{ok}}$$

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 are classified by their export name. Export descriptions $\mathit{exportdesc}$ are not classified by any type.

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

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

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

$\mathsf{func}\ x$

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

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

$\mathsf{table}\ x$

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

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

mem x

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

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

global x

- The global $C.\mathsf{globals}[x]$ must be defined in the context.
- Let $\mathit{mut}\ t$ be the *global type* $C.\mathsf{globals}[x]$.
- The mutability mut must be const.
- Then the export description is valid.

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

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 $[t_1^*] \to [t_2^*]$ be the *function type* $C.\mathsf{types}[x]$.
- Then the import description is valid with type $\mathsf{func}\ [t_1^*] \to [t_2^*]$.

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

table $\mathit{limits}\ \mathit{elemtype}$

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

$$\frac{\vdash \mathit{limits}\ \mathrm{ok}}{C \vdash \mathsf{table}\ \mathit{limits}\ \mathit{elemtype} : \mathsf{table}\ \mathit{limits}\ \mathit{elemtype}}$$

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

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

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

$\mathsf{global}~\mathit{mut}~t$

- The mutability mut must be const.
- Then the import description is valid with type $\mathsf{global}~t$.

$$\frac{}{C \vdash \mathsf{global}~\mathsf{const}~t : \mathsf{global}~\mathsf{const}~t}$$

3.4.10 Modules

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 $\mathrm{funcs}(\mathit{externtype}_i^*)$ concatenated with $\mathit{functype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{functype}_i^*$ as determined below,
  - $C.\mathsf{tables}$ is $\mathrm{tables}(\mathit{externtype}_i^*)$ concatenated with $\mathit{tabletype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{tabletype}_i^*$ as determined below,
  - $C.\mathsf{mems}$ is $\mathrm{mems}(\mathit{externtype}_i^*)$ concatenated with $\mathit{memtype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{memtype}_i^*$ as determined below,
  - $C.\mathsf{globals}$ is $\mathrm{globals}(\mathit{externtype}_i^*)$ concatenated with $\mathit{globaltype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{globaltype}_i^*$ as determined below.
  - $C.\mathsf{locals}$ is empty,
  - $C.\mathsf{labels}$ is empty.
  - $C.\mathsf{return}$ is empty.
- Let C' be the *context* where $C'.\mathsf{globals}$ is the sequence $\mathrm{globals}(\mathit{externtype}_i^*)$ and all other fields are empty.
- Under the context C:
  - For each $\mathit{functype}_i$ in $\mathit{module}.\mathsf{types}$, the *function type* $\mathit{functype}_i$ must be valid.
  - For each func_i in $\mathit{module}.\mathsf{funcs}$, the definition func_i must be *valid* with a *function type* $\mathit{functype}_i$.
  - For each table_i in $\mathit{module}.\mathsf{tables}$, the definition table_i must be *valid* with a *table type* $\mathit{tabletype}_i$.
  - For each mem_i in $\mathit{module}.\mathsf{mems}$, the definition mem_i must be *valid* with a *memory type* $\mathit{memtype}_i$.
  - For each global_i in $\mathit{module}.\mathsf{globals}$:
    * Under the context C', the definition global_i must be *valid* with a *global type* $\mathit{globaltype}_i$.
  - For each elem_i in $\mathit{module}.\mathsf{elem}$, the segment elem_i must be *valid*.
  - For each data_i in $\mathit{module}.\mathsf{data}$, the segment elem_i must be *valid*.
  - If $\mathit{module}.\mathsf{start}$ is non-empty, then $\mathit{module}.\mathsf{start}$ must be *valid*.

– For each $import_i$ in $module$.imports, the segment $import_i$ must be *valid* with an *external type* $externtype_i$.

– For each $export_i$ in $module$.exports, the segment $import_i$ must be *valid* with a *name* $name_i$.

- The length of C.tables must not be larger than 1.
- The length of C.mems must not be larger than 1.
- All export names $name_i$ must be different.
- Let $externtype^*$ be the concatenation of *external types* $externtype_i$ of the imports, in index order.
- Then the module is valid with *external types* $externtype^*$.

$$\frac{\begin{array}{c}(\vdash functype \text{ ok})^* \quad (C \vdash func : ft)^* \quad (C \vdash table : tt)^* \quad (C \vdash mem : mt)^* \quad (C' \vdash global : gt)^* \\ (C \vdash elem \text{ ok})^* \quad (C \vdash data \text{ ok})^* \quad (C \vdash start \text{ ok})^? \quad (C \vdash import : it)^* \quad (C \vdash export : name)^* \\ ift^* = \mathrm{funcs}(it^*) \qquad itt^* = \mathrm{tables}(it^*) \qquad imt^* = \mathrm{mems}(it^*) \qquad igt^* = \mathrm{globals}(it^*) \\ C = \{\mathsf{types}\ functype^*, \mathsf{funcs}\ ift^*\ ft^*, \mathsf{tables}\ itt^*\ tt^*, \mathsf{mems}\ imt^*\ mt^*, \mathsf{globals}\ igt^*\ gt^*\} \\ C' = \{\mathsf{globals}\ igt^*\} \qquad |C.\mathsf{tables}| \leq 1 \qquad |C.\mathsf{mems}| \leq 1 \qquad name^* \text{ disjoint}\end{array}}{\begin{array}{c}\vdash \{\mathsf{types}\ functype^*, \mathsf{funcs}\ func^*, \mathsf{tables}\ table^*, \mathsf{mems}\ mem^*, \mathsf{globals}\ global^*, \\ \mathsf{elem}\ elem^*, \mathsf{data}\ data^*, \mathsf{start}\ start^?, \mathsf{imports}\ import^*, \mathsf{exports}\ export^*\} : it^*\end{array}}$$

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. The effect of defining the limited context C' for validating the module's globals is that their initialization expressions can only access imported globals and nothing else.

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

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.

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.
- Both the store and the current frame are mutated by *replacing* some of its 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. [13]

The formal execution rules use a standard approach for specifying operational semantics, rendering them into *reduction rules*. Every rule has the following general form:

$$\mathit{configuration} \quad \hookrightarrow \quad \mathit{configuration}$$

A *configuration* is a syntactic description of a program state. Each rule specifies one *step* of execution. 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 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.

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 value 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*.

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 $x_6 = x_1 \cdot (-x_2 + x_3)$.

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

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 the four basic *value types*: *integers* and *floating-point data* of 32 or 64 bit width each, respectively.

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* producing them:

$$\begin{array}{lll} \mathit{val} & ::= & \mathsf{i32.const}\ \mathit{i32} \\ & | & \mathsf{i64.const}\ \mathit{i64} \\ & | & \mathsf{f32.const}\ \mathit{f32} \\ & | & \mathsf{f64.const}\ \mathit{f64} \end{array}$$

4.2.2 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* that have been *allocated* during the life time of the abstract machine. [15]

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

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

Convention

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

4.2.3 Addresses

Function instances, *table instances*, *memory instances*, and *global instances* in the *store* are referenced with abstract *addresses*. These are simply indices into the respective store component.

$$\begin{array}{lll} \mathit{addr} & ::= & 0 \mid 1 \mid 2 \mid \dots \\ \mathit{funcaddr} & ::= & \mathit{addr} \\ \mathit{tableaddr} & ::= & \mathit{addr} \\ \mathit{memaddr} & ::= & \mathit{addr} \\ \mathit{globaladdr} & ::= & \mathit{addr} \end{array}$$

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*).

[15] 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.

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.

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

4.2.4 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.

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

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*.

4.2.5 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.

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*.

4.2.6 Table Instances

A *table instance* is the runtime representation of a *table*. It holds a vector of *function elements* and an optional maximum size, if one was specified in the *table type* at the table's definition site.

Each function element is either empty, representing an uninitialized table entry, or a *function address*. Function elements can be mutated through the execution of an *element segment* or by external means provided by the *embedder*.

$$\begin{array}{lll} \mathit{tableinst} & ::= & \{\mathsf{elem}~\mathit{vec}(\mathit{funcelem}), \mathsf{max}~\mathit{u32}^?\} \\ \mathit{funcelem} & ::= & \mathit{funcaddr}^? \end{array}$$

It is an invariant of the semantics that the length of the element vector never exceeds the maximum size, if present.

Note: Other table elements may be added in future versions of WebAssembly.

4.2.7 Memory Instances

A *memory instance* is the runtime representation of a linear *memory*. It holds a vector of *bytes* and an optional maximum size, if one was specified at the definition site of the memory.

$$meminst \quad ::= \quad \{\mathsf{data}\ vec(byte), \mathsf{max}\ u32^?\}$$

The length of the vector always is a multiple of the WebAssembly *page size*, which is defined to be the constant 65536 – abbreviated $64\,\mathrm{Ki}$. Like in a *memory type*, the maximum size in a memory instance is given in units of this page size.

The bytes can be mutated through *memory instructions*, the execution of a *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, if present.

4.2.8 Global Instances

A *global instance* is the runtime representation of a *global* variable. It holds an individual *value* and a flag indicating whether it is mutable.

$$globalinst \quad ::= \quad \{\mathsf{value}\ val, \mathsf{mut}\ mut\}$$

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

4.2.9 Export Instances

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

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

4.2.10 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*.

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

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:

- $\mathrm{funcs}(externval^*) = [funcaddr \mid (\mathsf{func}\ funcaddr) \in externval^*]$

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

4.2.11 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 WebAssebmly 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:

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

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.

Frames

Activation frames carry the return arity 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*:

$$\begin{array}{lll} activation & ::= & \mathsf{frame}_n\{frame\} \\ frame & ::= & \{\mathsf{locals}\ val^*, \mathsf{module}\ moduleinst\} \end{array}$$

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:

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

4.2.12 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*:

$$\begin{array}{llcl} \mathit{instr} &::=& \dots \\ &|& \mathsf{trap} \\ &|& \mathsf{invoke}\ \mathit{funcaddr} \\ &|& \mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{instr}^*\ \mathsf{end} \\ &|& \mathsf{frame}_n\{\mathit{frame}\}\ \mathit{instr}^*\ \mathsf{end} \end{array}$$

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 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.

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*:

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

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 the hole:

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

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.

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:

$$E \quad ::= \quad [\_] \mid val^*\ E\ instr^* \mid \mathsf{label}_n\{instr^*\}\ E\ \mathsf{end} \mid \mathsf{frame}_n\{frame\}\ E\ \mathsf{end}$$

$$\begin{array}{lcll} S; F; E[instr^*] & \hookrightarrow & S'; F'; E[instr'^*] & (\text{if } S; F; instr^* \hookrightarrow S'; F'; instr'^*) \\ S; F; E[\mathsf{trap}] & \hookrightarrow & S; F; \mathsf{trap} & (\text{if } E \neq [\_]) \end{array}$$

Note: For example, the following instruction sequence,

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

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.

Module Instructions

Module *instantiation* is a complex operation. It is hence expressed in terms of reduction into smaller steps expressed by a sequence of administrative *module instructions* that are a superset of ordinary instructions and defined as follow.

$$\begin{array}{llll} moduleinstr & ::= & instr \\ & | & \mathsf{instantiate}\ module\ externval^* \\ & | & \mathsf{init\_table}\ tableaddr\ u32\ moduleinst\ funcidx^* \\ & | & \mathsf{init\_mem}\ memaddr\ u32\ byte^* \\ & | & \mathsf{init\_global}\ globaladdr\ val \\ & | & moduleinst \end{array}$$

The $\mathsf{instantiate}$ instruction expresses instantiation of a *module* itself, requiring a sequence of *external values* for the expected imports. It reduces into a sequence of initialization instructions for *tables*, *memories* and *globals*, and a possible *invocation* of the *start function*. The final instruction returns the newly created and initialized *module instance*.

Note: The reason for splitting instantiation into individual reduction steps is to provide a semantics that is compatible with future extensions like threads.

Unlike the administrative instructions above, module instructions *embed* ordinary instructions *instr* instead of extending them. Consequently, they can only occur at the top-level.

Evaluation contexts and additional structural reduction rules for module instructions are defined as follows:

$$M \quad ::= \quad E \ \mathit{moduleinstr}^* \ \mathit{moduleinst}$$

$$\begin{array}{lcll} S; M[\mathit{moduleinstr}] & \hookrightarrow & S'; M[\mathit{moduleinstr}'^*] & (\text{if } S; \mathit{moduleinstr} \hookrightarrow S'; \mathit{moduleinstr}'^*) \\ S; M[\mathsf{trap}] & \hookrightarrow & S; \mathsf{trap} & (\text{if } M \neq [\_]) \end{array}$$

Reduction terminates when the sequence has been reduced to a *moduleinst* or when a trap occurred.

Note: A trap may either arise from invocation of a *start function* or indicate failure of the instantiate instruction itself.

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:

$$\begin{array}{lcl} \mathrm{fcopysign}_N(+p_1, +p_2) & = & +p_1 \\ \mathrm{fcopysign}_N(-p_1, -p_2) & = & -p_1 \\ \mathrm{fcopysign}_N(+p_1, -p_2) & = & -p_1 \\ \mathrm{fcopysign}_N(-p_1, +p_2) & = & +p_1 \end{array}$$

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:

$$\mathrm{trunc}(\pm q) \quad = \quad \pm i \qquad (\text{if } i \in \mathbb{N} \wedge q - 1 < i \leq q)$$

4.3.1 Representations

Numbers have an underlying binary representation as a sequence of bits:

$$\begin{array}{lcl} \mathrm{bits}_{\mathsf{i}N}(i) & = & \mathrm{ibits}_N(i) \\ \mathrm{bits}_{\mathsf{f}N}(z) & = & \mathrm{fbits}_N(z) \end{array}$$

Each of these functions is a bijection, hence they are invertible.

Integers

Integers are represented as base two unsigned numbers:

$$\mathrm{ibits}_N(i) \quad = \quad d_{N-1}\ \dots\ d_0 \qquad (i = 2^{N-1} \cdot d_{N-1} + \dots + 2^0 \cdot d_0)$$

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[16]:

$$\begin{array}{lcl} \mathrm{fbits}_N(\pm(1 + m \cdot 2^{-M}) \cdot 2^e) & = & \mathrm{fsign}(\pm)\ \mathrm{ibits}_E(e + \mathrm{fbias}_N)\ \mathrm{ibits}_M(m) \\ \mathrm{fbits}_N(\pm(0 + m \cdot 2^{-M}) \cdot 2^e) & = & \mathrm{fsign}(\pm)\ (0)^E\ \mathrm{ibits}_M(m) \\ \mathrm{fbits}_N(\pm\infty) & = & \mathrm{fsign}(\pm)\ (1)^E\ (0)^M \\ \mathrm{fbits}_N(\pm\mathsf{nan}(n)) & = & \mathrm{fsign}(\pm)\ (1)^E\ \mathrm{ibits}_M(n) \\ \mathrm{fbias}_N & = & 2^{E-1} - 1 \\ \mathrm{fsign}(+) & = & 0 \\ \mathrm{fsign}(-) & = & 1 \end{array}$$

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

Storage

When a number is stored into *memory*, it is converted into a sequence of *bytes* in little endian[17] byte order:

$$\begin{array}{lcl} \mathrm{bytes}_t(i) & = & \mathrm{littleendian}(\mathrm{bits}_t(i)) \\ \mathrm{littleendian}(\epsilon) & = & \epsilon \\ \mathrm{littleendian}(d_1^8\ d_2^{N-8}) & = & \mathrm{ibits}_8^{-1}(d_1^8)\ \mathrm{littleendian}(d_2^{N-8}) \end{array}$$

Again these functions are invertable 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):

$$\begin{array}{lcll} \mathrm{signed}_N(i) & = & i & (0 \le i < 2^{N-1}) \\ \mathrm{signed}_N(i) & = & i - 2^N & (2^{N-1} \le i < 2^N) \end{array}$$

This function is bijective, and hence invertible.

[16] http://ieeexplore.ieee.org/document/4610935/
[17] https://en.wikipedia.org/wiki/Endianness#Little-endian

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.

$$\begin{array}{lcll} \mathrm{bool}(C) &=& 1 & (\text{if } C) \\ \mathrm{bool}(C) &=& 0 & (\text{otherwise}) \end{array}$$

$\mathrm{iadd}_N(i_1, i_2)$

- Return the result of adding i_1 and i_2 modulo 2^N.

$$\mathrm{iadd}_N(i_1, i_2) \quad = \quad (i_1 + i_2) \bmod 2^N$$

$\mathrm{isub}_N(i_1, i_2)$

- 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.

$$\begin{array}{lcl} \mathrm{irem\_s}_N(i_1, 0) &=& \{\} \\ \mathrm{irem\_s}_N(i_1, i_2) &=& \mathrm{signed}_N^{-1}(i_1 - i_2 \cdot \mathrm{trunc}(\mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2))) \end{array}$$

Note: This operator is *partial*.

As long as both operators are defined, it holds that $i_1 = i_2 \cdot \mathrm{idiv\_s}(i_1, i_2) + \mathrm{irem\_s}(i_1, i_2)$.

$\mathrm{iand}_N(i_1, i_2)$

- Return the bitwise conjunction of i_1 and i_2.

$$\mathrm{iand}_N(i_1, i_2) = \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \wedge \mathrm{ibits}_N(i_2))$$

$\mathrm{ior}_N(i_1, i_2)$

- Return the bitwise disjunction of i_1 and i_2.

$$\mathrm{ior}_N(i_1, i_2) = \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \vee \mathrm{ibits}_N(i_2))$$

$\mathrm{ixor}_N(i_1, i_2)$

- Return the bitwise exclusive disjunction of i_1 and i_2.

$$\mathrm{ixor}_N(i_1, i_2) = \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i_1) \veebar \mathrm{ibits}_N(i_2))$$

$\mathrm{ishl}_N(i_1, i_2)$

- Let k be i_2 modulo N.
- Return the result of shifting i_1 left by k bits, modulo 2^N.

$$\mathrm{ishl}_N(i_1, i_2) = \mathrm{ibits}_N^{-1}(d_2^{N-k}\ 0^k) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_1^k\ d_2^{N-k} \wedge k = i_2 \bmod N)$$

$\mathrm{ishr\_u}_N(i_1, i_2)$

- Let j_2 be i_2 modulo N.
- Return the result of shifting i_1 right by j_2 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 j_2 be i_2 modulo N.
- Return the result of shifting i_1 right by j_2 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 j_2 be i_2 modulo N.
- Return the result of rotating i_1 left by j_2 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 j_2 be i_2 modulo N.
- Return the result of rotating i_1 right by j_2 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.

$$\mathrm{ile\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) \le \mathrm{signed}_N(i_2))$$

$\mathrm{ige\_u}_N(i_1, i_2)$

- Return 1 if i_1 is greater than or equal to i_2, 0 otherwise.

$$\mathrm{ige\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \ge i_2)$$

$\mathrm{ige\_s}_N(i_1, i_2)$

- 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.

$$\mathrm{ige\_s}_N(i_1, i_2) \quad = \quad \mathrm{bool}(\mathrm{signed}_N(i_1) \ge \mathrm{signed}_N(i_2))$$

4.3.3 Floating-Point Operations

Floating-point arithmetic follows the IEEE 754-2008[18] 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

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.

[18] http://ieeexplore.ieee.org/document/4610935/

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:
  - 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.

$$
\begin{array}{lcll}
\mathrm{float}_N(0) & = & +0 & \\
\mathrm{float}_N(r) & = & r & (\mathrm{if}\ r \in \mathrm{exact}_N) \\
\mathrm{float}_N(r) & = & +\infty & (\mathrm{if}\ r \geq +\mathrm{limit}_N) \\
\mathrm{float}_N(r) & = & -\infty & (\mathrm{if}\ r \leq -\mathrm{limit}_N) \\
\mathrm{float}_N(r) & = & \mathrm{closest}_N(r, z_1, z_2) & (\mathrm{if}\ z_1 < r < z_2 \wedge (z_1, z_2) \in \mathrm{candidatepair}_N) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_1) & (\mathrm{if}\ |r - z_1| < |r - z_2|) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_2) & (\mathrm{if}\ |r - z_1| > |r - z_2|) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_1) & (\mathrm{if}\ |r - z_1| = |r - z_2| \wedge \mathrm{even}_N(z_1)) \\
\mathrm{closest}_N(r, z_1, z_2) & = & \mathrm{rectify}_N(r, z_2) & (\mathrm{if}\ |r - z_1| = |r - z_2| \wedge \mathrm{even}_N(z_2)) \\
\mathrm{rectify}_N(r, \pm\mathrm{limit}_N) & = & \pm\infty & \\
\mathrm{rectify}_N(r, 0) & = & +0 \quad (r \geq 0) & \\
\mathrm{rectify}_N(r, 0) & = & -0 \quad (r < 0) & \\
\mathrm{rectify}_N(r, z) & = & z &
\end{array}
$$

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* 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-determinsitically 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:

$$\begin{array}{lcll} \mathrm{nans}_N\{z^*\} & = & \{+\mathsf{nan}(n), -\mathsf{nan}(n) \mid n = \mathrm{canon}_N\} & (\text{if } \forall \mathsf{nan}(n) \in z^*,\ n = \mathrm{canon}_N) \\ \mathrm{nans}_N\{z^*\} & = & \{+\mathsf{nan}(n), -\mathsf{nan}(n) \mid n \geq \mathrm{canon}_N\} & (\text{otherwise}) \end{array}$$

$\mathrm{fadd}_N(z_1, z_2)$

- 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 $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return that infinity.
- Else if one of 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 one of 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 $\mathrm{nans}_N\{z_1, z_2\}$.
- 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 $\mathrm{nans}_N\{z_1, z_2\}$.
- 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 one of z_1 or z_2 is an infinity and the other a value with equal sign, then return positive infinity.
- Else if one of 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 $\mathrm{nans}_N\{z_1, z_2\}$.
- 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_2 by z_1, *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 one of z_1 or z_2 is a negative infinity, then return negative infinity.
- Else if one of 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.

$$
\begin{array}{lcll}
\mathrm{fmin}_N(\pm\mathsf{nan}(n), z_2) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} & \\
\mathrm{fmin}_N(z_1, \pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} & \\
\mathrm{fmin}_N(+\infty, z_2) &=& z_2 & \\
\mathrm{fmin}_N(-\infty, z_2) &=& -\infty & \\
\mathrm{fmin}_N(z_1, +\infty) &=& z_1 & \\
\mathrm{fmin}_N(z_1, -\infty) &=& -\infty & \\
\mathrm{fmin}_N(\pm 0, \mp 0) &=& -0 & \\
\mathrm{fmin}_N(z_1, z_2) &=& z_1 & (\text{if } z_1 \le z_2) \\
\mathrm{fmin}_N(z_1, z_2) &=& z_2 & (\text{if } z_2 \le z_1)
\end{array}
$$

$\mathrm{fmax}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if one of z_1 or z_2 is a positive infinity, then return positive infinity.
- Else if one of 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.

$$
\begin{array}{lcll}
\mathrm{fmax}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} & \\
\mathrm{fmax}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} & \\
\mathrm{fmax}_N(+\infty, z_2) & = & +\infty & \\
\mathrm{fmax}_N(-\infty, z_2) & = & z_2 & \\
\mathrm{fmax}_N(z_1, +\infty) & = & +\infty & \\
\mathrm{fmax}_N(z_1, -\infty) & = & z_1 & \\
\mathrm{fmax}_N(\pm 0, \mp 0) & = & +0 & \\
\mathrm{fmax}_N(z_1, z_2) & = & z_1 & (\text{if } z_1 \geq z_2) \\
\mathrm{fmax}_N(z_1, z_2) & = & z_2 & (\text{if } z_2 \geq z_1)
\end{array}
$$

$\mathrm{fcopysign}_N(z_1, z_2)$

- 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 has a negative sign, then return an element of $\mathrm{nans}_N\{z\}$.
- Else if z is positive infinity, then return positive infinity.
- Else if z is a zero, then return that zero.
- 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.

$$\begin{array}{lcll}
\mathrm{fnearest}_N(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} & \\
\mathrm{fnearest}_N(\pm\infty) & = & \pm\infty & \\
\mathrm{fnearest}_N(\pm 0) & = & \pm 0 & \\
\mathrm{fnearest}_N(+q) & = & +0 & (\text{if } 0 < +q \leq 0.5) \\
\mathrm{fnearest}_N(-q) & = & -0 & (\text{if } -0.5 \leq -q < 0) \\
\mathrm{fnearest}_N(\pm q) & = & \mathrm{float}_N(\pm i) & (\text{if } |i - q| < 0.5) \\
\mathrm{fnearest}_N(\pm q) & = & \mathrm{float}_N(\pm i) & (\text{if } |i - q| = 0.5 \wedge i \text{ even})
\end{array}$$

$\mathrm{feq}_N(z_1, z_2)$

- 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 0.
- 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.

$$\begin{array}{lcl}
\mathrm{fne}_N(\pm\mathsf{nan}(n), z_2) & = & 0 \\
\mathrm{fne}_N(z_1, \pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{fne}_N(\pm 0, \mp 0) & = & 0 \\
\mathrm{fne}_N(z_1, z_2) & = & \mathrm{bool}(z_1 \neq z_2)
\end{array}$$

$\mathrm{flt}_N(z_1, z_2)$

- 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.

$$\begin{array}{lcl}
\mathrm{fgt}_N(\pm\mathsf{nan}(n), z_2) &=& 0 \\
\mathrm{fgt}_N(z_1, \pm\mathsf{nan}(n)) &=& 0 \\
\mathrm{fgt}_N(z, z) &=& 0 \\
\mathrm{fgt}_N(+\infty, z_2) &=& 1 \\
\mathrm{fgt}_N(-\infty, z_2) &=& 0 \\
\mathrm{fgt}_N(z_1, +\infty) &=& 0 \\
\mathrm{fgt}_N(z_1, -\infty) &=& 1 \\
\mathrm{fgt}_N(\pm 0, \mp 0) &=& 0 \\
\mathrm{fgt}_N(z_1, z_2) &=& \mathrm{bool}(z_1 > z_2)
\end{array}$$

$\mathrm{fle}_N(z_1, z_2)$

- 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.

$$\begin{array}{lll}
\mathrm{fge}_N(\pm\mathsf{nan}(n), z_2) & = & 0 \\
\mathrm{fge}_N(z_1, \pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{fge}_N(z, z) & = & 1 \\
\mathrm{fge}_N(+\infty, z_2) & = & 1 \\
\mathrm{fge}_N(-\infty, z_2) & = & 0 \\
\mathrm{fge}_N(z_1, +\infty) & = & 0 \\
\mathrm{fge}_N(z_1, -\infty) & = & 1 \\
\mathrm{fge}_N(\pm 0, \mp 0) & = & 1 \\
\mathrm{fge}_N(z_1, z_2) & = & \mathrm{bool}(z_1 \geq z_2)
\end{array}$$

4.3.4 Conversions

$\mathrm{extend\_u}_{M,N}(i)$

- Return i.

$$\mathrm{extend\_u}_{M,N}(i) \quad = \quad i$$

Note: In the abstract syntax, unsigned extension just reinterprets the same value.

$\mathrm{extend\_s}_{M,N}(i)$

- Let j be the *signed interpretation* of i of size M.
- Return the two's complement of j relative to size N.

$$\mathrm{extend\_s}_{M,N}(i) \quad = \quad \mathrm{signed}_N^{-1}(\mathrm{signed}_M(i))$$

$\mathrm{wrap}_{M,N}(i)$

- Return i modulo N.

$$\mathrm{wrap}_{M,N}(i) = i \bmod 2^N$$

$\mathrm{trunc\_u}_{M,N}(z)$

- 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.

$$\begin{array}{lcll}
\mathrm{trunc_u}_{M,N}(\pm\mathsf{nan}(n)) &=& \{\} & \\
\mathrm{trunc_u}_{M,N}(\pm\infty) &=& \{\} & \\
\mathrm{trunc_u}_{M,N}(\pm q) &=& \mathrm{trunc}(\pm q) & (\text{if } -1 < \mathrm{trunc}(\pm q) < 2^N) \\
\mathrm{trunc_u}_{M,N}(\pm q) &=& \{\} & (\text{otherwise})
\end{array}$$

Note: This operator is *partial*. It is not defined for NaNs, infinities, or values for which the result is out of range.

$\mathrm{trunc\_s}_{M,N}(z)$

- 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.

$$\begin{array}{lcll}
\mathrm{trunc_s}_{M,N}(\pm\mathsf{nan}(n)) &=& \{\} & \\
\mathrm{trunc_s}_{M,N}(\pm\infty) &=& \{\} & \\
\mathrm{trunc_s}_{M,N}(\pm q) &=& \mathrm{trunc}(\pm q) & (\text{if } -2^{N-1} - 1 < \mathrm{trunc}(\pm q) < 2^{N-1}) \\
\mathrm{trunc_s}_{M,N}(\pm q) &=& \{\} & (\text{otherwise})
\end{array}$$

Note: This operator is *partial*. It is not defined for NaNs, infinities, or values for which the result is out of range.

$\mathrm{promote}_{M,N}(z)$

- 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 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 $\mathrm{float}_N(z)$.

$$\begin{array}{lcll}
\mathrm{demote}_{M,N}(\pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\} & (\text{if } n = \mathrm{canon}_N) \\
\mathrm{demote}_{M,N}(\pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{+\mathsf{nan}(1)\} & (\text{otherwise}) \\
\mathrm{demote}_{M,N}(\pm\infty) &=& \pm\infty \\
\mathrm{demote}_{M,N}(\pm 0) &=& \pm 0 \\
\mathrm{demote}_{M,N}(\pm q) &=& \mathrm{float}_N(\pm q)
\end{array}$$

$\mathrm{convert\_u}_{M,N}(i)$

- Return $\mathrm{float}_N(i)$.

$$\mathrm{convert\_u}_{M,N}(i) \quad = \quad \mathrm{float}_N(i)$$

$\mathrm{convert\_s}_{M,N}(i)$

- Let j be the *signed interpretation* of i.
- Return $\mathrm{float}_N(j)$.

$$\mathrm{convert\_u}_{M,N}(i) \quad = \quad \mathrm{float}_N(\mathrm{signed}_M(i))$$

$\mathrm{reinterpret}_{t_1,t_2}(c)$

- Let d^* be the bit sequence $\mathrm{bits}_{t_1}(c)$.
- Return the constant c' for which $\mathrm{bits}_{t_2}(c') = d^*$.

$$\mathrm{reinterpret}_{t_1,t_2}(c) \quad = \quad \mathrm{bits}_{t_2}^{-1}(\mathrm{bits}_{t_1}(c))$$

4.4 Instructions

WebAssembly computation is performed by executing individual *instructions*.

4.4.1 Numeric Instructions

Numeric instructions are defined in terms of the basic *numeric operators*. The mapping of numeric instructions to their underlying operators is expressed by the following definition:

$$\begin{array}{lcl}
op_{\mathsf{i}N}(n) &=& \mathrm{i}op_N(n) \\
op_{\mathsf{f}N}(z) &=& \mathrm{f}op_N(z)
\end{array}$$

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.

$t.\mathsf{const}\ c$

1. Push the value $t.\mathsf{const}\ c$ to the stack.

Note: No formal reduction rule is required for this instruction, since const instructions coincide with *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.

$$\begin{array}{lcll} (t.\mathsf{const}\ c_1)\ t.\mathit{unop} & \hookrightarrow & (t.\mathsf{const}\ c) & (\mathrm{if}\ c \in \mathit{unop}_t(c_1)) \\ (t.\mathsf{const}\ c_1)\ t.\mathit{unop} & \hookrightarrow & \mathsf{trap} & (\mathrm{if}\ \mathit{unop}_{t_1,t_2}(c_1) = \{\}) \end{array}$$

$t.\mathit{binop}$

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.

$$\begin{array}{lcll} (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & (t.\mathsf{const}\ c) & (\mathrm{if}\ c \in \mathit{binop}_t(c_1, c_2)) \\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & \mathsf{trap} & (\mathrm{if}\ \mathit{binop}_{t_1,t_2}(c_1) = \{\}) \end{array}$$

$t.\mathit{testop}$

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.

$$(t.\mathsf{const}\ c_1)\ t.\mathit{testop} \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad (\mathrm{if}\ c = \mathit{testop}_t(c_1))$$

$t.relop$

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 $relop_t(c_1, c_2)$.
5. Push the value $\mathsf{i32.const}\ c$ to the stack.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.relop \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad (\text{if } c = relop_t(c_1, c_2))$$

$t_2.cvtop/t_1$

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 $cvtop_{t_1,t_2}(c_1)$ is defined:
   (a) Let c_2 be a possible result of computing $cvtop_{t_1,t_2}(c_1)$.
   (b) Push the value $t_2.\mathsf{const}\ c_2$ to the stack.
4. Else:
   (a) Trap.

$$\begin{array}{lcll}
(t.\mathsf{const}\ c_1)\ t_2.cvtop/t_1 & \hookrightarrow & (t_2.\mathsf{const}\ c_2) & (\text{if } c_2 \in cvtop_{t_1,t_2}(c_1)) \\
(t.\mathsf{const}\ c_1)\ t_2.cvtop/t_1 & \hookrightarrow & \mathsf{trap} & (\text{if } cvtop_{t_1,t_2}(c_1) = \{\})
\end{array}$$

4.4.2 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.

$$val\ \mathsf{drop} \quad \hookrightarrow \quad \epsilon$$

select

1. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
2. Pop the value $\mathsf{i32.const}\ c$ 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.

$$\begin{array}{lcll}
val_1\ val_2\ (\mathsf{i32.const}\ c)\ \mathsf{select} & \hookrightarrow & val_1 & (\text{if } c \neq 0) \\
val_1\ val_2\ (\mathsf{i32.const}\ c)\ \mathsf{select} & \hookrightarrow & val_2 & (\text{if } c = 0)
\end{array}$$

4.4.3 Variable Instructions

$\mathsf{get\_local}\ x$

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.

$$F; (\mathsf{get\_local}\ x) \quad \hookrightarrow \quad F; val \qquad (\text{if } F.\mathsf{locals}[x] = val)$$

$\mathsf{set\_local}\ x$

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.

$$F; val\ (\mathsf{set\_local}\ x) \quad \hookrightarrow \quad F'; \epsilon \qquad (\text{if } F' = F \text{ with } \mathsf{locals}[x] = val)$$

$\mathsf{tee\_local}\ x$

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 $(\mathsf{set\_local}\ x)$.

$$val\ (\mathsf{tee\_local}\ x) \quad \hookrightarrow \quad val\ val\ (\mathsf{set\_local}\ x)$$

$\mathsf{get\_global}\ x$

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.

$$\begin{array}{l} S; F; (\mathsf{get\_global}\ x) \quad \hookrightarrow \quad S; F; val \\ \qquad (\text{if } S.\mathsf{globals}[F.\mathsf{module}.\mathsf{globaladdrs}[x]].\mathsf{value} = val) \end{array}$$

set_global x

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.globaladdrs[x] exists.
3. Let a be the *global address* F.module.globaladdrs[x].
4. Assert: due to *validation*, S.globals[a] exists.
5. Let $glob$ be the *global instance* S.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 $glob$.value with the value val.

$$S; F; val\ (\mathsf{set\_global}\ x) \quad \hookrightarrow \quad S'; F; \epsilon$$
$$(\text{if } S' = S \text{ with } \mathsf{globals}[F.\mathsf{module}.\mathsf{globaladdrs}[x]].\mathsf{value} = val)$$

4.4.4 Memory Instructions

t.load $memarg$ **and** t.load$N$_sx $memarg$

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.memaddrs[0] exists.
3. Let a be the *memory address* F.module.memaddrs[0].
4. Assert: due to *validation*, S.mems[a] exists.
5. Let mem be the *memory instance* S.mems[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. Let ea be $i + memarg$.offset.
9. If N is not part of the instruction, then:
   (a) Let N be the *bit width* $|t|$ of *value type* t.
10. If $ea + N/8$ is larger than the length of mem.data, then:
    (a) Trap.
11. Let b^* be the byte sequence mem.data[$ea : N/8$].
12. If N and sx are part of the instruction, then:
    (a) Let n be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
    (b) Let c be the result of computing $\mathrm{extend\_}sx_{N,|t|}(n)$.
13. Else:
    (a) Let c be the constant for which $\mathrm{bytes}_t(c) = b^*$.
14. Push the value t.const c to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}\ memarg) \quad \hookrightarrow \quad S; F; (t.\mathsf{const}\ c) \\
\qquad (\text{if } ea = i + memarg.\mathsf{offset} \\
\qquad\ \wedge\ ea + |t|/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}| \\
\qquad\ \wedge\ \mathrm{bytes}_t(c) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[ea : |t|/8]) \\
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}N\_sx\ memarg) \quad \hookrightarrow \quad S; F; (t.\mathsf{const}\ \mathrm{extend\_}sx_{N,|t|}(n)) \\
\qquad (\text{if } ea = i + memarg.\mathsf{offset} \\
\qquad\ \wedge\ ea + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}| \\
\qquad\ \wedge\ \mathrm{bytes}_{iN}(n) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[ea : N/8]) \\
S; F; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}(N\_sx)^?\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}
$$

Note: The alignment $memarg.\mathsf{align}$ does not affect the semantics. Unaligned access is supported for all types, and succeeds regardless of the annotation. The only purpose of the annotation is to provide optimizatons hints.

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

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* t is on the top of the stack.
7. Pop the value $t.\mathsf{const}\ c$ from the stack.
8. Assert: due to *validation*, a value of *value type* $\mathsf{i32}$ is on the top of the stack.
9. Pop the value $\mathsf{i32.const}\ i$ from the stack.
10. Let ea be $i + memarg.\mathsf{offset}$.
11. If N is not part of the instruction, then:
    (a) Let N be the *bit width* $|t|$ of *value type* t.
12. If $ea + N/8$ is larger than the length of $mem.\mathsf{data}$, then:
    (a) Trap.
13. If N is part of the instruction, then:
    (a) Let n be the result of computing $\mathrm{wrap}_{|t|,N}(c)$.
    (b) Let b^* be the byte sequence $\mathrm{bytes}_{iN}(n)$.
14. Else:
    (a) Let b^* be the byte sequence $\mathrm{bytes}_t(c)$.
15. Replace the bytes $mem.\mathsf{data}[ea : N/8]$ with b^*.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ \mathit{memarg}) \quad \hookrightarrow \quad S'; F; \epsilon \\
\qquad (\text{if } \mathit{ea} = i + \mathit{memarg}.\mathsf{offset} \\
\qquad \wedge\ \mathit{ea} + |t|/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}| \\
\qquad \wedge\ S' = S \text{ with } \mathsf{memaddrs}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[\mathit{ea} : |t|/8] = \mathrm{bytes}_t(c) \\
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N\ \mathit{memarg}) \quad \hookrightarrow \quad S'; F; \epsilon \\
\qquad (\text{if } \mathit{ea} = i + \mathit{memarg}.\mathsf{offset} \\
\qquad \wedge\ \mathit{ea} + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}| \\
\qquad \wedge\ S' = S \text{ with } \mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}[\mathit{ea} : N/8] = \mathrm{bytes}_{iN}(\mathrm{wrap}_{|t|,N}(c)) \\
S; F; (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}N^?\ \mathit{memarg}) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}
$$

current_memory

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. Let sz be the length of $\mathit{mem}.\mathsf{data}$ divided by the *page size*.
7. Push the value $\mathsf{i32.const}\ \mathit{sz}$ to the stack.

$$
\begin{array}{l}
S; F; \mathsf{current\_memory} \quad \hookrightarrow \quad S; F; (\mathsf{i32.const}\ \mathit{sz}) \\
\qquad (\text{if } |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{memaddrs}[0]].\mathsf{data}| = \mathit{sz} \cdot 64\,\mathrm{Ki})
\end{array}
$$

grow_memory

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. Let sz be the length of $S.\mathsf{mems}[a]$ divided by the *page size*.
7. Assert: due to *validation*, a value of *value type* $\mathsf{i32}$ is on the top of the stack.
8. Pop the value $\mathsf{i32.const}\ n$ from the stack.
9. If $\mathit{mem}.\mathsf{max}$ is not empty and $\mathit{sz} + n$ is larger than $\mathit{mem}.\mathsf{max}$, then:
1. Push the value $\mathsf{i32.const}\ (-1)$ to the stack.
10. Else, either:
    (a) Let len be n multiplied with the *page size*.
    (b) Append len bytes with value `0x00` to $S.\mathsf{mems}[a]$.
    (c) Push the value $\mathsf{i32.const}\ \mathit{sz}$ to the stack.
11. Or:
    (a) Push the value $\mathsf{i32.const}\ (-1)$ to the stack.

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ n)\ \mathsf{grow\_memory} &\hookrightarrow& S'; F; (\mathsf{i32.const}\ sz) \\
\qquad (\mathrm{if}\ F.\mathsf{module}.\mathsf{memaddrs}[0] = a \\
\qquad\ \wedge |S.\mathsf{mems}[a].\mathsf{data}| = sz \cdot 64\,\mathrm{Ki} \\
\qquad\ \wedge (sz + n \leq S.\mathsf{mems}[a].\mathsf{max} \vee S.\mathsf{mems}[a].\mathsf{max} = \epsilon) \\
\qquad\ \wedge S' = S\ \mathsf{with}\ \mathsf{mems}[a].\mathsf{data} = S.\mathsf{mems}[a].\mathsf{data}\ (\mathtt{0x00})^{n \cdot 64\,\mathrm{Ki}}) \\
S; F; (\mathsf{i32.const}\ n)\ \mathsf{grow\_memory} &\hookrightarrow& S; F; (\mathsf{i32.const}\ {-1})
\end{array}
$$

Note: The grow_memory instruction is non-deterministic. 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*.

4.4.5 Control Instructions

nop

1. Do nothing.

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

unreachable

1. Trap.

$$\mathsf{unreachable} \quad \hookrightarrow \quad \mathsf{trap}$$

block *blocktype* *instr** end

1. Assert: due to *validation*, $\mathrm{expand}_F(\mathit{blocktype})$ is defined.
2. Let $[t_1^m] \rightarrow [t_2^n]$ be the *function type* $\mathrm{expand}_F(\mathit{blocktype})$.
3. Let L be the label whose arity is n and whose continuation is the end of the block.
4. *Enter* the block *instr** with label L.

$$F; \mathsf{block}\ bt\ \mathit{instr}^*\ \mathsf{end} \quad \hookrightarrow \quad F; \mathsf{label}_n\{\epsilon\}\ \mathit{instr}^*\ \mathsf{end} \qquad (\mathrm{if}\ \mathrm{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n])$$

loop *blocktype* *instr** end

1. Assert: due to *validation*, $\mathrm{expand}_F(\mathit{blocktype})$ is defined.
2. Let $[t_1^m] \rightarrow [t_2^n]$ be the *function type* $\mathrm{expand}_F(\mathit{blocktype})$.
3. Let L be the label whose arity is m and whose continuation is the start of the loop.
4. *Enter* the block *instr** with label L.

$$F; \mathsf{loop}\ bt\ \mathit{instr}^*\ \mathsf{end} \quad \hookrightarrow \quad F; \mathsf{label}_m\{\mathsf{loop}\ bt\ \mathit{instr}^*\ \mathsf{end}\}\ \mathit{instr}^*\ \mathsf{end} \qquad (\mathrm{if}\ \mathrm{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n])$$

if $blocktype$ $instr_1^*$ else $instr_2^*$ end

1. Assert: due to *validation*, $\mathrm{expand}_F(blocktype)$ is defined.
2. Let $[t_1^m] \to [t_2^n]$ be the *function type* $\mathrm{expand}_F(blocktype)$.
3. Let L be the label whose arity is n and whose continuation is the end of the if instruction.
4. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
5. Pop the value i32.const c from the stack.
6. If c is non-zero, then:
   (a) *Enter* the block $instr_1^*$ with label L.
7. Else:
   (a) *Enter* the block $instr_2^*$ with label L.

$$\begin{array}{lcll}
F; (\mathsf{i32.const}\ c)\ \mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} & \hookrightarrow & F; \mathsf{label}_n\{\epsilon\}\ instr_1^*\ \mathsf{end} & (\text{if } c \neq 0 \wedge \mathrm{expand}_F(bt) = [t_1^m] \to [t_2^n]) \\
F; (\mathsf{i32.const}\ c)\ \mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} & \hookrightarrow & F; \mathsf{label}_n\{\epsilon\}\ instr_2^*\ \mathsf{end} & (\text{if } c = 0 \wedge \mathrm{expand}_F(bt) = [t_1^m] \to [t_2^n])
\end{array}$$

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 $(\mathsf{br}\ l)$.
4. Else:
   (a) Do nothing.

$$\begin{array}{lcll}
(\mathsf{i32.const}\ c)\ (\mathsf{br_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & (\text{if } c \neq 0) \\
(\mathsf{i32.const}\ c)\ (\mathsf{br_if}\ l) & \hookrightarrow & \epsilon & (\text{if } c = 0)
\end{array}$$

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).

$$\begin{array}{lcll} (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_i) & (\text{if } l^*[i] = l_i) \\ (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_N) & (\text{if } |l^*| \leq i) \end{array}$$

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.module.funcaddrs$[x]$ exists.
3. Let a be the *function address* F.module.funcaddrs$[x]$.
4. *Invoke* the function instance at address a.

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

call_indirect x

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.tableaddrs$[0]$ exists.
3. Let ta be the *table address* F.module.tableaddrs$[0]$.
4. Assert: due to *validation*, S.tables$[ta]$ exists.

5. Let tab be the *table instance* $S.\mathsf{tables}[ta]$.
6. Assert: due to *validation*, $F.\mathsf{module}.\mathsf{types}[x]$ exists.
7. Let ft_{expect} be the *function type* $F.\mathsf{module}.\mathsf{types}[x]$.
8. Assert: due to *validation*, a value with *value type* $\mathsf{i32}$ 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. If $tab.\mathsf{elem}[i]$ is uninitialized, then:
    (a) Trap.
12. Let a be the *function address* $tab.\mathsf{elem}[i]$.
13. Assert: due to *validation*, $S.\mathsf{funcs}[a]$ exists.
14. Let f be the *function instance* $S.\mathsf{funcs}[a]$.
15. Let ft_{actual} be the *function type* $f.\mathsf{type}$.
16. If ft_{actual} and ft_{expect} differ, then:
    (a) Trap.
17. *Invoke* the function instance at address a.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call_indirect}\ x) & \hookrightarrow & S; F; (\mathsf{invoke}\ a) \\
\quad (\mathrm{if}\ S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[0]].\mathsf{elem}[i] = a & & \\
\quad \wedge\ S.\mathsf{funcs}[a] = f & & \\
\quad \wedge\ F.\mathsf{module}.\mathsf{types}[x] = f.\mathsf{type}) & & \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call_indirect}\ x) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\mathrm{otherwise}) & &
\end{array}$$

4.4.6 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. Let n be the arity of L.
2. Assert: due to *validation*, there are n values on the top of the stack.
3. Pop the results val^n from the stack.
4. Assert: due to *validation*, the label L is now on the top of the stack.

5. Pop the label from the stack.
6. Push val^n back to the stack.
7. Jump to the position after the end of the *structured control instruction* associated with the label L.

$$\mathsf{label}_n\{instr^*\}\ val^n\ \mathsf{end} \quad\hookrightarrow\quad val^n$$

Note: This semantics also applies to the instruction sequence contained in a loop instruction. Therefor, execution of a loop falls off the end, unless a backwards branch is performed explicitly.

4.4.7 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 $[t_1^n] \to [t_2^m]$ be the *function type* $f.\mathsf{type}$.
4. Let t^* be the list of *value types* $f.\mathsf{code}.\mathsf{locals}$.
5. Let $instr^*\ \mathsf{end}$ 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 val_0^* be the list of zero values of types t^*.
9. Let F be the *frame* $\{\mathsf{module}\ f.\mathsf{module}, \mathsf{locals}\ val^n\ val_0^*\}$.
10. Push the activation of F with arity m to the stack.
11. Let L be the *label* whose arity is m and whose continuation is the end of the function.
12. *Enter* the instruction sequence $instr^*$ with label L.

$$\begin{array}{l} S; val^n\ (\mathsf{invoke}\ a) \quad\hookrightarrow\quad S; \mathsf{frame}_m\{F\}\ \mathsf{label}_m\{\}\ instr^*\ \mathsf{end}\ \mathsf{end} \\ \quad (\text{if } S.\mathsf{funcs}[a] = f \\ \quad\ \wedge f.\mathsf{type} = [t_1^n] \to [t_2^m] \\ \quad\ \wedge f.\mathsf{code} = \{\mathsf{type}\ x, \mathsf{locals}\ t^k, \mathsf{body}\ instr^*\ \mathsf{end}\} \\ \quad\ \wedge F = \{\mathsf{module}\ f.\mathsf{module},\ \mathsf{locals}\ val^n\ (t.\mathsf{const}\ 0)^k\}) \end{array}$$

Returning from a function

When the end of a funtion 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.

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

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 is assumed that it consumes and produces the right number and types of WebAssembly *values* on the stack, according to its *function type*. A host function may also modify the *store*.

$$\begin{array}{lcl}
S; val_1^n\ (\mathsf{invoke}\ a) & \hookrightarrow & S'; val_2^m \\
\quad (\mathrm{if}\ S.\mathsf{funcs}[a] = \{\mathsf{type}\ [t_1^n] \to [t_2^m], \mathsf{hostcode}\ \dots\} \\
\quad \wedge\ val_1^n = (t_1.\mathsf{const}\ c_1)^n \\
\quad \wedge\ val_2^m = (t_2.\mathsf{const}\ c_2)^m \\
\quad \wedge\ S' \succ S) \\
S; val^n\ (\mathsf{invoke}\ a) & \hookrightarrow & S'; \mathsf{trap} \\
\quad (\mathrm{if}\ S.\mathsf{funcs}[a] = \{\mathsf{type}\ \mathit{ft}, \mathsf{hostcode}\ \dots\} \\
\quad \wedge\ S' \succ S)
\end{array}$$

Here, $S' \succ S$ expresses that the new store S' is *reachable* from S. Such a store must not contain fewer addresses than the original store, it must not differ in elements that are not mutable, and it must still be well-typed.

Todo

Define this relation more precisely.

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.

4.4.8 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 of the instruction sequence.
3. Assert: due to *validation*, the top of the stack contains a *value*.
4. Pop the the *value* val from the stack.

The value val is the result of the evaluation.

$$\frac{S; F; instr^* \hookrightarrow^* S'; F'; v}{S; F; instr^*\ \mathsf{end} \hookrightarrow^* S'; F'; v}$$

4.5 Modules

For modules, the execution semantics primarily defines *instantiation*, which *allocates* instances for a module and its contained definitions, inititializes *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 external value lives.

$\mathsf{func}\ a$

- The store entry $S.\mathsf{funcs}[a]$ must be a *function instance* $\{\mathsf{type}\ \mathit{functype}, \dots\}$.
- Then $\mathsf{func}\ a$ is valid with *external type* $\mathsf{func}\ \mathit{functype}$.

$$\frac{S.\mathsf{funcs}[a] = \{\mathsf{type}\ \mathit{functype}, \dots\}}{S \vdash \mathsf{func}\ a : \mathsf{func}\ \mathit{functype}}$$

$\mathsf{table}\ a$

- The store entry $S.\mathsf{tables}[a]$ must be a *table instance* $\{\mathsf{elem}\ (\mathit{fa}^?)^n, \mathsf{max}\ m^?\}$.
- Then $\mathsf{table}\ a$ is valid with *external type* $\mathsf{table}\ (\{\mathsf{min}\ n, \mathsf{max}\ m^?\}\ \mathsf{anyfunc})$.

$$\frac{S.\mathsf{tables}[a] = \{\mathsf{elem}\ (\mathit{fa}^?)^n, \mathsf{max}\ m^?\}}{S \vdash \mathsf{table}\ a : \mathsf{table}\ (\{\mathsf{min}\ n, \mathsf{max}\ m^?\}\ \mathsf{anyfunc})}$$

$\mathsf{mem}\ a$

- The store entry $S.\mathsf{mems}[a]$ must be a *memory instance* $\{\mathsf{data}\ b^{n \cdot 64\,\mathrm{Ki}}, \mathsf{max}\ m^?\}$, for some n.
- Then $\mathsf{mem}\ a$ is valid with *external type* $\mathsf{mem}\ (\{\mathsf{min}\ n, \mathsf{max}\ m^?\})$.

$$\frac{S.\mathsf{mems}[a] = \{\mathsf{data}\ b^{n \cdot 64\,\mathrm{Ki}}, \mathsf{max}\ m^?\}}{S \vdash \mathsf{mem}\ a : \mathsf{mem}\ \{\mathsf{min}\ n, \mathsf{max}\ m^?\}}$$

$\mathsf{global}\ a$

- The store entry $S.\mathsf{globals}[a]$ must be a *global instance* $\{\mathsf{value}\ (t.\mathsf{const}\ c), \mathsf{mut}\ \mathit{mut}\}$.
- Then $\mathsf{global}\ a$ is valid with *external type* $\mathsf{global}\ (\mathit{mut}\ t)$.

$$\frac{S.\mathsf{globals}[a] = \{\mathsf{value}\ (t.\mathsf{const}\ c), \mathsf{mut}\ \mathit{mut}\}}{S \vdash \mathsf{global}\ a : \mathsf{global}\ (\mathit{mut}\ t)}$$

4.5.2 Import Matching

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, as defined below.

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* $\mathsf{table}\ (\mathit{limits}_1\ \mathit{elemtype}_1)$ matches $\mathsf{table}\ (\mathit{limits}_2\ \mathit{elemtype}_2)$ if and only if:

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

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

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.

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

4.5.3 Allocation

New instances of *functions*, *tables*, *memories*, *globals*, and *modules* are *allocated* in a *store* S, as defined by the following auxiliary functions.

Functions

1. Let $func$ be the *function* to allocate and $moduleinst$ its *module instance*.
2. Let a be the first free *function address* in S.
3. Let $functype$ be the *function type* $moduleinst.\mathsf{types}[func.\mathsf{type}]$.
4. Let $funcinst$ be the *function instance* $\{\mathsf{type}~functype, \mathsf{module}~moduleinst, \mathsf{code}~func\}$.
5. Append $funcinst$ to the funcs of S.
6. Return a.

$$
\begin{array}{rcl}
\mathrm{allocfunc}(S, func, moduleinst) & = & S', funcaddr \\
\text{where:} & & \\
funcaddr & = & |S.\mathsf{funcs}| \\
functype & = & moduleinst.\mathsf{types}[func.\mathsf{type}] \\
funcinst & = & \{\mathsf{type}~functype, \mathsf{module}~moduleinst, \mathsf{code}~func\} \\
S' & = & S \oplus \{\mathsf{funcs}~funcinst\}
\end{array}
$$

Host Functions

1. Let $hostfunc$ be the *host function* to allocate and $functype$ its *function type*.
2. Let a be the first free *function address* in S.
4. Let $funcinst$ be the *function instance* $\{\mathsf{type}~functype, \mathsf{hostcode}~hostfunc\}$.
5. Append $funcinst$ to the funcs of S.
6. Return a.

$$
\begin{array}{rcl}
\mathrm{allochostfunc}(S, hostfunc, functype) & = & S', funcaddr \\
\text{where:} & & \\
funcaddr & = & |S.\mathsf{funcs}| \\
funcinst & = & \{\mathsf{type}~functype, \mathsf{hostcode}~hostfunc\} \\
S' & = & S \oplus \{\mathsf{funcs}~tableinst\}
\end{array}
$$

Note: Host functions are never allocated by the WebAssembly semantics itself, but may be allocated by the *embedder*.

Tables

1. Let $table$ be the *table* to allocate.
2. Let $(\{\mathsf{min}~n, \mathsf{max}~m^?\}~elemtype)$ be the *table type* $table.\mathsf{type}$.
3. Let a be the first free *table address* in S.
4. Let $tableinst$ be the *table instance* $\{\mathsf{elem}~(\epsilon)^n, \mathsf{max}~m^?\}$ with n empty elements.
5. Append $tableinst$ to the tables of S.

6. Return a.

$$\begin{array}{rcl} \mathrm{alloctable}(S, \mathit{table}) & = & S', \mathit{tableaddr} \\ & \mathrm{where:} & \\ \mathit{table}.\mathsf{type} & = & \{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathit{elemtype} \\ \mathit{tableaddr} & = & |S.\mathsf{tables}| \\ \mathit{tableinst} & = & \{\mathsf{elem}~(\epsilon)^n, \mathsf{max}~m^?\} \\ S' & = & S \oplus \{\mathsf{tables}~\mathit{tableinst}\} \end{array}$$

Memories

1. Let mem be the *memory* to allocate.
2. Let $\{\mathsf{min}~n, \mathsf{max}~m^?\}$ be the *table type* $\mathit{mem}.\mathsf{type}$.
3. Let a be the first free *memory address* in S.
4. Let $\mathit{meminst}$ be the *memory instance* $\{\mathsf{data}~(\mathtt{0x00})^{n \cdot 64\,\mathrm{Ki}}, \mathsf{max}~m^?\}$ that contains n pages of zeroed *bytes*.
5. Append $\mathit{meminst}$ to the mems of S.
6. Return a.

$$\begin{array}{rcl} \mathrm{allocmem}(S, \mathit{mem}) & = & S', \mathit{memaddr} \\ & \mathrm{where:} & \\ \mathit{mem}.\mathsf{type} & = & \{\mathsf{min}~n, \mathsf{max}~m^?\} \\ \mathit{memaddr} & = & |S.\mathsf{mems}| \\ \mathit{meminst} & = & \{\mathsf{data}~(\mathtt{0x00})^{n \cdot 64\,\mathrm{Ki}}, \mathsf{max}~m^?\} \\ S' & = & S \oplus \{\mathsf{mems}~\mathit{meminst}\} \end{array}$$

Globals

1. Let global be the *global* to allocate.
2. Let $\mathit{mut}~t$ be the *global type* $\mathit{global}.\mathsf{type}$.
3. Let a be the first free *global address* in S.
4. Let $\mathit{globalinst}$ be the *global instance* $\{\mathsf{value}~(t.\mathsf{const}~0), \mathsf{mut}~\mathit{mut}\}$ whose contents is a zero *value* of *value type* t.
5. Append $\mathit{globalinst}$ to the $\mathsf{globals}$ of S.
6. Return a.

$$\begin{array}{rcl} \mathrm{allocglobal}(S, \mathit{global}) & = & S', \mathit{globaladdr} \\ & \mathrm{where:} & \\ \mathit{global}.\mathsf{type} & = & \mathit{mut}~t \\ \mathit{globaladdr} & = & |S.\mathsf{globals}| \\ \mathit{globalinst} & = & \{\mathsf{value}~(t.\mathsf{const}~0), \mathsf{mut}~\mathit{mut}\} \\ S' & = & S \oplus \{\mathsf{globals}~\mathit{globalinst}\} \end{array}$$

Modules

The allocation function for *modules* requires a suitable list of *external values* that are assumed to *match* the *import* vector of the module.

1. Let module be the *module* to allocate and $\mathit{externval}^*_{\mathrm{im}}$ the vector of *external values* providing the module's imports.

2. For each *function* $func_i$ in $module.\mathsf{funcs}$, do:
   (a) Let $funcaddr_i$ be the *function address* resulting from *allocating* $func_i$ for the *module instance* $moduleinst$ defined below.
3. For each *table* $table_i$ in $module.\mathsf{tables}$, do:
   (a) Let $tableaddr_i$ be the *table address* resulting from *allocating* $table_i$.
4. For each *memory* mem_i in $module.\mathsf{mems}$, do:
   (a) Let $memaddr_i$ be the *memory address* resulting from *allocating* mem_i.
5. For each *global* $global_i$ in $module.\mathsf{globals}$, do:
   (a) Let $globaladdr_i$ be the *global address* resulting from *allocating* $global_i$.
6. Let $funcaddr^*$ be the the concatenation of the *function addresses* $funcaddr_i$ in index order.
7. Let $tableaddr^*$ be the the concatenation of the *table addresses* $tableaddr_i$ in index order.
8. Let $memaddr^*$ be the the concatenation of the *memory addresses* $memaddr_i$ in index order.
9. Let $globaladdr^*$ be the the concatenation of the *global addresses* $globaladdr_i$ in index order.
10. Let $funcaddr^*_{\mathrm{mod}}$ be the list of *function addresses* extracted from $externval^*_{\mathrm{im}}$, concatenated with $funcaddr^*$.
11. Let $tableaddr^*_{\mathrm{mod}}$ be the list of *table addresses* extracted from $externval^*_{\mathrm{im}}$, concatenated with $tableaddr^*$.
12. Let $memaddr^*_{\mathrm{mod}}$ be the list of *memory addresses* extracted from $externval^*_{\mathrm{im}}$, concatenated with $memaddr^*$.
13. Let $globaladdr^*_{\mathrm{mod}}$ be the list of *global addresses* extracted from $externval^*_{\mathrm{im}}$, concatenated with $globaladdr^*$.
14. For each *export* $export_i$ in $module.\mathsf{exports}$, do:
   (a) If $export_i$ is a function export for *function index* x, then let $externval_i$ be the *external value* $\mathsf{func}\ (funcaddr^*_{\mathrm{mod}}[x])$.
   (b) Else, if $export_i$ is a table export for *table index* x, then let $externval_i$ be the *external value* $\mathsf{table}\ (tableaddr^*_{\mathrm{mod}}[x])$.
   (c) Else, if $export_i$ is a memory export for *memory index* x, then let $externval_i$ be the *external value* $\mathsf{mem}\ (memaddr^*_{\mathrm{mod}}[x])$.
   (d) Else, if $export_i$ is a global export for *global index* x, then let $externval_i$ be the *external value* $\mathsf{global}\ (globaladdr^*_{\mathrm{mod}}[x])$.
   (e) Let $exportinst_i$ be the *export instance* $\{\mathsf{name}\ (export_i.\mathsf{name}), \mathsf{value}\ externval_i\}$.
15. Let $exportinst^*$ be the the concatenation of the *export instances* $exportinst_i$ in index order.
16. Let $moduleinst$ be the *module instance* $\{\mathsf{types}\ (module.\mathsf{types}),\ \mathsf{funcaddrs}\ funcaddr^*_{\mathrm{mod}},\ \mathsf{tableaddrs}\ tableaddr^*_{\mathrm{mod}},\ \mathsf{memaddrs}\ memaddr^*_{\mathrm{mod}},\ \mathsf{globaladdrs}\ globaladdr^*_{\mathrm{mod}},\ \mathsf{exports}\ exportinst^*\}$.
17. Return $moduleinst$.

$$\mathrm{allocmodule}(S, module, externval^*_{\mathrm{im}}) \quad = \quad S', moduleinst$$

where:

$$
\begin{array}{rcl}
\mathit{moduleinst} &=& \{\ \mathsf{types}\ \mathit{module}.\mathsf{types}, \\
&& \ \ \mathsf{funcaddrs}\ \mathrm{funcs}(\mathit{externval}_{\mathrm{im}}^*)\ \mathit{funcaddr}^*, \\
&& \ \ \mathsf{tableaddrs}\ \mathrm{tables}(\mathit{externval}_{\mathrm{im}}^*)\ \mathit{tableaddr}^*, \\
&& \ \ \mathsf{memaddrs}\ \mathrm{mems}(\mathit{externval}_{\mathrm{im}}^*)\ \mathit{memaddr}^*, \\
&& \ \ \mathsf{globaladdrs}\ \mathrm{globals}(\mathit{externval}_{\mathrm{im}}^*)\ \mathit{globaladdr}^*, \\
&& \ \ \mathsf{exports}\ \mathit{exportinst}^*\ \} \\
S_1, \mathit{funcaddr}^* &=& \mathrm{allocfunc}^*(S, \mathit{module}.\mathsf{funcs}, \mathit{moduleinst}) \\
S_2, \mathit{tableaddr}^* &=& \mathrm{alloctable}^*(S_1, \mathit{module}.\mathsf{tables}) \\
S_3, \mathit{memaddr}^* &=& \mathrm{allocmem}^*(S_2, \mathit{module}.\mathsf{mems}) \\
S', \mathit{globaladdr}^* &=& \mathrm{allocglobal}^*(S_3, \mathit{module}.\mathsf{globals}) \\
\mathit{exportinst}^* &=& \{\mathsf{name}\ (\mathit{export}.\mathsf{name}), \mathsf{value}\ \mathit{externval}_{\mathrm{ex}}\}^* \quad (\text{where } \mathit{export}^* = \mathit{module}.\mathsf{exports}) \\
\mathrm{funcs}(\mathit{externval}_{\mathrm{ex}}^*) &=& (\mathit{moduleinst}.\mathsf{funcaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{funcs}(\mathit{module}.\mathsf{exports})) \\
\mathrm{tables}(\mathit{externval}_{\mathrm{ex}}^*) &=& (\mathit{moduleinst}.\mathsf{tableaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{tables}(\mathit{module}.\mathsf{exports})) \\
\mathrm{mems}(\mathit{externval}_{\mathrm{ex}}^*) &=& (\mathit{moduleinst}.\mathsf{memaddrs}[x])^* \qquad (\text{where } x^* = \mathrm{mems}(\mathit{module}.\mathsf{exports})) \\
\mathrm{globals}(\mathit{externval}_{\mathrm{ex}}^*) &=& (\mathit{moduleinst}.\mathsf{globaladdrs}[x])^* \qquad (\text{where } x^* = \mathrm{globals}(\mathit{module}.\mathsf{exports}))
\end{array}
$$

Here, the notation allocX^* is shorthand for multiple *allocations* of object kind X, defined as follows:

$$
\begin{array}{rcl}
\mathrm{allocX}^*(S_0, X^n, \dots) &=& S_n, a^n \\
\text{where for all } i < n: && \\
S_{i+1}, a^n[i] &=& \mathrm{allocX}(S_i, X^n[i], \dots)
\end{array}
$$

Note: The definition of module allocation is mutually recursive with the allocation of its associated functions, because the resulting module instance $\mathit{moduleinst}$ 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 may *fail* with an error if the module is not *valid* or the imports do not *match*. Instantiation can also result in a *trap* 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* $\mathit{externtype}^m$ 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 $\mathit{externtype}^n$, do:
   (a) Assert: $\mathit{externval}_i$ is *valid* with *external type* $\mathit{externtype}'_i$ in store S.
   (b) If $\mathit{externtype}'_i$ does not *match* $\mathit{externtype}_i$, then:
      i. Fail.
5. Let $\mathit{moduleinst}$ be a new module instance *allocated* from module in store S.
6. Let F be the *frame* $\{\mathsf{module}\ \mathit{moduleinst}, \mathsf{locals}\ \epsilon\}$.
7. Push the frame F to the stack.

8. For each *element segment* $elem_i$ in $module$.elem, do:
   (a) Let $eoval_i$ be the result of *evaluating* the expression $elem_i$.offset.
   (b) Assert: due to *validation*, $eoval_i$ is of the form i32.const eo_i.
   (c) Let $tableidx_i$ be the *table index* $elem_i$.table.
   (d) Assert: due to *validation*, $moduleinst$.tableaddrs[$tableidx_i$] exists.
   (e) Let $tableaddr_i$ be the *table address* $moduleinst$.tableaddrs[$tableidx_i$].
   (f) Assert: due to *validation*, S.tables[$tableaddr_i$] exists.
   (g) Let $tableinst_i$ be the *table instance* S.tables[$tableaddr_i$].
   (h) Let $eend_i$ be eo_i plus the length of $elem_i$.init.
   (i) If $eend_i$ is larger than the length of $tableinst_i$.elem, then:
      i. Fail.
9. For each *data segment* $data_i$ in $module$.data, do:
   (a) Let $doval_i$ be the result of *evaluating* the expression $data_i$.offset.
   (b) Assert: due to *validation*, $doval_i$ is of the form i32.const do_i.
   (c) Let $memidx_i$ be the *memory index* $data_i$.data.
   (d) Assert: due to *validation*, $moduleinst$.memaddrs[$memidx_i$] exists.
   (e) Let $memaddr_i$ be the *memory address* $moduleinst$.memaddrs[$memidx_i$].
   (f) Assert: due to *validation*, S.mems[$memaddr_i$] exists.
   (g) Let $meminst_i$ be the *memory instance* S.mems[$memaddr_i$].
   (h) Let $dend_i$ be do_i plus the length of $data_i$.init.
   (i) If $dend_i$ is larger than the length of $meminst_i$.data, then:
      i. Fail.
10. Let $globalidx_{\mathrm{new}}$ be the *global index* that corresponds to the number of global *imports* in $module$.imports (i.e., the index of the first non-imported global).
11. For each *global* $global_i$ in $module$.globals, do:
   (a) Let val_i be the result of *evaluating* the initializer expression $global_i$.init.
   (b) Let $globalidx_i$ be the *global index* $globalidx_{\mathrm{new}} + i$.
   (c) Assert: due to *validation*, $moduleinst$.globaladdrs[$globalidx_i$] exists.
   (d) Let $globaladdr_i$ be the *global address* $moduleinst$.globaladdrs[$globalidx_i$].
   (e) Assert: due to *validation*, S.globals[$globaladdr_i$] exists.
   (f) Let $globalinst_i$ be the *global instance* S.globals[$globaladdr_i$].
12. Assert: due to *validation*, the frame F is now on the top of the stack.
13. Pop the frame from the stack.
14. For each *element segment* $elem_i$ in $module$.elem, do:
   (a) For each *function index* $funcidx_{ij}$ in $elem_i$.init (starting with $j = 0$), do:
      i. Assert: due to *validation*, $moduleinst$.funcaddrs[$funcidx_{ij}$] exists.
      ii. Let $funcaddr_{ij}$ be the *function address* $moduleinst$.funcaddrs[$funcidx_{ij}$].
      iii. Replace $tableinst_i$.elem[$eo_i + j$] with $funcaddr_{ij}$.
15. For each *data segment* $data_i$ in $module$.data, do:

(a) For each *byte* b_{ij} in $data_i$.init (starting with $j = 0$), do:

i. Replace $meminst_i$.data$[do_i + j]$ with b_{ij}.

16. For each *global* $global_i$ in $module$.globals, do:

(a) Replace $globalinst_i$.value with val_i.

17. If the *start function* $module$.start is not empty, then:

(a) Assert: due to *validation*, $moduleinst$.funcaddrs$[module$.start.func$]$ exists.

(b) Let $funcaddr$ be the *function address* $moduleinst$.funcaddrs$[module$.start.func$]$.

(c) *Invoke* the function instance at $funcaddr$.

$$
\begin{array}{rcl}
S; \mathsf{instantiate}\ module\ externval^n & \hookrightarrow & S'; (\mathsf{init\_table}\ tableaddr\ eo\ moduleinst\ elem.\mathsf{init})^* \\
& & \quad (\mathsf{init\_mem}\ memaddr\ do\ data.\mathsf{init})^* \\
& & \quad (\mathsf{init\_global}\ globaladdr\ v)^* \\
& & \quad (\mathsf{invoke}\ funcaddr)^? \\
& & \quad moduleinst \\
& (\mathrm{if} & \vdash module : externtype^n \\
& \wedge & (\vdash externval : externtype')^n \\
& \wedge & (\vdash externtype' \leq externtype)^n \\
& \wedge & module.\mathsf{globals} = global^k \\
& \wedge & module.\mathsf{elem} = elem^* \\
& \wedge & module.\mathsf{data} = data^* \\
& \wedge & module.\mathsf{start} = start^? \\
& \wedge & S', moduleinst = \mathrm{allocmodule}(S, module, externval^n) \\
& \wedge & F = \{\mathsf{module}\ moduleinst, \mathsf{locals}\ \epsilon\} \\
& \wedge & (S'; F; elem.\mathsf{offset} \hookrightarrow^* S'; F; \mathsf{i32.const}\ eo)^* \\
& \wedge & (S'; F; data.\mathsf{offset} \hookrightarrow^* S'; F; \mathsf{i32.const}\ do)^* \\
& \wedge & (S'; F; global.\mathsf{init} \hookrightarrow^* S'; F; v)^* \\
& \wedge & (tableaddr = moduleinst.\mathsf{tableaddrs}[elem.\mathsf{table}])^* \\
& \wedge & (memaddr = moduleinst.\mathsf{memaddrs}[data.\mathsf{data}])^* \\
& \wedge & globaladdr^* = moduleinst.\mathsf{globaladdrs}[|moduleinst.\mathsf{globaladdrs}| - k : k] \\
& \wedge & (funcaddr = moduleinst.\mathsf{funcaddrs}[start.\mathsf{func}])^? \\
& \wedge & (eo + |elem.\mathsf{init}| \leq |S'.\mathsf{tables}[tableaddr].\mathsf{elem}|)^* \\
& \wedge & (do + |data.\mathsf{init}| \leq |S'.\mathsf{mems}[memaddr].\mathsf{data}|)^*) \\
S; \mathsf{instantiate}\ module\ externval^n & \hookrightarrow & S'; \mathsf{trap} \qquad (\mathrm{otherwise}) \\
S; \mathsf{init\_table}\ a\ i\ m\ \epsilon & \hookrightarrow & S; \epsilon \\
S; \mathsf{init\_table}\ a\ i\ m\ (x_0\ x^*) & \hookrightarrow & S'; \mathsf{init\_table}\ a\ (i+1)\ m\ x^* \\
& & (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{tables}[a].\mathsf{elem}[i] = m.\mathsf{funcaddrs}[x_0]) \\
S; \mathsf{init\_mem}\ a\ i\ \epsilon & \hookrightarrow & S; \epsilon \\
S; \mathsf{init\_mem}\ a\ i\ (b_0\ b^*) & \hookrightarrow & S'; \mathsf{init\_mem}\ a\ (i+1)\ b^* \\
& & (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{mems}[a].\mathsf{data}[i] = b_0) \\
S; \mathsf{init\_global}\ a\ v & \hookrightarrow & S'; \epsilon \\
& & (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{globals}[a] = v)
\end{array}
$$

Note: 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.

4.5.5 Invocation

Once a *module* has been *instantiated*, any exported function can be *invoked* externally via its *function address* $\mathit{funcaddr}$ 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: $S.\mathsf{funcs}[\mathit{funcaddr}]$ exists.
2. Let $\mathit{funcinst}$ be the *function instance* $S.\mathsf{funcs}[\mathit{funcaddr}]$.
3. Let $[t_1^n] \rightarrow [t_2^m]$ be the *function type* $\mathit{funcinst}.\mathsf{type}$.
4. If the length $|\mathit{val}^*|$ 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 $t_i.\mathsf{const}\ c_i$ for some c_i, then:
      i. Fail.
6. Push the values val^* to the stack.
7. *Invoke* the function instance at address $\mathit{funcaddr}$.

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 $\mathit{val}_{\mathrm{res}}^m$ from the stack.

The values $\mathit{val}_{\mathrm{res}}^m$ are returned as the results of the invocation.

$$
\begin{array}{lcl}
\mathrm{invoke}(S, \mathit{funcaddr}, \mathit{val}^n) & = & \mathit{val}_{\mathrm{res}}^m / \mathsf{trap} \\
& (\mathrm{if} & S.\mathsf{funcs}[\mathit{funcaddr}].\mathsf{type} = [t_1^n] \rightarrow [t_2^m] \\
& \wedge & \mathit{val}^n = (t_1.\mathsf{const}\ c)^n \\
& \wedge & S; \mathit{val}^n\ (\mathsf{invoke}\ \mathit{funcaddr}) \hookrightarrow^* S'; \mathit{val}_{\mathrm{res}}^m / \mathsf{trap})
\end{array}
$$

CHAPTER 5

Binary Format

5.1 Conventions

The binary format for WebAssembly *modules* is a dense linear *encoding* of their *abstract syntax*. [19]

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.

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`".

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.)
- $B^?$ 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.
- 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 `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.

[19] 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.

Note: For example, the *binary grammar* for *value types* is given as follows:

$$\begin{array}{llcll} \mathtt{valtype} & ::= & \mathtt{0x7F} & \Rightarrow & \mathsf{i32} \\ & | & \mathtt{0x7E} & \Rightarrow & \mathsf{i64} \\ & | & \mathtt{0x7D} & \Rightarrow & \mathsf{f32} \\ & | & \mathtt{0x7C} & \Rightarrow & \mathsf{f64} \end{array}$$

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 value type.

The *binary grammar* for *limits* is defined as follows:

$$\begin{array}{llll} \mathtt{limits} & ::= & \mathtt{0x00}~~n{:}\mathtt{u32} & \Rightarrow \{\mathsf{min}~n, \mathsf{max}~\epsilon\} \\ & | & \mathtt{0x01}~~n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow \{\mathsf{min}~n, \mathsf{max}~m\} \end{array}$$

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.

$$\mathtt{vec}(\mathtt{B}) \quad ::= \quad n{:}\mathtt{u32}~~(x{:}\mathtt{B})^n \quad \Rightarrow \quad x^n$$

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[20] variable-length integer encoding, in either unsigned or signed variant.

[20] https://en.wikipedia.org/wiki/LEB128

Unsigned integers are encoded in unsigned LEB128[21] format. As an additional constraint, the total number of bytes encoding a value of type uN must not exceed $\mathrm{ceil}(N/7)$ bytes.

$$\begin{array}{llllll} \mathtt{u}N & ::= & n{:}\mathtt{byte} & \Rightarrow & n & (\text{if } n < 2^7 \wedge n < 2^N) \\ & | & n{:}\mathtt{byte}\ \ m{:}\mathtt{u}(N{-}7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (\text{if } n \geq 2^7 \wedge N > 7) \end{array}$$

Signed integers are encoded in signed LEB128[22] 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.

$$\begin{array}{llllll} \mathtt{s}N & ::= & n{:}\mathtt{byte} & \Rightarrow & n & (\text{if } n < 2^6 \wedge n < 2^{N-1}) \\ & | & n{:}\mathtt{byte} & \Rightarrow & n - 2^7 & (\text{if } 2^6 \leq n < 2^7 \wedge n \geq 2^7 - 2^{N-1}) \\ & | & n{:}\mathtt{byte}\ \ m{:}\mathtt{s}(N{-}7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (\text{if } n \geq 2^7 \wedge N > 7) \end{array}$$

Uninterpreted integers are encoded as signed integers.

$$\mathtt{i}N \quad ::= \quad n{:}\mathtt{s}N \quad \Rightarrow \quad i \qquad (\text{if } n = \mathrm{signed}_{iN}(i))$$

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[23] bit pattern in little endian[24] byte order:

$$\mathtt{f}N \quad ::= \quad b^*{:}\,\mathtt{byte}^{N/8} \quad \Rightarrow \quad \mathrm{bytes}_{fN}^{-1}(b^*)$$

5.2.4 Names

Names are encoded as a *vector* of bytes containing the Unicode[25] UTF-8 encoding of the name's code point sequence.

$$\mathtt{name} \quad ::= \quad b^*{:}\mathtt{vec}(\mathtt{byte}) \quad \Rightarrow \quad name \qquad (\text{if } \mathrm{utf8}(name) = b^*)$$

The auxiliary utf8 function expressing this encoding is defined as follows:

$$\begin{array}{llll} \mathrm{utf8}(c^*) & = & (\mathrm{utf8}(c))^* & \\ \mathrm{utf8}(c) & = & b & (\text{if } c < \mathrm{U{+}80} \\ & & & \wedge\, c = b) \\ \mathrm{utf8}(c) & = & b_1\, b_2 & (\text{if } \mathrm{U{+}80} \leq c < \mathrm{U{+}800} \\ & & & \wedge\, c = 2^6(b_1 - \mathtt{0xC0}) + (b_2 - \mathtt{0x80})) \\ \mathrm{utf8}(c) & = & b_1\, b_2\, b_3 & (\text{if } \mathrm{U{+}800} \leq c < \mathrm{U{+}10000} \\ & & & \wedge\, c = 2^{12}(b_1 - \mathtt{0xC0}) + 2^6(b_2 - \mathtt{0x80}) + (b_3 - \mathtt{0x80})) \\ \mathrm{utf8}(c) & = & b_1\, b_2\, b_3\, b_4 & (\text{if } \mathrm{U{+}10000} \leq c < \mathrm{U{+}110000} \\ & & & \wedge\, c = 2^{18}(b_1 - \mathtt{0xC0}) + 2^{12}(b_2 - \mathtt{0x80}) + 2^6(b_3 - \mathtt{0x80}) + (b_4 - \mathtt{0x80})) \end{array}$$

[21] https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128
[22] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128
[23] http://ieeexplore.ieee.org/document/4610935/
[24] https://en.wikipedia.org/wiki/Endianness#Little-endian
[25] http://www.unicode.org/versions/latest/

5.3 Types

5.3.1 Value Types

Value types are encoded by a single byte.

$$\begin{array}{llcll} \mathtt{valtype} & ::= & \mathtt{0x7F} & \Rightarrow & \mathsf{i32} \\ & | & \mathtt{0x7E} & \Rightarrow & \mathsf{i64} \\ & | & \mathtt{0x7D} & \Rightarrow & \mathsf{f32} \\ & | & \mathtt{0x7C} & \Rightarrow & \mathsf{f64} \end{array}$$

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 encodings of small negative sN values, so that they can be distinguished from (positive) type indices.

5.3.2 Result Types

Result types are encoded by the respective *vectors* of *value types* '.

$$\mathtt{resulttype} ::= t^*{:}\mathtt{vec}(\mathtt{valtype}) \Rightarrow [t^*]$$

5.3.3 Function Types

Function types are encoded by the byte `0x60` followed by the respective *vectors* of parameter and result types.

$$\mathtt{functype} ::= \mathtt{0x60}\ rt_1{:}\mathtt{resulttype}\ rt_2{:}\mathtt{resulttype} \Rightarrow rt_1 \rightarrow rt_2$$

5.3.4 Limits

Limits are encoded with a preceding flag indicating whether a maximum is present.

$$\begin{array}{llll} \mathtt{limits} & ::= & \mathtt{0x00}\ n{:}\mathtt{u32} & \Rightarrow \{\mathsf{min}\ n, \mathsf{max}\ \epsilon\} \\ & | & \mathtt{0x01}\ n{:}\mathtt{u32}\ m{:}\mathtt{u32} & \Rightarrow \{\mathsf{min}\ n, \mathsf{max}\ m\} \end{array}$$

5.3.5 Memory Types

Memory types are encoded with their *limits*.

$$\mathtt{memtype} ::= lim{:}\mathtt{limits} \Rightarrow lim$$

5.3.6 Table Types

Table types are encoded with their *limits* and a constant byte indicating their *element type*.

$$\begin{array}{llll} \mathtt{tabletype} & ::= & et{:}\mathtt{elemtype}\ lim{:}\mathtt{limits} & \Rightarrow lim\ et \\ \mathtt{elemtype} & ::= & \mathtt{0x70} & \Rightarrow \mathsf{anyfunc} \end{array}$$

5.3.7 Global Types

Global types are encoded by their *value type* and a flag for their *mutability*.

$$
\begin{array}{llcll}
\mathtt{globaltype} & ::= & t{:}\mathtt{valtype}~~m{:}\mathtt{mut} & \Rightarrow & m~t \\
\mathtt{mut} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{const} \\
& | & \mathtt{0x01} & \Rightarrow & \mathsf{var}
\end{array}
$$

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*.

$$
\begin{array}{llcll}
\mathtt{blocktype} & ::= & \mathtt{0x40} & \Rightarrow & \epsilon \\
& | & t{:}\mathtt{valtype} & \Rightarrow & t \\
& | & x{:}\mathtt{s33} & \Rightarrow & x \\
\mathtt{instr} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{unreachable} \\
& | & \mathtt{0x01} & \Rightarrow & \mathsf{nop} \\
& | & \mathtt{0x02}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{block}~bt~in^*~\mathsf{end} \\
& | & \mathtt{0x03}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{loop}~bt~in^*~\mathsf{end} \\
& | & \mathtt{0x04}~~bt{:}\mathtt{blocktype}~~(in{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{if}~bt~in^*~\mathsf{else}~\epsilon~\mathsf{end} \\
& | & \mathtt{0x04}~~bt{:}\mathtt{blocktype}~~(in_1{:}\mathtt{instr})^*~~\mathtt{0x05}~~(in_2{:}\mathtt{instr})^*~~\mathtt{0x0B} & \Rightarrow & \mathsf{if}~bt~in_1^*~\mathsf{else}~in_2^*~\mathsf{end} \\
& | & \mathtt{0x0C}~~l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br}~l \\
& | & \mathtt{0x0D}~~l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br\_if}~l \\
& | & \mathtt{0x0E}~~l^*{:}\mathtt{vec}(\mathtt{labelidx})~~l_N{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br\_table}~l^*~l_N \\
& | & \mathtt{0x0F} & \Rightarrow & \mathsf{return} \\
& | & \mathtt{0x10}~~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{call}~x \\
& | & \mathtt{0x11}~~x{:}\mathtt{typeidx} & \Rightarrow & \mathsf{call\_indirect}~x
\end{array}
$$

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 LEB128[26] 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.

[26] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

5.4.2 Parametric Instructions

Parametric instructions are represented by single byte codes.

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots & & \\
& | & \mathtt{0x1A} & \Rightarrow & \mathsf{drop} \\
& | & \mathtt{0x1B} & \Rightarrow & \mathsf{select}
\end{array}
$$

5.4.3 Variable Instructions

Variable instructions are represented by byte codes followed by the encoding of the respective *index*.

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots & & \\
& | & \mathtt{0x20}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{get\_local}~x \\
& | & \mathtt{0x21}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{set\_local}~x \\
& | & \mathtt{0x22}~~x{:}\mathtt{localidx} & \Rightarrow & \mathsf{tee\_local}~x \\
& | & \mathtt{0x23}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{get\_global}~x \\
& | & \mathtt{0x24}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{set\_global}~x
\end{array}
$$

5.4.4 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}{llcll}
\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{current\_memory} \\
& | & \mathtt{0x40}~~\mathtt{0x00} & \Rightarrow & \mathsf{grow\_memory}
\end{array}
$$

Note: In future versions of WebAssembly, the additional zero bytes occurring in the encoding of the current_memory and grow_memory instructions may be used to index additional memories.

5.4.5 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.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x45} & \Rightarrow & \mathsf{i32.eqz} \\
& | & \mathtt{0x46} & \Rightarrow & \mathsf{i32.eq} \\
& | & \mathtt{0x47} & \Rightarrow & \mathsf{i32.ne} \\
& | & \mathtt{0x48} & \Rightarrow & \mathsf{i32.lt\_s} \\
& | & \mathtt{0x49} & \Rightarrow & \mathsf{i32.lt\_u} \\
& | & \mathtt{0x4A} & \Rightarrow & \mathsf{i32.gt\_s} \\
& | & \mathtt{0x4B} & \Rightarrow & \mathsf{i32.gt\_u} \\
& | & \mathtt{0x4C} & \Rightarrow & \mathsf{i32.le\_s} \\
& | & \mathtt{0x4D} & \Rightarrow & \mathsf{i32.le\_u} \\
& | & \mathtt{0x4E} & \Rightarrow & \mathsf{i32.ge\_s} \\
& | & \mathtt{0x4F} & \Rightarrow & \mathsf{i32.ge\_u} \\
\\
& | & \mathtt{0x50} & \Rightarrow & \mathsf{i64.eqz} \\
& | & \mathtt{0x51} & \Rightarrow & \mathsf{i64.eq} \\
& | & \mathtt{0x52} & \Rightarrow & \mathsf{i64.ne} \\
& | & \mathtt{0x53} & \Rightarrow & \mathsf{i64.lt\_s} \\
& | & \mathtt{0x54} & \Rightarrow & \mathsf{i64.lt\_u} \\
& | & \mathtt{0x55} & \Rightarrow & \mathsf{i64.gt\_s} \\
& | & \mathtt{0x56} & \Rightarrow & \mathsf{i64.gt\_u} \\
& | & \mathtt{0x57} & \Rightarrow & \mathsf{i64.le\_s} \\
& | & \mathtt{0x58} & \Rightarrow & \mathsf{i64.le\_u} \\
& | & \mathtt{0x59} & \Rightarrow & \mathsf{i64.ge\_s} \\
& | & \mathtt{0x5A} & \Rightarrow & \mathsf{i64.ge\_u} \\
\\
& | & \mathtt{0x5B} & \Rightarrow & \mathsf{f32.eq} \\
& | & \mathtt{0x5C} & \Rightarrow & \mathsf{f32.ne} \\
& | & \mathtt{0x5D} & \Rightarrow & \mathsf{f32.lt} \\
& | & \mathtt{0x5E} & \Rightarrow & \mathsf{f32.gt} \\
& | & \mathtt{0x5F} & \Rightarrow & \mathsf{f32.le} \\
& | & \mathtt{0x60} & \Rightarrow & \mathsf{f32.ge} \\
\\
& | & \mathtt{0x61} & \Rightarrow & \mathsf{f64.eq} \\
& | & \mathtt{0x62} & \Rightarrow & \mathsf{f64.ne} \\
& | & \mathtt{0x63} & \Rightarrow & \mathsf{f64.lt} \\
& | & \mathtt{0x64} & \Rightarrow & \mathsf{f64.gt} \\
& | & \mathtt{0x65} & \Rightarrow & \mathsf{f64.le} \\
& | & \mathtt{0x66} & \Rightarrow & \mathsf{f64.ge}
\end{array}
$$

$$
\begin{array}{llcl}
| & \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} \\
\\
| & \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} \\
\\
| & \mathtt{0x8B} & \Rightarrow & \mathsf{f32.abs} \\
| & \mathtt{0x8C} & \Rightarrow & \mathsf{f32.neg} \\
| & \mathtt{0x8D} & \Rightarrow & \mathsf{f32.ceil} \\
| & \mathtt{0x8E} & \Rightarrow & \mathsf{f32.floor} \\
| & \mathtt{0x8F} & \Rightarrow & \mathsf{f32.trunc} \\
| & \mathtt{0x90} & \Rightarrow & \mathsf{f32.nearest} \\
| & \mathtt{0x91} & \Rightarrow & \mathsf{f32.sqrt} \\
| & \mathtt{0x92} & \Rightarrow & \mathsf{f32.add} \\
| & \mathtt{0x93} & \Rightarrow & \mathsf{f32.sub} \\
| & \mathtt{0x94} & \Rightarrow & \mathsf{f32.mul} \\
| & \mathtt{0x95} & \Rightarrow & \mathsf{f32.div} \\
| & \mathtt{0x96} & \Rightarrow & \mathsf{f32.min} \\
| & \mathtt{0x97} & \Rightarrow & \mathsf{f32.max} \\
| & \mathtt{0x98} & \Rightarrow & \mathsf{f32.copysign}
\end{array}
$$

$$
\begin{array}{llll}
| & \texttt{0x99} & \Rightarrow & \mathsf{f64.abs} \\
| & \texttt{0x9A} & \Rightarrow & \mathsf{f64.neg} \\
| & \texttt{0x9B} & \Rightarrow & \mathsf{f64.ceil} \\
| & \texttt{0x9C} & \Rightarrow & \mathsf{f64.floor} \\
| & \texttt{0x9D} & \Rightarrow & \mathsf{f64.trunc} \\
| & \texttt{0x9E} & \Rightarrow & \mathsf{f64.nearest} \\
| & \texttt{0x9F} & \Rightarrow & \mathsf{f64.sqrt} \\
| & \texttt{0xA0} & \Rightarrow & \mathsf{f64.add} \\
| & \texttt{0xA1} & \Rightarrow & \mathsf{f64.sub} \\
| & \texttt{0xA2} & \Rightarrow & \mathsf{f64.mul} \\
| & \texttt{0xA3} & \Rightarrow & \mathsf{f64.div} \\
| & \texttt{0xA4} & \Rightarrow & \mathsf{f64.min} \\
| & \texttt{0xA5} & \Rightarrow & \mathsf{f64.max} \\
| & \texttt{0xA6} & \Rightarrow & \mathsf{f64.copysign}
\end{array}
$$

$$
\begin{array}{llll}
| & \texttt{0xA7} & \Rightarrow & \mathsf{i32.wrap/i64} \\
| & \texttt{0xA8} & \Rightarrow & \mathsf{i32.trunc\_s/f32} \\
| & \texttt{0xA9} & \Rightarrow & \mathsf{i32.trunc\_u/f32} \\
| & \texttt{0xAA} & \Rightarrow & \mathsf{i32.trunc\_s/f64} \\
| & \texttt{0xAB} & \Rightarrow & \mathsf{i32.trunc\_u/f64} \\
| & \texttt{0xAC} & \Rightarrow & \mathsf{i64.extend\_s/i32} \\
| & \texttt{0xAD} & \Rightarrow & \mathsf{i64.extend\_u/i32} \\
| & \texttt{0xAE} & \Rightarrow & \mathsf{i64.trunc\_s/f32} \\
| & \texttt{0xAF} & \Rightarrow & \mathsf{i64.trunc\_u/f32} \\
| & \texttt{0xB0} & \Rightarrow & \mathsf{i64.trunc\_s/f64} \\
| & \texttt{0xB1} & \Rightarrow & \mathsf{i64.trunc\_u/f64} \\
| & \texttt{0xB2} & \Rightarrow & \mathsf{f32.convert\_s/i32} \\
| & \texttt{0xB3} & \Rightarrow & \mathsf{f32.convert\_u/i32} \\
| & \texttt{0xB4} & \Rightarrow & \mathsf{f32.convert\_s/i64} \\
| & \texttt{0xB5} & \Rightarrow & \mathsf{f32.convert\_u/i64} \\
| & \texttt{0xB6} & \Rightarrow & \mathsf{f32.demote/f64} \\
| & \texttt{0xB7} & \Rightarrow & \mathsf{f64.convert\_s/i32} \\
| & \texttt{0xB8} & \Rightarrow & \mathsf{f64.convert\_u/i32} \\
| & \texttt{0xB9} & \Rightarrow & \mathsf{f64.convert\_s/i64} \\
| & \texttt{0xBA} & \Rightarrow & \mathsf{f64.convert\_u/i64} \\
| & \texttt{0xBB} & \Rightarrow & \mathsf{f64.promote/f32} \\
| & \texttt{0xBC} & \Rightarrow & \mathsf{i32.reinterpret/f32} \\
| & \texttt{0xBD} & \Rightarrow & \mathsf{i64.reinterpret/f64} \\
| & \texttt{0xBE} & \Rightarrow & \mathsf{f32.reinterpret/i32} \\
| & \texttt{0xBF} & \Rightarrow & \mathsf{f64.reinterpret/i64}
\end{array}
$$

5.4.6 Expressions

Expressions are encoded by their instruction sequence terminated with an explicit `0x0B` opcode for end.

$$\mathtt{expr} ::= (in{:}\mathtt{instr})^* \ \texttt{0x0B} \ \Rightarrow \ in^* \ \mathsf{end}$$

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.

$$
\begin{array}{llcll}
\mathtt{typeidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{funcidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{tableidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{memidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{globalidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{localidx} & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
\mathtt{labelidx} & ::= & l{:}\mathtt{u32} & \Rightarrow & l
\end{array}
$$

5.5.2 Sections

Each section consists of

- a one-byte section *id*,
- the *u32* *size* of the contents, in bytes,
- the actual *contents*, whose structure is depended 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*

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 contents of a custom section, then errors in that contents, 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*.

$$\mathtt{typesec} \ ::= \ ft^*{:}\mathtt{section}_1(\mathtt{vec}(\mathtt{functype})) \ \Rightarrow \ ft^*$$

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*.

$$\begin{array}{llll} \mathtt{importsec} & ::= & im^*{:}\mathtt{section}_2(\mathtt{vec}(\mathtt{import})) & \Rightarrow \ im^* \\ \mathtt{import} & ::= & mod{:}\mathtt{name}\ \ nm{:}\mathtt{name}\ \ d{:}\mathtt{importdesc} & \Rightarrow \ \{\mathsf{module}\ mod, \mathsf{name}\ nm, \mathsf{desc}\ d\} \\ \mathtt{importdesc} & ::= & \mathtt{0x00}\ \ x{:}\mathtt{typeidx} & \Rightarrow \ \mathsf{func}\ x \\ & | & \mathtt{0x01}\ \ tt{:}\mathtt{tabletype} & \Rightarrow \ \mathsf{table}\ tt \\ & | & \mathtt{0x02}\ \ mt{:}\mathtt{memtype} & \Rightarrow \ \mathsf{mem}\ mt \\ & | & \mathtt{0x03}\ \ gt{:}\mathtt{globaltype} & \Rightarrow \ \mathsf{global}\ gt \end{array}$$

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*.

$$\mathtt{funcsec} \ ::= \ x^*{:}\mathtt{section}_3(\mathtt{vec}(\mathtt{typeidx})) \ \Rightarrow \ x^*$$

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*.

$$\begin{array}{llll} \mathtt{tablesec} & ::= & tab^*{:}\mathtt{section}_4(\mathtt{vec}(\mathtt{table})) & \Rightarrow \ tab^* \\ \mathtt{table} & ::= & tt{:}\mathtt{tabletype} & \Rightarrow \ \{\mathsf{type}\ tt\} \end{array}$$

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*.

$$\begin{array}{llll} \mathtt{memsec} & ::= & mem^*{:}\mathtt{section}_5(\mathtt{vec}(\mathtt{mem})) & \Rightarrow \ mem^* \\ \mathtt{mem} & ::= & mt{:}\mathtt{memtype} & \Rightarrow \ \{\mathsf{type}\ mt\} \end{array}$$

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*.

$$
\begin{array}{llclll}
\mathtt{globalsec} & ::= & \mathit{glob}^*{:}\mathtt{section}_6(\mathtt{vec}(\mathtt{global})) & \Rightarrow & \mathit{glob}^* \\
\mathtt{global} & ::= & \mathit{gt}{:}\mathtt{globaltype}~~e{:}\mathtt{expr} & \Rightarrow & \{\mathsf{type}~\mathit{gt}, \mathsf{init}~e\}
\end{array}
$$

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*.

$$
\begin{array}{llclll}
\mathtt{exportsec} & ::= & \mathit{ex}^*{:}\mathtt{section}_7(\mathtt{vec}(\mathtt{export})) & \Rightarrow & \mathit{ex}^* \\
\mathtt{export} & ::= & \mathit{nm}{:}\mathtt{name}~~d{:}\mathtt{exportdesc} & \Rightarrow & \{\mathsf{name}~\mathit{nm}, \mathsf{desc}~d\} \\
\mathtt{exportdesc} & ::= & \mathtt{0x00}~~x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{func}~x \\
 & | & \mathtt{0x01}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table}~x \\
 & | & \mathtt{0x02}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{mem}~x \\
 & | & \mathtt{0x03}~~x{:}\mathtt{globalidx} & \Rightarrow & \mathsf{global}~x
\end{array}
$$

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*.

$$
\begin{array}{llclll}
\mathtt{startsec} & ::= & \mathit{st}^?{:}\mathtt{section}_8(\mathtt{start}) & \Rightarrow & \mathit{st}^? \\
\mathtt{start} & ::= & x{:}\mathtt{funcidx} & \Rightarrow & \{\mathsf{func}~x\}
\end{array}
$$

5.5.12 Element Section

The *element section* has the id 9. It decodes into a vector of *element segments* that represent the elem component of a *module*.

$$
\begin{array}{llclll}
\mathtt{elemsec} & ::= & \mathit{seg}^*{:}\mathtt{section}_9(\mathtt{vec}(\mathtt{elem})) & \Rightarrow & \mathit{seg} \\
\mathtt{elem} & ::= & x{:}\mathtt{tableidx}~~e{:}\mathtt{expr}~~y^*{:}\mathtt{vec}(\mathtt{funcidx}) & \Rightarrow & \{\mathsf{table}~x, \mathsf{offset}~e, \mathsf{init}~y^*\}
\end{array}
$$

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 $\mathit{u32}$ *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 $\mathit{u32}$ *count*,

- a *value type*,

denoting *count* locals of the same value type.

$$
\begin{array}{llcllll}
\mathtt{codesec} & ::= & \mathit{code}^*{:}\mathtt{section}_{10}(\mathtt{vec}(\mathtt{code})) & \Rightarrow & \mathit{code}^* & \\
\mathtt{code} & ::= & \mathit{size}{:}\mathtt{u32}~~\mathit{code}{:}\mathtt{func} & \Rightarrow & \mathit{code} & (\text{if } \mathit{size} = ||\mathtt{func}||) \\
\mathtt{func} & ::= & (t^*)^*{:}\mathtt{vec}(\mathtt{locals})~~e{:}\mathtt{expr} & \Rightarrow & \mathrm{concat}((t^*)^*), e^* & (\text{if } |\mathrm{concat}((t^*)^*)| < 2^{32}) \\
\mathtt{locals} & ::= & n{:}\mathtt{u32}~~t{:}\mathtt{valtype} & \Rightarrow & t^n &
\end{array}
$$

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 data component of a *module*.

$$
\begin{array}{llcll}
\mathtt{datasec} & ::= & \mathit{seg}^*{:}\mathtt{section}_{11}(\mathtt{vec}(\mathtt{data})) & \Rightarrow & \mathit{seg} \\
\mathtt{data} & ::= & x{:}\mathtt{memidx}~~e{:}\mathtt{expr}~~b^*{:}\mathtt{vec}(\mathtt{byte}) & \Rightarrow & \{\mathsf{data}~x, \mathsf{offset}~e, \mathsf{init}~b^*\}
\end{array}
$$

5.5.15 Modules

The encoding of a *module* starts with a preamble containing a 4-byte magic number 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.

$$
\begin{array}{llll}
\mathtt{magic} & ::= & \mathtt{0x00}\ \mathtt{0x61}\ \mathtt{0x73}\ \mathtt{0x6D} & \\
\mathtt{version} & ::= & \mathtt{0x01}\ \mathtt{0x00}\ \mathtt{0x00}\ \mathtt{0x00} & \\
\mathtt{module} & ::= & \mathtt{magic} & \\
& & \mathtt{version} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{functype}^*{:}\,\mathtt{typesec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{import}^*{:}\,\mathtt{importsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{typeidx}^n{:}\,\mathtt{funcsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{table}^*{:}\,\mathtt{tablesec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{mem}^*{:}\,\mathtt{memsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{global}^*{:}\,\mathtt{globalsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{export}^*{:}\,\mathtt{exportsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{start}^?{:}\,\mathtt{startsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{elem}^*{:}\,\mathtt{elemsec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{code}^n{:}\,\mathtt{codesec} & \\
& & \mathtt{customsec}^* & \\
& & \mathit{data}^*{:}\,\mathtt{datasec} & \\
& & \mathtt{customsec}^* \quad \Rightarrow & \{\ \mathsf{types}\ \mathit{functype}^*, \\
& & & \ \ \mathsf{funcs}\ \mathit{func}^n, \\
& & & \ \ \mathsf{tables}\ \mathit{table}^*, \\
& & & \ \ \mathsf{mems}\ \mathit{mem}^*, \\
& & & \ \ \mathsf{globals}\ \mathit{global}^*, \\
& & & \ \ \mathsf{elem}\ \mathit{elem}^*, \\
& & & \ \ \mathsf{data}\ \mathit{data}^*, \\
& & & \ \ \mathsf{start}\ \mathit{start}^?, \\
& & & \ \ \mathsf{imports}\ \mathit{import}^*, \\
& & & \ \ \mathsf{exports}\ \mathit{export}^*\ \}
\end{array}
$$

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.

Text Format

6.1 Conventions

The textual format for WebAssembly *modules* is a rendering of their *abstract syntax* into S-expressions[27].

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 *identifers*.

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 source files containing WebAssembly modules in text format is "`.wat`". Files with this extension are assumed to be encoded in Unicode UTF-8[28].

6.1.1 Grammar

The following conventions are adopted in defining grammar rules for 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: '`module`'; or expressed as Unicode[29] code points: U+0A. (All characters written literally are unambiguously drawn from the 7-bit ASCII[30] 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 $n \leq 1$.)
- $x{:}T$ denotes the same language as the nonterminal T, but also binds the variable x to the attribute synthesized for 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.

[27] https://en.wikipedia.org/wiki/S-expression
[28] http://www.unicode.org/versions/latest/
[29] http://www.unicode.org/versions/latest/
[30] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

- 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.
- 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 *value types* is given as follows:

$$\begin{array}{llcll} \mathtt{valtype} & ::= & \text{‘}\mathtt{i32}\text{’} & \Rightarrow & \mathsf{i32} \\ & | & \text{‘}\mathtt{i64}\text{’} & \Rightarrow & \mathsf{i64} \\ & | & \text{‘}\mathtt{f32}\text{’} & \Rightarrow & \mathsf{f32} \\ & | & \text{‘}\mathtt{f64}\text{’} & \Rightarrow & \mathsf{f64} \end{array}$$

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 $\mathtt{u32}$ 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:

$$abbreviation~syntax \quad \equiv \quad expanded~syntax$$

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 to use symbolic *identifiers* in place of *indices*. To resolve these identifiers into concrete indices, some grammar production 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:

$$\begin{array}{llll} I & ::= & \{ \mathsf{types} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{funcs} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{tables} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{mems} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{globals} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{locals} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{labels} & (\mathtt{id}^?)^*, \\ & & \phantom{\{}\mathsf{typedefs} & functype^*~\} \end{array}$$

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.

$$\mathtt{vec}(\mathtt{A}) \quad ::= \quad (x{:}\mathtt{A})^n \quad \Rightarrow \quad x^n \qquad\qquad (\text{if } n < 2^{32})$$

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[31] *code points*.

$$\mathtt{char} \quad ::= \quad \mathrm{U{+}00} \mid \ldots \mid \mathrm{U{+}D7FF} \mid \mathrm{U{+}E000} \mid \ldots \mid \mathrm{U{+}10FFFF}$$

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[32] subset of Unicode.

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.

$$\begin{array}{lcll}
\mathtt{token} & ::= & \mathtt{keyword} \mid \mathtt{u}N \mid \mathtt{s}N \mid \mathtt{f}N \mid \mathtt{string} \mid \mathtt{id} \mid \text{‘(’} \mid \text{‘)’} \mid \mathtt{reserved} & \\
\mathtt{keyword} & ::= & (\text{‘}\mathtt{a}\text{’} \mid \ldots \mid \text{‘}\mathtt{z}\text{’})\ \mathtt{idchar}^* & (\text{if occurring as a literal terminal in the grammar}) \\
\mathtt{reserved} & ::= & \mathtt{idchar}^+ &
\end{array}$$

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.

The set of *keyword* tokens is defined implicitly, by all occurrences of a *terminal symbol* in literal form ‘`keyword`’ in a *syntactic* production of 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 or *white space*. For example, ‘`0$x`’ is a single reserved token. Consequently, it is not recognized as two separate tokens ‘`0`’ and ‘`$x`’, but instead disallowed. This property of tokenization is not affected by the fact that the definition of reserved tokens overlaps with other token classes.

---

[31] http://www.unicode.org/versions/latest/
[32] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

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[33] *format effectors*, namely, *horizontal tabulation* (U+09), *line feed* (U+0A), and *carriage return* (U+0D).

$$\begin{array}{lll} \mathtt{space} & ::= & (\text{` '} \mid \mathtt{format} \mid \mathtt{comment})^* \\ \mathtt{format} & ::= & \text{U+09} \mid \text{U+0A} \mid \text{U+0D} \end{array}$$

The only relevance of white space is to separate *tokens*, it is ignored otherwise.

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.

$$\begin{array}{llll} \mathtt{comment} & ::= & \mathtt{linecomment} \mid \mathtt{blockcomment} & \\ \mathtt{linecomment} & ::= & \text{`;;'}\ \mathtt{linechar}^*\ (\text{U+0A} \mid \mathtt{eof}) & \\ \mathtt{linechar} & ::= & c{:}\mathtt{char} & (\text{if } c \neq \text{U+0A}) \\ \mathtt{blockcomment} & ::= & \text{`(;'}\ \mathtt{blockchar}^*\ \text{`;)'} & \\ \mathtt{blockchar} & ::= & c{:}\mathtt{char} & (\text{if } c \neq \text{`;'} \wedge c \neq \text{`('}) \\ & \mid & \text{`;'} & (\text{if the next character is not `)'}) \\ & \mid & \text{`('} & (\text{if the next character is not `;'}) \\ & \mid & \mathtt{blockcomment} & \end{array}$$

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.

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.

$$\begin{array}{lll} \mathtt{sign} & ::= & \epsilon \Rightarrow + \mid \text{`+'} \Rightarrow + \mid \text{`-'} \Rightarrow - \\ \mathtt{digit} & ::= & \text{`0'} \Rightarrow 0 \mid \dots \mid \text{`9'} \Rightarrow 9 \\ \mathtt{hexdigit} & ::= & d{:}\mathtt{digit} \Rightarrow d \\ & \mid & \text{`A'} \Rightarrow 10 \mid \dots \mid \text{`F'} \Rightarrow 15 \\ & \mid & \text{`a'} \Rightarrow 10 \mid \dots \mid \text{`f'} \Rightarrow 15 \end{array}$$

$$\begin{array}{lllll} \mathtt{num} & ::= & d{:}\mathtt{digit} & \Rightarrow & d \\ & \mid & n{:}\mathtt{num}\ d{:}\mathtt{digit} & \Rightarrow & 10 \cdot n + d \\ \mathtt{hexnum} & ::= & h{:}\mathtt{hexdigit} & \Rightarrow & h \\ & \mid & n{:}\mathtt{hexnum}\ h{:}\mathtt{hexdigit} & \Rightarrow & 16 \cdot n + h \end{array}$$

The allowed syntax for integer literals depends on size and signedness. Moreover, their value must lie within the range of the respective type.

$$\begin{array}{llllll} \mathtt{u}N & ::= & n{:}\mathtt{num} & \Rightarrow & n & (\text{if } n < 2^N) \\ & \mid & \text{`0x'}\ n{:}\mathtt{hexnum} & \Rightarrow & n & (\text{if } n < 2^N) \\ \mathtt{s}N & ::= & \pm{:}\mathtt{sign}\ n{:}\mathtt{num} & \Rightarrow & \pm n & (\text{if } -2^{N-1} \leq \pm n < 2^{N-1}) \\ & \mid & \pm{:}\mathtt{sign}\ \text{`0x'}\ n{:}\mathtt{hexnum} & \Rightarrow & \pm n & (\text{if } -2^{N-1} \leq \pm n < 2^{N-1}) \end{array}$$

[33] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

Uninterpreted integers can be written as either signed or unsigned, and are normalized to unsigned in the abstract syntax.

$$\begin{array}{llcll} \mathtt{i}N & ::= & n{:}\mathtt{u}N & \Rightarrow & n \\ & | & i{:}\mathtt{s}N & \Rightarrow & n \qquad (\text{if } i = \mathrm{signed}(n)) \end{array}$$

6.3.2 Floating-Point

Floating-point values can be represented in either decimal or hexadecimal notation.

$$\begin{array}{llll} \mathtt{frac} & ::= & \epsilon & \Rightarrow 0 \\ & | & d{:}\mathtt{digit}\ q{:}\mathtt{frac} & \Rightarrow (d+q)/10 \\ \mathtt{hexfrac} & ::= & \epsilon & \Rightarrow 0 \\ & | & h{:}\mathtt{hexdigit}\ q{:}\mathtt{hexfrac} & \Rightarrow (h+q)/16 \\ \mathtt{float} & ::= & p{:}\mathtt{num}\ \text{`.'}\ q{:}\mathtt{frac} & \Rightarrow p+q \\ & | & p{:}\mathtt{num}\ (\text{`}\mathtt{E}\text{'} \mid \text{`}\mathtt{e}\text{'})\ \pm{:}\mathtt{sign}\ e{:}\mathtt{num} & \Rightarrow p \cdot 10^{\pm e} \\ & | & p{:}\mathtt{num}\ \text{`.'}\ q{:}\mathtt{frac}\ (\text{`}\mathtt{E}\text{'} \mid \text{`}\mathtt{e}\text{'})\ \pm{:}\mathtt{sign}\ e{:}\mathtt{num} & \Rightarrow (p+q) \cdot 10^{\pm e} \\ \mathtt{hexfloat} & ::= & \text{`}\mathtt{0x}\text{'}\ p{:}\mathtt{hexnum}\ \text{`.'}\ q{:}\mathtt{hexfrac} & \Rightarrow p+q \\ & | & \text{`}\mathtt{0x}\text{'}\ p{:}\mathtt{hexnum}\ (\text{`}\mathtt{P}\text{'} \mid \text{`}\mathtt{p}\text{'})\ \pm{:}\mathtt{sign}\ e{:}\mathtt{num} & \Rightarrow p \cdot 2^{\pm e} \\ & | & \text{`}\mathtt{0x}\text{'}\ p{:}\mathtt{hexnum}\ \text{`.'}\ q{:}\mathtt{hexfrac}\ (\text{`}\mathtt{P}\text{'} \mid \text{`}\mathtt{p}\text{'})\ \pm{:}\mathtt{sign}\ e{:}\mathtt{num} & \Rightarrow (p+q) \cdot 2^{\pm e} \end{array}$$

The value of a literal must not lie outside the representable range of the corresponding IEEE 754[34] type (that is, a numeric value must not overflow to ±infinity), but it may be *rounded* to the nearest representable value.

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.

$$\begin{array}{lllll} \mathtt{f}N & ::= & \pm{:}\mathtt{sign}\ z{:}\mathtt{f}N\mathtt{mag} & \Rightarrow \pm z & \\ \mathtt{f}N\mathtt{mag} & ::= & z{:}\mathtt{float} & \Rightarrow \mathrm{float}_N(z) & (\text{if } \mathrm{float}_N(z) \neq \pm\infty) \\ & | & z{:}\mathtt{hexfloat} & \Rightarrow \mathrm{float}_N(z) & (\text{if } \mathrm{float}_N(z) \neq \pm\infty) \\ & | & \text{`}\mathtt{inf}\text{'} & \Rightarrow \infty & \\ & | & \text{`}\mathtt{nan}\text{'} & \Rightarrow \mathtt{nan}(2^{\mathrm{signif}(N)-1}) & \\ & | & \text{`}\mathtt{nan{:}0x}\text{'}\ n{:}\mathtt{hexnum} & \Rightarrow \mathtt{nan}(n) & (\text{if } 1 \leq n < 2^{\mathrm{signif}(N)}) \end{array}$$

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[35] control characters, quotation marks (‘"’), or backslash (‘\’), except when expressed with an *escape sequence*.

$$\begin{array}{lllll} \mathtt{string} & ::= & \text{`}\mathtt{"}\text{'}\ (b^*{:}\mathtt{stringelem})^*\ \text{`}\mathtt{"}\text{'} & \Rightarrow \mathrm{concat}((b^*)^*) & (\text{if } |\mathrm{concat}((b^*)^*)| < 2^{32}) \\ \mathtt{stringelem} & ::= & c{:}\mathtt{stringchar} & \Rightarrow \mathrm{utf8}(c) & \\ & | & \text{`}\backslash\text{'}\ n{:}\mathtt{hexdigit}\ m{:}\mathtt{hexdigit} & \Rightarrow 16 \cdot n + m & \end{array}$$

[34] http://ieeexplore.ieee.org/document/4610935/

[35] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

Each character in a string literal represents the byte sequence corresponding to its Unicode[36] UTF-8 encoding, except for hexadecimal escape sequences '\hh', which represent raw bytes of the respective value.

$$
\begin{array}{llcll}
\mathtt{stringchar} & ::= & c{:}\mathtt{char} & \Rightarrow & c \qquad (\text{if } c \geq \text{U+20} \wedge c \neq \text{U+7F} \wedge c \neq \text{`"'} c \neq \text{`\textbackslash'}) \\
& | & \text{`\textbackslash t'} & \Rightarrow & \text{U+09} \\
& | & \text{`\textbackslash n'} & \Rightarrow & \text{U+0A} \\
& | & \text{`\textbackslash r'} & \Rightarrow & \text{U+0D} \\
& | & \text{`\textbackslash "'} & \Rightarrow & \text{U+22} \\
& | & \text{`\textbackslash ''} & \Rightarrow & \text{U+27} \\
& | & \text{`\textbackslash\textbackslash'} & \Rightarrow & \text{U+5C} \\
& | & \text{`\textbackslash u\{'}\ n{:}\mathtt{hexnum}\ \text{`\}'} & \Rightarrow & \text{U+(n)} \qquad (\text{if } n < \mathtt{0xD800} \vee \mathtt{0xE000} \leq n < \mathtt{0x110000})
\end{array}
$$

6.3.4 Names

Names are strings denoting a literal character sequence. A name string must form a valid UTF-8[37] encoding that is interpreted as a string of Unicode code points.

$$
\mathtt{name} \ ::= \ b^*{:}\mathtt{string} \ \Rightarrow \ c^* \qquad (\text{if } b^* = \mathrm{utf8}(c^*))
$$

Note: Presuming the source text is itself encoded correctly, strings that do not contain any uses of hexadecimal byte escapes are always valid names.

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[38] characters that does not contain a space, quotation mark, comma, semicolon, or bracket.

$$
\begin{array}{lll}
\mathtt{id} & ::= & \text{`\$'}\ \mathtt{idchar}^+ \\
\mathtt{idchar} & ::= & \text{`0'} \mid \dots \mid \text{`9'} \\
& | & \text{`A'} \mid \dots \mid \text{`Z'} \\
& | & \text{`a'} \mid \dots \mid \text{`z'} \\
& | & \text{`!'} \mid \text{`\#'} \mid \text{`\$'} \mid \text{`\%'} \mid \text{`\&'} \mid \text{`''} \mid \text{`*'} \mid \text{`+'} \mid \text{`-'} \mid \text{`.'} \mid \text{`/'} \\
& | & \text{`:'} \mid \text{`<'} \mid \text{`='} \mid \text{`>'} \mid \text{`?'} \mid \text{`@'} \mid \text{`\textbackslash'} \mid \text{`\^{}'} \mid \text{`\_'} \mid \text{``'} \mid \text{`|'} \mid \text{`\~{}'}
\end{array}
$$

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 Value Types

$$
\begin{array}{llcll}
\mathtt{valtype} & ::= & \text{`i32'} & \Rightarrow & \mathsf{i32} \\
& | & \text{`i64'} & \Rightarrow & \mathsf{i64} \\
& | & \text{`f32'} & \Rightarrow & \mathsf{f32} \\
& | & \text{`f64'} & \Rightarrow & \mathsf{f64}
\end{array}
$$

[36] http://www.unicode.org/versions/latest/
[37] http://www.unicode.org/versions/latest/
[38] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

6.4.2 Function Types

$$\begin{array}{llll} \mathtt{functype} & ::= & \text{`('}~\text{`}\mathtt{func}\text{'}~~t_1^*{:}\mathtt{vec}(\mathtt{param})~~t_2^*{:}\mathtt{vec}(\mathtt{result})~\text{`)'} & \Rightarrow~~[t_1^*] \to [t_2^*] \\ \mathtt{param} & ::= & \text{`('}~\text{`}\mathtt{param}\text{'}~~\mathtt{id}^?~~t{:}\mathtt{valtype}~\text{`)'} & \Rightarrow~~t \\ \mathtt{result} & ::= & \text{`('}~\text{`}\mathtt{result}\text{'}~~t{:}\mathtt{valtype}~\text{`)'} & \Rightarrow~~t \end{array}$$

Abbreviations

Multiple anonymous parameters or results may be combined into a single declaration:

$$\begin{array}{lll} \text{`('}~~\text{`}\mathtt{param}\text{'}~~\mathtt{valtype}^*~~\text{`)'} & \equiv & (\text{`('}~~\text{`}\mathtt{param}\text{'}~~\mathtt{valtype}~~\text{`)'})^* \\ \text{`('}~~\text{`}\mathtt{result}\text{'}~~\mathtt{valtype}^*~~\text{`)'} & \equiv & (\text{`('}~~\text{`}\mathtt{result}\text{'}~~\mathtt{valtype}~~\text{`)'})^* \end{array}$$

6.4.3 Limits

$$\begin{array}{llll} \mathtt{limits} & ::= & n{:}\mathtt{u32} & \Rightarrow~~\{\mathsf{min}~n, \mathsf{max}~\epsilon\} \\ & | & n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow~~\{\mathsf{min}~n, \mathsf{max}~m\} \end{array}$$

6.4.4 Memory Types

$$\mathtt{memtype} ~~::=~~ \mathit{lim}{:}\mathtt{limits} ~~\Rightarrow~~ \mathit{lim}$$

6.4.5 Table Types

$$\begin{array}{llll} \mathtt{tabletype} & ::= & \mathit{lim}{:}\mathtt{limits}~~\mathit{et}{:}\mathtt{elemtype} & \Rightarrow~~\mathit{lim}~\mathit{et} \\ \mathtt{elemtype} & ::= & \text{`}\mathtt{anyfunc}\text{'} & \Rightarrow~~\mathsf{anyfunc} \end{array}$$

Note: Additional element types may be introduced in future versions of WebAssembly.

6.4.6 Global Types

$$\begin{array}{llll} \mathtt{globaltype} & ::= & t{:}\mathtt{valtype} & \Rightarrow~~\mathsf{const}~t \\ & | & \text{`('}~~\text{`}\mathtt{mut}\text{'}~~t{:}\mathtt{valtype}~\text{`)'} & \Rightarrow~~\mathsf{var}~t \end{array}$$

6.5 Instructions

Instructions are syntactically distinguished into *plain* and *structured* instructions.

$$\begin{array}{llll} \mathtt{instr}_I & ::= & \mathit{in}{:}\mathtt{plaininstr}_I & \Rightarrow~~\mathit{in} \\ & | & \mathit{in}{:}\mathtt{blockinstr}_I & \Rightarrow~~\mathit{in} \end{array}$$

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.

$$\begin{array}{llcll} \mathtt{label}_I & ::= & v{:}\mathtt{id} & \Rightarrow & \{\mathsf{labels}~v\} \oplus I \qquad (\text{if } v \notin I.\mathsf{labels}) \\ & | & \epsilon & \Rightarrow & \{\mathsf{labels}~(\epsilon)\} \oplus I \end{array}$$

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.

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*.

$$\begin{array}{llll} \mathtt{blocktype}_I & ::= & (t{:}\mathtt{result})^? \quad \Rightarrow \quad t^? \\ & | & x, I'{:}\mathtt{typeuse}_I \quad \Rightarrow \quad x \quad (\text{if } I' = \{\}) \\ \mathtt{blockinstr}_I & ::= & \text{`}\mathtt{block}\text{'}~I'{:}\mathtt{label}_I~bt{:}\mathtt{blocktype}~(in{:}\mathtt{instr}_{I'})^*~\text{`}\mathtt{end}\text{'}~\mathtt{id}^? \\ & & \quad \Rightarrow \quad \mathsf{block}~bt~in^*~\mathsf{end} \qquad (\text{if } \mathtt{id}^? = \epsilon \vee \mathtt{id}^? = \mathtt{label}) \\ & | & \text{`}\mathtt{loop}\text{'}~I'{:}\mathtt{label}_I~bt{:}\mathtt{blocktype}~(in{:}\mathtt{instr}_{I'})^*~\text{`}\mathtt{end}\text{'}~\mathtt{id}^? \\ & & \quad \Rightarrow \quad \mathsf{loop}~bt~in^*~\mathsf{end} \qquad (\text{if } \mathtt{id}^? = \epsilon \vee \mathtt{id}^? = \mathtt{label}) \\ & | & \text{`}\mathtt{if}\text{'}~I'{:}\mathtt{label}_I~bt{:}\mathtt{blocktype}~(in_1{:}\mathtt{instr}_{I'})^*~\text{`}\mathtt{else}\text{'}~\mathtt{id}_1^?~(in_2{:}\mathtt{instr}_{I'})^*~\text{`}\mathtt{end}\text{'}~\mathtt{id}_2^? \\ & & \quad \Rightarrow \quad \mathsf{if}~bt~in_1^*~\mathsf{else}~in_2^*~\mathsf{end} \qquad (\text{if } \mathtt{id}_1^? = \epsilon \vee \mathtt{id}_1^? = \mathtt{label}, \mathtt{id}_2^? = \epsilon \vee \mathtt{id}_2^? = \mathtt{label}) \end{array}$$

Note: The side condition stating that the *identifier context* I' must be empty 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.

$$\begin{array}{llll} \mathtt{plaininstr}_I & ::= & \text{`}\mathtt{unreachable}\text{'} & \Rightarrow \quad \mathsf{unreachable} \\ & | & \text{`}\mathtt{nop}\text{'} & \Rightarrow \quad \mathsf{nop} \\ & | & \text{`}\mathtt{br}\text{'}~l{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br}~l \\ & | & \text{`}\mathtt{br\_if}\text{'}~l{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br\_if}~l \\ & | & \text{`}\mathtt{br\_table}\text{'}~l^*{:}\mathtt{vec}(\mathtt{labelidx}_I)~l_N{:}\mathtt{labelidx}_I & \Rightarrow \quad \mathsf{br\_table}~l^*~l_N \\ & | & \text{`}\mathtt{return}\text{'} & \Rightarrow \quad \mathsf{return} \\ & | & \text{`}\mathtt{call}\text{'}~x{:}\mathtt{funcidx}_I & \Rightarrow \quad \mathsf{call}~x \\ & | & \text{`}\mathtt{call\_indirect}\text{'}~x{:}\mathtt{typeidx}_I & \Rightarrow \quad \mathsf{call\_indirect}~x \end{array}$$

Abbreviations

The ‘`else`’ keyword of an ‘`if`’ instruction can be omitted if the following instruction sequence is empty.

$$\text{`}\mathtt{if}\text{'}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{`}\mathtt{end}\text{'} \quad \equiv \quad \text{`}\mathtt{if}\text{'}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{`}\mathtt{else}\text{'}~\text{`}\mathtt{end}\text{'}$$

6.5.3 Parametric Instructions

$$
\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots & & \\
& | & \text{`}\mathtt{drop}\text{'} & \Rightarrow & \mathsf{drop} \\
& | & \text{`}\mathtt{select}\text{'} & \Rightarrow & \mathsf{select}
\end{array}
$$

6.5.4 Variable Instructions

$$
\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots & & \\
& | & \text{`}\mathtt{get\_local}\text{'}~~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{get\_local}~x \\
& | & \text{`}\mathtt{set\_local}\text{'}~~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{set\_local}~x \\
& | & \text{`}\mathtt{tee\_local}\text{'}~~x{:}\mathtt{localidx}_I & \Rightarrow & \mathsf{tee\_local}~x \\
& | & \text{`}\mathtt{get\_global}\text{'}~~x{:}\mathtt{globalidx}_I & \Rightarrow & \mathsf{get\_global}~x \\
& | & \text{`}\mathtt{set\_global}\text{'}~~x{:}\mathtt{globalidx}_I & \Rightarrow & \mathsf{set\_global}~x
\end{array}
$$

6.5.5 Memory Instructions

The offset and alignment immediates to memory instructions are optional. The offset defaults to 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}{llll}
\mathtt{memarg}_N & ::= & o{:}\mathtt{offset}~~a{:}\mathtt{align}_N & \Rightarrow \{\mathsf{align}~a,~\mathsf{offset}~o\} \\
\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{current\_memory}\text{'} & \Rightarrow \mathsf{current\_memory} \\
& | & \text{`}\mathtt{grow\_memory}\text{'} & \Rightarrow \mathsf{grow\_memory}
\end{array}
$$

6.5.6 Numeric Instructions

$$
\begin{array}{llclll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{`}\mathtt{i32.const}\text{'}~~n{:}\mathtt{i32} & \Rightarrow & \mathsf{i32.const}~n \\
& | & \text{`}\mathtt{i64.const}\text{'}~~n{:}\mathtt{i64} & \Rightarrow & \mathsf{i64.const}~n \\
& | & \text{`}\mathtt{f32.const}\text{'}~~z{:}\mathtt{f32} & \Rightarrow & \mathsf{f32.const}~z \\
& | & \text{`}\mathtt{f64.const}\text{'}~~z{:}\mathtt{f64} & \Rightarrow & \mathsf{f64.const}~z \\
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\mathtt{i32.clz}\text{'} & \Rightarrow & \mathsf{i32.clz} \\
| & \text{`}\mathtt{i32.ctz}\text{'} & \Rightarrow & \mathsf{i32.ctz} \\
| & \text{`}\mathtt{i32.popcnt}\text{'} & \Rightarrow & \mathsf{i32.popcnt} \\
| & \text{`}\mathtt{i32.add}\text{'} & \Rightarrow & \mathsf{i32.add} \\
| & \text{`}\mathtt{i32.sub}\text{'} & \Rightarrow & \mathsf{i32.sub} \\
| & \text{`}\mathtt{i32.mul}\text{'} & \Rightarrow & \mathsf{i32.mul} \\
| & \text{`}\mathtt{i32.div\_s}\text{'} & \Rightarrow & \mathsf{i32.div\_s} \\
| & \text{`}\mathtt{i32.div\_u}\text{'} & \Rightarrow & \mathsf{i32.div\_u} \\
| & \text{`}\mathtt{i32.rem\_s}\text{'} & \Rightarrow & \mathsf{i32.rem\_s} \\
| & \text{`}\mathtt{i32.rem\_u}\text{'} & \Rightarrow & \mathsf{i32.rem\_u} \\
| & \text{`}\mathtt{i32.and}\text{'} & \Rightarrow & \mathsf{i32.and} \\
| & \text{`}\mathtt{i32.or}\text{'} & \Rightarrow & \mathsf{i32.or} \\
| & \text{`}\mathtt{i32.xor}\text{'} & \Rightarrow & \mathsf{i32.xor} \\
| & \text{`}\mathtt{i32.shl}\text{'} & \Rightarrow & \mathsf{i32.shl} \\
| & \text{`}\mathtt{i32.shr\_s}\text{'} & \Rightarrow & \mathsf{i32.shr\_s} \\
| & \text{`}\mathtt{i32.shr\_u}\text{'} & \Rightarrow & \mathsf{i32.shr\_u} \\
| & \text{`}\mathtt{i32.rotl}\text{'} & \Rightarrow & \mathsf{i32.rotl} \\
| & \text{`}\mathtt{i32.rotr}\text{'} & \Rightarrow & \mathsf{i32.rotr} \\
\end{array}
$$

$$
\begin{array}{llcl}
| & \text{`}\mathtt{i64.clz}\text{'} & \Rightarrow & \mathsf{i64.clz} \\
| & \text{`}\mathtt{i64.ctz}\text{'} & \Rightarrow & \mathsf{i64.ctz} \\
| & \text{`}\mathtt{i64.popcnt}\text{'} & \Rightarrow & \mathsf{i64.popcnt} \\
| & \text{`}\mathtt{i64.add}\text{'} & \Rightarrow & \mathsf{i64.add} \\
| & \text{`}\mathtt{i64.sub}\text{'} & \Rightarrow & \mathsf{i64.sub} \\
| & \text{`}\mathtt{i64.mul}\text{'} & \Rightarrow & \mathsf{i64.mul} \\
| & \text{`}\mathtt{i64.div\_s}\text{'} & \Rightarrow & \mathsf{i64.div\_s} \\
| & \text{`}\mathtt{i64.div\_u}\text{'} & \Rightarrow & \mathsf{i64.div\_u} \\
| & \text{`}\mathtt{i64.rem\_s}\text{'} & \Rightarrow & \mathsf{i64.rem\_s} \\
| & \text{`}\mathtt{i64.rem\_u}\text{'} & \Rightarrow & \mathsf{i64.rem\_u} \\
| & \text{`}\mathtt{i64.and}\text{'} & \Rightarrow & \mathsf{i64.and} \\
| & \text{`}\mathtt{i64.or}\text{'} & \Rightarrow & \mathsf{i64.or} \\
| & \text{`}\mathtt{i64.xor}\text{'} & \Rightarrow & \mathsf{i64.xor} \\
| & \text{`}\mathtt{i64.shl}\text{'} & \Rightarrow & \mathsf{i64.shl} \\
| & \text{`}\mathtt{i64.shr\_s}\text{'} & \Rightarrow & \mathsf{i64.shr\_s} \\
| & \text{`}\mathtt{i64.shr\_u}\text{'} & \Rightarrow & \mathsf{i64.shr\_u} \\
| & \text{`}\mathtt{i64.rotl}\text{'} & \Rightarrow & \mathsf{i64.rotl} \\
| & \text{`}\mathtt{i64.rotr}\text{'} & \Rightarrow & \mathsf{i64.rotr} \\
\end{array}
$$

$$
\begin{array}{llll}
| & \text{‘}\mathtt{f32.abs}\text{’} & \Rightarrow & \mathsf{f32.abs} \\
| & \text{‘}\mathtt{f32.neg}\text{’} & \Rightarrow & \mathsf{f32.neg} \\
| & \text{‘}\mathtt{f32.ceil}\text{’} & \Rightarrow & \mathsf{f32.ceil} \\
| & \text{‘}\mathtt{f32.floor}\text{’} & \Rightarrow & \mathsf{f32.floor} \\
| & \text{‘}\mathtt{f32.trunc}\text{’} & \Rightarrow & \mathsf{f32.trunc} \\
| & \text{‘}\mathtt{f32.nearest}\text{’} & \Rightarrow & \mathsf{f32.nearest} \\
| & \text{‘}\mathtt{f32.sqrt}\text{’} & \Rightarrow & \mathsf{f32.sqrt} \\
| & \text{‘}\mathtt{f32.add}\text{’} & \Rightarrow & \mathsf{f32.add} \\
| & \text{‘}\mathtt{f32.sub}\text{’} & \Rightarrow & \mathsf{f32.sub} \\
| & \text{‘}\mathtt{f32.mul}\text{’} & \Rightarrow & \mathsf{f32.mul} \\
| & \text{‘}\mathtt{f32.div}\text{’} & \Rightarrow & \mathsf{f32.div} \\
| & \text{‘}\mathtt{f32.min}\text{’} & \Rightarrow & \mathsf{f32.min} \\
| & \text{‘}\mathtt{f32.max}\text{’} & \Rightarrow & \mathsf{f32.max} \\
| & \text{‘}\mathtt{f32.copysign}\text{’} & \Rightarrow & \mathsf{f32.copysign} \\
\\
| & \text{‘}\mathtt{f64.abs}\text{’} & \Rightarrow & \mathsf{f64.abs} \\
| & \text{‘}\mathtt{f64.neg}\text{’} & \Rightarrow & \mathsf{f64.neg} \\
| & \text{‘}\mathtt{f64.ceil}\text{’} & \Rightarrow & \mathsf{f64.ceil} \\
| & \text{‘}\mathtt{f64.floor}\text{’} & \Rightarrow & \mathsf{f64.floor} \\
| & \text{‘}\mathtt{f64.trunc}\text{’} & \Rightarrow & \mathsf{f64.trunc} \\
| & \text{‘}\mathtt{f64.nearest}\text{’} & \Rightarrow & \mathsf{f64.nearest} \\
| & \text{‘}\mathtt{f64.sqrt}\text{’} & \Rightarrow & \mathsf{f64.sqrt} \\
| & \text{‘}\mathtt{f64.add}\text{’} & \Rightarrow & \mathsf{f64.add} \\
| & \text{‘}\mathtt{f64.sub}\text{’} & \Rightarrow & \mathsf{f64.sub} \\
| & \text{‘}\mathtt{f64.mul}\text{’} & \Rightarrow & \mathsf{f64.mul} \\
| & \text{‘}\mathtt{f64.div}\text{’} & \Rightarrow & \mathsf{f64.div} \\
| & \text{‘}\mathtt{f64.min}\text{’} & \Rightarrow & \mathsf{f64.min} \\
| & \text{‘}\mathtt{f64.max}\text{’} & \Rightarrow & \mathsf{f64.max} \\
| & \text{‘}\mathtt{f64.copysign}\text{’} & \Rightarrow & \mathsf{f64.copysign}
\end{array}
$$

$$
\begin{array}{llll}
| & \text{‘}\mathtt{i32.eqz}\text{’} & \Rightarrow & \mathsf{i32.eqz} \\
| & \text{‘}\mathtt{i32.eq}\text{’} & \Rightarrow & \mathsf{i32.eq} \\
| & \text{‘}\mathtt{i32.ne}\text{’} & \Rightarrow & \mathsf{i32.ne} \\
| & \text{‘}\mathtt{i32.lt\_s}\text{’} & \Rightarrow & \mathsf{i32.lt\_s} \\
| & \text{‘}\mathtt{i32.lt\_u}\text{’} & \Rightarrow & \mathsf{i32.lt\_u} \\
| & \text{‘}\mathtt{i32.gt\_s}\text{’} & \Rightarrow & \mathsf{i32.gt\_s} \\
| & \text{‘}\mathtt{i32.gt\_u}\text{’} & \Rightarrow & \mathsf{i32.gt\_u} \\
| & \text{‘}\mathtt{i32.le\_s}\text{’} & \Rightarrow & \mathsf{i32.le\_s} \\
| & \text{‘}\mathtt{i32.le\_u}\text{’} & \Rightarrow & \mathsf{i32.le\_u} \\
| & \text{‘}\mathtt{i32.ge\_s}\text{’} & \Rightarrow & \mathsf{i32.ge\_s} \\
| & \text{‘}\mathtt{i32.ge\_u}\text{’} & \Rightarrow & \mathsf{i32.ge\_u} \\
\\
| & \text{‘}\mathtt{i64.eqz}\text{’} & \Rightarrow & \mathsf{i64.eqz} \\
| & \text{‘}\mathtt{i64.eq}\text{’} & \Rightarrow & \mathsf{i64.eq} \\
| & \text{‘}\mathtt{i64.ne}\text{’} & \Rightarrow & \mathsf{i64.ne} \\
| & \text{‘}\mathtt{i64.lt\_s}\text{’} & \Rightarrow & \mathsf{i64.lt\_s} \\
| & \text{‘}\mathtt{i64.lt\_u}\text{’} & \Rightarrow & \mathsf{i64.lt\_u} \\
| & \text{‘}\mathtt{i64.gt\_s}\text{’} & \Rightarrow & \mathsf{i64.gt\_s} \\
| & \text{‘}\mathtt{i64.gt\_u}\text{’} & \Rightarrow & \mathsf{i64.gt\_u} \\
| & \text{‘}\mathtt{i64.le\_s}\text{’} & \Rightarrow & \mathsf{i64.le\_s} \\
| & \text{‘}\mathtt{i64.le\_u}\text{’} & \Rightarrow & \mathsf{i64.le\_u} \\
| & \text{‘}\mathtt{i64.ge\_s}\text{’} & \Rightarrow & \mathsf{i64.ge\_s} \\
| & \text{‘}\mathtt{i64.ge\_u}\text{’} & \Rightarrow & \mathsf{i64.ge\_u}
\end{array}
$$

$$
\begin{array}{llll}
| & \text{`}\mathtt{f32.eq}\text{'} & \Rightarrow & \mathsf{f32.eq} \\
| & \text{`}\mathtt{f32.ne}\text{'} & \Rightarrow & \mathsf{f32.ne} \\
| & \text{`}\mathtt{f32.lt}\text{'} & \Rightarrow & \mathsf{f32.lt} \\
| & \text{`}\mathtt{f32.gt}\text{'} & \Rightarrow & \mathsf{f32.gt} \\
| & \text{`}\mathtt{f32.le}\text{'} & \Rightarrow & \mathsf{f32.le} \\
| & \text{`}\mathtt{f32.ge}\text{'} & \Rightarrow & \mathsf{f32.ge} \\
\\
| & \text{`}\mathtt{f64.eq}\text{'} & \Rightarrow & \mathsf{f64.eq} \\
| & \text{`}\mathtt{f64.ne}\text{'} & \Rightarrow & \mathsf{f64.ne} \\
| & \text{`}\mathtt{f64.lt}\text{'} & \Rightarrow & \mathsf{f64.lt} \\
| & \text{`}\mathtt{f64.gt}\text{'} & \Rightarrow & \mathsf{f64.gt} \\
| & \text{`}\mathtt{f64.le}\text{'} & \Rightarrow & \mathsf{f64.le} \\
| & \text{`}\mathtt{f64.ge}\text{'} & \Rightarrow & \mathsf{f64.ge}
\end{array}
$$

$$
\begin{array}{llll}
| & \text{`}\mathtt{i32.wrap/i64}\text{'} & \Rightarrow & \mathsf{i32.wrap/i64} \\
| & \text{`}\mathtt{i32.trunc\_s/f32}\text{'} & \Rightarrow & \mathsf{i32.trunc\_s/f32} \\
| & \text{`}\mathtt{i32.trunc\_u/f32}\text{'} & \Rightarrow & \mathsf{i32.trunc\_u/f32} \\
| & \text{`}\mathtt{i32.trunc\_s/f64}\text{'} & \Rightarrow & \mathsf{i32.trunc\_s/f64} \\
| & \text{`}\mathtt{i32.trunc\_u/f64}\text{'} & \Rightarrow & \mathsf{i32.trunc\_u/f64} \\
| & \text{`}\mathtt{i64.extend\_s/i32}\text{'} & \Rightarrow & \mathsf{i64.extend\_s/i32} \\
| & \text{`}\mathtt{i64.extend\_u/i32}\text{'} & \Rightarrow & \mathsf{i64.extend\_u/i32} \\
| & \text{`}\mathtt{i64.trunc\_s/f32}\text{'} & \Rightarrow & \mathsf{i64.trunc\_s/f32} \\
| & \text{`}\mathtt{i64.trunc\_u/f32}\text{'} & \Rightarrow & \mathsf{i64.trunc\_u/f32} \\
| & \text{`}\mathtt{i64.trunc\_s/f64}\text{'} & \Rightarrow & \mathsf{i64.trunc\_s/f64} \\
| & \text{`}\mathtt{i64.trunc\_u/f64}\text{'} & \Rightarrow & \mathsf{i64.trunc\_u/f64} \\
| & \text{`}\mathtt{f32.convert\_s/i32}\text{'} & \Rightarrow & \mathsf{f32.convert\_s/i32} \\
| & \text{`}\mathtt{f32.convert\_u/i32}\text{'} & \Rightarrow & \mathsf{f32.convert\_u/i32} \\
| & \text{`}\mathtt{f32.convert\_s/i64}\text{'} & \Rightarrow & \mathsf{f32.convert\_s/i64} \\
| & \text{`}\mathtt{f32.convert\_u/i64}\text{'} & \Rightarrow & \mathsf{f32.convert\_u/i64} \\
| & \text{`}\mathtt{f32.demote/f64}\text{'} & \Rightarrow & \mathsf{f32.demote/f64} \\
| & \text{`}\mathtt{f64.convert\_s/i32}\text{'} & \Rightarrow & \mathsf{f64.convert\_s/i32} \\
| & \text{`}\mathtt{f64.convert\_u/i32}\text{'} & \Rightarrow & \mathsf{f64.convert\_u/i32} \\
| & \text{`}\mathtt{f64.convert\_s/i64}\text{'} & \Rightarrow & \mathsf{f64.convert\_s/i64} \\
| & \text{`}\mathtt{f64.convert\_u/i64}\text{'} & \Rightarrow & \mathsf{f64.convert\_u/i64} \\
| & \text{`}\mathtt{f64.demote/f32}\text{'} & \Rightarrow & \mathsf{f64.promote/f32} \\
| & \text{`}\mathtt{i32.reinterpret/f32}\text{'} & \Rightarrow & \mathsf{i32.reinterpret/f32} \\
| & \text{`}\mathtt{i64.reinterpret/f64}\text{'} & \Rightarrow & \mathsf{i64.reinterpret/f64} \\
| & \text{`}\mathtt{f32.reinterpret/i32}\text{'} & \Rightarrow & \mathsf{f32.reinterpret/i32} \\
| & \text{`}\mathtt{f64.reinterpret/i64}\text{'} & \Rightarrow & \mathsf{f64.reinterpret/i64}
\end{array}
$$

6.5.7 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 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.

$$
\begin{array}{lll}
\text{`('}~\mathtt{plaininstr}~\mathtt{foldedinstr}^*~\text{`)'} & \equiv & \mathtt{foldedinstr}^*~\mathtt{plaininstr} \\
\text{`('}~\text{`}\mathtt{block}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{instr}^*~\text{`)'} & \equiv & \text{`}\mathtt{block}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{instr}^*~\text{`}\mathtt{end}\text{'} \\
\text{`('}~\text{`}\mathtt{loop}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{instr}^*~\text{`)'} & \equiv & \text{`}\mathtt{loop}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{instr}^*~\text{`}\mathtt{end}\text{'}
\end{array}
$$

$$
\begin{array}{l}
\text{`('}~\text{`}\mathtt{if}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{foldedinstr}^*~\text{`('}~\text{`}\mathtt{then}\text{'}~\mathtt{instr}_1^*~\text{`)'}~(\text{`('}~\text{`}\mathtt{else}\text{'}~\mathtt{instr}_2^*~\text{`)'})^?~\text{`)'} \quad \equiv \\
\quad \mathtt{foldedinstr}^*~\text{`}\mathtt{if}\text{'}~\mathtt{label}~\mathtt{resulttype}~\mathtt{instr}_1^*~\text{`}\mathtt{else}\text{'}~(\mathtt{instr}_2^*)^?~\text{`}\mathtt{end}\text{'}
\end{array}
$$

Note: Folded instructions are solely syntactic sugar, no additional syntactic or type-based checking is implied.

6.5.8 Expressions

Expressions are written as instruction sequences. No explicit '`end`' keyword is included, since they only occur in bracketed positions.

$$\mathtt{expr} \ ::= \ (in{:}\mathtt{instr})^* \ \Rightarrow \ in^*\ \mathsf{end}$$

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.

$$\begin{array}{llll}
\mathtt{typeidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{types}[x] = v) \\
\mathtt{funcidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{funcs}[x] = v) \\
\mathtt{tableidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{tables}[x] = v) \\
\mathtt{memidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{mems}[x] = v) \\
\mathtt{globalidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{globals}[x] = v) \\
\mathtt{localidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow \ x \\
 & | & v{:}\mathtt{id} & \Rightarrow \ x \quad (\text{if } I.\mathsf{locals}[x] = v) \\
\mathtt{labelidx}_I & ::= & l{:}\mathtt{u32} & \Rightarrow \ l \\
 & | & v{:}\mathtt{id} & \Rightarrow \ l \quad (\text{if } I.\mathsf{labels}[l] = v)
\end{array}$$

6.6.2 Types

Type definitions can bind a symbolic *type identifier*.

$$\mathtt{type} \ ::= \ \text{`('}\ \text{`}\mathtt{type}\text{'}\ \mathtt{id}^?\ \mathit{ft}{:}\mathtt{functype}\ \text{`)'} \ \Rightarrow \ \mathit{ft}$$

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*.

$$\begin{array}{lll}
\mathtt{typeuse}_I & ::= & \text{`('}\ \text{`}\mathtt{type}\text{'}\ x{:}\mathtt{typeidx}_I\ \text{`)'} \ \Rightarrow \ x, I' \\
 & & \qquad (\text{if } I.\mathsf{typedefs}[x] = [t_1^n] \rightarrow [t_2^*] \wedge I' = \{\mathsf{locals}\ (\epsilon)^n\}) \\
 & | & \text{`('}\ \text{`}\mathtt{type}\text{'}\ x{:}\mathtt{typeidx}_I\ \text{`)'}\ (t_1{:}\mathtt{param})^*\ (t_2{:}\mathtt{result})^* \ \Rightarrow \ x, I' \\
 & & \qquad (\text{if } I.\mathsf{typedefs}[x] = [t_1^*] \rightarrow [t_2^*] \wedge I' = \{\mathsf{locals}\ \mathrm{id}(\mathtt{param})\}^*\ \text{well-formed})
\end{array}$$

The synthesized attribute of a typeuse is a pair consisting of both the used *type index* and the updated *identifier context* including 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 identifier.

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* $[t_1^*] \to [t_2^*]$. 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*.

$$\begin{array}{llll}
\mathtt{import}_I & ::= & \text{‘('}\ \text{‘}\mathtt{import}\text{'}\ \ \mathit{mod}{:}\mathtt{name}\ \ \mathit{nm}{:}\mathtt{name}\ \ d{:}\mathtt{importdesc}_I\ \text{‘)'} & \\
 & & \quad\Rightarrow\quad \{\mathsf{module}\ \mathit{mod}, \mathsf{name}\ \mathit{nm}, \mathsf{desc}\ d\} & \\
\mathtt{importdesc}_I & ::= & \text{‘('}\ \text{‘}\mathtt{func}\text{'}\ \ \mathtt{id}^?\ \ x, I'{:}\mathtt{typeuse}_I\ \text{‘)'} & \Rightarrow\quad \mathsf{func}\ x \\
 & | & \text{‘('}\ \text{‘}\mathtt{table}\text{'}\ \ \mathtt{id}^?\ \ \mathit{tt}{:}\mathtt{tabletype}\ \text{‘)'} & \Rightarrow\quad \mathsf{table}\ \mathit{tt} \\
 & | & \text{‘('}\ \text{‘}\mathtt{memory}\text{'}\ \ \mathtt{id}^?\ \ \mathit{mt}{:}\mathtt{memtype}\ \text{‘)'} & \Rightarrow\quad \mathsf{mem}\ \ \mathit{mt} \\
 & | & \text{‘('}\ \text{‘}\mathtt{global}\text{'}\ \ \mathtt{id}^?\ \ \mathit{gt}{:}\mathtt{globaltype}\ \text{‘)'} & \Rightarrow\quad \mathsf{global}\ \mathit{gt}
\end{array}$$

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.

$$\begin{array}{lll}
\mathtt{func}_I & ::= & \text{‘('}\ \text{‘}\mathtt{func}\text{'}\ \ \mathtt{id}^?\ \ x, I'{:}\mathtt{typeuse}_I\ \ (t{:}\mathtt{local})^*\ \ (\mathit{in}{:}\mathtt{instr}_{I''})^*\ \text{‘)'} \\
 & & \quad\Rightarrow\quad \{\mathsf{type}\ x, \mathsf{locals}\ t^*, \mathsf{body}\ \mathit{in}^*\ \mathsf{end}\} \\
 & & \qquad\qquad (\text{if } I'' = I' \oplus \{\mathsf{locals}\ \mathrm{id}(\mathtt{local})^*\}\ \text{well-formed}) \\
\mathtt{local} & ::= & \text{‘('}\ \text{‘}\mathtt{local}\text{'}\ \ \mathtt{id}^?\ \ t{:}\mathtt{valtype}\ \text{‘)'} \quad\Rightarrow\quad t
\end{array}$$

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:

$$\text{`('}\ \text{`}\mathtt{local}\text{'}\ \mathtt{valtype}^*\ \text{`)'} \quad \equiv \quad (\text{`('}\ \text{`}\mathtt{local}\text{'}\ \mathtt{valtype}\ \text{`)'})^*$$

Moreover, functions can be defined as *imports* or *exports* inline:

$$\begin{array}{l}
\text{`('}\ \text{`}\mathtt{func}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`)'}\ \mathtt{typeuse}\ \text{`)'} \quad \equiv \\
\qquad \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`('}\ \text{`}\mathtt{func}\text{'}\ \mathtt{id}^?\ \mathtt{typeuse}\ \text{`)'}\ \text{`)'} \\
\text{`('}\ \text{`}\mathtt{func}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`)'}\ \dots\ \text{`)'} \quad \equiv \\
\qquad \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`('}\ \text{`}\mathtt{func}\text{'}\ \mathtt{id}'\ \text{`)'}\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{func}\text{'}\ \mathtt{id}'\ \dots\ \text{`)'} \\
\qquad\qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}' \text{ fresh})
\end{array}$$

The latter abbreviation can be applied repeatedly, with "…" containing another import or export.

6.6.6 Tables

Table definitions can bind a symbolic *table identifier*.

$$\mathtt{table}_I \quad ::= \quad \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathit{tt}{:}\mathtt{tabletype}\ \text{`)'} \quad \Rightarrow \quad \{\mathsf{type}\ \mathit{tt}\}$$

Abbreviations

An *element segment* can be given inline with a table definition, in which case the *limits* of the *table type* are inferred from the length of the given segment:

$$\begin{array}{l}
\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathtt{elemtype}\ \text{`('}\ \text{`}\mathtt{elem}\text{'}\ x^n{:}\mathtt{vec}(\mathtt{funcidx})\ \text{`)'}\ \text{`)'} \quad \equiv \\
\qquad \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}'\ n\ n\ \mathtt{elemtype}\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{elem}\text{'}\ \mathtt{id}'\ \text{`('}\ \text{`}\mathtt{i32.const}\text{'}\ \text{`0'}\ \text{`)'}\ \mathtt{vec}(\mathtt{funcidx})\ \text{`)'} \\
\qquad\qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}' \text{ fresh})
\end{array}$$

Moreover, tables can be defined as *imports* or *exports* inline:

$$\begin{array}{l}
\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`)'}\ \mathtt{tabletype}\ \text{`)'} \quad \equiv \\
\qquad \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathtt{tabletype}\ \text{`)'}\ \text{`)'} \\
\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`)'}\ \dots\ \text{`)'} \quad \equiv \\
\qquad \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}'\ \text{`)'}\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}'\ \dots\ \text{`)'} \\
\qquad\qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}' \text{ fresh})
\end{array}$$

The latter abbreviation can be applied repeatedly, with "…" containing another import or export or an inline elements segment.

6.6.7 Memories

Memory definitions can bind a symbolic *memory identifier*.

$$\mathtt{mem}_I \ ::= \ \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}^?\ mt{:}\mathtt{memtype}\ \text{`)'} \ \Rightarrow\ \{\mathsf{type}\ mt\}$$

Abbreviations

A *data segment* can be given inline with a memory definition, in which case the *limits* of the *memory type* are inferred from the length of the data, rounded up to *page size*:

$$\begin{array}{l} \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{data}\text{'}\ b^n{:}\mathtt{datastring}\ \text{`)'}\ \text{`)'} \quad \equiv \\ \quad \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}'\ m\ m\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{data}\text{'}\ \mathtt{id}'\ \text{`('}\ \text{`}\mathtt{i32.const}\text{'}\ \text{`}\mathtt{0}\text{'}\ \text{`)'}\ \mathtt{datastring}\ \text{`)'} \\ \qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}'\ \mathrm{fresh}, m = \mathrm{ceil}(n/64\mathrm{Ki})) \end{array}$$

Moreover, memories can be defined as *imports* or *exports* inline:

$$\begin{array}{l} \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`)'}\ \mathtt{memtype}\ \text{`)'} \quad \equiv \\ \quad \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}^?\ \mathtt{memtype}\ \text{`)'}\ \text{`)'} \\ \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`)'}\ \dots\ \text{`)'} \quad \equiv \\ \quad \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}'\ \text{`)'}\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{memory}\text{'}\ \mathtt{id}'\ \dots\ \text{`)'} \\ \qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}'\ \mathrm{fresh}) \end{array}$$

The latter abbreviation can be applied repeatedly, with "…" containing another import or export or an inline data segment.

6.6.8 Globals

Global definitions can bind a symbolic *global identifier*.

$$\mathtt{global}_I \ ::= \ \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}^?\ gt{:}\mathtt{globaltype}\ e{:}\mathtt{expr}_I\ \text{`)'} \ \Rightarrow\ \{\mathsf{type}\ gt, \mathsf{init}\ e\}$$

Abbreviations

Globals can be defined as *imports* or *exports* inline:

$$\begin{array}{l} \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`)'}\ \mathtt{globaltype}\ \text{`)'} \quad \equiv \\ \quad \text{`('}\ \text{`}\mathtt{import}\text{'}\ \mathtt{name}_1\ \mathtt{name}_2\ \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}^?\ \mathtt{globaltype}\ \text{`)'}\ \text{`)'} \\ \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}^?\ \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`)'}\ \dots\ \text{`)'} \quad \equiv \\ \quad \text{`('}\ \text{`}\mathtt{export}\text{'}\ \mathtt{name}\ \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}'\ \text{`)'}\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{global}\text{'}\ \mathtt{id}'\ \dots\ \text{`)'} \\ \qquad (\text{if } \mathtt{id}' = \mathtt{id}^? \neq \epsilon \vee \mathtt{id}'\ \mathrm{fresh}) \end{array}$$

The latter abbreviation can be applied repeatedly, with "…" containing another import or export.

6.6.9 Exports

The syntax for exports mirrors their *abstract syntax* directly.

$$\begin{array}{llll} \mathtt{export}_I & ::= & \text{`('}\ \text{`}\mathtt{export}\text{'}\ nm{:}\mathtt{name}\ d{:}\mathtt{exportdesc}_I\ \text{`)'} & \Rightarrow\ \{\mathsf{name}\ nm, \mathsf{desc}\ d\} \\ \mathtt{exportdesc}_I & ::= & \text{`('}\ \text{`}\mathtt{func}\text{'}\ x{:}\mathtt{funcidx}_I\ \text{`)'} & \Rightarrow\ \mathsf{func}\ x \\ & | & \text{`('}\ \text{`}\mathtt{table}\text{'}\ x{:}\mathtt{tableidx}_I\ \text{`)'} & \Rightarrow\ \mathsf{table}\ x \\ & | & \text{`('}\ \text{`}\mathtt{memory}\text{'}\ x{:}\mathtt{memidx}_I\ \text{`)'} & \Rightarrow\ \mathsf{mem}\ x \\ & | & \text{`('}\ \text{`}\mathtt{global}\text{'}\ x{:}\mathtt{globalidx}_I\ \text{`)'} & \Rightarrow\ \mathsf{global}\ x \end{array}$$

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.

$$\mathtt{start}_I \ ::= \ \text{`('}\ \text{`}\mathtt{start}\text{'}\ x{:}\mathtt{funcidx}_I\ \text{`)'} \ \Rightarrow \ \{\mathsf{func}\ x\}$$

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. When omitted, `0` is assumed.

$$\begin{array}{llll}
\mathtt{elem}_I & ::= & \multicolumn{2}{l}{\text{`('}\ \text{`}\mathtt{elem}\text{'}\ (x{:}\mathtt{tableidx}_I)^?\ \text{`('}\ \text{`}\mathtt{offset}\text{'}\ e{:}\mathtt{expr}_I\ \text{`)'}\ y^*{:}\mathtt{vec}(\mathtt{funcidx}_I)\ \text{`)'}} \\
&& \Rightarrow & \{\mathsf{table}\ x', \mathsf{offset}\ e, \mathsf{init}\ y^*\} \\
&&& \quad (\mathrm{if}\ x' = x^? \neq \epsilon \vee x' = 0)
\end{array}$$

Note: In the current version of WebAssembly, the only valid table index is 0 or a symbolic *table identifier* resolving to the same value.

Abbreviations

As an 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. When omitted, `0` is assumed. The data is written as a *string*, which may be split up into a possibly empty sequence of individual string literals.

$$\begin{array}{llll}
\mathtt{data}_I & ::= & \multicolumn{2}{l}{\text{`('}\ \text{`}\mathtt{data}\text{'}\ (x{:}\mathtt{memidx}_I)^?\ \text{`('}\ \text{`}\mathtt{offset}\text{'}\ e{:}\mathtt{expr}_I\ \text{`)'}\ b^*{:}\mathtt{datastring}\ \text{`)'}} \\
&& \Rightarrow & \{\mathsf{data}\ x', \mathsf{offset}\ e, \mathsf{init}\ b^*\} \\
&&& \quad (\mathrm{if}\ x' = x^? \neq \epsilon \vee x' = 0) \\
\mathtt{datastring} & ::= & (b^*{:}\mathtt{string})^* \ \Rightarrow & \mathrm{concat}((b^*)^*)
\end{array}$$

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, 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*.

$$
\begin{array}{llcll}
\mathtt{module} & ::= & \text{‘(’}~\text{‘}\mathtt{module}\text{’}~\mathtt{id}^?~(m{:}\mathtt{modulefield}_I)^*~\text{‘)’} & \Rightarrow & \bigoplus m^* \\
& & \qquad (\text{if } I = \bigoplus \mathrm{idc}(\mathtt{modulefield})^* ~\text{well-formed}) & & \\
\mathtt{modulefield}_I & ::= & ty{:}\mathtt{type} & \Rightarrow & \{\mathsf{types}~ty\} \\
& | & im{:}\mathtt{import}_I & \Rightarrow & \{\mathsf{imports}~im\} \\
& | & fn{:}\mathtt{func}_I & \Rightarrow & \{\mathsf{funcs}~fn\} \\
& | & ta{:}\mathtt{table}_I & \Rightarrow & \{\mathsf{tables}~ta\} \\
& | & me{:}\mathtt{mem}_I & \Rightarrow & \{\mathsf{mems}~me\} \\
& | & gl{:}\mathtt{global}_I & \Rightarrow & \{\mathsf{globals}~gl\} \\
& | & ex{:}\mathtt{export}_I & \Rightarrow & \{\mathsf{exports}~ex\} \\
& | & st{:}\mathtt{start}_I & \Rightarrow & \{\mathsf{start}~st\} \\
& | & el{:}\mathtt{elem}_I & \Rightarrow & \{\mathsf{elem}~el\} \\
& | & da{:}\mathtt{data}_I & \Rightarrow & \{\mathsf{data}~da\}
\end{array}
$$

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:

$$
\begin{array}{lcl}
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{type}\text{’}~\mathtt{id}^?~ft{:}\mathtt{functype}~\text{‘)’}) & = & \{\mathsf{types}~(\mathtt{id}^?), \mathsf{typedefs}~ft\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}) & = & \{\mathsf{funcs}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}) & = & \{\mathsf{tables}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{memory}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}) & = & \{\mathsf{mems}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{global}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}) & = & \{\mathsf{globals}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\dots~\text{‘(’}~\text{‘}\mathtt{func}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}~\text{‘)’}) & = & \{\mathsf{funcs}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\dots~\text{‘(’}~\text{‘}\mathtt{table}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}~\text{‘)’}) & = & \{\mathsf{tables}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\dots~\text{‘(’}~\text{‘}\mathtt{memory}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}~\text{‘)’}) & = & \{\mathsf{mems}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\text{‘}\mathtt{import}\text{’}~\dots~\text{‘(’}~\text{‘}\mathtt{global}\text{’}~\mathtt{id}^?~\dots~\text{‘)’}~\text{‘)’}) & = & \{\mathsf{globals}~(\mathtt{id}^?)\} \\
\mathrm{idc}(\text{‘(’}~\dots~\text{‘)’}) & = & \{\}
\end{array}
$$

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 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.

7.1.1 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* form a *module*
- the number of parameters in a *function type*
- the number of results in a *function 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 *code points* 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[39] to limit the names of *imports* and *exports* to common subsets like ASCII[40].

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* (code points) allowed in the *source text*

7.1.2 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.

---

[39] http://www.unicode.org/versions/latest/

[40] http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

7.1.3 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 *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.

7.2 Name Section

Todo

Describe

7.3 Formal Properties

7.3.1 Representation

Todo

bijection between abstract and binary

7.3.2 Validation

Todo

progress, preservation

7.4 Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

Index of Instructions

Instruction	Opcode	Type	Validation	Execution
unreachable	0x00	$[t_1^*] \to [t_2^*]$	*validation*	*execution*
nop	0x01	$[] \to []$	*validation*	*execution*
block	0x02	$[t_1^*] \to [t_2^*]$	*validation*	*execution*
loop	0x03	$[t_1^*] \to [t_2^*]$	*validation*	*execution*
if	0x04	$[t_1^*] \to [t_2^*]$	*validation*	*execution*
else	0x05			
(reserved)	0x06			
(reserved)	0x07			
(reserved)	0x08			
(reserved)	0x09			
(reserved)	0x0A			
end	0x0B			
br l	0x0C	$[t_1^*\ t^*] \to [t_2^*]$	*validation*	*execution*
br_if l	0x0D	$[t^*\ \mathsf{i32}] \to [t^*]$	*validation*	*execution*
br_table $l^*\ l$	0x0E	$[t_1^*\ t^*\ \mathsf{i32}] \to [t_2^*]$	*validation*	*execution*
return	0x0F	$[t_1^*\ t^*] \to [t_2^*]$	*validation*	*execution*
call x	0x10	$[t_1^*] \to [t_2^*]$	*validation*	*execution*
call_indirect x	0x11	$[t_1^*\ \mathsf{i32}] \to [t_2^*]$	*validation*	*execution*
(reserved)	0x12			
(reserved)	0x13			
(reserved)	0x14			
(reserved)	0x15			
(reserved)	0x16			
(reserved)	0x17			
(reserved)	0x18			
(reserved)	0x19			
drop	0x1A	$[t] \to []$	*validation*	*execution*
select	0x1B	$[t\ t\ \mathsf{i32}] \to [t]$	*validation*	*execution*
(reserved)	0x1C			
(reserved)	0x1D			
(reserved)	0x1E			
(reserved)	0x1F			
get_local x	0x20	$[] \to [t]$	*validation*	*execution*
set_local x	0x21	$[t] \to []$	*validation*	*execution*
tee_local x	0x22	$[t] \to [t]$	*validation*	*execution*
get_global x	0x23	$[] \to [t]$	*validation*	*execution*
set_global x	0x24	$[t] \to []$	*validation*	*execution*
(reserved)	0x25			
(reserved)	0x26			
(reserved)	0x27			

Continued on next page

Table 8.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
i32.load *memarg*	0x28	[i32] → [i32]	*validation*	*execution*
i64.load *memarg*	0x29	[i32] → [i64]	*validation*	*execution*
f32.load *memarg*	0x2A	[i32] → [f32]	*validation*	*execution*
f64.load *memarg*	0x2B	[i32] → [f64]	*validation*	*execution*
i32.load8_s *memarg*	0x2C	[i32] → [i32]	*validation*	*execution*
i32.load8_u *memarg*	0x2D	[i32] → [i32]	*validation*	*execution*
i32.load16_s *memarg*	0x2E	[i32] → [i32]	*validation*	*execution*
i32.load16_u *memarg*	0x2F	[i32] → [i32]	*validation*	*execution*
i64.load8_s *memarg*	0x30	[i32] → [i64]	*validation*	*execution*
i64.load8_u *memarg*	0x31	[i32] → [i64]	*validation*	*execution*
i64.load16_s *memarg*	0x32	[i32] → [i64]	*validation*	*execution*
i64.load16_u *memarg*	0x33	[i32] → [i64]	*validation*	*execution*
i64.load32_s *memarg*	0x34	[i32] → [i64]	*validation*	*execution*
i64.load32_u *memarg*	0x35	[i32] → [i64]	*validation*	*execution*
i32.store *memarg*	0x36	[i32 i32] → []	*validation*	*execution*
i64.store *memarg*	0x37	[i32 i64] → []	*validation*	*execution*
f32.store *memarg*	0x38	[i32 f32] → []	*validation*	*execution*
f64.store *memarg*	0x39	[i32 f64] → []	*validation*	*execution*
i32.store8 *memarg*	0x3A	[i32 i32] → []	*validation*	*execution*
i32.store16 *memarg*	0x3B	[i32 i32] → []	*validation*	*execution*
i64.store8 *memarg*	0x3C	[i32 i64] → []	*validation*	*execution*
i64.store16 *memarg*	0x3D	[i32 i64] → []	*validation*	*execution*
i64.store32 *memarg*	0x3E	[i32 i64] → []	*validation*	*execution*
current_memory	0x3F	[] → [i32]	*validation*	*execution*
grow_memory	0x40	[i32] → [i32]	*validation*	*execution*
i32.const *i32*	0x41	[] → [i32]	*validation*	*execution*
i64.const *i64*	0x42	[] → [i64]	*validation*	*execution*
f32.const *f32*	0x43	[] → [f32]	*validation*	*execution*
f64.const *f64*	0x44	[] → [f64]	*validation*	*execution*
i32.eqz	0x45	[i32] → [i32]	*validation*	*execution*, *operator*
i32.eq	0x46	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.ne	0x47	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.lt_s	0x48	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.lt_u	0x49	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.gt_s	0x4A	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.gt_u	0x4B	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.le_s	0x4C	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.le_u	0x4D	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.ge_s	0x4E	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.ge_u	0x4F	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i64.eqz	0x50	[i64] → [i32]	*validation*	*execution*, *operator*
i64.eq	0x51	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.ne	0x52	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.lt_s	0x53	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.lt_u	0x54	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.gt_s	0x55	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.gt_u	0x56	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.le_s	0x57	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.le_u	0x58	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.ge_s	0x59	[i64 i64] → [i32]	*validation*	*execution*, *operator*
i64.ge_u	0x5A	[i64 i64] → [i32]	*validation*	*execution*, *operator*
f32.eq	0x5B	[f32 f32] → [i32]	*validation*	*execution*, *operator*
f32.ne	0x5C	[f32 f32] → [i32]	*validation*	*execution*, *operator*

Continued on next page

Table 8.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
f32.lt	0x5D	[f32 f32] → [i32]	*validation*	*execution*, *operator*
f32.gt	0x5E	[f32 f32] → [i32]	*validation*	*execution*, *operator*
f32.le	0x5F	[f32 f32] → [i32]	*validation*	*execution*, *operator*
f32.ge	0x60	[f32 f32] → [i32]	*validation*	*execution*, *operator*
f64.eq	0x61	[f64 f64] → [i32]	*validation*	*execution*, *operator*
f64.ne	0x62	[f64 f64] → [i32]	*validation*	*execution*, *operator*
f64.lt	0x63	[f64 f64] → [i32]	*validation*	*execution*, *operator*
f64.gt	0x64	[f64 f64] → [i32]	*validation*	*execution*, *operator*
f64.le	0x65	[f64 f64] → [i32]	*validation*	*execution*, *operator*
f64.ge	0x66	[f64 f64] → [i32]	*validation*	*execution*, *operator*
i32.clz	0x67	[i32] → [i32]	*validation*	*execution*, *operator*
i32.ctz	0x68	[i32] → [i32]	*validation*	*execution*, *operator*
i32.popcnt	0x69	[i32] → [i32]	*validation*	*execution*, *operator*
i32.add	0x6A	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.sub	0x6B	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.mul	0x6C	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.div_s	0x6D	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.div_u	0x6E	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.rem_s	0x6F	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.rem_u	0x70	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.and	0x71	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.or	0x72	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.xor	0x73	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.shl	0x74	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.shr_s	0x75	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.shr_u	0x76	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.rotl	0x77	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i32.rotr	0x78	[i32 i32] → [i32]	*validation*	*execution*, *operator*
i64.clz	0x79	[i64] → [i64]	*validation*	*execution*, *operator*
i64.ctz	0x7A	[i64] → [i64]	*validation*	*execution*, *operator*
i64.popcnt	0x7B	[i64] → [i64]	*validation*	*execution*, *operator*
i64.add	0x7C	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.sub	0x7D	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.mul	0x7E	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.div_s	0x7F	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.div_u	0x80	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.rem_s	0x81	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.rem_u	0x82	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.and	0x83	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.or	0x84	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.xor	0x85	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.shl	0x86	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.shr_s	0x87	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.shr_u	0x88	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.rotl	0x89	[i64 i64] → [i64]	*validation*	*execution*, *operator*
i64.rotr	0x8A	[i64 i64] → [i64]	*validation*	*execution*, *operator*
f32.abs	0x8B	[f32] → [f32]	*validation*	*execution*, *operator*
f32.neg	0x8C	[f32] → [f32]	*validation*	*execution*, *operator*
f32.ceil	0x8D	[f32] → [f32]	*validation*	*execution*, *operator*
f32.floor	0x8E	[f32] → [f32]	*validation*	*execution*, *operator*
f32.trunc	0x8F	[f32] → [f32]	*validation*	*execution*, *operator*
f32.nearest	0x90	[f32] → [f32]	*validation*	*execution*, *operator*
f32.sqrt	0x91	[f32] → [f32]	*validation*	*execution*, *operator*

Continued on next page

Table 8.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
f32.add	0x92	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.sub	0x93	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.mul	0x94	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.div	0x95	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.min	0x96	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.max	0x97	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f32.copysign	0x98	[f32 f32] → [f32]	*validation*	*execution*, *operator*
f64.abs	0x99	[f64] → [f64]	*validation*	*execution*, *operator*
f64.neg	0x9A	[f64] → [f64]	*validation*	*execution*, *operator*
f64.ceil	0x9B	[f64] → [f64]	*validation*	*execution*, *operator*
f64.floor	0x9C	[f64] → [f64]	*validation*	*execution*, *operator*
f64.trunc	0x9D	[f64] → [f64]	*validation*	*execution*, *operator*
f64.nearest	0x9E	[f64] → [f64]	*validation*	*execution*, *operator*
f64.sqrt	0x9F	[f64] → [f64]	*validation*	*execution*, *operator*
f64.add	0xA0	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.sub	0xA1	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.mul	0xA2	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.div	0xA3	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.min	0xA4	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.max	0xA5	[f64 f64] → [f64]	*validation*	*execution*, *operator*
f64.copysign	0xA6	[f64 f64] → [f64]	*validation*	*execution*, *operator*
i32.wrap/i64	0xA7	[i64] → [i32]	*validation*	*execution*, *operator*
i32.trunc_s/f32	0xA8	[f32] → [i32]	*validation*	*execution*, *operator*
i32.trunc_u/f32	0xA9	[f32] → [i32]	*validation*	*execution*, *operator*
i32.trunc_s/f64	0xAA	[f64] → [i32]	*validation*	*execution*, *operator*
i32.trunc_u/f64	0xAB	[f64] → [i32]	*validation*	*execution*, *operator*
i64.extend_s/i32	0xAC	[i32] → [i64]	*validation*	*execution*, *operator*
i64.extend_u/i32	0xAD	[i32] → [i64]	*validation*	*execution*, *operator*
i64.trunc_s/f32	0xAE	[f32] → [i64]	*validation*	*execution*, *operator*
i64.trunc_u/f32	0xAF	[f32] → [i64]	*validation*	*execution*, *operator*
i64.trunc_s/f64	0xB0	[f64] → [i64]	*validation*	*execution*, *operator*
i64.trunc_u/f64	0xB1	[f64] → [i64]	*validation*	*execution*, *operator*
f32.convert_s/i32	0xB2	[i32] → [f32]	*validation*	*execution*, *operator*
f32.convert_u/i32	0xB3	[i32] → [f32]	*validation*	*execution*, *operator*
f32.convert_s/i64	0xB4	[i64] → [f32]	*validation*	*execution*, *operator*
f32.convert_u/i64	0xB5	[i64] → [f32]	*validation*	*execution*, *operator*
f32.demote/f64	0xB6	[f64] → [f32]	*validation*	*execution*, *operator*
f64.convert_s/i32	0xB7	[i32] → [f64]	*validation*	*execution*, *operator*
f64.convert_u/i32	0xB8	[i32] → [f64]	*validation*	*execution*, *operator*
f64.convert_s/i64	0xB9	[i64] → [f64]	*validation*	*execution*, *operator*
f64.convert_u/i64	0xBA	[i64] → [f64]	*validation*	*execution*, *operator*
f64.promote/f32	0xBB	[f32] → [f64]	*validation*	*execution*, *operator*
i32.reinterpret/f32	0xBC	[f32] → [i32]	*validation*	*execution*, *operator*
i64.reinterpret/f64	0xBD	[f64] → [i64]	*validation*	*execution*, *operator*
f32.reinterpret/i32	0xBE	[i32] → [f32]	*validation*	*execution*, *operator*
f64.reinterpret/i64	0xBF	[i64] → [f64]	*validation*	*execution*, *operator*

Symbols

K

L

M

N

O

P

R

S

T

W