WebAssembly Specification

Release 3.0 (Draft 2024-07-10)

WebAssembly Community Group

Andreas Rossberg (editor)

Jul 10, 2024

Contents

CHAPTER 1

Introduction

1.1 Introduction

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

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

This document describes version 3.0 (Draft 2024-07-10) of the core WebAssembly standard. It is intended that it will be superseded by new incremental releases with additional features in the future.

1.1.1 Design Goals

The design goals of WebAssembly are the following:

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

[2] A contraction of "WebAssembly", not an acronym, hence not using all-caps.
[1] https://www.w3.org/community/webassembly/
[3] No program can break WebAssembly's memory model. Of course, it cannot guarantee that an unsafe language compiling to WebAssembly does not corrupt its own memory layout, e.g. inside WebAssembly's linear memory.

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

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

1.1.2 Scope

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

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

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

1.1.3 Security Considerations

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

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

1.1.4 Dependencies

WebAssembly depends on two existing standards:

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

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

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

1.2 Overview

1.2.1 Concepts

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

Values

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

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

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

Instructions

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

Traps

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

Functions

Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns

[4] https://ieeexplore.ieee.org/document/8766229
[5] https://www.unicode.org/versions/latest/
[6] https://ieeexplore.ieee.org/document/8766229
[7] https://ieeexplore.ieee.org/document/8766229
[8] In practice, implementations need not maintain an actual operand stack. Instead, the stack can be viewed as a set of anonymous registers that are implicitly referenced by instructions. The type system ensures that the stack height, and thus any referenced register, is always known statically.

a sequence of values as results. Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable *local variables* that are usable as virtual registers.

Tables
A *table* is an array of opaque values of a particular *reference type*. It allows programs to select such values indirectly through a dynamic index operand. Thereby, for example, a program can call functions indirectly through a dynamic index into a table. This allows emulating function pointers by way of table indices.

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

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

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

1.2.2 Semantic Phases

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

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

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

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

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

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

Instantiation and invocation are operations within the embedding environment.

CHAPTER 2

Structure

2.1 Conventions

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

2.1.1 Grammar Notation

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

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

2.1.2 Auxiliary Notation

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

- ϵ denotes the empty sequence.
- $|s|$ denotes the length of a sequence s.
- $s[i]$ denotes the i-th element of a sequence s, starting from 0.
- $s[i:n]$ denotes the sub-sequence $s[i] \dots s[i+n-1]$ of a sequence s.
- $s[[i] = A]$ denotes the same sequence as s, except that the i-th element is replaced with A.
- $s[[i:n] = A^n]$ denotes the same sequence as s, except that the sub-sequence $s[i:n]$ is replaced with A^n.
- $s_1 \oplus s_2$ denotes the sequence s_1 concatenated with s_2; this is equivalent to $s_1\ s_2$, but used for clarity.
- $\bigoplus s^*$ denotes the flat sequence formed by concatenating all sequences s_i in s^*.
- $A \in s$ denotes that A is contained in the sequence s, that is, s is of the form $s_1\ A\ s_2$ for some sequences s_1, s_2.

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 an iterated sequence $(\dots x \dots)^n$ 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:

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

The following notation is adopted for manipulating such records:

- Where the type of a record is clear from context, empty fields with value ϵ are often omitted.
- $r.\mathsf{field}$ denotes the contents of the field component of r.
- $r[.\mathsf{field} = A]$ denotes the same record as r, except that the value of the field component is replaced with A.
- $r[.\mathsf{field} =\oplus\ A^*]$ denotes the same record as r, except that A^* is appended to the sequence value of the field component, that is, it is short for $r[.\mathsf{field} = r.\mathsf{field} \oplus A^*]$.
- $r_1 \oplus r_2$ denotes the composition of two identically shaped records by concatenating each field of sequences point-wise:

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

- $\bigoplus r^*$ 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" $pth ::= ([i] \mid .\mathsf{field})^+$:

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

2.1.3 Lists

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

$$\mathit{list}(X) \quad ::= \quad X^* \qquad \text{if } |X^*| < 2^{32}$$

2.2 Values

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

2.2.1 Bytes

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

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

Conventions

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

2.2.2 Integers

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

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

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

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

Conventions

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

2.2.3 Floating-Point

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

Every value has a *sign* and a *magnitude*. Magnitudes can either be expressed as *normal* numbers of the form $m_0 . m_1 m_2 \ldots m_\mathsf{m} \cdot 2^e$, where e is the exponent and m is the *significand* whose most significant bit m_0 is 1, or as a *subnormal* number where the exponent is fixed to the smallest possible value and m_0 is 0; among the subnormals are positive and negative zero values. Since the significands are binary values, normals are represented in the form $(1 + m \cdot 2^{-M}) \cdot 2^e$, where M is the bit width of m; similarly for subnormals.

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.

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

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

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

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

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

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

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

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

Conventions

- The meta variable z ranges over floating-point values where clear from context.
- Where clear from context, shorthands like $+1$ denote floating point values like $+(1 + 1 \cdot 2^{-M}) \cdot 2^0$.

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

2.2.4 Vectors

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

2.2.5 Names

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

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

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

Convention

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

2.3 Types

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

2.3.1 Number Types

Number types classify numeric values.

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

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

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

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

Conventions

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

[11] https://www.unicode.org/versions/latest/
[12] https://ieeexplore.ieee.org/document/8766229

2.3.2 Vector Types

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

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

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

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

Conventions

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

2.3.3 Heap Types

Heap types classify objects in the runtime store. There are three disjoint hierarchies of heap types:

- *function types* classify functions,
- *aggregate types* classify dynamically allocated *managed* data, such as *structures*, *arrays*, or *unboxed scalars*,
- *external types* classify *external* references possibly owned by the embedder.

The values from the latter two hierarchies are interconvertible by ways of the extern.convert_any and any.convert_extern instructions. That is, both type hierarchies are inhabited by an isomorphic set of values, but may have different, incompatible representations in practice.

$$\begin{array}{rcl}
\mathit{absheaptype} & ::= & \mathsf{any} \mid \mathsf{eq} \mid \mathsf{i31} \mid \mathsf{struct} \mid \mathsf{array} \mid \mathsf{none} \\
& | & \mathsf{func} \mid \mathsf{nofunc} \\
& | & \mathsf{extern} \mid \mathsf{noextern} \\
& | & \dots \\
\mathit{heaptype} & ::= & \mathit{absheaptype} \mid \mathit{typeuse} \\
\mathit{typeuse} & ::= & \mathit{typeidx} \mid \dots
\end{array}$$

A heap type is either *abstract* or *concrete*. A concrete heap type consists of a *type use*, which is a type index. It classifies an object of the respective type defined in a module. Abstract types are denoted by individual keywords.

The type func denotes the common supertype of all function types, regardless of their concrete definition. Dually, the type nofunc denotes the common subtype of all function types, regardless of their concrete definition. This type has no values.

The type extern denotes the common supertype of all external references received through the embedder. This type has no concrete subtypes. Dually, the type noextern denotes the common subtype of all forms of external references. This type has no values.

The type any denotes the common supertype of all aggregate types, as well as possibly abstract values produced by *internalizing* an external reference of type extern. Dually, the type none denotes the common subtype of all forms of aggregate types. This type has no values.

The type eqt is a subtype of any that includes all types for which references can be compared, i.e., aggregate values and i31.

The types struct and array denote the common supertypes of all structure and array aggregates, respectively.

The type i31 denotes *unboxed scalars*, that is, integers injected into references. Their observable value range is limited to 31 bits.

Note: An i31 value is not actually allocated in the store, but represented in a way that allows them to be mixed with actual references into the store without ambiguity. Engines need to perform some form of *pointer tagging* to achieve this, which is why one bit is reserved.

Although the types none, nofunc, and noextern are not inhabited by any values, they can be used to form the types of all null references in their respective hierarchy. For example, (ref null nofunc) is the generic type of a null reference compatible with all function reference types.

The syntax of abstract heap types and type uses is extended with additional forms for the purpose of specifying validation and execution.

2.3.4 Reference Types

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

$$reftype ::= \mathsf{ref}\ \mathsf{null}^?\ heaptype$$

A reference type is characterised by the heap type it points to.

In addition, a reference type of the form ref null ht is *nullable*, meaning that it can either be a proper reference to ht or null. Other references are *non-null*.

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

Conventions

- The reference type anyref is an abbreviation for (ref null any).
- The reference type eqref is an abbreviation for (ref null eq).
- The reference type i31ref is an abbreviation for (ref null i31).
- The reference type structref is an abbreviation for (ref null struct).
- The reference type arrayref is an abbreviation for (ref null array).
- The reference type funcref is an abbreviation for (ref null func).
- The reference type externref is an abbreviation for (ref null extern).
- The reference type nullref is an abbreviation for (ref null none).
- The reference type nullfuncref is an abbreviation for (ref null nofunc).
- The reference type nullexternref is an abbreviation for (ref null noextern).

2.3.5 Value Types

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

$$\begin{array}{rcl} consttype & ::= & numtype \mid vectype \\ valtype & ::= & numtype \mid vectype \mid reftype \mid \dots \end{array}$$

The syntax of value types is extended with additional forms for the purpose of specifying validation.

Conventions

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

2.3.6 Result Types

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

$$\mathit{resulttype} \quad ::= \quad \mathit{list}(\mathit{valtype})$$

2.3.7 Function Types

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

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

2.3.8 Aggregate Types

Aggregate types describe compound objects consisting of multiple values. These are either *structures* or *arrays*, which both consist of a list of possibly mutable and possibly packed *fields*. Structures are heterogeneous, but require static indexing, while arrays need to be homogeneous, but allow dynamic indexing.

$$\begin{array}{rcl}
\mathit{structtype} & ::= & \mathit{list}(\mathit{fieldtype}) \\
\mathit{arraytype} & ::= & \mathit{fieldtype} \\
\mathit{fieldtype} & ::= & \mathsf{mut}^? ~ \mathit{storagetype} \\
\mathit{storagetype} & ::= & \mathit{valtype} ~|~ \mathit{packtype} \\
\mathit{packtype} & ::= & \mathsf{i8} ~|~ \mathsf{i16}
\end{array}$$

Conventions

- The notation $|t|$ for bit width extends to packed types as well, that is, $|\mathsf{i8}| = 8$ and $|\mathsf{i16}| = 16$.
- The auxiliary function unpack maps a storage type to the value type obtained when accessing a field:

$$\begin{array}{rcl}
\mathrm{unpack}(\mathit{valtype}) & = & \mathit{valtype} \\
\mathrm{unpack}(\mathit{packtype}) & = & \mathsf{i32}
\end{array}$$

2.3.9 Composite Types

Composite types are all types composed from simpler types, including function types and aggregate types.

$$\begin{array}{rcl}
\mathit{comptype} & ::= & \mathsf{struct} ~ \mathit{structtype} \\
& | & \mathsf{array} ~ \mathit{arraytype} \\
& | & \mathsf{func} ~ \mathit{functype}
\end{array}$$

2.3.10 Recursive Types

Recursive types denote a group of mutually recursive composite types, each of which can optionally declare a list of type indices of supertypes that it matches. Each type can also be declared *final*, preventing further subtyping.

$$\begin{array}{lll} \mathit{rectype} & ::= & \mathsf{rec}\ \mathit{list}(\mathit{subtype}) \\ \mathit{subtype} & ::= & \mathsf{sub}\ \mathsf{final}^?\ \mathit{typeuse}^*\ \mathit{comptype} \end{array}$$

In a module, each member of a recursive type is assigned a separate type index.

2.3.11 Limits

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

$$\mathit{limits} \quad ::= \quad [\mathit{u32} \,..\, \mathit{u32}]$$

2.3.12 Memory Types

Memory types classify linear memories and their size range.

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

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

2.3.13 Table Types

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

$$\mathit{tabletype} \quad ::= \quad \mathit{limits}\ \mathit{reftype}$$

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

2.3.14 Global Types

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

$$\mathit{globaltype} \quad ::= \quad \mathsf{mut}^?\ \mathit{valtype}$$

2.3.15 Element Types

Element types classify element segments by a reference type of its elements.

$$\mathit{elemtype} \quad ::= \quad \mathit{reftype}$$

2.3.16 Data Types

Data types classify data segments. Since the contents of a data segment requires no further classification, they merely consist of a universal marker ok indicating well-formedness.

$$\mathit{datatype} ::= \mathsf{ok}$$

2.3.17 External Types

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

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

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:

$$\begin{array}{lcll}
\mathrm{funcs}(\epsilon) & = & \epsilon & \\
\mathrm{funcs}((\mathsf{func}\ dt)\ xt^*) & = & dt\ \mathrm{funcs}(xt^*) & \\
\mathrm{funcs}(\mathit{externtype}\ xt^*) & = & \mathrm{funcs}(xt^*) & \text{otherwise} \\
\mathrm{tables}(\epsilon) & = & \epsilon & \\
\mathrm{tables}((\mathsf{table}\ tt)\ xt^*) & = & tt\ \mathrm{tables}(xt^*) & \\
\mathrm{tables}(\mathit{externtype}\ xt^*) & = & \mathrm{tables}(xt^*) & \text{otherwise} \\
\mathrm{mems}(\epsilon) & = & \epsilon & \\
\mathrm{mems}((\mathsf{mem}\ mt)\ xt^*) & = & mt\ \mathrm{mems}(xt^*) & \\
\mathrm{mems}(\mathit{externtype}\ xt^*) & = & \mathrm{mems}(xt^*) & \text{otherwise} \\
\mathrm{globals}(\epsilon) & = & \epsilon & \\
\mathrm{globals}((\mathsf{global}\ gt)\ xt^*) & = & gt\ \mathrm{globals}(xt^*) & \\
\mathrm{globals}(\mathit{externtype}\ xt^*) & = & \mathrm{globals}(xt^*) & \text{otherwise}
\end{array}$$

2.4 Instructions

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

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

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

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

The syntax of instruction is further extended with additional forms for the purpose of specifying execution.

2.4.1 Parametric Instructions

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

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

The nop instruction does nothing.

The unreachable instruction causes an unconditional trap.

The drop instruction simply throws away a single operand.

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

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

2.4.2 Numeric Instructions

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

$$
\begin{array}{llll}
\mathit{sz} & ::= & \mathsf{8} ~|~ \mathsf{16} ~|~ \mathsf{32} ~|~ \mathsf{64} \\
\mathit{sx} & ::= & \mathsf{u} ~|~ \mathsf{s} \\
\mathit{num}_{\mathsf{i}N} & ::= & \mathit{iN} \\
\mathit{num}_{\mathsf{f}N} & ::= & \mathit{fN} \\
\mathit{instr} & ::= & \dots \\
& | & \mathit{numtype}.\mathsf{const}~\mathit{num}_{\mathit{numtype}} \\
& | & \mathit{numtype}.\mathit{unop}_{\mathit{numtype}} \\
& | & \mathit{numtype}.\mathit{binop}_{\mathit{numtype}} \\
& | & \mathit{numtype}.\mathit{testop}_{\mathit{numtype}} \\
& | & \mathit{numtype}.\mathit{relop}_{\mathit{numtype}} \\
& | & \mathit{numtype}_1.\mathit{cvtop}_{\mathit{numtype}_2,\mathit{numtype}_1}\mathsf{\_}\mathit{numtype}_2\mathsf{\_}\mathit{sx}^? & \text{if } \mathit{numtype}_1 \neq \mathit{numtype}_2 \\
& | & \dots \\
\mathit{unop}_{\mathsf{i}N} & ::= & \mathsf{clz} ~|~ \mathsf{ctz} ~|~ \mathsf{popcnt} ~|~ \mathsf{extend}\mathit{sz}\mathsf{\_s} & \text{if } \mathit{sz} < N \\
\mathit{unop}_{\mathsf{f}N} & ::= & \mathsf{abs} ~|~ \mathsf{neg} ~|~ \mathsf{sqrt} ~|~ \mathsf{ceil} ~|~ \mathsf{floor} ~|~ \mathsf{trunc} ~|~ \mathsf{nearest} \\
\mathit{binop}_{\mathsf{i}N} & ::= & \mathsf{add} ~|~ \mathsf{sub} ~|~ \mathsf{mul} ~|~ \mathsf{div\_}\mathit{sx} ~|~ \mathsf{rem\_}\mathit{sx} \\
& | & \mathsf{and} ~|~ \mathsf{or} ~|~ \mathsf{xor} ~|~ \mathsf{shl} ~|~ \mathsf{shr\_}\mathit{sx} ~|~ \mathsf{rotl} ~|~ \mathsf{rotr} \\
\mathit{binop}_{\mathsf{f}N} & ::= & \mathsf{add} ~|~ \mathsf{sub} ~|~ \mathsf{mul} ~|~ \mathsf{div} ~|~ \mathsf{min} ~|~ \mathsf{max} ~|~ \mathsf{copysign} \\
\mathit{testop}_{\mathsf{i}N} & ::= & \mathsf{eqz} \\
\mathit{relop}_{\mathsf{i}N} & ::= & \mathsf{eq} ~|~ \mathsf{ne} ~|~ \mathsf{lt\_}\mathit{sx} ~|~ \mathsf{gt\_}\mathit{sx} ~|~ \mathsf{le\_}\mathit{sx} ~|~ \mathsf{ge\_}\mathit{sx} \\
\mathit{relop}_{\mathsf{f}N} & ::= & \mathsf{eq} ~|~ \mathsf{ne} ~|~ \mathsf{lt} ~|~ \mathsf{gt} ~|~ \mathsf{le} ~|~ \mathsf{ge} \\
\mathit{cvtop}_{\mathit{nt}_1,\mathit{nt}_2} & ::= & \mathsf{convert} \\
& | & \mathsf{convert\_sat} & \text{if } \mathit{nt}_1 = \mathsf{i}N \wedge \mathit{nt}_2 = \mathsf{f}N \\
& | & \mathsf{reinterpret} & \text{if } |\mathit{nt}_1| = |\mathit{nt}_2|
\end{array}
$$

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

- *Constants*: return a static constant.
- *Unary Operations*: consume one operand and produce one result of the respective type.
- *Binary Operations*: consume two operands and produce one result of the respective type.

- *Tests*: consume one operand of the respective type and produce a Boolean integer result.
- *Comparisons*: consume two operands of the respective type and produce a Boolean integer result.
- *Conversions*: consume a value of one type and produce a result of another (the source type of the conversion is the one after the "_").

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

2.4.3 Vector Instructions

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

$$
\begin{array}{rcll}
\mathit{lanetype} & ::= & \mathit{numtype} \mid \mathit{packtype} & \\
\mathit{dim} & ::= & \mathsf{1} \mid \mathsf{2} \mid \mathsf{4} \mid \mathsf{8} \mid \mathsf{16} & \\
\mathit{shape} & ::= & \mathit{lanetype}\mathsf{x}\mathit{dim} & \\
\mathit{ishape} & ::= & \mathsf{i}N\mathsf{x}\mathit{dim} & \\
\mathit{half}_{\mathsf{i}N_1\mathsf{x}M_1,\mathsf{i}N_2\mathsf{x}M_2} & ::= & \mathsf{low} \mid \mathsf{high} & \text{if } 2 \cdot N_1 = N_1 \\
\mathit{half}_{\mathsf{i}N_1\mathsf{x}M_1,\mathsf{f}N_2\mathsf{x}M_2} & ::= & \mathsf{low} & \text{if } 2 \cdot N_1 = N_1 = \mathsf{64} \\
\mathit{zero}_{\mathsf{f}N_1\mathsf{x}M_1,\mathsf{i}N_2\mathsf{x}M_2} & ::= & \mathsf{zero} & \text{if } 2 \cdot N_2 = N_1 = \mathsf{64} \\
\mathit{laneidx} & ::= & \mathit{u8} & \\
\mathit{instr} & ::= & \dots & \\
& | & \mathit{vectype}.\mathsf{const}\ \mathit{vec}_{\mathit{vectype}} & \\
& | & \mathit{vectype}.\mathit{vvunop} & \\
& | & \mathit{vectype}.\mathit{vvbinop} & \\
& | & \mathit{vectype}.\mathit{vvternop} & \\
& | & \mathit{vectype}.\mathit{vvtestop} & \\
& | & \mathit{shape}.\mathit{vunop}_{\mathit{shape}} & \\
& | & \mathit{shape}.\mathit{vbinop}_{\mathit{shape}} & \\
& | & \mathit{shape}.\mathit{vtestop}_{\mathit{shape}} & \\
& | & \mathit{shape}.\mathit{vrelop}_{\mathit{shape}} & \\
& | & \mathit{ishape}.\mathit{vshiftop}_{\mathit{ishape}} & \\
& | & \mathit{ishape}.\mathsf{bitmask} & \\
& | & \mathit{ishape}.\mathsf{swizzle} & \text{if } \mathit{ishape} = \mathsf{i8x16} \\
& | & \mathit{ishape}.\mathsf{shuffle}\ \mathit{laneidx}^* & \text{if } \mathit{ishape} = \mathsf{i8x16} \land |\mathit{laneidx}^*| = \mathsf{16} \\
& | & \mathit{ishape}_1.\mathit{vextunop}_{\mathit{ishape}_2,\mathit{ishape}_1}\_\mathit{ishape}_2\_\mathit{sx} & \text{if } |\mathrm{lanetype}(\mathit{ishape}_1)| = 2 \cdot |\mathrm{lanetype}(\\
& | & \mathit{ishape}_1.\mathit{vextbinop}_{\mathit{ishape}_2,\mathit{ishape}_1}\_\mathit{ishape}_2\_\mathit{sx} & \text{if } |\mathrm{lanetype}(\mathit{ishape}_1)| = 2 \cdot |\mathrm{lanetype}(\\
& | & \mathit{ishape}_1.\mathsf{narrow}\_\mathit{ishape}_2\_\mathit{sx} & \text{if } |\mathrm{lanetype}(\mathit{ishape}_2)| = 2 \cdot |\mathrm{lanetype}(\\
& | & \mathit{shape}_1.\mathit{vcvtop}_{\mathit{shape}_2,\mathit{shape}_1}\_\mathit{sx}^?\_\mathit{shape}_2\_\mathit{half}^?_{\mathit{shape}_2,\mathit{shape}_1} & \text{if } \mathrm{lanetype}(\mathit{shape}_1) \neq \mathrm{lanetype}(\mathit{shape} \\
& & & \land\ \mathit{sx}^? \neq \epsilon \Leftrightarrow \mathrm{lanetype}(\mathit{shape}_1) = \mathsf{i}N_1 \\
& | & \mathit{shape}.\mathsf{splat} & \\
& | & \mathit{shape}.\mathsf{extract\_lane}\_\mathit{sx}^?\ \mathit{laneidx} & \text{if } \mathrm{lanetype}(\mathit{shape}) = \mathit{numtype} \Leftrightarrow \mathit{sx}^? \\
& | & \mathit{shape}.\mathsf{replace\_lane}\ \mathit{laneidx} & \\
& | & \dots &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{vvunop} & ::= & \mathsf{not} & \\
\mathit{vvbinop} & ::= & \mathsf{and} \mid \mathsf{andnot} \mid \mathsf{or} \mid \mathsf{xor} & \\
\mathit{vvternop} & ::= & \mathsf{bitselect} & \\
\mathit{vvtestop} & ::= & \mathsf{any\_true} & \\
\mathit{vunop}_{\mathsf{i}N\times M} & ::= & \mathsf{abs} \mid \mathsf{neg} & \\
 & \mid & \mathsf{popcnt} & \text{if } N = 8 \\
\mathit{vunop}_{\mathsf{f}N\times M} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} & \\
\mathit{vbinop}_{\mathsf{i}N\times M} & ::= & \mathsf{add} & \\
 & \mid & \mathsf{sub} & \\
 & \mid & \mathsf{add\_sat\_}\mathit{sx} & \text{if } N \leq 16 \\
 & \mid & \mathsf{sub\_sat\_}\mathit{sx} & \text{if } N \leq 16 \\
 & \mid & \mathsf{mul} & \text{if } N \geq 16 \\
 & \mid & \mathsf{avgr\_u} & \text{if } N \leq 16 \\
 & \mid & \mathsf{q15mulr\_sat\_s} & \text{if } N = 16 \\
 & \mid & \mathsf{min\_}\mathit{sx} & \text{if } N \leq 32 \\
 & \mid & \mathsf{max\_}\mathit{sx} & \text{if } N \leq 32 \\
\mathit{vbinop}_{\mathsf{f}N\times M} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{pmin} \mid \mathsf{pmax} & \\
\mathit{vtestop}_{\mathsf{i}N\times M} & ::= & \mathsf{all\_true} & \\
\mathit{vrelop}_{\mathsf{i}N\times M} & ::= & \mathsf{eq} \mid \mathsf{ne} & \\
 & \mid & \mathsf{lt\_}\mathit{sx} & \text{if } N \neq 64 \vee \mathit{sx} = \mathsf{s} \\
 & \mid & \mathsf{gt\_}\mathit{sx} & \text{if } N \neq 64 \vee \mathit{sx} = \mathsf{s} \\
 & \mid & \mathsf{le\_}\mathit{sx} & \text{if } N \neq 64 \vee \mathit{sx} = \mathsf{s} \\
 & \mid & \mathsf{ge\_}\mathit{sx} & \text{if } N \neq 64 \vee \mathit{sx} = \mathsf{s} \\
\mathit{vrelop}_{\mathsf{f}N\times M} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} & \\
\mathit{vshiftop}_{\mathsf{i}N\times M} & ::= & \mathsf{shl} \mid \mathsf{shr\_}\mathit{sx} & \\
\mathit{vextunop}_{\mathsf{i}N_1\times M_1,\mathsf{i}N_2\times M_2} & ::= & \mathsf{extadd\_pairwise} & \text{if } 16 \leq 2 \cdot N_1 = N_2 \leq 32 \\
\mathit{vextbinop}_{\mathsf{i}N_1\times M_1,\mathsf{i}N_2\times M_2} & ::= & \mathsf{extmul\_}\mathit{half}_{\mathsf{i}N_1\times M_1,\mathsf{i}N_2\times M_2} & \text{if } 2 \cdot N_1 = N_2 \geq 16 \\
 & \mid & \mathsf{dot} & \text{if } 2 \cdot N_1 = N_2 = 32 \\
\mathit{vcvtop}_{\mathsf{i}N_1\times M_1,\mathsf{i}N_2\times M_2} & ::= & \mathsf{extend} & \text{if } N_2 = 2 \cdot N_1 \\
\mathit{vcvtop}_{\mathsf{i}N_1\times M_1,\mathsf{f}N_2\times M_2} & ::= & \mathsf{convert} & \text{if } N_2 \geq N_1 = 32 \\
\mathit{vcvtop}_{\mathsf{f}N_1\times M_1,\mathsf{i}N_2\times M_2} & ::= & \mathsf{trunc\_sat} & \text{if } N_1 \geq N_2 = 32 \\
\mathit{vcvtop}_{\mathsf{f}N_1\times M_1,\mathsf{f}N_2\times M_2} & ::= & \mathsf{demote} & \text{if } N_1 > N_2 \\
 & \mid & \mathsf{promote} & \text{if } N_1 < N_2
\end{array}
$$

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

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

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

Vector instructions can be grouped into several subcategories:

- *Constants*: return a static constant.
- *Unary Operations*: consume one $\mathsf{v128}$ operand and produce one $\mathsf{v128}$ result.
- *Binary Operations*: consume two $\mathsf{v128}$ operands and produce one $\mathsf{v128}$ result.
- *Ternary Operations*: consume three $\mathsf{v128}$ operands and produce one $\mathsf{v128}$ result.
- *Tests*: consume one $\mathsf{v128}$ operand and produce a Boolean integer result.
- *Shifts*: consume a $\mathsf{v128}$ operand and an $\mathsf{i32}$ operand, producing one $\mathsf{v128}$ result.

- *Splats*: consume a value of numeric type and produce a v128 result of a specified shape.
- *Extract lanes*: consume a v128 operand and return the numeric value in a given lane.
- *Replace lanes*: consume a v128 operand and a numeric value for a given lane, and produce a v128 result.

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

Conventions

- The function $\mathrm{lanetype}(shape)$ extracts the lane type of a shape.
- The function $\mathrm{dim}(shape)$ extracts the dimension of a shape.

2.4.4 Reference Instructions

Instructions in this group are concerned with accessing references.

$$\begin{array}{llll} instr & ::= & \dots \\ & | & \mathsf{ref.func}\ funcidx \\ & | & \mathsf{ref.null}\ heaptype \\ & | & \mathsf{ref.is\_null} \\ & | & \mathsf{ref.as\_non\_null} \\ & | & \mathsf{ref.eq} \\ & | & \mathsf{ref.test}\ reftype \\ & | & \mathsf{ref.cast}\ reftype \\ & | & \dots \end{array}$$

The ref.null and ref.func instructions produce a null value or a reference to a given function, respectively.

The instruction ref.is_null checks for null, while ref.as_non_null converts a nullable to a non-null one, and traps if it encounters null.

The ref.eq compares two references.

The instructions ref.test and ref.cast test the dynamic type of a reference operand. The former merely returns the result of the test, while the latter performs a downcast and traps if the operand's type does not match.

Note: The br_on_null and br_on_non_null instructions provide versions of ref.as_null that branch depending on the success of failure of a null test instead of trapping. Similarly, the br_on_cast and br_on_cast_fail instructions provides versions of ref.cast that branch depending on the success of the downcast instead of trapping.

An additional instruction operating on function references is the control instruction call_ref.

2.4.5 Aggregate Instructions

Instructions in this group are concerned with creating and accessing references to aggregate types.

$$
\begin{array}{llll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{struct.new}~\mathit{typeidx} \\
& | & \mathsf{struct.new\_default}~\mathit{typeidx} \\
& | & \mathsf{struct.get\_}\mathit{sx}^?~\mathit{typeidx}~\mathit{u32} \\
& | & \mathsf{struct.set}~\mathit{typeidx}~\mathit{u32} \\
& | & \mathsf{array.new}~\mathit{typeidx} \\
& | & \mathsf{array.new\_default}~\mathit{typeidx} \\
& | & \mathsf{array.new\_fixed}~\mathit{typeidx}~\mathit{u32} \\
& | & \mathsf{array.new\_data}~\mathit{typeidx}~\mathit{dataidx} \\
& | & \mathsf{array.new\_elem}~\mathit{typeidx}~\mathit{elemidx} \\
& | & \mathsf{array.get\_}\mathit{sx}^?~\mathit{typeidx} \\
& | & \mathsf{array.set}~\mathit{typeidx} \\
& | & \mathsf{array.len} \\
& | & \mathsf{array.fill}~\mathit{typeidx} \\
& | & \mathsf{array.copy}~\mathit{typeidx}~\mathit{typeidx} \\
& | & \mathsf{array.init\_data}~\mathit{typeidx}~\mathit{dataidx} \\
& | & \mathsf{array.init\_elem}~\mathit{typeidx}~\mathit{elemidx} \\
& | & \mathsf{ref.i31} \\
& | & \mathsf{i31.get\_}\mathit{sx} \\
& | & \mathsf{extern.convert\_any} \\
& | & \mathsf{any.convert\_extern} \\
& | & \dots
\end{array}
$$

The instructions struct.new and struct.new_default allocate a new structure, initializing them either with operands or with default values. The remaining instructions on structs access individual fields, allowing for different sign extension modes in the case of packed storage types.

Similarly, arrays can be allocated either with an explicit initialization operand or a default value. Furthermore, array.new_fixed allocates an array with statically fixed size, and array.new_data and array.new_elem allocate an array and initialize it from a data or element segment, respectively. The instructions array.get, array.get sx, and array.set access individual slots, again allowing for different sign extension modes in the case of a packed storage type; array.len produces the length of an array; array.fill fills a specified slice of an array with a given value and array.copy, array.init_data, and array.init_elem copy elements to a specified slice of an array from a given array, data segment, or element segment, respectively.

The instructions ref.i31 and i31.get sx convert between type i32 and an unboxed scalar.

The instructions any.convert_extern and extern.convert_any allow lossless conversion between references represented as type (ref null extern) and as (ref null any).

2.4.6 Variable Instructions

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

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

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

2.4.7 Table Instructions

Instructions in this group are concerned with tables table.

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

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

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

The table.fill instruction sets all entries in a range to a given value. The table.copy instruction copies elements from a source table region to a possibly overlapping destination region; the first index denotes the destination. The table.init instruction copies elements from a passive element segment into a table.

The elem.drop instruction prevents further use of a passive element segment. This instruction is intended to be used as an optimization hint. After an element segment is dropped its elements can no longer be retrieved, so the memory used by this segment may be freed.

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

2.4.8 Memory Instructions

Instructions in this group are concerned with linear memory.

$$
\begin{array}{rcll}
\mathit{memarg} & ::= & \{\mathsf{align}~\mathit{u32},~\mathsf{offset}~\mathit{u32}\} & \\
\mathit{loadop}_{\mathsf{i}N} & ::= & \mathit{sz}~\mathit{sx} & \text{if } \mathit{sz} < N \\
\mathit{vloadop}_{\mathit{vectype}} & ::= & \mathit{sz}\mathsf{x}M\_\mathit{sx} & \text{if } \mathit{sz} \cdot M = |\mathit{vectype}|/2 \\
& | & \mathit{sz}\_\mathsf{splat} & \\
& | & \mathit{sz}\_\mathsf{zero} & \text{if } \mathit{sz} \geq 32 \\
\mathit{instr} & ::= & \dots & \\
& | & \mathit{numtype}.\mathsf{load}\mathit{loadop}^?_{\mathit{numtype}}~\mathit{memidx}~\mathit{memarg} & \\
& | & \mathit{numtype}.\mathsf{store}\mathit{sz}^?~\mathit{memidx}~\mathit{memarg} & \\
& | & \mathit{vectype}.\mathsf{load}\mathit{vloadop}^?_{\mathit{vectype}}~\mathit{memidx}~\mathit{memarg} & \\
& | & \mathit{vectype}.\mathsf{load}\mathit{sz}\_\mathsf{lane}~\mathit{memidx}~\mathit{memarg}~\mathit{laneidx} & \\
& | & \mathit{vectype}.\mathsf{store}~\mathit{memidx}~\mathit{memarg} & \\
& | & \mathit{vectype}.\mathsf{store}\mathit{sz}\_\mathsf{lane}~\mathit{memidx}~\mathit{memarg}~\mathit{laneidx} & \\
& | & \mathsf{memory.size}~\mathit{memidx} & \\
& | & \mathsf{memory.grow}~\mathit{memidx} & \\
& | & \mathsf{memory.fill}~\mathit{memidx} & \\
& | & \mathsf{memory.copy}~\mathit{memidx}~\mathit{memidx} & \\
& | & \mathsf{memory.init}~\mathit{memidx}~\mathit{dataidx} & \\
& | & \mathsf{data.drop}~\mathit{dataidx} & \\
& | & \dots &
\end{array}
$$

Memory is accessed with load and store instructions for the different number types and *vector types <syntax-vectype>*. They all take a memory index and a *memory argument* memarg that contains an address *offset* and the expected *alignment* (expressed as the exponent of a power of 2).

Integer loads and stores can optionally specify a *storage size* sz that is smaller than the bit width of the respective value type. In the case of loads, a sign extension mode sx is then required to select appropriate behavior.

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

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

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

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

The memory.fill instruction sets all values in a regionof a memory to a given byte. The memory.copy instruction copies data from a source memory region to a possibly overlapping destination region in another or the same memory; the first index denotes the destination The memory.init instruction copies data from a passive data segment into a memory.

The data.drop instruction prevents further use of a passive data segment. This instruction is intended to be used as an optimization hint. After a data segment is dropped its data can no longer be retrieved, so the memory used by this segment may be freed.

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

2.4.9 Control Instructions

Instructions in this group affect the flow of control.

$$
\begin{array}{llll}
\mathit{blocktype} & ::= & \mathit{valtype}^? \\
& | & \mathit{funcidx} \\
\mathit{instr} & ::= & \dots \\
& | & \mathsf{block}~\mathit{blocktype}~\mathit{instr}^* \\
& | & \mathsf{loop}~\mathit{blocktype}~\mathit{instr}^* \\
& | & \mathsf{if}~\mathit{blocktype}~\mathit{instr}^*~\mathsf{else}~\mathit{instr}^* \\
& | & \mathsf{br}~\mathit{labelidx} \\
& | & \mathsf{br\_if}~\mathit{labelidx} \\
& | & \mathsf{br\_table}~\mathit{labelidx}^*~\mathit{labelidx} \\
& | & \mathsf{br\_on\_null}~\mathit{labelidx} \\
& | & \mathsf{br\_on\_non\_null}~\mathit{labelidx} \\
& | & \mathsf{br\_on\_cast}~\mathit{labelidx}~\mathit{reftype}~\mathit{reftype} \\
& | & \mathsf{br\_on\_cast\_fail}~\mathit{labelidx}~\mathit{reftype}~\mathit{reftype} \\
& | & \mathsf{call}~\mathit{funcidx} \\
& | & \mathsf{call\_ref}~\mathit{typeuse} \\
& | & \mathsf{call\_indirect}~\mathit{tableidx}~\mathit{typeuse} \\
& | & \mathsf{return} \\
& | & \mathsf{return\_call}~\mathit{funcidx} \\
& | & \mathsf{return\_call\_ref}~\mathit{typeuse} \\
& | & \mathsf{return\_call\_indirect}~\mathit{tableidx}~\mathit{typeuse} \\
& | & \dots
\end{array}
$$

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

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

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

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 in most C-like languages, while a branch targeting a loop behaves like a continue statement.

Branch instructions come in several flavors: br performs an unconditional branch, br_if performs a conditional branch, and br_table performs an indirect branch through an operand indexing into the label list that is an immediate to the instruction, or to a default target if the operand is out of bounds. The br_on_null and br_on_non_null instructions check whether a reference operand is null and branch if that is the case or not the case, respectively. Similarly, br_on_cast and br_on_cast_fail attempt a downcast on a reference operand and branch if that succeeds, or fails, respectively.

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_ref instruction invokes a function indirectly through a function reference operand. The call_indirect instruction calls a function indirectly through an operand indexing into a table that is denoted by a table index and must contain function references. Since it may contain functions of heterogeneous type, the callee is dynamically checked against the function type indexed by the instruction's second immediate, and the call is aborted with a trap if it does not match.

The return_call, return_call_ref, and return_call_indirect instructions are *tail-call* variants of the previous ones. That is, they first return from the current function before actually performing the respective call. It is guaranteed that no sequence of nested calls using only these instructions can cause resource exhaustion due to hitting an implementation's limit on the number of active calls.

2.4.10 Expressions

Function bodies, initialization values for globals, elements and offsets of element segments, and offsets of data segments are given as expressions, which are sequences of instructions.

$$\mathit{expr} \quad ::= \quad \mathit{instr}^*$$

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

2.5 Modules

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

$$module \quad ::= \quad \mathsf{module}\ type^*\ import^*\ func^*\ global^*\ table^*\ mem^*\ elem^*\ data^*\ start^*\ export^*$$

Each of the lists — 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.

$$\begin{array}{rcl} typeidx & ::= & idx \\ funcidx & ::= & idx \\ globalidx & ::= & idx \\ tableidx & ::= & idx \\ memidx & ::= & idx \\ elemidx & ::= & idx \\ dataidx & ::= & idx \\ labelidx & ::= & idx \\ localidx & ::= & idx \\ fieldidx & ::= & idx \end{array}$$

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

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

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

Label indices reference structured control instructions inside an instruction sequence.

Each aggregate type provides an index space for its fields.

Conventions

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

Note: For example, if $instr^*$ is $(\mathsf{data.drop}\ 1)\ (\mathsf{memory.init}\ 2\ 3)$, then $\mathrm{dataidx}_{instrs}(instr^*) = 1\ 3$, or equivalently, the set $\{1, 3\}$.

2.5.2 Types

The *type* section of a module defines a list of recursive types, each consisting of a list of sub types referenced by individual type indices. All function or aggregate types used in a module must be defined in this section.

$$type \quad ::= \quad \mathsf{type}\ rectype$$

2.5.3 Functions

The *func* section of a module defines a list of *functions* with the following structure:

$$\begin{array}{lll} func & ::= & \mathsf{func}\ typeidx\ local^*\ expr \\ local & ::= & \mathsf{local}\ valtype \end{array}$$

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

The locals declare a list 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.

A function's expression is an instruction sequence that represents the body of the function. Upon termination it 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 *table* section of a module defines a list of *tables* described by their table type:

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

A table is an array of opaque values of a particular reference type that is specified by the table type. Each table slot is initialized with a value given by a constant initializer expression. Tables can further be initialized through element segments.

The minimum size in the limits of the table type specifies the initial size of that table, while its maximum restricts the size to which it can grow later.

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

2.5.5 Memories

The *mem* section of a module defines a list of *linear memories* (or *memories* for short) as described by their memory type:

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

A memory is a list of raw uninterpreted bytes. The minimum size in the limits of its memory type specifies the initial size of that memory, while its maximum, 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.

2.5.6 Globals

The *global* section of a module defines a list of *global variables* (or *globals* for short):

$$global ::= \mathsf{global}\ globaltype\ expr$$

Each global stores a single value of the type specified in the global type. It also specifies whether a global is immutable or mutable. Moreover, each global is initialized with a 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 *elem* section of a module defines a list of *element segments*, which can be used to initialize a subrange of a table from a static list of elements.

$$\begin{array}{rcl} elem & ::= & \mathsf{elem}\ reftype\ expr^*\ elemmode \\ elemmode & ::= & \mathsf{active}\ tableidx\ expr \mid \mathsf{passive} \mid \mathsf{declare} \end{array}$$

Each element segment defines a reference type and a corresponding list of constant element expressions.

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

Element segments are referenced through element indices.

2.5.8 Data Segments

The *data* section of a module defines a list of *data segments*, which can be used to initialize a range of memory from a static list of bytes.

$$\begin{array}{rcl} data & ::= & \mathsf{data}\ byte^*\ datamode \\ datamode & ::= & \mathsf{active}\ memidx\ expr \mid \mathsf{passive} \end{array}$$

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

Data segments are referenced through data indices.

2.5.9 Start Function

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

$$start ::= \mathsf{start}\ funcidx$$

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

2.5.10 Exports

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

$$\begin{array}{lll} \mathit{export} & ::= & \mathsf{export}\ \mathit{name}\ \mathit{externidx} \\ \mathit{externidx} & ::= & \mathsf{func}\ \mathit{funcidx} \mid \mathsf{global}\ \mathit{globalidx} \mid \mathsf{table}\ \mathit{tableidx} \mid \mathsf{memory}\ \mathit{memidx} \end{array}$$

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

Conventions

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

$$\begin{array}{llll}
\mathrm{funcs}(\epsilon) & = & \epsilon & \\
\mathrm{funcs}((\mathsf{func}\ x)\ xx^*) & = & x\ \mathrm{funcs}(xx^*) & \\
\mathrm{funcs}(\mathit{externidx}\ xx^*) & = & \mathrm{funcs}(xx^*) & \text{otherwise} \\
\mathrm{tables}(\epsilon) & = & \epsilon & \\
\mathrm{tables}((\mathsf{table}\ x)\ xx^*) & = & x\ \mathrm{tables}(xx^*) & \\
\mathrm{tables}(\mathit{externidx}\ xx^*) & = & \mathrm{tables}(xx^*) & \text{otherwise} \\
\mathrm{mems}(\epsilon) & = & \epsilon & \\
\mathrm{mems}((\mathsf{memory}\ x)\ xx^*) & = & x\ \mathrm{mems}(xx^*) & \\
\mathrm{mems}(\mathit{externidx}\ xx^*) & = & \mathrm{mems}(xx^*) & \text{otherwise} \\
\mathrm{globals}(\epsilon) & = & \epsilon & \\
\mathrm{globals}((\mathsf{global}\ x)\ xx^*) & = & x\ \mathrm{globals}(xx^*) & \\
\mathrm{globals}(\mathit{externidx}\ xx^*) & = & \mathrm{globals}(xx^*) & \text{otherwise}
\end{array}$$

2.5.11 Imports

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

$$\mathit{import} \quad ::= \quad \mathsf{import}\ \mathit{name}\ \mathit{name}\ \mathit{externtype}$$

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

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

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

CHAPTER 3

Validation

3.1 Conventions

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

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

1. In *prose*, describing the meaning in intuitive form.
2. In *formal notation*, describing the rule in mathematical form.[14]

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

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

3.1.1 Types

To define the semantics, the definition of some sorts of types is extended to include additional forms. By virtue of not being representable in either the binary format or the text format, these forms cannot be used in a program; they only occur during validation or execution.

$$\begin{array}{rcl} \mathit{valtype} & ::= & \dots \mid \mathsf{bot} \\ \mathit{absheaptype} & ::= & \dots \mid \mathsf{bot} \\ \mathit{typeuse} & ::= & \dots \mid \mathit{deftype} \mid \mathsf{rec}\ n \end{array}$$

The unique value type bot is a *bottom type* that matches all value types. Similarly, bot is also used as a bottom type of all heap types.

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

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

Note: No validation rule uses bottom types explicitly, but various rules can pick any value or heap type, including bottom. This ensures the existence of principal types, and thus a validation algorithm without back tracking.

A type use can consist directly of a defined type. This occurs as the result of substituting a type index with its definition.

A type use may also be a *recursive type index*. Such an index refers to the i-th component of a surrounding recursive type. It occurs as the result of rolling up the definition of a recursive type.

Both extensions affect the occurrence of type uses in concrete heap types, in sub types and in instructions.

Note: It is an invariant of the semantics that sub types occur only in one of two forms: either as "syntactic" types as in a source module, where all supertypes are type indices, or as "semantic" types, where all supertypes are resolved to either defined types or recursive type indices.

A type of any form is *closed* when it does not contain a heap type that is a type index or a recursive type index without a surrounding recursive type, i.e., all type indices have been substituted with their defined type and all free recursive type indices have been unrolled.

Note: Recursive type indices are internal to a recursive type. They are distinguished from regular type indices and represented such that two closed types are syntactically equal if and only if they have the same recursive structure.

Convention

- The *difference* $rt_1 \setminus rt_2$ between two reference types is defined as follows:

$$\begin{array}{lcl} (\mathsf{ref}~\mathsf{null}_1^?~ht_1) \setminus (\mathsf{ref}~\mathsf{null}~ht_2) &=& (\mathsf{ref}~ht_1) \\ (\mathsf{ref}~\mathsf{null}_1^?~ht_1) \setminus (\mathsf{ref}~ht_2) &=& (\mathsf{ref}~\mathsf{null}_1^?~ht_1) \end{array}$$

Note: This definition computes an approximation of the reference type that is inhabited by all values from rt_1 except those from rt_2. Since the type system does not have general union types, the defnition only affects the presence of null and cannot express the absence of other values.

3.1.2 Defined Types

Defined types denote the individual types defined in a module. Each such type is represented as a projection from the recursive type group it originates from, indexed by its position in that group.

$$deftype \quad ::= \quad rectype.n$$

Defined types do not occur in the binary or text format, but are formed by rolling up the recursive types defined in a module.

It is hence an invariant of the semantics that all recursive types occurring in defined types are rolled up.

Conventions

- $t[x^* := dt^*]$ denotes the parallel *substitution* of type indices x^* with defined types dt^* in type t, provided $|x^*| = |dt^*|$.
- $t[(\mathsf{rec}\ i)^* := dt^*]$ denotes the parallel substitution of recursive type indices $(\mathsf{rec}\ i)^*$ with defined types dt^* in type t, provided $|(\mathsf{rec}\ i)^*| = |dt^*|$.
- $t[:= dt^*]$ is shorthand for the substitution $t[x^* := dt^*]$, where $x^* = 0 \ldots (|dt^*| - 1)$.

3.1.3 Rolling and Unrolling

In order to allow comparing recursive types for equivalence, their representation is changed such that all type indices internal to the same recursive type are replaced by recursive type indices.

Note: This representation is independent of the type index space, so that it is meaningful across module boundaries. Moreover, this representation ensures that types with equivalent recursive structure are also syntactically equal, hence allowing a simple equality check on (closed) types. It gives rise to an *iso-recursive* interpretation of types.

The representation change is performed by two auxiliary operations on the syntax of recursive types:

- *Rolling up* a recursive type substitutes its internal type indices with corresponding recursive type indices.
- *Unrolling* a recursive type substitutes its recursive type indices with the corresponding defined types.

These operations are extended to defined types and defined as follows:

$$\begin{array}{llll}
\mathrm{roll}_x(\mathit{rectype}) & = & \mathsf{rec}\ (\mathit{subtype}[(x + i)^{i<n} := (\mathsf{rec}\ i)^{i<n}])^n & \text{if } \mathit{rectype} = \mathsf{rec}\ \mathit{subtype}^n \\
\mathrm{unroll}(\mathit{rectype}) & = & \mathsf{rec}\ (\mathit{subtype}[(\mathsf{rec}\ i)^{i<n} := (\mathit{rectype}.i)^{i<n}])^n & \text{if } \mathit{rectype} = \mathsf{rec}\ \mathit{subtype}^n \\
\mathrm{roll}_x^*(\mathit{rectype}) & = & ((\mathsf{rec}\ \mathit{subtype}^n).i)^{i<n} & \text{if } \mathrm{roll}_x(\mathit{rectype}) = \mathsf{rec}\ \mathit{subtype}^n \\
\mathrm{unroll}(\mathit{rectype}.i) & = & \mathit{subtype}^*[i] & \text{if } \mathrm{unroll}(\mathit{rectype}) = \mathsf{rec}\ \mathit{subtype}^*
\end{array}$$

In addition, the following auxiliary relation denotes the *expansion* of a defined type:

$$\mathit{deftype} \approx \mathit{comptype} \qquad \text{if } \mathrm{unroll}(\mathit{deftype}) = \mathsf{sub}\ \mathsf{final}^?\ \mathit{typeuse}^*\ \mathit{comptype}$$

3.1.4 Instruction Types

Instruction types classify the behaviour of instructions or instruction sequences, by describing how they manipulate the operand stack and the initialization status of locals:

$$\mathit{instrtype} ::= \mathit{resulttype} \rightarrow_{\mathit{localidx}^*} \mathit{resulttype}$$

An instruction type $t_1^* \rightarrow_{x^*} t_2^*$ describes 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. Moreover, it enumerates the indices x^* of locals that have been set by the instruction or sequence.

Note: Instruction types are only used for validation, they do not occur in programs.

3.1.5 Local Types

Local types classify locals, by describing their value type as well as their *initialization status*:

$$\begin{array}{rcl} \mathit{localtype} & ::= & \mathit{init}~\mathit{valtype} \\ \mathit{init} & ::= & \mathsf{set} \mid \mathsf{unset} \end{array}$$

Note: Local types are only used for validation, they do not occur in programs.

3.1.6 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.
- *Recursive Types*: the list of sub types in the current group of recursive types.
- *Functions*: the list of functions declared in the current module, represented by a defined type that expands to 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.
- *Element Segments*: the list of element segments declared in the current module, represented by the elements' reference type.
- *Data Segments*: the list of data segments declared in the current module, each represented by an ok entry.
- *Locals*: the list of locals declared in the current function (including parameters), represented by their local type.
- *Labels*: the stack of labels accessible from the current position, represented by their result type.
- *Return*: the return type of the current function, represented as an optional result type that is absent when no return is allowed, as in free-standing expressions.
- *References*: the list of function indices that occur in the module outside functions and can hence be used to form references inside them.

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

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

$$\begin{array}{rcl} \mathit{context} & ::= & \{\mathsf{types}~\mathit{deftype}^*, \\ & & \phantom{\{}\mathsf{recs}~\mathit{subtype}^*, \\ & & \phantom{\{}\mathsf{funcs}~\mathit{deftype}^*, \\ & & \phantom{\{}\mathsf{globals}~\mathit{globaltype}^*, \\ & & \phantom{\{}\mathsf{tables}~\mathit{tabletype}^*, \\ & & \phantom{\{}\mathsf{mems}~\mathit{memtype}^*, \\ & & \phantom{\{}\mathsf{elems}~\mathit{elemtype}^*, \\ & & \phantom{\{}\mathsf{datas}~\mathit{datatype}^*, \\ & & \phantom{\{}\mathsf{locals}~\mathit{localtype}^*, \\ & & \phantom{\{}\mathsf{labels}~\mathit{resulttype}^*, \\ & & \phantom{\{}\mathsf{return}~\mathit{resulttype}^?, \\ & & \phantom{\{}\mathsf{refs}~\mathit{funcidx}^*\} \end{array}$$

Convention

A type of any shape can be *closed* to bring it into closed form relative to a context it is valid in by substituting each type index x occurring in it with its own corresponding defined type $C.\mathsf{types}[x]$, after first closing the the types in $C.\mathsf{types}$ themselves.

$$\begin{array}{lcll} \mathsf{clos}_C(t) & = & t[:= dt^*] & \text{if } dt^* = \mathsf{clos}^*(C.\mathsf{types}) \\ \mathsf{clos}^*(\epsilon) & = & \epsilon & \\ \mathsf{clos}^*(dt^*~dt_n) & = & dt'^*~dt_n[:= dt'^*] & \text{if } dt'^* = \mathsf{clos}^*(dt^*) \end{array}$$

3.1.7 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 the syntactic class of A.

  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.

3.1.8 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.[16]

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

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

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

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 $\mathsf{i32.add}$ 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 $\mathsf{i32}$ values and produces one), independent of any side conditions.

[16] For example: Benjamin Pierce. Types and Programming Languages[Page 31, 17]. The MIT Press 2002
[17] https://www.cis.upenn.edu/~bcpierce/tapl/

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

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

Here, the premise enforces that the immediate global index x exists in the context. The instruction produces a value of its respective type t (and does not consume any values). If $C.\mathsf{globals}[x]$ 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:

$$\frac{C \vdash \mathit{blocktype} : t_1^* \to t_2^* \qquad \{\mathsf{labels}\ (t_2^*)\} \oplus C \vdash \mathit{instr}^* : t_1^* \to t_2^*}{C \vdash \mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^* : t_1^* \to t_2^*}$$

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

Simple types, such as number types are universally valid. However, restrictions apply to most other types, such as reference types, function types, as well as the limits of table types and memory types, which must be checked during validation.

Moreover, block types are converted to plain function types for ease of processing.

3.2.1 Number Types

Number types are always valid.

$$\frac{}{C \vdash \mathit{numtype} : \mathsf{ok}}$$

3.2.2 Vector Types

Vector types are always valid.

$$\frac{}{C \vdash \mathit{vectype} : \mathsf{ok}}$$

3.2.3 Heap Types

Concrete heap types are only valid when the type index is, while abstract ones are vacuously valid.

$absheaptype$

- The heap type is valid.

$$\frac{}{C \vdash absheaptype : \mathsf{ok}}$$

$typeidx$

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

$$\frac{C.\mathsf{types}[typeidx] = dt}{C \vdash typeidx : \mathsf{ok}}$$

3.2.4 Reference Types

Reference types are valid when the referenced heap type is.

$\mathsf{ref}\ \mathsf{null}^?\ heaptype$

- The heap type $heaptype$ must be valid.
- Then the reference type is valid.

$$\frac{C \vdash heaptype : \mathsf{ok}}{C \vdash \mathsf{ref}\ \mathsf{null}^?\ heaptype : \mathsf{ok}}$$

3.2.5 Value Types

Valid value types are either valid number types, valid vector types, or valid reference types.

3.2.6 Block Types

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

$typeidx$

- The type $C.\mathsf{types}[typeidx]$ must be defined in the context.
- The expansion of $C.\mathsf{funcs}[typeidx]$ must be a function type $\mathsf{func}\ [t_1^*] \to [t_2^*]$.
- Then the block type is valid as instruction type $[t_1^*] \to [t_2^*]$.

$$\frac{C.\mathsf{types}[typeidx] \approx \mathsf{func}\ (t_1^* \to t_2^*)}{C \vdash typeidx : t_1^* \to t_2^*}$$

$[valtype^?]$

- The value type $valtype$ must either be absent, or valid.
- Then the block type is valid as instruction type $[] \rightarrow [valtype^?]$.

$$\frac{(C \vdash valtype : \mathsf{ok})^?}{C \vdash valtype^? : \epsilon \rightarrow valtype^?}$$

3.2.7 Result Types

$[t^*]$

- Each value type t_i in the type sequence t^* must be valid.
- Then the result type is valid.

$$\frac{(C \vdash t : \mathsf{ok})^*}{C \vdash t^* : \mathsf{ok}}$$

3.2.8 Instruction Types

$[t_1^*] \rightarrow_{x^*} [t_2^*]$

- The result type $[t_1^*]$ must be valid.
- The result type $[t_2^*]$ must be valid.
- Each local index x_i in x^* must be defined in the context.
- Then the instruction type is valid.

$$\frac{C \vdash t_1^* : \mathsf{ok} \qquad C \vdash t_2^* : \mathsf{ok} \qquad (C.\mathsf{locals}[x] = lt)^*}{C \vdash t_1^* \rightarrow_{x^*} t_2^* : \mathsf{ok}}$$

3.2.9 Function Types

$[t_1^*] \rightarrow [t_2^*]$

- The result type $[t_1^*]$ must be valid.
- The result type $[t_2^*]$ must be valid.
- Then the function type is valid.

$$\frac{C \vdash t_1^* : \mathsf{ok} \qquad C \vdash t_2^* : \mathsf{ok}}{C \vdash t_1^* \rightarrow t_2^* : \mathsf{ok}}$$

3.2.10 Composite Types

$\mathsf{func}\ \mathit{functype}$

- The function type $\mathit{functype}$ must be valid.
- Then the composite type is valid.

$$\frac{C \vdash \mathit{functype} : \mathsf{ok}}{C \vdash \mathsf{func}\ \mathit{functype} : \mathsf{ok}}$$

$\mathsf{struct}\ \mathit{fieldtype}^*$

- For each field type $\mathit{fieldtype}_i$ in $\mathit{fieldtype}^*$:
  - The field type $\mathit{fieldtype}_i$ must be valid.
- Then the composite type is valid.

$$\frac{(C \vdash \mathit{fieldtype} : \mathsf{ok})^*}{C \vdash \mathsf{struct}\ \mathit{fieldtype}^* : \mathsf{ok}}$$

$\mathsf{array}\ \mathit{fieldtype}$

- The field type $\mathit{fieldtype}$ must be valid.
- Then the composite type is valid.

$$\frac{C \vdash \mathit{fieldtype} : \mathsf{ok}}{C \vdash \mathsf{array}\ \mathit{fieldtype} : \mathsf{ok}}$$

3.2.11 Field Types

$\mathit{mut}\ \mathit{storagetype}$

- The storage type $\mathit{storagetype}$ must be valid.
- Then the field type is valid.

$$\frac{C \vdash \mathit{storagetype} : \mathsf{ok}}{C \vdash \mathsf{mut}^?\ \mathit{storagetype} : \mathsf{ok}}$$

$\mathit{packtype}$

- The packed type is valid.

$$\frac{}{C \vdash \mathit{packtype} : \mathsf{ok}}$$

3.2.12 Recursive Types

Recursive types are validated for a specific type index that denotes the index of the type defined by the recursive group.

$\mathsf{rec}\ \mathit{subtype}^*$

- Either the sequence $\mathit{subtype}^*$ is empty.
- Or:
  - The first sub type of the sequence $\mathit{subtype}^*$ must be valid for the type index x.
  - The remaining sequence $\mathit{subtype}^*$ must be valid for the type index $x + 1$.
- Then the recursive type is valid for the type index x.

$$\frac{}{C \vdash \mathsf{rec}\ \epsilon : \mathsf{ok}(x)} \qquad \frac{C \vdash \mathit{subtype}_1 : \mathsf{ok}(x) \qquad C \vdash \mathsf{rec}\ \mathit{subtype}^* : \mathsf{ok}(x+1)}{C \vdash \mathsf{rec}\ (\mathit{subtype}_1\ \mathit{subtype}^*) : \mathsf{ok}(x)}$$

$\mathsf{sub}\ \mathsf{final}^?\ y^*\ \mathit{comptype}$

- The composite type $\mathit{comptype}$ must be valid.
- The sequence y^* may be no longer than 1.
- For every type index y_i in y^*:
  - The type index y_i must be smaller than x.
  - The type index y_i must exist in the context C.
  - Let $\mathit{subtype}_i$ be the unrolling of the defined type $C.\mathsf{types}[y_i]$.
  - The sub type $\mathit{subtype}_i$ must not contain final.
  - Let $\mathit{comptype}'_i$ be the composite type in $\mathit{subtype}_i$.
  - The composite type $\mathit{comptype}$ must match $\mathit{comptype}'_i$.
- Then the sub type is valid for the type index x.

$$\frac{\begin{array}{c}|x^*| \leq 1 \qquad (x < x_0)^* \qquad (\mathrm{unroll}(C.\mathsf{types}[x]) = \mathsf{sub}\ x'^*\ \mathit{comptype}')^* \\ C \vdash \mathit{comptype} : \mathsf{ok} \qquad (C \vdash \mathit{comptype} \leq \mathit{comptype}')^*\end{array}}{C \vdash \mathsf{sub}\ \mathsf{final}^?\ \mathit{typeidx}^*\ \mathit{comptype} : \mathsf{ok}(x_0)}$$

Note: The side condition on the index ensures that a declared supertype is a previously defined types, preventing cyclic subtype hierarchies.

Future versions of WebAssembly may allow more than one supertype.

3.2.13 Defined Types

$\mathit{rectype}.i$

- The recursive type $\mathit{rectype}$ must be valid for some type index x.
- Let $\mathsf{rec}\ \mathit{subtype}^*$ be the defined type $\mathit{rectype}$.
- The number i must be smaller than the length of the sequence $\mathit{subtype}^*$ of sub types.
- Then the defined type is valid.

$$\frac{C \vdash \mathit{rectype} : \mathsf{ok}(x) \qquad \mathit{rectype} = \mathsf{rec}\ \mathit{subtype}^n \qquad i < n}{C \vdash \mathit{rectype}.i : \mathsf{ok}}$$

3.2.14 Limits

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

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

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

$$\frac{n \leq m \leq k}{C \vdash [n\,..\,m] : k}$$

3.2.15 Table Types

$\mathit{limits}\ \mathit{reftype}$

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

$$\frac{C \vdash \mathit{limits} : 2^{32} - 1 \qquad C \vdash \mathit{reftype} : \mathsf{ok}}{C \vdash \mathit{limits}\ \mathit{reftype} : \mathsf{ok}}$$

3.2.16 Memory Types

limits

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

$$\frac{C \vdash \mathit{limits} : 2^{16}}{C \vdash \mathit{limits}\ \mathsf{page} : \mathsf{ok}}$$

3.2.17 Global Types

$mut\ valtype$

- The value type $valtype$ must be valid.
- Then the global type is valid.

$$\frac{C \vdash t : \mathsf{ok}}{C \vdash \mathsf{mut}^? \ t : \mathsf{ok}}$$

3.2.18 External Types

$\mathsf{func}\ deftype$

- The defined type $deftype$ must be valid.
- The defined type $deftype$ must be a function type.
- Then the external type is valid.

$$\frac{C \vdash deftype : \mathsf{ok} \qquad deftype \approx \mathsf{func}\ functype}{C \vdash \mathsf{func}\ deftype : \mathsf{ok}}$$

$\mathsf{table}\ tabletype$

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

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

$\mathsf{mem}\ memtype$

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

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

$\mathsf{global}\ globaltype$

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

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

3.3 Matching

On most types, a notion of *subtyping* is defined that is applicable in validation rules, during module instantiation when checking the types of imports, or during execution, when performing casts.

3.3.1 Number Types

A number type $numtype_1$ matches a number type $numtype_2$ if and only if:

- Both $numtype_1$ and $numtype_2$ are the same.

$$\frac{}{C \vdash numtype \leq numtype}$$

3.3.2 Vector Types

A vector type $vectype_1$ matches a vector type $vectype_2$ if and only if:

- Both $vectype_1$ and $vectype_2$ are the same.

$$\frac{}{C \vdash vectype \leq vectype}$$

3.3.3 Heap Types

A heap type $heaptype_1$ matches a heap type $heaptype_2$ if and only if:

- Either both $heaptype_1$ and $heaptype_2$ are the same.
- Or there exists a valid heap type $heaptype'$, such that $heaptype_1$ matches $heaptype'$ and $heaptype'$ matches $heaptype_2$.
- Or $heaptype_1$ is eq and $heaptype_2$ is any.
- Or $heaptype_1$ is one of i31, struct, or array and $heaptype_2$ is eq.
- Or $heaptype_1$ is a defined type which expands to a structure type and $heaptype_2$ is struct.
- Or $heaptype_1$ is a defined type which expands to an array type and $heaptype_2$ is array.
- Or $heaptype_1$ is a defined type which expands to a function type and $heaptype_2$ is func.
- Or $heaptype_1$ is a defined type $deftype_1$ and $heaptype_2$ is a defined type $deftype_2$, and $deftype_1$ matches $deftype_2$.
- Or $heaptype_1$ is a type index x_1, and the defined type $C.\mathsf{types}[x_1]$ matches $heaptype_2$.
- Or $heaptype_2$ is a type index x_2, and $heaptype_1$ matches the defined type $C.\mathsf{types}[x_2]$.
- Or $heaptype_1$ is none and $heaptype_2$ matches any.
- Or $heaptype_1$ is nofunc and $heaptype_2$ matches func.
- Or $heaptype_1$ is noextern and $heaptype_2$ matches extern.
- Or $heaptype_1$ is bot.

$$\frac{}{C \vdash \mathit{heaptype} \leq \mathit{heaptype}} \qquad \frac{C \vdash \mathit{heaptype}' : \mathsf{ok} \qquad C \vdash \mathit{heaptype}_1 \leq \mathit{heaptype}' \qquad C \vdash \mathit{heaptype}' \leq \mathit{heaptype}_2}{C \vdash \mathit{heaptype}_1 \leq \mathit{heaptype}_2}$$

$$\frac{}{C \vdash \mathsf{eq} \leq \mathsf{any}} \qquad \frac{}{C \vdash \mathsf{i31} \leq \mathsf{eq}} \qquad \frac{}{C \vdash \mathsf{struct} \leq \mathsf{eq}} \qquad \frac{}{C \vdash \mathsf{array} \leq \mathsf{eq}}$$

$$\frac{\mathit{deftype} \approx \mathsf{struct}\ \mathit{fieldtype}^*}{C \vdash \mathit{deftype} \leq \mathsf{struct}} \qquad \frac{\mathit{deftype} \approx \mathsf{array}\ \mathit{fieldtype}}{C \vdash \mathit{deftype} \leq \mathsf{array}} \qquad \frac{\mathit{deftype} \approx \mathsf{func}\ \mathit{functype}}{C \vdash \mathit{deftype} \leq \mathsf{func}}$$

$$\frac{C \vdash C.\mathsf{types}[\mathit{typeidx}] \leq \mathit{heaptype}}{C \vdash \mathit{typeidx} \leq \mathit{heaptype}} \qquad \frac{C \vdash \mathit{heaptype} \leq C.\mathsf{types}[\mathit{typeidx}]}{C \vdash \mathit{heaptype} \leq \mathit{typeidx}}$$

$$\frac{C.\mathsf{recs}[i] = \mathsf{sub}\ \mathsf{final}^?\ \mathit{typeuse}^*\ \mathit{ct}}{C \vdash \mathsf{rec}\ i \leq \mathit{typeuse}^*[j]}$$

$$\frac{C \vdash \mathit{heaptype} \leq \mathsf{any}}{C \vdash \mathsf{none} \leq \mathit{heaptype}} \qquad \frac{C \vdash \mathit{heaptype} \leq \mathsf{func}}{C \vdash \mathsf{nofunc} \leq \mathit{heaptype}} \qquad \frac{C \vdash \mathit{heaptype} \leq \mathsf{extern}}{C \vdash \mathsf{noextern} \leq \mathit{heaptype}}$$

$$\frac{}{C \vdash \mathsf{bot} \leq \mathit{heaptype}}$$

3.3.4 Reference Types

A reference type $\mathsf{ref}\ \mathsf{null}_1^?\ \mathit{heaptype}_1$ matches a reference type $\mathsf{ref}\ \mathsf{null}_2^?\ \mathit{heaptype}_2$ if and only if:

- The heap type $\mathit{heaptype}_1$ matches $\mathit{heaptype}_2$.
- null_1 is absent or null_2 is present.

$$\frac{C \vdash \mathit{ht}_1 \leq \mathit{ht}_2}{C \vdash \mathsf{ref}\ \mathit{ht}_1 \leq \mathsf{ref}\ \mathit{ht}_2} \qquad \frac{C \vdash \mathit{ht}_1 \leq \mathit{ht}_2}{C \vdash \mathsf{ref}\ \mathsf{null}^?\ \mathit{ht}_1 \leq \mathsf{ref}\ \mathsf{null}\ \mathit{ht}_2}$$

3.3.5 Value Types

A value type $\mathit{valtype}_1$ matches a value type $\mathit{valtype}_2$ if and only if:

- Either both $\mathit{valtype}_1$ and $\mathit{valtype}_2$ are number types and $\mathit{valtype}_1$ matches $\mathit{valtype}_2$.
- Or both $\mathit{valtype}_1$ and $\mathit{valtype}_2$ are reference types and $\mathit{valtype}_1$ matches $\mathit{valtype}_2$.
- Or $\mathit{valtype}_1$ is bot.

$$\frac{}{C \vdash \mathsf{bot} \leq \mathit{valtype}}$$

3.3.6 Result Types

Subtyping is lifted to result types in a pointwise manner. That is, a result type t_1^* matches a result type t_2^* if and only if:

- Every value type t_1 in $[t_1^*]$ matches the corresponding value type t_2 in $[t_2^*]$.

$$\frac{(C \vdash t_1 \leq t_2)^*}{C \vdash t_1^* \leq t_2^*}$$

3.3.7 Instruction Types

Subtyping is further lifted to instruction types. An instruction type $t_{11}^* \to_{x_1^*} t_{12}^*$ matches a type $t_{21}^* \to_{x_2^*} t_{22}^*$ if and only if:

- There is a common sequence of value types t^* such that t_{21}^* equals $t^*\ t'^*_{21}$ and t_{22}^* equals $t^*\ t'^*_{22}$.
- The result type $[t'^*_{21}]$ matches $[t_{11}^*]$.
- The result type $[t_{12}^*]$ matches $[t'^*_{22}]$.
- For every local index x that is in x_2^* but not in x_1^*, the local type $C.\mathsf{locals}[x]$ is $\mathsf{set}\ t_x$ for some value type t_x.

$$\frac{C \vdash t_{21}^* \leq t_{11}^* \qquad C \vdash t_{12}^* \leq t_{22}^* \qquad x^* = x_2^* \setminus x_1^* \qquad (C.\mathsf{locals}[x] = \mathsf{set}\ t)^*}{C \vdash t_{11}^* \to_{x_1^*} t_{12}^* \leq t_{21}^* \to_{x_2^*} t_{22}^*}$$

Note: Instruction types are contravariant in their input and covariant in their output. Subtyping also incorporates a sort of "frame" condition, which allows adding arbitrary invariant stack elements on both sides in the super type.

Finally, the supertype may ignore variables from the init set x_1^*. It may also *add* variables to the init set, provided these are already set in the context, i.e., are vacuously initialized.

3.3.8 Function Types

A function type $t_{11}^* \to t_{12}^*$ matches a type $t_{21}^* \to t_{22}^*$ if and only if:

- The result type $[t_{21}^*]$ matches $[t_{11}^*]$.
- The result type $[t_{12}^*]$ matches $[t_{22}^*]$.

$$\frac{C \vdash t_{21}^* \leq t_{11}^* \qquad C \vdash t_{12}^* \leq t_{22}^*}{C \vdash t_{11}^* \to t_{12}^* \leq t_{21}^* \to t_{22}^*}$$

3.3.9 Composite Types

A composite type $\mathit{comptype}_1$ matches a type $\mathit{comptype}_2$ if and only if:

- Either the composite type $\mathit{comptype}_1$ is $\mathsf{func}\ \mathit{functype}_1$ and $\mathit{comptype}_2$ is $\mathsf{func}\ \mathit{functype}_2$ and:
  - The function type $\mathit{functype}_1$ matches $\mathit{functype}_2$.
- Or the composite type $\mathit{comptype}_1$ is $\mathsf{struct}\ \mathit{fieldtype}_1^{n_1}$ and $\mathit{comptype}_2$ is $\mathsf{struct}\ \mathit{fieldtype}_2$ and:
  - The arity n_1 is greater than or equal to n_2.
  - For every field type $\mathit{fieldtype}_{2i}$ in $\mathit{fieldtype}_2^{n_2}$ and corresponding $\mathit{fieldtype}_{1i}$ in $\mathit{fieldtype}_1^{n_1}$
    * The field type $\mathit{fieldtype}_{1i}$ matches $\mathit{fieldtype}_{2i}$.
- Or the composite type $\mathit{comptype}_1$ is $\mathsf{array}\ \mathit{fieldtype}_1$ and $\mathit{comptype}_2$ is $\mathsf{array}\ \mathit{fieldtype}_2$ and:
  - The field type $\mathit{fieldtype}_1$ matches $\mathit{fieldtype}_2$.

$$\frac{(C \vdash \mathit{yt}_1 \leq \mathit{yt}_2)^*}{C \vdash \mathsf{struct}\ (\mathit{yt}_1^*\ \mathit{yt'}_1) \leq \mathsf{struct}\ \mathit{yt}_2^*} \qquad \frac{C \vdash \mathit{yt}_1 \leq \mathit{yt}_2}{C \vdash \mathsf{array}\ \mathit{yt}_1 \leq \mathsf{array}\ \mathit{yt}_2} \qquad \frac{C \vdash \mathit{ft}_1 \leq \mathit{ft}_2}{C \vdash \mathsf{func}\ \mathit{ft}_1 \leq \mathsf{func}\ \mathit{ft}_2}$$

3.3.10 Field Types

A field type $(\mathsf{mut}_1^? \; storagetype_1)$ matches a type $(\mathsf{mut}_2^? \; storagetype_2)$ if and only if:

- Storage type $storagetype_1$ matches $storagetype_2$.
- Either both mut_1 and mut_2 are const.
- Or both mut_1 and mut_2 are var and $storagetype_2$ matches $storagetype_1$ as well.

$$\frac{C \vdash zt_1 \leq zt_2}{C \vdash zt_1 \leq zt_2} \qquad \frac{C \vdash zt_1 \leq zt_2 \qquad C \vdash zt_2 \leq zt_1}{C \vdash \mathsf{mut}\; zt_1 \leq \mathsf{mut}\; zt_2}$$

A storage type $storagetype_1$ matches a type $storagetype_2$ if and only if:

- Either $storagetype_1$ is a value type $valtype_1$ and $storagetype_2$ is a value type $valtype_2$ and $valtype_1$ matches $valtype_2$.
- Or $storagetype_1$ is a packed type $packtype_1$ and $storagetype_2$ is a packed type $packtype_2$ and $packtype_1$ matches $packtype_2$.

A packed type $packtype_1$ matches a type $packtype_2$ if and only if:

- The packed type $packtype_1$ is the same as $packtype_2$.

$$\frac{}{C \vdash packtype \leq packtype}$$

3.3.11 Defined Types

A defined type $deftype_1$ matches a type $deftype_2$ if and only if:

- Either $deftype_1$ and $deftype_2$ are equal when closed under context C.
- Or:
  - Let the sub type $\mathsf{sub}\; \mathsf{final}^? \; heaptype^* \; comptype$ be the result of unrolling $deftype_1$.
  - Then there must exist a heap type $heaptype_i$ in $heaptype^*$ that matches $deftype_2$.

$$\frac{\mathsf{clos}_C(deftype_1) = \mathsf{clos}_C(deftype_2)}{C \vdash deftype_1 \leq deftype_2}$$

$$\frac{\mathrm{unroll}(deftype_1) = \mathsf{sub}\; \mathsf{final}^? \; typeuse^* \; ct \qquad C \vdash typeuse^*[i] \leq deftype_2}{C \vdash deftype_1 \leq deftype_2}$$

Note: Note that there is no explicit definition of type *equivalence*, since it coincides with syntactic equality, as used in the premise of the fomer rule above.

3.3.12 Limits

Limits $[n_1 \,..\, m_1]$ match limits $[n_2 \,..\, m_2]$ 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.

$$\frac{n_1 \geq n_2 \qquad m_1 \leq m_2}{C \vdash [n_1 \,..\, m_1] \leq [n_2 \,..\, m_2]}$$

3.3.13 Table Types

A table type $(\mathit{limits}_1\ \mathit{reftype}_1)$ matches $(\mathit{limits}_1\ \mathit{reftype}_1)$ if and only if:

- Limits limits_1 match limits_2.
- The reference type $\mathit{reftype}_1$ matches $\mathit{reftype}_2$, and vice versa.

$$\frac{C \vdash \mathit{limits}_1 \leq \mathit{limits}_2 \qquad C \vdash \mathit{reftype}_1 \leq \mathit{reftype}_2 \qquad C \vdash \mathit{reftype}_2 \leq \mathit{reftype}_1}{C \vdash \mathit{limits}_1\ \mathit{reftype}_1 \leq \mathit{limits}_2\ \mathit{reftype}_2}$$

3.3.14 Memory Types

A memory type $(\mathit{limits}_1\ \mathsf{page})$ matches $(\mathit{limits}_2\ \mathsf{page})$ if and only if:

- Limits limits_1 match limits_2.

$$\frac{C \vdash \mathit{limits}_1 \leq \mathit{limits}_2}{C \vdash \mathit{limits}_1\ \mathsf{page} \leq \mathit{limits}_2\ \mathsf{page}}$$

3.3.15 Global Types

A global type $(\mathsf{mut}_1^?\ \mathit{valtype}_1)$ matches $(\mathsf{mut}_2^?\ \mathit{valtype}_2)$ if and only if:

- Either both mut_1 and mut_2 are var and t_1 matches t_2 and vice versa.
- Or both mut_1 and mut_2 are const and t_1 matches t_2.

$$\frac{C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2}{C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2} \qquad \frac{C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2 \qquad C \vdash \mathit{valtype}_2 \leq \mathit{valtype}_1}{C \vdash \mathsf{mut}\ \mathit{valtype}_1 \leq \mathsf{mut}\ \mathit{valtype}_2}$$

3.3.16 External Types

Functions

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

- The defined type $\mathit{deftype}_1$ matches $\mathit{deftype}_2$.

$$\frac{C \vdash \mathit{deftype}_1 \leq \mathit{deftype}_2}{C \vdash \mathsf{func}\ \mathit{deftype}_1 \leq \mathsf{func}\ \mathit{deftype}_2}$$

Tables

An external type $\mathsf{table}\ \mathit{tabletype}_1$ matches $\mathsf{table}\ \mathit{tabletype}_2$ if and only if:

- Table type $\mathit{tabletype}_1$ matches $\mathit{tabletype}_2$.

$$\frac{C \vdash \mathit{tabletype}_1 \leq \mathit{tabletype}_2}{C \vdash \mathsf{table}\ \mathit{tabletype}_1 \leq \mathsf{table}\ \mathit{tabletype}_2}$$

Memories

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

- Memory type $\mathit{memtype}_1$ matches $\mathit{memtype}_2$.

$$\frac{C \vdash \mathit{memtype}_1 \leq \mathit{memtype}_2}{C \vdash \mathsf{mem}\ \mathit{memtype}_1 \leq \mathsf{mem}\ \mathit{memtype}_2}$$

Globals

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

- Global type $\mathit{globaltype}_1$ matches $\mathit{globaltype}_2$.

$$\frac{C \vdash \mathit{globaltype}_1 \leq \mathit{globaltype}_2}{C \vdash \mathsf{global}\ \mathit{globaltype}_1 \leq \mathsf{global}\ \mathit{globaltype}_2}$$

3.4 Instructions

Instructions are classified by instruction types that describe how they manipulate the operand stack and initialize locals: A type $t_1^* \rightarrow_{x^*} t_2^*$ describes 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. Moreover, it enumerates the indices x^* of locals that have been set by the instruction. In most cases, this is empty.

Note: For example, the instruction $\mathsf{binop}\ \mathsf{i32}\ \mathsf{add}$ has type $\mathsf{i32}\ \mathsf{i32} \rightarrow \mathsf{i32}$, consuming two $\mathsf{i32}$ values and producing one. The instruction $\mathsf{local.set}\ x$ has type $t \rightarrow_x \epsilon$, provided t is the type declared for the local x.

Typing extends to instruction sequences instr^*. Such a sequence has an instruction type $t_1^* \rightarrow_{x^*} t_2^*$ if the accumulative effect of executing the instructions is consuming values of types t_1^* off the operand stack, pushing new values of types t_2^*, and setting all locals x^*.

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

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

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

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

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

and

$$(\mathsf{const}\ \mathsf{f64}\ 1.0)\ (\mathsf{const}\ \mathsf{f64}\ 2.0)\ (\mathsf{const}\ \mathsf{f64}\ 3.0)\ \mathsf{select}$$

are valid, with t in the typing of select being instantiated to $\mathsf{i32}$ or $\mathsf{f64}$, respectively.

The $\mathsf{unreachable}$ instruction is stack-polymorphic, and hence valid with type $t_1^* \rightarrow t_2^*$ for any possible sequences of value types t_1^* and t_2^*. Consequently,

$$\mathsf{unreachable}\ \mathsf{binop}\ \mathsf{i32}\ \mathsf{add}$$

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

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

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

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

3.4.1 Parametric Instructions

nop

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

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \rightarrow \epsilon}$$

unreachable

- The instruction is valid with any valid type of the form $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{C \vdash t_1^* \rightarrow t_2^* : \mathsf{ok}}{C \vdash \mathsf{unreachable} : t_1^* \rightarrow t_2^*}$$

Note: The unreachable instruction is stack-polymorphic.

---

drop

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

$$\frac{C \vdash t : \mathsf{ok}}{C \vdash \mathsf{drop} : t \rightarrow \epsilon}$$

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

---

select $(t^*)^?$

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

$$\frac{C \vdash t : \mathsf{ok}}{C \vdash \mathsf{select}\ t : t\ t\ \mathsf{i32} \rightarrow t} \qquad \frac{C \vdash t : \mathsf{ok} \qquad C \vdash t \leq t' \qquad t' = \mathit{numtype} \vee t' = \mathit{vectype}}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \rightarrow t}$$

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

3.4.2 Numeric Instructions

$t.\mathsf{const}\ c$

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

$$\frac{}{C \vdash nt.\mathsf{const}\ c_{nt} : \epsilon \rightarrow nt}$$

$t.unop$

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

$$\frac{}{C \vdash nt.unop_{nt} : nt \rightarrow nt}$$

$t.binop$

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

$$\frac{}{C \vdash nt.binop_{nt} : nt\ nt \rightarrow nt}$$

$t.testop$

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

$$\frac{}{C \vdash nt.testop_{nt} : nt \rightarrow \mathsf{i32}}$$

$t.relop$

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

$$\frac{}{C \vdash nt.relop_{nt} : nt\ nt \rightarrow \mathsf{i32}}$$

$t_2.cvtop\_t_1\_sx^?$

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

$$\frac{|nt_1| = |nt_2|}{C \vdash nt_1.\mathsf{reinterpret\_}nt_2 : nt_2 \rightarrow nt_1}$$

$$\frac{sx^? = \epsilon \Leftrightarrow nt_1 = \mathsf{i}N_1 \wedge nt_2 = \mathsf{i}N_2 \wedge |nt_1| > |nt_2| \vee nt_1 = \mathsf{f}N_1 \wedge nt_2 = \mathsf{f}N_2}{C \vdash nt_1.\mathsf{convert\_}nt_2\mathsf{\_}sx^? : nt_2 \rightarrow nt_1}$$

3.4.3 Reference Instructions

$\mathsf{ref.null}\ ht$

- The heap type ht must be valid.
- Then the instruction is valid with type $[] \rightarrow [(\mathsf{ref\ null}\ ht)]$.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.null}\ ht : \epsilon \rightarrow (\mathsf{ref\ null}\ ht)}$$

$\mathsf{ref.func}\ x$

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

$$\frac{C.\mathsf{funcs}[x] = dt \qquad x \in C.\mathsf{refs}}{C \vdash \mathsf{ref.func}\ x : \epsilon \to (\mathsf{ref}\ dt)}$$

$\mathsf{ref.is\_null}$

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ ht)] \to [\mathsf{i32}]$, for any valid heap type ht.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.is\_null} : (\mathsf{ref}\ \mathsf{null}\ ht) \to \mathsf{i32}}$$

$\mathsf{ref.as\_non\_null}$

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ ht)] \to [(\mathsf{ref}\ ht)]$, for any valid heap type ht.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.as\_non\_null} : (\mathsf{ref}\ \mathsf{null}\ ht) \to (\mathsf{ref}\ ht)}$$

$\mathsf{ref.eq}$

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ \mathsf{eq})(\mathsf{ref}\ \mathsf{null}\ \mathsf{eq})] \to [\mathsf{i32}]$.

$$\frac{}{C \vdash \mathsf{ref.eq} : (\mathsf{ref}\ \mathsf{null}\ \mathsf{eq})\ (\mathsf{ref}\ \mathsf{null}\ \mathsf{eq}) \to \mathsf{i32}}$$

$\mathsf{ref.test}\ rt$

- The reference type rt must be valid.
- Then the instruction is valid with type $[rt'] \to [\mathsf{i32}]$ for any valid reference type rt' for which rt matches rt'.

$$\frac{C \vdash rt : \mathsf{ok} \qquad C \vdash rt' : \mathsf{ok} \qquad C \vdash rt \leq rt'}{C \vdash \mathsf{ref.test}\ rt : rt' \to \mathsf{i32}}$$

Note: The liberty to pick a supertype rt' allows typing the instruction with the least precise super type of rt as input, that is, the top type in the corresponding heap subtyping hierarchy.

$\mathsf{ref.cast}\ rt$

- The reference type rt must be valid.
- Then the instruction is valid with type $[rt'] \to [rt]$ for any valid reference type rt' for which rt matches rt'.

$$\frac{C \vdash rt : \mathsf{ok} \qquad C \vdash rt' : \mathsf{ok} \qquad C \vdash rt \leq rt'}{C \vdash \mathsf{ref.cast}\ rt : rt' \to rt}$$

Note: The liberty to pick a supertype rt' allows typing the instruction with the least precise super type of rt as input, that is, the top type in the corresponding heap subtyping hierarchy.

3.4.4 Aggregate Reference Instructions

$\mathsf{struct.new}\ x$

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be a structure type $\mathsf{struct}\ \mathit{fieldtype}^*$.
- For each field type $\mathit{fieldtype}_i$ in $\mathit{fieldtype}^*$:
  - Let $\mathit{fieldtype}_i$ be $\mathit{mut}\ \mathit{storagetype}_i$.
  - Let t_i be the value type $\mathrm{unpack}(\mathit{storagetype}_i)$.
- Let t^* be the concatenation of all t_i.
- Then the instruction is valid with type $[t^*] \rightarrow [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ (\mathsf{mut}^?\ \mathit{zt})^*}{C \vdash \mathsf{struct.new}\ x : \mathrm{unpack}(\mathit{zt})^* \rightarrow (\mathsf{ref}\ x)}$$

$\mathsf{struct.new\_default}\ x$

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be a structure type $\mathsf{struct}\ \mathit{fieldtype}^*$.
- For each field type $\mathit{fieldtype}_i$ in $\mathit{fieldtype}^*$:
  - Let $\mathit{fieldtype}_i$ be $\mathit{mut}\ \mathit{storagetype}_i$.
  - Let t_i be the value type $\mathrm{unpack}(\mathit{storagetype}_i)$.
  - The type t_i must be defaultable.
- Let t^* be the concatenation of all t_i.
- Then the instruction is valid with type $[] \rightarrow [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ (\mathsf{mut}^?\ \mathit{zt})^* \qquad (\mathrm{default}_{\mathrm{unpack}(\mathit{zt})} = \mathit{val})^*}{C \vdash \mathsf{struct.new\_default}\ x : \epsilon \rightarrow (\mathsf{ref}\ x)}$$

$\mathsf{struct.get\_}\mathit{sx}^?\ x\ y$

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be a structure type $\mathsf{struct}\ \mathit{fieldtype}^*$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}^*[y]$.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- The extension sx must be present if and only if $\mathit{storagetype}$ is a packed type.
- Then the instruction is valid with type $[(\mathsf{ref\ null}\ x)] \rightarrow [t]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ \mathit{yt}^* \qquad \mathit{yt}^*[i] = \mathsf{mut}^?\ \mathit{zt} \qquad \mathit{sx}^? = \epsilon \Leftrightarrow \mathit{zt} = \mathrm{unpack}(\mathit{zt})}{C \vdash \mathsf{struct.get\_}\mathit{sx}^?\ x\ i : (\mathsf{ref\ null}\ x) \rightarrow \mathrm{unpack}(\mathit{zt})}$$

struct.set x y

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be a structure type $\mathsf{struct}\ \mathit{fieldtype}^*$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}^*[y]$.
- The prefix mut must be var.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ t] \to []$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ \mathit{yt}^* \qquad \mathit{yt}^*[i] = \mathsf{mut}\ \mathit{zt}}{C \vdash \mathsf{struct.set}\ x\ i : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathrm{unpack}(\mathit{zt}) \to \epsilon}$$

array.new x

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let $\mathit{fieldtype}$ be $\mathit{mut}\ \mathit{storagetype}$.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- Then the instruction is valid with type $[t\ \mathsf{i32}] \to [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ \mathit{zt})}{C \vdash \mathsf{array.new}\ x : \mathrm{unpack}(\mathit{zt})\ \mathsf{i32} \to (\mathsf{ref}\ x)}$$

array.new_default x

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let $\mathit{fieldtype}$ be $\mathit{mut}\ \mathit{storagetype}$.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- The type t must be defaultable.
- Then the instruction is valid with type $[\mathsf{i32}] \to [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ \mathit{zt}) \qquad \mathrm{default}_{\mathrm{unpack}(\mathit{zt})} = \mathit{val}}{C \vdash \mathsf{array.new\_default}\ x : \mathsf{i32} \to (\mathsf{ref}\ x)}$$

array.new_fixed x n

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let $\mathit{fieldtype}$ be $\mathit{mut}\ \mathit{storagetype}$.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- Then the instruction is valid with type $[t^n] \to [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ \mathit{zt})}{C \vdash \mathsf{array.new\_fixed}\ x\ n : \mathrm{unpack}(\mathit{zt})^n \to (\mathsf{ref}\ x)}$$

array.new_elem x y

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type array *fieldtype*.
- Let *fieldtype* be *mut storagetype*.
- The storage type *storagetype* must be a reference type rt.
- The element segment $C.\mathsf{elems}[y]$ must exist.
- Let rt' be the reference type $C.\mathsf{elems}[y]$.
- The reference type rt' must match rt.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}] \rightarrow [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ rt) \qquad C \vdash C.\mathsf{elems}[y] \leq rt}{C \vdash \mathsf{array.new\_elem}\ x\ y : \mathsf{i32}\ \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

array.new_data x y

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type array *fieldtype*.
- Let *fieldtype* be *mut storagetype*.
- Let t be the value type $\mathrm{unpack}(storagetype)$.
- The type t must be a numeric type or a vector type.
- The data segment $C.\mathsf{datas}[y]$ must exist.
- Then the instruction is valid with type $[\mathsf{i32}\ \mathsf{i32}] \rightarrow [(\mathsf{ref}\ x)]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \qquad \mathrm{unpack}(zt) = numtype \lor \mathrm{unpack}(zt) = vectype \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{array.new\_data}\ x\ y : \mathsf{i32}\ \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

array.get_$sx^?$ x

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type array *fieldtype*.
- Let the field type *mut storagetype* be *fieldtype*.
- Let t be the value type $\mathrm{unpack}(storagetype)$.
- The extension sx must be present if and only if *storagetype* is a packed type.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}] \rightarrow [t]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \qquad sx^? = \epsilon \Leftrightarrow zt = \mathrm{unpack}(zt)}{C \vdash \mathsf{array.get\_}sx^?\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32} \rightarrow \mathrm{unpack}(zt)}$$

array.set x

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}$.
- The prefix mut must be var.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ t] \rightarrow []$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ \mathit{zt})}{C \vdash \mathsf{array.set}\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathrm{unpack}(\mathit{zt}) \rightarrow \epsilon}$$

array.len

- The the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ \mathsf{array})] \rightarrow [\mathsf{i32}]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ \mathit{zt})}{C \vdash \mathsf{array.len} : (\mathsf{ref}\ \mathsf{null}\ \mathsf{array}) \rightarrow \mathsf{i32}}$$

array.fill x

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}$.
- The prefix mut must be var.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ t\ \mathsf{i32}] \rightarrow []$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ \mathit{zt})}{C \vdash \mathsf{array.fill}\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathrm{unpack}(\mathit{zt})\ \mathsf{i32} \rightarrow \epsilon}$$

array.copy x y

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}_1$.
- Let the field type $\mathit{mut}_1\ \mathit{storagetype}_1$ be $\mathit{fieldtype}_1$.
- The prefix mut_1 must be var.
- The defined type $C.\mathsf{types}[y]$ must exist.
- The expansion of $C.\mathsf{types}[y]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}_2$.
- Let the field type $\mathit{mut}_2\ \mathit{storagetype}_2$ be $\mathit{fieldtype}_2$.
- The storage type $\mathit{storagetype}_2$ must match $\mathit{storagetype}_1$.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ (\mathsf{ref}\ \mathsf{null}\ y)\ \mathsf{i32}\ \mathsf{i32}] \rightarrow []$.

$$\frac{C.\mathsf{types}[x_1] \approx \mathsf{array}\ (\mathsf{mut}\ \mathit{zt}_1) \qquad C.\mathsf{types}[x_2] \approx \mathsf{array}\ (\mathsf{mut}^?\ \mathit{zt}_2) \qquad C \vdash \mathit{zt}_2 \leq \mathit{zt}_1}{C \vdash \mathsf{array.copy}\ x_1\ x_2 : (\mathsf{ref}\ \mathsf{null}\ x_1)\ \mathsf{i32}\ (\mathsf{ref}\ \mathsf{null}\ x_2)\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

$\mathsf{array.init\_elem}\ x\ y$

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}$.
- The prefix mut must be var.
- The storage type $\mathit{storagetype}$ must be a reference type rt.
- The element segment $C.\mathsf{elems}[y]$ must exist.
- Let rt' be the reference type $C.\mathsf{elems}[y]$.
- The reference type rt' must match rt.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt) \qquad C \vdash C.\mathsf{elems}[y] \leq zt}{C \vdash \mathsf{array.init\_elem}\ x\ y : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}$$

$\mathsf{array.init\_data}\ x\ y$

- The defined type $C.\mathsf{types}[x]$ must exist.
- The expansion of $C.\mathsf{types}[x]$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let the field type $\mathit{mut}\ \mathit{storagetype}$ be $\mathit{fieldtype}$.
- The prefix mut must be var.
- Let t be the value type $\mathrm{unpack}(\mathit{storagetype})$.
- The value type t must be a numeric type or a vector type.
- The data segment $C.\mathsf{datas}[y]$ must exist.
- Then the instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32}] \to []$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt) \qquad \mathrm{unpack}(zt) = \mathit{numtype} \vee \mathrm{unpack}(zt) = \mathit{vectype} \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{array.init\_data}\ x\ y : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}$$

3.4.5 Scalar Reference Instructions

$\mathsf{ref.i31}$

- The instruction is valid with type $[\mathsf{i32}] \to [(\mathsf{ref}\ \mathsf{i31})]$.

$$\frac{}{C \vdash \mathsf{ref.i31} : \mathsf{i32} \to (\mathsf{ref}\ \mathsf{i31})}$$

$\mathsf{i31.get\_}sx$

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}\ \mathsf{i31})] \to [\mathsf{i32}]$.

$$\frac{}{C \vdash \mathsf{i31.get\_}sx : (\mathsf{ref}\ \mathsf{null}\ \mathsf{i31}) \to \mathsf{i32}}$$

3.4.6 External Reference Instructions

any.convert_extern

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}_1^?\ \mathsf{extern})] \to [(\mathsf{ref}\ \mathsf{null}_2^?\ \mathsf{any})]$ for any $\mathsf{null}_1^?$ that equals $\mathsf{null}_2^?$.

$$\frac{}{C \vdash \mathsf{any.convert\_extern} : (\mathsf{ref}\ \mathsf{null}^?\ \mathsf{extern}) \to (\mathsf{ref}\ \mathsf{null}^?\ \mathsf{any})}$$

extern.convert_any

- The instruction is valid with type $[(\mathsf{ref}\ \mathsf{null}_1^?\ \mathsf{any})] \to [(\mathsf{ref}\ \mathsf{null}_2^?\ \mathsf{extern})]$ for any $\mathsf{null}_1^?$ that equals $\mathsf{null}_2^?$.

$$\frac{}{C \vdash \mathsf{extern.convert\_any} : (\mathsf{ref}\ \mathsf{null}^?\ \mathsf{any}) \to (\mathsf{ref}\ \mathsf{null}^?\ \mathsf{extern})}$$

3.4.7 Vector Instructions

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

$$\mathrm{unpack}(\mathsf{i}N\mathsf{x}N) \quad = \quad \mathrm{unpack}(\mathsf{i}N)$$

v128.const c

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

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

v128.$vvunop$

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

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

v128.$vvbinop$

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

v128.$vvternop$

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

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

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

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

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

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathit{vunop} : \mathsf{v128} \to \mathsf{v128}}$$

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathit{vbinop} : \mathsf{v128}~\mathsf{v128} \to \mathsf{v128}}$$

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathit{vtestop} : \mathsf{v128} \to \mathsf{i32}}$$

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathit{vrelop} : \mathsf{v128}~\mathsf{v128} \to \mathsf{v128}}$$

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathit{vshiftop} : \mathsf{v128}~\mathsf{i32} \to \mathsf{v128}}$$

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

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

$$\frac{}{C \vdash \mathit{sh}.\mathsf{bitmask} : \mathsf{v128} \to \mathsf{i32}}$$

$\mathsf{i8x16}.\mathsf{swizzle}$

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

$$\frac{}{C \vdash \mathit{sh}.\mathsf{swizzle} : \mathsf{v128}~\mathsf{v128} \to \mathsf{v128}}$$

i8x16.shuffle $laneidx^{16}$

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

$$\frac{(i < 2 \cdot \mathrm{dim}(sh))^*}{C \vdash sh.\mathsf{shuffle}\ i^* : \mathsf{v128}\ \mathsf{v128} \to \mathsf{v128}}$$

$shape$.splat

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

$$\frac{}{C \vdash sh.\mathsf{splat} : \mathrm{unpack}(sh) \to \mathsf{v128}}$$

$shape$.extract_lane_$sx^?$ $laneidx$

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

$$\frac{i < \mathrm{dim}(sh)}{C \vdash sh.\mathsf{extract\_lane\_}sx^?\ i : \mathsf{v128} \to \mathrm{unpack}(sh)}$$

$shape$.replace_lane $laneidx$

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

$$\frac{i < \mathrm{dim}(sh)}{C \vdash sh.\mathsf{replace\_lane}\ i : \mathsf{v128}\ \mathrm{unpack}(sh) \to \mathsf{v128}}$$

$ishape_1$.extadd_pairwise_$ishape_2$_sx

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

$$\frac{}{C \vdash sh_1.vextunop\_sh_2\_sx : \mathsf{v128} \to \mathsf{v128}}$$

$ishape_1$.extmul_$half$_$ishape_2$_sx

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

$$\frac{}{C \vdash sh_1.vextbinop\_sh_2\_sx : \mathsf{v128}\ \mathsf{v128} \to \mathsf{v128}}$$

$ishape_1.\mathsf{narrow\_}ishape_2\_sx$

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

$$\frac{}{C \vdash sh_1.\mathsf{narrow\_}sh_2\_sx : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$shape.vcvtop\_half^?\_shape\_sx^?\_\mathsf{zero}^?$

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

$$\frac{}{C \vdash sh_1.vcvtop\_sx^?\_sh_2\_half^? : \mathsf{v128} \rightarrow \mathsf{v128}}$$

3.4.8 Variable Instructions

$\mathsf{local.get}\ x$

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

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

$\mathsf{local.set}\ x$

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

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

$\mathsf{local.tee}\ x$

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

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

global.get 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 $[] \to [t]$.

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

global.set 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 $[t] \to []$.

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

3.4.9 Table Instructions

table.get x

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}}{C \vdash \mathsf{table.get}\ x : \mathsf{i32} \to \mathit{rt}}$$

table.set x

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}}{C \vdash \mathsf{table.set}\ x : \mathsf{i32}\ \mathit{rt} \to \epsilon}$$

table.size x

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}}{C \vdash \mathsf{table.size}\ x : \epsilon \to \mathsf{i32}}$$

$\mathsf{table.grow}\ x$

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}}{C \vdash \mathsf{table.grow}\ x : \mathit{rt}\ \mathsf{i32} \to \mathsf{i32}}$$

$\mathsf{table.fill}\ x$

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}}{C \vdash \mathsf{table.fill}\ x : \mathsf{i32}\ \mathit{rt}\ \mathsf{i32} \to \epsilon}$$

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

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

$$\frac{C.\mathsf{tables}[x_1] = \mathit{lim}_1\ \mathit{rt}_1 \qquad C.\mathsf{tables}[x_2] = \mathit{lim}_2\ \mathit{rt}_2 \qquad C \vdash \mathit{rt}_2 \leq \mathit{rt}_1}{C \vdash \mathsf{table.copy}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}$$

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

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}_1 \qquad C.\mathsf{elems}[y] = \mathit{rt}_2 \qquad C \vdash \mathit{rt}_2 \leq \mathit{rt}_1}{C \vdash \mathsf{table.init}\ x\ y : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}$$

$\mathsf{elem.drop}\ x$

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

$$\frac{C.\mathsf{elems}[x] = rt}{C \vdash \mathsf{elem.drop}\ x : \epsilon \to \epsilon}$$

3.4.10 Memory Instructions

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

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq |nt|/8}{C \vdash nt.\mathsf{load}\ x\ memarg : \mathsf{i32} \to nt}$$

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

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq M/8}{C \vdash \mathsf{i}N.\mathsf{load}M\ sx\ x\ memarg : \mathsf{i32} \to \mathsf{i}N}$$

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

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq |nt|/8}{C \vdash nt.\mathsf{store}\ x\ memarg : \mathsf{i32}\ nt \to \epsilon}$$

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

- The memory $C.\mathsf{mems}[x]$ 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 $[\mathsf{i32}\ t] \to []$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq M/8}{C \vdash \mathsf{i}N.\mathsf{store}M\ x\ memarg : \mathsf{i32}\ \mathsf{i}N \to \epsilon}$$

v128..load x $memarg$

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq |\mathsf{v128}|/8}{C \vdash \mathsf{v128.load}\ x\ memarg : \mathsf{i32} \to \mathsf{v128}}$$

v128.load$N$xM_sx x $memarg$

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

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

v128.load$N$_splat x $memarg$

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

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

v128.load$N$_zero x $memarg$

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

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

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

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

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

$\mathsf{v128.store}\ x\ memarg$

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq |\mathsf{v128}|/8}{C \vdash \mathsf{v128.store}\ x\ memarg : \mathsf{i32}\ \mathsf{v128} \to \epsilon}$$

$\mathsf{v128.store}N\mathsf{\_lane}\ x\ memarg\ laneidx$

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memarg.\mathsf{align}} \leq N/8 \qquad i < 128/N}{C \vdash \mathsf{v128.store}N\mathsf{\_lane}\ x\ memarg\ i : \mathsf{i32}\ \mathsf{v128} \to \epsilon}$$

$\mathsf{memory.size}\ x$

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

$$\frac{C.\mathsf{mems}[x] = mt}{C \vdash \mathsf{memory.size}\ x : \epsilon \to \mathsf{i32}}$$

$\mathsf{memory.grow}\ x$

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

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

$\mathsf{memory.fill}\ x$

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

$$\frac{C.\mathsf{mems}[x] = mt}{C \vdash \mathsf{memory.fill}\ x : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}$$

memory.copy x y

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

$$\frac{C.\mathsf{mems}[x_1] = mt_1 \qquad C.\mathsf{mems}[x_2] = mt_2}{C \vdash \mathsf{memory.copy}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

memory.init x y

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{memory.init}\ x\ y : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

data.drop x

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

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

3.4.11 Control Instructions

block *blocktype* $instr^*$ end

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

$$\frac{C \vdash bt : t_1^* \rightarrow t_2^* \qquad \{\mathsf{labels}\ (t_2^*)\} \oplus C \vdash instr^* : t_1^* \rightarrow_{x^*} t_2^*}{C \vdash \mathsf{block}\ bt\ instr^* : t_1^* \rightarrow t_2^*}$$

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

loop *blocktype* $instr^*$ end

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

$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad \{\mathsf{labels}\ (t_1^*)\} \oplus C \vdash instr^* : t_1^* \to_{x^*} t_2^*}{C \vdash \mathsf{loop}\ bt\ instr^* : t_1^* \to t_2^*}$$

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

$\mathsf{if}\ blocktype\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end}$

- The block type must be valid as some instruction 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 list.
- 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 $[t_1^*\ \mathsf{i32}] \to [t_2^*]$.

$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad \{\mathsf{labels}\ (t_2^*)\} \oplus C \vdash instr_1^* : t_1^* \to_{x_1^*} t_2^* \qquad \{\mathsf{labels}\ (t_2^*)\} \oplus C \vdash instr_2^* : t_1^* \to_{x_2^*} t_2^*}{C \vdash \mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^* : t_1^*\ \mathsf{i32} \to t_2^*}$$

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

$\mathsf{br}\ l$

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

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

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

The br instruction is stack-polymorphic.

$\mathsf{br\_if}\ l$

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

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

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

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

- The label $C.\mathsf{labels}[l_N]$ must be defined in the context.
- For each label l_i in l^*, the label $C.\mathsf{labels}[l_i]$ must be defined in the context.
- There must be a sequence t^* of value types, such that:
  - The result type $[t^*]$ matches $C.\mathsf{labels}[l_N]$.
  - For all l_i in l^*, the result type $[t^*]$ matches $C.\mathsf{labels}[l_i]$.
- Then the instruction is valid with any valid type of the form $[t_1^*\ t^*\ \mathsf{i32}] \to [t_2^*]$.

$$\frac{(C \vdash t^* \leq C.\mathsf{labels}[l])^* \qquad C \vdash t^* \leq C.\mathsf{labels}[l'] \qquad C \vdash t_1^* \to t_2^* : \mathsf{ok}}{C \vdash \mathsf{br\_table}\ l^*\ l' : t_1^*\ t^* \to t_2^*}$$

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

The br_table instruction is stack-polymorphic.

Furthermore, the result type t^* is also chosen non-deterministically in this rule. Although it may seem necessary to compute t^* as the greatest lower bound of all label types in practice, a simple linear algorithm does not require this.

$\mathsf{br\_on\_null}\ l$

- 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 $[t^*\ (\mathsf{ref\ null}\ ht)] \to [t^*\ (\mathsf{ref}\ ht)]$ for any valid heap type ht.

$$\frac{C.\mathsf{labels}[l] = t^* \qquad C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{br\_on\_null}\ l : t^*\ (\mathsf{ref\ null}\ ht) \to t^*\ (\mathsf{ref}\ ht)}$$

$\mathsf{br\_on\_non\_null}\ l$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t'^*]$ be the result type $C.\mathsf{labels}[l]$.
- The result type $[t'^*]$ must contain at least one type.
- Let the value type t_l be the last element in the sequence t'^*, and $[t^*]$ the remainder of the sequence preceding it.
- The value type t_l must be a reference type of the form $\mathsf{ref\ null}^?\ ht$.
- Then the instruction is valid with type $[t^*\ (\mathsf{ref\ null}\ ht)] \to [t^*]$.

$$\frac{C.\mathsf{labels}[l] = t^*\ (\mathsf{ref}\ ht)}{C \vdash \mathsf{br\_on\_non\_null}\ l : t^*\ (\mathsf{ref\ null}\ ht) \to t^*}$$

$\mathsf{br\_on\_cast}\ l\ rt_1\ rt_2$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t_l^*]$ be the result type $C.\mathsf{labels}[l]$.
- The type sequence t_l^* must be of the form $t^*\ rt'$.
- The reference type rt_1 must be valid.
- The reference type rt_2 must be valid.
- The reference type rt_2 must match rt_1.
- The reference type rt_2 must match rt'.
- Let rt_1' be the type difference between rt_1 and rt_2.
- Then the instruction is valid with type $[t^*\ rt_1] \to [t^*\ rt_1']$.

$$\frac{C.\mathsf{labels}[l] = t^*\ rt \qquad C \vdash rt_1 : \mathsf{ok} \qquad C \vdash rt_2 : \mathsf{ok} \qquad C \vdash rt_2 \leq rt_1 \qquad C \vdash rt_2 \leq rt}{C \vdash \mathsf{br\_on\_cast}\ l\ rt_1\ rt_2 : t^*\ rt_1 \to t^*\ (rt_1 \setminus rt_2)}$$

$\mathsf{br\_on\_cast\_fail}\ l\ rt_1\ rt_2$

- The label $C.\mathsf{labels}[l]$ must be defined in the context.
- Let $[t_l^*]$ be the result type $C.\mathsf{labels}[l]$.
- The type sequence t_l^* must be of the form $t^*\ rt'$.
- The reference type rt_1 must be valid.
- The reference type rt_2 must be valid.
- The reference type rt_2 must match rt_1.
- Let rt_1' be the type difference between rt_1 and rt_2.
- The reference type rt_1' must match rt'.
- Then the instruction is valid with type $[t^*\ rt_1] \to [t^*\ rt_2]$.

$$\frac{C.\mathsf{labels}[l] = t^*\ rt \qquad C \vdash rt_1 : \mathsf{ok} \qquad C \vdash rt_2 : \mathsf{ok} \qquad C \vdash rt_2 \leq rt_1 \qquad C \vdash rt_1 \setminus rt_2 \leq rt}{C \vdash \mathsf{br\_on\_cast\_fail}\ l\ rt_1\ rt_2 : t^*\ rt_1 \to t^*\ rt_2}$$

$\mathsf{call}\ x$

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- The expansion of $C.\mathsf{funcs}[x]$ must be a function type $\mathsf{func}\ [t_1^*] \to [t_2^*]$.
- Then the instruction is valid with type $[t_1^*] \to [t_2^*]$.

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}\ (t_1^* \to t_2^*)}{C \vdash \mathsf{call}\ x : t_1^* \to t_2^*}$$

call_ref x

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- The expansion of $C.\mathsf{funcs}[x]$ must be a function type $\mathsf{func}~[t_1^*] \rightarrow [t_2^*]$.
- Then the instruction is valid with type $[t_1^*~(\mathsf{ref~null}~x)] \rightarrow [t_2^*]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}~(t_1^* \rightarrow t_2^*)}{C \vdash \mathsf{call\_ref}~x : t_1^*~(\mathsf{ref~null}~x) \rightarrow t_2^*}$$

call_indirect x y

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

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}~\mathit{rt} \qquad C \vdash \mathit{rt} \leq (\mathsf{ref~null~func}) \qquad C.\mathsf{types}[y] \approx \mathsf{func}~(t_1^* \rightarrow t_2^*)}{C \vdash \mathsf{call\_indirect}~x~y : t_1^*~\mathsf{i32} \rightarrow t_2^*}$$

return

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

$$\frac{C.\mathsf{return} = (t^*) \qquad C \vdash t_1^* \rightarrow t_2^* : \mathsf{ok}}{C \vdash \mathsf{return} : t_1^*~t^* \rightarrow t_2^*}$$

Note: The return instruction is stack-polymorphic.

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

---

return_call x

- The return type $C.\mathsf{return}$ must not be absent in the context.
- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- The expansion of $C.\mathsf{funcs}[x]$ must be a function type $\mathsf{func}~[t_1^*] \rightarrow [t_2^*]$.
- The result type $[t_2^*]$ must match $C.\mathsf{return}$.
- Then the instruction is valid with any valid type $[t_3^*~t_1^*] \rightarrow [t_4^*]$.

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}~(t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = (t'^*_2) \qquad C \vdash t_2^* \leq t'^*_2 \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok}}{C \vdash \mathsf{return\_call}~x : t_3^*~t_1^* \rightarrow t_4^*}$$

Note: The return_call instruction is stack-polymorphic.

$\mathsf{return\_call\_ref}\ x$

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- The expansion of $C.\mathsf{types}[x]$ must be a function type $\mathsf{func}\ [t_1^*] \rightarrow [t_2^*]$.
- The result type $[t_2^*]$ must match $C.\mathsf{return}$.
- Then the instruction is valid with any valid type $[t_3^*\ t_1^*\ (\mathsf{ref\ null}\ x)] \rightarrow [t_4^*]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = (t'^*_2) \qquad C \vdash t_2^* \leq t'^*_2 \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok}}{C \vdash \mathsf{return\_call\_ref}\ x : t_3^*\ t_1^*\ (\mathsf{ref\ null}\ x) \rightarrow t_4^*}$$

Note: The return_call_ref instruction is stack-polymorphic.

$\mathsf{return\_call\_indirect}\ x\ y$

- The return type $C.\mathsf{return}$ must not be empty in the context.
- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}\ t$ be the table type $C.\mathsf{tables}[x]$.
- The reference type t must match type $\mathsf{ref\ null\ func}$.
- The type $C.\mathsf{types}[y]$ must be defined in the context.
- The expansion of $C.\mathsf{types}[y]$ must be a function type $\mathsf{func}\ [t_1^*] \rightarrow [t_2^*]$.
- The result type $[t_2^*]$ must match $C.\mathsf{return}$.
- Then the instruction is valid with type $[t_3^*\ t_1^*\ \mathsf{i32}] \rightarrow [t_4^*]$, for any sequences of value types t_3^* and t_4^*.

$$\frac{\begin{array}{c}C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt} \qquad C \vdash \mathit{rt} \leq (\mathsf{ref\ null\ func}) \\ C.\mathsf{types}[y] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = (t'^*_2) \qquad C \vdash t_2^* \leq t'^*_2 \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok}\end{array}}{C \vdash \mathsf{return\_call\_indirect}\ x\ y : t_3^*\ t_1^*\ \mathsf{i32} \rightarrow t_4^*}$$

Note: The return_call_indirect instruction is stack-polymorphic.

3.4.12 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

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

$$\frac{}{C \vdash \epsilon : \epsilon \rightarrow \epsilon}$$

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

- The instruction $instr$ must be valid with some type $[t_1^*] \to_{x_1^*} [t_2^*]$.
- Let C' be the same context as C, but with:
  - locals the same as in C, except that for every local index x in x_1^*, the local type $\mathsf{locals}[x]$ has been updated to initialization status set.
- Under the context C', the instruction sequence $instr'^*$ must be valid with some type $[t_2^*] \to_{x_2^*} [t_3^*]$.
- Then the combined instruction sequence is valid with type $[t_1^*] \to_{x_1^* x_2^*} [t_3^*]$.

$$\frac{C \vdash instr_1 : t_1^* \to_{x_1^*} t_2^* \qquad (C.\mathsf{locals}[x_1] = init\ t)^* \qquad C[.\mathsf{local}[x_1^*] = (\mathsf{set}\ t)^*] \vdash instr_2^* : t_2^* \to_{x_2^*} t_3^*}{C \vdash instr_1\ instr_2^* : t_1^* \to_{x_1^*\ x_2^*} t_3^*}$$

Subsumption for $instr^*$

- The instruction sequence $instr^*$ must be valid with some type $instrtype$.
- The instruction type $instrtype'$: must be a valid
- The instruction type $instrtype$ must match the type $instrtype'$.
- Then the instruction sequence $instr^*$ is also valid with type $instrtype'$.

$$\frac{C \vdash instr^* : it \qquad C \vdash it \leq it' \qquad C \vdash it' : \mathsf{ok}}{C \vdash instr^* : it'}$$

$$\frac{C \vdash instr^* : t_1^* \to_{x^*} t_2^* \qquad C \vdash t^* : \mathsf{ok}}{C \vdash instr^* : (t^*\ t_1^*) \to_{x^*} (t^*\ t_2^*)}$$

Note: In combination with the previous rule, subsumption allows to compose instructions whose types would not directly fit otherwise. For example, consider the instruction sequence

$$(\mathsf{const}\ \mathsf{i31}\ 1)\ (\mathsf{const}\ \mathsf{i32}\ 2)\ \mathsf{binop}\ \mathsf{i32}\ \mathsf{add}$$

$$(\mathsf{i32.const}\ 1)\ (\mathsf{i32.const}\ 2)\ \mathsf{i32.add}$$

To type this sequence, its subsequence $(\mathsf{const}\ \mathsf{i32}\ 2)\ \mathsf{binop}\ \mathsf{i32}\ \mathsf{add}$ needs to be valid with an intermediate type. But the direct type of $(\mathsf{const}\ \mathsf{i32}\ 2)$ is $\epsilon \to \mathsf{i32}$, not matching the two inputs expected by $\mathsf{binop}\ \mathsf{i32}\ \mathsf{add}$. The subsumption rule allows to weaken the type of $(\mathsf{const}\ \mathsf{i32}\ 2)$ to the supertype $\mathsf{i32} \to \mathsf{i32}\ \mathsf{i32}$, such that it can be composed with $\mathsf{add}\ \mathsf{i32}$ and yields the intermediate type $\mathsf{i32} \to \mathsf{i32}\ \mathsf{i32}$ for the subsequence. That can in turn be composed with the first constant.

Furthermore, subsumption allows to drop init variables x^* from the instruction type in a context where they are not needed, for example, at the end of the body of a block.

3.4.13 Expressions

Expressions $expr$ are classified by result types t^*.

$instr^*$ end

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

$$\frac{C \vdash instr^* : \epsilon \to_\epsilon t^*}{C \vdash instr^* : t^*}$$

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 $\mathsf{i}nn.ibinop$, where $ibinop$ is limited to add, sub, or mul.
  - or of the form ref.null,
  - or of the form ref.i31,
  - or of the form ref.func x,
  - or of the form struct.new x,
  - or of the form struct.new_default x,
  - or of the form array.new x,
  - or of the form array.new_default x,
  - or of the form array.new_fixed x,
  - or of the form any.convert_extern,
  - or of the form extern.convert_any,
  - or of the form global.get x, in which case $C.\mathsf{globals}[x]$ must be a global type of the form const t.

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

$$\frac{}{C \vdash (nt.\mathsf{const}\ c_{nt})\ \mathsf{const}} \qquad \frac{}{C \vdash (vt.\mathsf{const}\ c_{vt})\ \mathsf{const}} \qquad \frac{\mathsf{i}N \in \mathsf{i32}\ \mathsf{i64} \qquad binop \in \mathsf{add}\ \mathsf{sub}\ \mathsf{mul}}{C \vdash (\mathsf{i}N.binop)\ \mathsf{const}}$$

$$\frac{}{C \vdash (\mathsf{ref.null}\ ht)\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{ref.i31})\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{ref.func}\ x)\ \mathsf{const}}$$

$$\frac{}{C \vdash (\mathsf{struct.new}\ x)\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{struct.new\_default}\ x)\ \mathsf{const}}$$

$$\frac{}{C \vdash (\mathsf{array.new}\ x)\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{array.new\_default}\ x)\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{array.new\_fixed}\ x\ n)\ \mathsf{const}}$$

$$\frac{}{C \vdash (\mathsf{any.convert\_extern})\ \mathsf{const}} \qquad \frac{}{C \vdash (\mathsf{extern.convert\_any})\ \mathsf{const}}$$

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

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

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

3.5 Modules

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

3.5.1 Types

The sequence of types defined in a module is validated incrementally, yielding a sequence of defined types representing them individually.

$type$

$$\frac{x = |C.\mathsf{types}| \qquad dt^* = \mathrm{roll}_x^*(rectype) \qquad C \oplus \{\mathsf{types}\ dt^*\} \vdash rectype : \mathsf{ok}(x)}{C \vdash \mathsf{type}\ rectype : dt^*}$$

$type^*$

- If the sequence is empty, then:
  - The context C must be empty.
  - Then the type sequence is valid.
- Otherwise:
  - Let the recursive type $rectype$ be the last element in the sequence.
  - The sequence without $rectype$ must be valid for some context C'.
  - Let the type index x be the length of $C'.\mathsf{types}$, i.e., the first type index free in C'.
  - Let the sequence of defined types $deftype^*$ be the result $\mathrm{roll}_x^*(rectype)$ of rolling up into its sequence of defined types.
  - The recursive type $rectype$ must be valid under the context C for type index x.
  - The current context C be the same as C', but with $deftype^*$ appended to types.
  - Then the type sequence is valid.

$$\frac{}{C \vdash \epsilon : \epsilon} \qquad \frac{C \vdash type_1 : dt_1^* \qquad C \oplus \{\mathsf{types}\ dt_1^*\} \vdash type^* : dt^*}{C \vdash type_1\ type^* : dt_1^*\ dt^*}$$

3.5.2 Functions

Functions $func$ are classified by defined types that expand to function types of the form $\mathsf{func}\ (t_1^* \to t_2^*)$.

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

- The defined type $C.\mathsf{types}[x]$ must be a function type.
- Let $\mathsf{func}\ [t_1^*] \to [t_2^*]$ be the expansion of the defined type $C.\mathsf{types}[x]$.
- For each local declared by a value type t in t^*:
  - The local for type t must be valid with local type $localtype_i$.
- Let $localtype^*$ be the concatenation of all $localtype_i$.
- Let C' be the same context as C, but with:

– locals set to the sequence of value types $(\mathsf{set}\ t_1)^*\ \mathit{localtype}^*$, 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 $C.\mathsf{types}[x]$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}\ (t_1^* \to t_2^*) \qquad (C \vdash \mathit{local} : \mathit{lt})^* \qquad C \oplus \{\mathsf{locals}\ (\mathsf{set}\ t_1)^*\ \mathit{lt}^*,\ \mathsf{labels}\ (t_2^*),\ \mathsf{return}\ (t_2^*)\} \vdash \mathit{expr} : t_2^*}{C \vdash \mathsf{func}\ x\ \mathit{local}^*\ \mathit{expr} : C.\mathsf{types}[x]}$$

3.5.3 Locals

Locals local are classified with local types.

$\{\mathsf{type}\ \mathit{valtype}\}$

- The value type $\mathit{valtype}$ must be valid.
- If $\mathit{valtype}$ is defaultable, then:
  – The local is valid with local type set $\mathit{valtype}$.
- Else:
  – The local is valid with local type unset $\mathit{valtype}$.

$$\frac{\mathrm{default}_t \neq \epsilon}{C \vdash \mathsf{local}\ t : \mathsf{set}\ t} \qquad \frac{\mathrm{default}_t = \epsilon}{C \vdash \mathsf{local}\ t : \mathsf{unset}\ t}$$

Note: For cases where both rules are applicable, the former yields the more permissable type.

3.5.4 Tables

Tables table are classified by table types.

$\{\mathsf{type}\ \mathit{tabletype}, \mathsf{init}\ \mathit{expr}\}$

- The table type $\mathit{tabletype}$ must be valid.
- Let t be the element reference type of $\mathit{tabletype}$.
- The expression expr must be valid with result type $[t]$.
- The expression expr must be constant.
- Then the table definition is valid with type $\mathit{tabletype}$.

$$\frac{C \vdash \mathit{tt} : \mathsf{ok} \qquad \mathit{tabletype} = \mathit{lim}\ \mathit{rt} \qquad C \vdash \mathit{expr} : \mathit{rt}\ \mathsf{const}}{C \vdash \mathsf{table}\ \mathit{tabletype}\ \mathit{expr} : \mathit{tabletype}}$$

3.5.5 Memories

Memories mem are classified by memory types.

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

- The memory type $memtype$ must be valid.
- Then the memory definition is valid with type $memtype$.

$$\frac{C \vdash memtype : \mathsf{ok}}{C \vdash \mathsf{memory}~memtype : memtype}$$

3.5.6 Globals

Globals $global$ are classified by global types.

Sequences of globals are handled incrementally, such that each definition has access to previous definitions.

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

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

$$\frac{C \vdash gt : \mathsf{ok} \qquad globaltype = \mathsf{mut}^?~t \qquad C \vdash expr : t~\mathsf{const}}{C \vdash \mathsf{global}~globaltype~expr : globaltype}$$

$global^*$

- If the sequence is empty, then it is valid with the empty sequence of global types.
- Else:
  - The first global definition must be valid with some type global type gt_1.
  - Let C' be the same context as C, but with the global type gt_1 apppended to the globals list.
  - Under context C', the remainder of the sequence must be valid with some sequence gt^* of global types.
  - Then the sequence is valid with the sequence of global types consisting of gt_1 prepended to gt^*.

$$\frac{}{C \vdash \epsilon : \epsilon} \qquad \frac{C \vdash global : gt_1 \qquad C \oplus \{\mathsf{globals}~gt_1\} \vdash global^* : gt^*}{C \vdash global_1~global^* : gt_1~gt^*}$$

3.5.7 Element Segments

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

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

- The reference type t must be valid.
- For each e_i in e^*,
  - The expression e_i must be valid with some result type $[t]$.
  - The expression e_i must be constant.
- The element mode $elemmode$ must be valid with some reference type t'.
- The reference type t must match the reference type t'.
- Then the element segment is valid with reference type t.

$$\frac{C \vdash elemtype : \mathsf{ok} \qquad (C \vdash expr : elemtype\ \mathsf{const})^* \qquad C \vdash elemmode : elemtype}{C \vdash \mathsf{elem}\ elemtype\ expr^*\ elemmode : elemtype}$$

passive

- The element mode is valid with any valid reference type.

$$\frac{}{C \vdash \mathsf{passive} : rt}$$

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

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

$$\frac{C.\mathsf{tables}[x] = lim\ rt' \qquad C \vdash rt \leq rt' \qquad C \vdash expr : \mathsf{i32}\ \mathsf{const}}{C \vdash \mathsf{active}\ x\ expr : rt}$$

declare

- The element mode is valid with any valid reference type.

$$\frac{}{C \vdash \mathsf{declare} : rt}$$

3.5.8 Data Segments

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

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

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

$$\frac{C \vdash \mathit{datamode} : \mathsf{ok}}{C \vdash \mathsf{data}\ b^*\ \mathit{datamode} : \mathsf{ok}}$$

$\mathsf{passive}$

- The data mode is valid.

$$\frac{}{C \vdash \mathsf{passive} : \mathsf{ok}}$$

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

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

$$\frac{C.\mathsf{mems}[x] = mt \qquad C \vdash \mathit{expr} : \mathsf{i32}\ \mathsf{const}}{C \vdash \mathsf{active}\ x\ \mathit{expr} : \mathsf{ok}}$$

3.5.9 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 expansion of $C.\mathsf{funcs}[x]$ must be a function type $\mathsf{func}\ [] \to []$.
- Then the start function is valid.

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}\ (\epsilon \to \epsilon)}{C \vdash \mathsf{start}\ x : \mathsf{ok}}$$

3.5.10 Exports

Exports export are classified by their external type.

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

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

$$\frac{C \vdash \mathit{externidx} : \mathit{xt}}{C \vdash \mathsf{export}\ \mathit{name}\ \mathit{externidx} : \mathit{name}\ \mathit{xt}}$$

$\mathsf{func}\ x$

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

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

$\mathsf{table}\ x$

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

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

$\mathsf{mem}\ x$

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

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

$\mathsf{global}\ x$

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

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

3.5.11 Imports

Imports import are classified by external types.

$\{\mathsf{module}\ name_1, \mathsf{name}\ name_2, \mathsf{desc}\ importdesc\}$

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

$$\frac{C \vdash xt : \mathsf{ok}}{C \vdash \mathsf{import}\ name_1\ name_2\ xt : xt}$$

3.5.12 Modules

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

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

The external types classifying a module may contain free type indices that refer to types defined within the module.

- Let $module$ be the module to validate.
- The types $module.\mathsf{types}$ must be valid yielding a context C_0.
- Let C be a context where:
  - $C.\mathsf{types}$ is $C_0.\mathsf{types}$,
  - $C.\mathsf{funcs}$ is $\mathrm{funcs}(it^*)$ concatenated with dt^*, with the import's external types it^* and the internal defined types dt^* as determined below,
  - $C.\mathsf{tables}$ is $\mathrm{tables}(it^*)$ concatenated with tt^*, with the import's external types it^* and the internal table types tt^* as determined below,
  - $C.\mathsf{mems}$ is $\mathrm{mems}(it^*)$ concatenated with mt^*, with the import's external types it^* and the internal memory types mt^* as determined below,
  - $C.\mathsf{globals}$ is $\mathrm{globals}(it^*)$ concatenated with gt^*, with the import's external types it^* and the internal global types gt^* as determined below,
  - $C.\mathsf{elems}$ is rt^* as determined below,
  - $C.\mathsf{datas}$ is ok^* as determined below,
  - $C.\mathsf{locals}$ is empty,
  - $C.\mathsf{labels}$ is empty,
  - $C.\mathsf{return}$ is empty.
  - $C.\mathsf{refs}$ is the set $\mathrm{funcidx}(module\ \mathsf{with}\ \mathsf{funcs} = \epsilon\ \mathsf{with}\ \mathsf{start} = \epsilon)$, i.e., the set of function indices occurring in the module, except in its functions or start function.
- Let C' be the context where:
  - $C'.\mathsf{globals}$ is the sequence $\mathrm{globals}(it^*)$,
  - $C'.\mathsf{types}$ is the same as $C.\mathsf{types}$,
  - $C'.\mathsf{funcs}$ is the same as $C.\mathsf{funcs}$,
  - $C'.\mathsf{tables}$ is the same as $C.\mathsf{tables}$,
  - $C'.\mathsf{mems}$ is the same as $C.\mathsf{mems}$,
  - $C'.\mathsf{refs}$ is the same as $C.\mathsf{refs}$,
  - all other fields are empty.
- Under the context C':

  - The sequence $module$.globals of globals must be valid with a sequence gt^* of global types.
  - For each $table_i$ in $module$.tables, the definition $table_i$ must be valid with a table type tt_i.
  - For each mem_i in $module$.mems, the definition mem_i must be valid with a memory type mt_i.
- Under the context C:
  - For each $func_i$ in $module$.funcs, the definition $func_i$ must be valid with a defined type dt_i.
  - For each $elem_i$ in $module$.elems, the segment $elem_i$ must be valid with reference type rt_i.
  - For each $data_i$ in $module$.datas, the segment $data_i$ must be valid with data type ok_i.
  - If $module$.start is non-empty, then $module$.start must be valid.
  - For each $import_i$ in $module$.imports, the segment $import_i$ must be valid with an external type it_i.
  - For each $export_i$ in $module$.exports, the segment $export_i$ must be valid with external type et_i.
- Let dt^* be the concatenation of the internal function types dt_i, in index order.
- Let tt^* be the concatenation of the internal table types tt_i, in index order.
- Let mt^* be the concatenation of the internal memory types mt_i, in index order.
- Let rt^* be the concatenation of the reference types rt_i, in index order.
- Let ok^* be the concatenation of the data types ok_i, in index order.
- Let it^* be the concatenation of external types it_i of the imports, in index order.
- Let et^* be the concatenation of external types et_i of the exports, in index order.
- The length of C.mems must not be larger than 1.
- All export names $export_i$.name must be different.
- Then the module is valid with external types $it^* \rightarrow et^*$.

$$\frac{\begin{array}{c}\{\} \vdash type^* : {dt'}^* \qquad (\{\mathsf{types}\ {dt'}^*\} \vdash import : xt_\mathsf{i})^* \\ C' \vdash global^* : gt^* \qquad (C' \vdash table : tt)^* \qquad (C' \vdash mem : mt)^* \qquad (C \vdash func : dt)^* \\ (C \vdash elem : rt)^* \qquad (C \vdash data : ok)^* \qquad (C \vdash start : \mathsf{ok})^? \qquad (C \vdash export : nm\ xt_\mathsf{e})^* \qquad nm^*\ \text{disjoint} \\ C = \{\mathsf{types}\ {dt'}^*,\ \mathsf{funcs}\ dt_\mathsf{i}^*\ dt^*,\ \mathsf{globals}\ gt_\mathsf{i}^*\ gt^*,\ \mathsf{tables}\ tt_\mathsf{i}^*\ tt^*,\ \mathsf{mems}\ mt_\mathsf{i}^*\ mt^*,\ \mathsf{elems}\ rt^*,\ \mathsf{datas}\ ok^*,\ \mathsf{refs}\ x^*\} \\ C' = \{\mathsf{types}\ {dt'}^*,\ \mathsf{funcs}\ dt_\mathsf{i}^*\ dt^*,\ \mathsf{globals}\ gt_\mathsf{i}^*,\ \mathsf{refs}\ x^*\} \qquad x^* = \mathrm{funcidx}(global^*\ table^*\ mem^*\ elem^*\ data^*) \\ dt_\mathsf{i}^* = \mathrm{funcs}(xt_\mathsf{i}^*) \qquad gt_\mathsf{i}^* = \mathrm{globals}(xt_\mathsf{i}^*) \qquad tt_\mathsf{i}^* = \mathrm{tables}(xt_\mathsf{i}^*) \qquad mt_\mathsf{i}^* = \mathrm{mems}(xt_\mathsf{i}^*)\end{array}}{\vdash \mathsf{module}\ type^*\ import^*\ func^*\ global^*\ table^*\ mem^*\ elem^*\ data^*\ start^?\ export^* : \mathsf{clos}_C(xt_\mathsf{i}^* \rightarrow xt_\mathsf{e}^*)}$$

Todo: Check refs; check export names

Note: All functions in a module 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 but evaluated sequentially, such that each constant expressions only has access to imported or previously defined globals.

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

CHAPTER 4

Execution

4.1 Conventions

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

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

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

1. In *prose*, describing the execution in intuitive form.
2. In *formal notation*, describing the rule in mathematical form.[18]

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.

[18] The semantics is derived from the following article: Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman. Bringing the Web up to Speed with WebAssembly[Page 79, 19]. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM 2017.

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

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

4.1.2 Formal Notation

Note: This section gives a brief explanation of the notation for specifying execution formally. For the interested reader, a more thorough introduction can be found in respective text books.[20]

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

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

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

To avoid unnecessary clutter, the store s and the frame f are often combined into a *state* z, which is a pair $(s; f)$. Moreover, z is 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:

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

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

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

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

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

The order of reduction is determined by the details of the reduction rules. Usually, the left-most instruction that is not a constant will be the subject of the next reduction *step*.

[20] For example: Benjamin Pierce. Types and Programming Languages[Page 80, 21]. The MIT Press 2002
[21] https://www.cis.upenn.edu/~bcpierce/tapl/

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

Note: For example, the following instruction sequence,

$$(\mathsf{f64.const}\ q_1)\ (\mathsf{f64.const}\ q_2)\ \mathsf{f64.neg}\ (\mathsf{f64.const}\ q_3)\ \mathsf{f64.add}\ \mathsf{f64.mul}$$

terminates after three steps:

$$\begin{array}{ll} \hookrightarrow & (\mathsf{f64.const}\ q_1)\ (\mathsf{f64.const}\ q_4)\ (\mathsf{f64.const}\ q_3)\ \mathsf{f64.add}\ \mathsf{f64.mul} \\ \hookrightarrow & (\mathsf{f64.const}\ q_1)\ (\mathsf{f64.const}\ q_5)\ \mathsf{f64.mul} \\ \hookrightarrow & (\mathsf{f64.const}\ q_6) \end{array}$$

where $q_4 = -q_2$ and $q_5 = -q_2 + q_3$ and $q_6 = q_1 \cdot (-q_2 + q_3)$.

4.2 Runtime Structure

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

4.2.1 Values

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

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

References other than null are represented with additional administrative instructions. They either are *scalar references*, containing a 31-bit integer, *structure references*, pointing to a specific structure address, *array references*, pointing to a specific array address, *function references*, pointing to a specific function address, or *host references* pointing to an uninterpreted form of host address defined by the embedder. Any of the aforementioned references can furthermore be wrapped up as an *external reference*.

$$\begin{array}{llll} \mathit{val} & ::= & \mathit{num} \mid \mathit{vec} \mid \mathit{ref} \\ \mathit{num} & ::= & \mathit{numtype}.\mathsf{const}\ \mathit{num}_{\mathit{numtype}} \\ \mathit{vec} & ::= & \mathit{vectype}.\mathsf{const}\ \mathit{vec}_{\mathit{vectype}} \\ \mathit{ref} & ::= & \mathit{addrref} \\ & | & \mathsf{ref.null}\ \mathit{heaptype} \\ \mathit{addrref} & ::= & \mathsf{ref.i31}\ \mathit{u31} \\ & | & \mathsf{ref.struct}\ \mathit{structaddr} \\ & | & \mathsf{ref.array}\ \mathit{arrayaddr} \\ & | & \mathsf{ref.func}\ \mathit{funcaddr} \\ & | & \mathsf{ref.host}\ \mathit{hostaddr} \\ & | & \mathsf{ref.extern}\ \mathit{addrref} \end{array}$$

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

Value types can have an associated *default value*; it is the respective value 0 for number types, 0 for vector types, and null for nullable reference types. For other references, no default value is defined, $\mathrm{default}_t$ hence is an optional value $val^?$.

$$\begin{array}{lcl}
\mathrm{default}_{\mathsf{i}N} &=& (\mathsf{i}N.\mathsf{const}~0) \\
\mathrm{default}_{\mathsf{f}N} &=& (\mathsf{f}N.\mathsf{const}~{+0}) \\
\mathrm{default}_{\mathsf{v}N} &=& (\mathsf{v}N.\mathsf{const}~0) \\
\mathrm{default}_{\mathsf{ref}~\mathsf{null}~ht} &=& (\mathsf{ref.null}~ht) \\
\mathrm{default}_{\mathsf{ref}~ht} &=& \epsilon
\end{array}$$

Convention

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

4.2.2 Results

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

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

4.2.3 Store

The *store* represents all global state that can be manipulated by WebAssembly programs. It consists of the runtime representation of all *instances* of functions, tables, memories, and globals, element segments, data segments, and structures or arrays that have been allocated during the life time of the abstract machine.[22]

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

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

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

Convention

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

4.2.4 Addresses

Function instances, table instances, memory instances, and global instances, element instances, data instances and structure or array instances in the store are referenced with abstract *addresses*. These are simply indices into the

[22] 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.

respective store component. In addition, an embedder may supply an uninterpreted set of *host addresses*.

$$\begin{array}{rcl}
\mathit{addr} & ::= & \mathbb{N} \\
\mathit{funcaddr} & ::= & \mathit{addr} \\
\mathit{tableaddr} & ::= & \mathit{addr} \\
\mathit{memaddr} & ::= & \mathit{addr} \\
\mathit{globaladdr} & ::= & \mathit{addr} \\
\mathit{elemaddr} & ::= & \mathit{addr} \\
\mathit{dataaddr} & ::= & \mathit{addr} \\
\mathit{structaddr} & ::= & \mathit{addr} \\
\mathit{arrayaddr} & ::= & \mathit{addr} \\
\mathit{hostaddr} & ::= & \mathit{addr}
\end{array}$$

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

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

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

Conventions

- The notation $\mathrm{addr}(A)$ denotes the set of addresses from address space addr occurring free in A. We sometimes reinterpret this set as the list of its elements.

4.2.5 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.

$$\begin{array}{rcl}
\mathit{moduleinst} & ::= & \{\mathsf{types}~\mathit{deftype}^*, \\
& & \phantom{\{}\mathsf{funcs}~\mathit{funcaddr}^*, \\
& & \phantom{\{}\mathsf{globals}~\mathit{globaladdr}^*, \\
& & \phantom{\{}\mathsf{tables}~\mathit{tableaddr}^*, \\
& & \phantom{\{}\mathsf{mems}~\mathit{memaddr}^*, \\
& & \phantom{\{}\mathsf{elems}~\mathit{elemaddr}^*, \\
& & \phantom{\{}\mathsf{datas}~\mathit{dataaddr}^*, \\
& & \phantom{\{}\mathsf{exports}~\mathit{exportinst}^*\}
\end{array}$$

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.

4.2.6 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.

$$\begin{array}{rcl} \mathit{funcinst} & ::= & \{\mathsf{type}~\mathit{deftype},~\mathsf{module}~\mathit{moduleinst},~\mathsf{code}~\mathit{code}\} \\ \mathit{code} & ::= & \mathit{func} \mid \mathit{hostfunc} \end{array}$$

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

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

4.2.7 Table Instances

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

$$\mathit{tableinst} ~~::=~~ \{\mathsf{type}~\mathit{tabletype},~\mathsf{elem}~\mathit{ref}^*\}$$

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

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

4.2.8 Memory Instances

A *memory instance* is the runtime representation of a linear memory. It records its type and holds a list of bytes.

$$\mathit{meminst} ~~::=~~ \{\mathsf{type}~\mathit{memtype},~\mathsf{bytes}~\mathit{byte}^*\}$$

The length of the list always is a multiple of the WebAssembly *page size*, which is defined to be the constant 65536 – abbreviated 64^* Ki.

The bytes can be mutated through memory instructions, the execution of an active data segment, or by external means provided by the embedder.

It is an invariant of the semantics that the length of the byte list, divided by page size, never exceeds the maximum size of $\mathit{memtype}$.

4.2.9 Global Instances

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

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

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

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

4.2.10 Element Instances

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

$$\mathit{eleminst} ::= \{\mathsf{type}\ \mathit{elemtype},\ \mathsf{elem}\ \mathit{ref}^*\}$$

4.2.11 Data Instances

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

$$\mathit{datainst} ::= \{\mathsf{bytes}\ \mathit{byte}^*\}$$

4.2.12 Export Instances

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

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

4.2.13 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.

$$\mathit{externval} ::= \mathsf{func}\ \mathit{funcaddr} \mid \mathsf{global}\ \mathit{globaladdr} \mid \mathsf{table}\ \mathit{tableaddr} \mid \mathsf{mem}\ \mathit{memaddr}$$

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:

$$\begin{array}{lcll}
\mathrm{funcs}(\epsilon) & = & \epsilon & \\
\mathrm{funcs}((\mathsf{func}\ fa)\ xv^*) & = & fa\ \mathrm{funcs}(xv^*) & \\
\mathrm{funcs}(\mathit{externval}\ xv^*) & = & \mathrm{funcs}(xv^*) & \text{otherwise} \\
\mathrm{tables}(\epsilon) & = & \epsilon & \\
\mathrm{tables}((\mathsf{table}\ ta)\ xv^*) & = & ta\ \mathrm{tables}(xv^*) & \\
\mathrm{tables}(\mathit{externval}\ xv^*) & = & \mathrm{tables}(xv^*) & \text{otherwise} \\
\mathrm{mems}(\epsilon) & = & \epsilon & \\
\mathrm{mems}((\mathsf{mem}\ ma)\ xv^*) & = & ma\ \mathrm{mems}(xv^*) & \\
\mathrm{mems}(\mathit{externval}\ xv^*) & = & \mathrm{mems}(xv^*) & \text{otherwise} \\
\mathrm{globals}(\epsilon) & = & \epsilon & \\
\mathrm{globals}((\mathsf{global}\ ga)\ xv^*) & = & ga\ \mathrm{globals}(xv^*) & \\
\mathrm{globals}(\mathit{externval}\ xv^*) & = & \mathrm{globals}(xv^*) & \text{otherwise}
\end{array}$$

4.2.14 Aggregate Instances

A *structure instance* is the runtime representation of a heap object allocated from a structure type. Likewise, an *array instance* is the runtime representation of a heap object allocated from an array type. Both record their respective defined type and hold a list of the values of their *fields*.

$$\begin{array}{rcl} \mathit{structinst} & ::= & \{\mathsf{type}\ \mathit{deftype},\ \mathsf{fields}\ \mathit{fieldval}^*\} \\ \mathit{arrayinst} & ::= & \{\mathsf{type}\ \mathit{deftype},\ \mathsf{fields}\ \mathit{fieldval}^*\} \\ \mathit{fieldval} & ::= & \mathit{val} \mid \mathit{packval} \\ \mathit{packval} & ::= & \mathit{packtype}.\mathsf{pack}\ \mathit{iN} \end{array}$$

Conventions

- Conversion of a regular value to a field value is defined as follows:

$$\begin{array}{lcl} \mathrm{pack}_{\mathit{valtype}}(\mathit{val}) & = & \mathit{val} \\ \mathrm{pack}_{\mathit{packtype}}(\mathsf{i32.const}\ i) & = & \mathit{packtype}.\mathsf{pack}\ \mathrm{wrap}_{32,|\mathit{packtype}|}(i) \end{array}$$

- The inverse conversion of a field value to a regular value is defined as follows:

$$\begin{array}{lcl} \mathrm{unpack}^{\epsilon}_{\mathit{valtype}}(\mathit{val}) & = & \mathit{val} \\ \mathrm{unpack}^{sx}_{\mathit{packtype}}(\mathit{packtype}.\mathsf{pack}\ i) & = & \mathsf{i32.const}\ \mathrm{extend}^{sx}_{|\mathit{packtype}|,32}(i) \end{array}$$

4.2.15 Stack

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

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

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

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

Values

Values are represented by themselves.

Labels

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

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

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

Note: For example, a loop label has the form

$$\mathsf{label}_n\{(\mathsf{loop}\ \mathit{bt}\ \ldots)\}$$

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.

Call Frames

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

$$\begin{array}{rcl} \mathit{callframe} & ::= & \mathsf{frame}_n\{\mathit{frame}\} \\ \mathit{frame} & ::= & \{\mathsf{locals}~(\mathit{val}^?)^*,~\mathsf{module}~\mathit{moduleinst}\} \end{array}$$

Locals may be uninitialized, in which case they are empty. 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 frame states where clear from context.
- The following auxiliary definition takes a block type and looks up the instruction type that it denotes in the current frame:

$$\begin{array}{rcll} \mathrm{instrtype}_z(x) & = & \mathit{ft} & \text{if } z.\mathsf{types}[x] \approx \mathsf{func}~\mathit{ft} \\ \mathrm{instrtype}_z(t^?) & = & \epsilon \to t^? & \end{array}$$

4.2.16 Administrative Instructions

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

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

$$\begin{array}{rcl} \mathit{instr} & ::= & \dots \\ & | & \mathit{addrref} \\ & | & \mathsf{label}_n\{\mathit{instr}^*\}~\mathit{instr}^* \\ & | & \mathsf{frame}_n\{\mathit{frame}\}~\mathit{instr}^* \\ & | & \mathsf{trap} \end{array}$$

An address reference represents an allocated reference value of respective form "on the stack".

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.

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.

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

$$(\mathsf{block}~\mathit{bt}~\mathit{instr}^*) \hookrightarrow (\mathsf{label}_n\{\epsilon\}~\mathit{instr}^*)$$

if the block type bt denotes a function type $t_1^m \to t_2^n$, such that n is the block's result arity. This rule replaces the block with a label instruction, which can be interpreted as "pushing" the label on the stack. When its end is reached, i.e., the inner instruction sequence has been reduced to the empty sequence – or rather, a sequence of n values representing the results – then the label instruction is eliminated courtesy of its own reduction rule:

$$(\mathsf{label}_n\{instr^*\}\ val^*) \hookrightarrow val^*$$

This can be interpreted as removing the label from the stack and only leaving the locally accumulated operand values. Validation guarantees that n matches the number $|val^*|$ of resulting values at this point.

Configurations

A *configuration* describes the current computation. It consists of the computations's *state* and the sequence of instructions left to execute. The state in turn consists of a global store and a current frame referring to the module instance in which the computation runs, i.e., where the current function originates from.

$$\begin{array}{rcl} config & ::= & state; instr^* \\ state & ::= & store; frame \end{array}$$

Note: The current version of WebAssembly is single-threaded, but configurations with multiple threads may be supported in the future.

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:

$$\begin{array}{rcl} \mathrm{fcopysign}_N(\pm p_1, \pm p_2) & = & \pm p_1 \\ \mathrm{fcopysign}_N(\pm p_1, \mp p_2) & = & \mp p_1 \end{array}$$

This definition is to be read as a shorthand for the following expansion of each clause into two separate ones:

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

Numeric operators are lifted to input sequences by applying the operator element-wise, returning a sequence of results. When there are multiple inputs, they must be of equal length.

$$op(c_1^n, \dots, c_k^n) \quad = \quad op(c_1^n[0], \dots, c_k^n[0]) \ \dots \ op(c_1^n[n-1], \dots, c_k^n[n-1])$$

Note: For example, the unary operator fabs, when given a sequence of floating-point values, return a sequence of floating-point results:

$$\mathrm{fabs}_N(z^n) \quad = \quad \mathrm{fabs}_N(z[0]) \ \ldots \ \mathrm{fabs}_N(z[n])$$

The binary operator iadd, when given two sequences of integers of the same length, n, return a sequence of integer results:

$$\mathrm{iadd}_N(i_1^n, i_2^n) \quad = \quad \mathrm{iadd}_N(i_1[0], i_2[0]) \ \ldots \ \mathrm{iadd}_N(i_1[n], i_2[n])$$

Conventions:

- The meta variable d is used to range over single bits.
- The meta variable p is used to range over (signless) magnitudes of floating-point values, including nan and ∞.
- The meta variable q is used to range over (signless) *rational* magnitudes, excluding nan or ∞.
- The notation f^{-1} denotes the inverse of a bijective function f.
- Truncation of rational values is written $\mathrm{trunc}(\pm q)$, with the usual mathematical definition:

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

- Saturation of integers is written $\mathrm{sat\_u}_N(i)$ and $\mathrm{sat\_s}_N(i)$. The arguments to these two functions range over arbitrary signed integers.
  - Unsigned saturation, $\mathrm{sat\_u}_N(i)$ clamps i to between 0 and $2^N - 1$:

$$\begin{array}{lcll} \mathrm{sat\_u}_N(i) & = & 2^N - 1 & (\text{if } i > 2^N - 1) \\ \mathrm{sat\_u}_N(i) & = & 0 & (\text{if } i < 0) \\ \mathrm{sat\_u}_N(i) & = & i & (\text{otherwise}) \end{array}$$

  - Signed saturation, $\mathrm{sat\_s}_N(i)$ clamps i to between -2^{N-1} and $2^{N-1} - 1$:

$$\begin{array}{lcll} \mathrm{sat\_s}_N(i) & = & \mathrm{signed}_N^{-1}(-2^{N-1}) & (\text{if } i < -2^{N-1}) \\ \mathrm{sat\_s}_N(i) & = & \mathrm{signed}_N^{-1}(2^{N-1} - 1) & (\text{if } i > 2^{N-1} - 1) \\ \mathrm{sat\_s}_N(i) & = & i & (\text{otherwise}) \end{array}$$

4.3.1 Representations

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

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

The first case of these applies to representations of both integer value types and packed types.

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

Integers

Integers are represented as base two unsigned numbers:

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

Boolean operators like $\wedge$, $\vee$, or $\veebar$ are lifted to bit sequences of equal length by applying them pointwise.

Floating-Point

Floating-point values are represented in the respective binary format defined by IEEE 754[23] (Section 3.4):

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

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

Vectors

Numeric vectors of type $\mathsf{v}N$ have the same underlying representation as an $\mathsf{i}N$. They can also be interpreted as a sequence of numeric values packed into a $\mathsf{v}N$ with a particular *shape* $t\mathsf{x}M$, provided that $N = |t| \cdot M$.

$$\begin{array}{l}
\mathrm{lanes}_{t\mathsf{x}M}(c) \quad = \quad c_0\ \ldots\ c_{M-1} \\
\qquad (\text{where } w = |t|/8 \\
\qquad\qquad \wedge\ b^* = \mathrm{bytes}_{\mathsf{i}N}(c) \\
\qquad\qquad \wedge\ c_i = \mathrm{bytes}_t^{-1}(b^*[i \cdot w : w]))
\end{array}$$

This function is a bijection on $\mathsf{i}N$, hence it is invertible.

Todo: pack/unpacknum

Storage

When a number is stored into memory, it is converted into a sequence of bytes in little endian[24] byte order:

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

Again these functions are invertible bijections.

[23] https://ieeexplore.ieee.org/document/8766229
[24] https://en.wikipedia.org/wiki/Endianness#Little-endian

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):

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

This function is bijective, and hence invertible.

Boolean Interpretation

The integer result of predicates – i.e., tests and relational operators – is defined with the help of the following auxiliary function producing the value 1 or 0 depending on a condition.

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

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

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

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

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

- Return the result of subtracting i_2 from i_1 modulo 2^N.

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

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

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

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

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

- If i_2 is 0, then the result is undefined.
- Else, return the result of dividing i_1 by i_2, truncated toward zero.

$$\begin{array}{lcl} \mathrm{idiv\_u}_N(i_1, 0) & = & \{\} \\ \mathrm{idiv\_u}_N(i_1, i_2) & = & \mathrm{trunc}(i_1 / i_2) \end{array}$$

Note: This operator is partial.

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- If j_2 is 0, then the result is undefined.
- Else if j_1 divided by j_2 is 2^{N-1}, then the result is undefined.
- Else, return the result of dividing j_1 by j_2, truncated toward zero.

$$\begin{array}{lcll}
\mathrm{idiv_s}_N(i_1, 0) & = & \{\} & \\
\mathrm{idiv_s}_N(i_1, i_2) & = & \{\} & (\text{if } \mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2) = 2^{N-1}) \\
\mathrm{idiv_s}_N(i_1, i_2) & = & \mathrm{signed}_N^{-1}(\mathrm{trunc}(\mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2))) &
\end{array}$$

Note: This operator is partial. Besides division by 0, the result of $(-2^{N-1})/(-1) = +2^{N-1}$ is not representable as an N-bit signed integer.

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

- If i_2 is 0, then the result is undefined.
- Else, return the remainder of dividing i_1 by i_2.

$$\begin{array}{lcl}
\mathrm{irem_u}_N(i_1, 0) & = & \{\} \\
\mathrm{irem_u}_N(i_1, i_2) & = & i_1 - i_2 \cdot \mathrm{trunc}(i_1/i_2)
\end{array}$$

Note: This operator is partial.

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

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- If i_2 is 0, then the result is undefined.
- Else, return the remainder of dividing j_1 by j_2, with the sign of the dividend j_1.

$$\begin{array}{lcl}
\mathrm{irem_s}_N(i_1, 0) & = & \{\} \\
\mathrm{irem_s}_N(i_1, i_2) & = & \mathrm{signed}_N^{-1}(j_1 - j_2 \cdot \mathrm{trunc}(j_1/j_2)) \\
 & & (\text{where } j_1 = \mathrm{signed}_N(i_1) \wedge j_2 = \mathrm{signed}_N(i_2))
\end{array}$$

Note: This operator is partial.

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

$\mathrm{inot}_N(i)$

- Return the bitwise negation of i.

$$\mathrm{inot}_N(i) \quad = \quad \mathrm{ibits}_N^{-1}(\mathrm{ibits}_N(i) \veebar \mathrm{ibits}_N(2^N - 1))$$

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

- Return the bitwise conjunction of i_1 and i_2.

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

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

- Return the bitwise conjunction of i_1 and the bitwise negation of i_2.

$$\mathrm{iandnot}_N(i_1, i_2) \quad = \quad \mathrm{iand}_N(i_1, \mathrm{inot}_N(i_2))$$

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

- Return the bitwise disjunction of i_1 and i_2.

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

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

- Return the bitwise exclusive disjunction of i_1 and i_2.

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

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

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

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

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

- Let k be i_2 modulo N.
- Return the result of shifting i_1 right by k bits, extended with 0 bits.

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

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

- Let k be i_2 modulo N.
- Return the result of shifting i_1 right by k bits, extended with the most significant bit of the original value.

$$\mathrm{ishr\_s}_N(i_1, i_2) \quad = \quad \mathrm{ibits}_N^{-1}(d_0^{k+1}\ d_1^{N-k-1}) \quad (\text{if } \mathrm{ibits}_N(i_1) = d_0\ d_1^{N-k-1}\ d_2^k \wedge k = i_2 \bmod N)$$

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

- Let k be i_2 modulo N.
- Return the result of rotating i_1 left by k bits.

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

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

- Let k be i_2 modulo N.
- Return the result of rotating i_1 right by k bits.

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

$\mathrm{iclz}_N(i)$

- Return the count of leading zero bits in i; all bits are considered leading zeros if i is 0.

$$\mathrm{iclz}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = 0^k\ (1\ d^*)^?)$$

$\mathrm{ictz}_N(i)$

- Return the count of trailing zero bits in i; all bits are considered trailing zeros if i is 0.

$$\mathrm{ictz}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = (d^*\ 1)^?\ 0^k)$$

$\mathrm{ipopcnt}_N(i)$

- Return the count of non-zero bits in i.

$$\mathrm{ipopcnt}_N(i) \quad = \quad k \quad (\text{if } \mathrm{ibits}_N(i) = (0^*\ 1)^k\ 0^*)$$

$\mathrm{ieqz}_N(i)$

- Return 1 if i is zero, 0 otherwise.

$$\mathrm{ieqz}_N(i) \quad = \quad \mathrm{bool}(i = 0)$$

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

- Return 1 if i_1 equals i_2, 0 otherwise.

$$\mathrm{ieq}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 = i_2)$$

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

- Return 1 if i_1 does not equal i_2, 0 otherwise.

$$\mathrm{ine}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \neq i_2)$$

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

- Return 1 if i_1 is less than i_2, 0 otherwise.

$$\mathrm{ilt\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 < i_2)$$

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is less than j_2, 0 otherwise.

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

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

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

$$\mathrm{igt\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 > i_2)$$

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is greater than j_2, 0 otherwise.

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

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

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

$$\mathrm{ile\_u}_N(i_1, i_2) \quad = \quad \mathrm{bool}(i_1 \leq i_2)$$

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is less than or equal to j_2, 0 otherwise.

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

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

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

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

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

- Let j_1 be the signed interpretation of i_1.
- Let j_2 be the signed interpretation of i_2.
- Return 1 if j_1 is greater than or equal to j_2, 0 otherwise.

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

$\mathrm{iextend}M\mathrm{\_s}_N(i)$

- Let j be the result of computing $\mathrm{wrap}_{N,M}(i)$.
- Return $\mathrm{extend}^{\mathsf{s}}{}_{M,N}(j)$.

$$\mathrm{iextend}M\mathrm{\_s}_N(i) = \mathrm{extend}^{\mathsf{s}}{}_{M,N}(\mathrm{wrap}_{N,M}(i))$$

$\mathrm{ibitselect}_N(i_1, i_2, i_3)$

- Let j_1 be the bitwise conjunction of i_1 and i_3.
- Let j_3' be the bitwise negation of i_3.
- Let j_2 be the bitwise conjunction of i_2 and j_3'.
- Return the bitwise disjunction of j_1 and j_2.

$$\mathrm{ibitselect}_N(i_1, i_2, i_3) = \mathrm{ior}_N(\mathrm{iand}_N(i_1, i_3), \mathrm{iand}_N(i_2, \mathrm{inot}_N(i_3)))$$

$\mathrm{iabs}_N(i)$

- Let j be the signed interpretation of i.
- If j is greater than or equal to 0, then return i.
- Else return the negation of j, modulo 2^N.

$$\begin{array}{lcll} \mathrm{iabs}_N(i) &=& i & (\text{if } \mathrm{signed}_N(i) \geq 0) \\ \mathrm{iabs}_N(i) &=& -\mathrm{signed}_N(i) \bmod 2^N & (\text{otherwise}) \end{array}$$

$\mathrm{ineg}_N(i)$

- Return the result of negating i, modulo 2^N.

$$\mathrm{ineg}_N(i) \quad = \quad (2^N - i) \bmod 2^N$$

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

- Return i_1 if $\mathrm{ilt\_u}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll} \mathrm{imin\_u}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{ilt\_u}_N(i_1, i_2) = 1) \\ \mathrm{imin\_u}_N(i_1, i_2) & = & i_2 & (\text{otherwise}) \end{array}$$

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

- Return i_1 if $\mathrm{ilt\_s}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll} \mathrm{imin\_s}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{ilt\_s}_N(i_1, i_2) = 1) \\ \mathrm{imin\_s}_N(i_1, i_2) & = & i_2 & (\text{otherwise}) \end{array}$$

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

- Return i_1 if $\mathrm{igt\_u}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll} \mathrm{imax\_u}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{igt\_u}_N(i_1, i_2) = 1) \\ \mathrm{imax\_u}_N(i_1, i_2) & = & i_2 & (\text{otherwise}) \end{array}$$

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

- Return i_1 if $\mathrm{igt\_s}_N(i_1, i_2)$ is 1, return i_2 otherwise.

$$\begin{array}{lcll} \mathrm{imax\_s}_N(i_1, i_2) & = & i_1 & (\text{if } \mathrm{igt\_s}_N(i_1, i_2) = 1) \\ \mathrm{imax\_s}_N(i_1, i_2) & = & i_2 & (\text{otherwise}) \end{array}$$

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

- Let i be the result of adding i_1 and i_2.
- Return $\mathrm{sat\_u}_N(i)$.

$$\mathrm{iadd\_sat\_u}_N(i_1, i_2) \quad = \quad \mathrm{sat\_u}_N(i_1 + i_2)$$

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

- Let j_1 be the signed interpretation of i_1
- Let j_2 be the signed interpretation of i_2
- Let j be the result of adding j_1 and j_2.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{iadd\_sat\_s}_N(i_1, i_2) \quad = \quad \mathrm{sat\_s}_N(\mathrm{signed}_N(i_1) + \mathrm{signed}_N(i_2))$$

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

- Let i be the result of subtracting i_2 from i_1.
- Return $\mathrm{sat\_u}_N(i)$.

$$\mathrm{isub\_sat\_u}_N(i_1, i_2) \quad = \quad \mathrm{sat\_u}_N(i_1 - i_2)$$

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

- Let j_1 be the signed interpretation of i_1
- Let j_2 be the signed interpretation of i_2
- Let j be the result of subtracting j_2 from j_1.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{isub\_sat\_s}_N(i_1, i_2) \quad = \quad \mathrm{sat\_s}_N(\mathrm{signed}_N(i_1) - \mathrm{signed}_N(i_2))$$

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

- Let j be the result of adding i_1, i_2, and 1.
- Return the result of dividing j by 2, truncated toward zero.

$$\mathrm{iavgr\_u}_N(i_1, i_2) \quad = \quad \mathrm{trunc}((i_1 + i_2 + 1)/2)$$

$\mathrm{iq15mulrsat\_s}_N(i_1, i_2)$

- Return the result of $\mathrm{sat\_s}_N(\mathrm{ishr\_s}_N(i_1 \cdot i_2 + 2^{14}, 15))$.

$$\mathrm{iq15mulrsat\_s}_N(i_1, i_2) \quad = \quad \mathrm{sat\_s}_N(\mathrm{ishr\_s}_N(i_1 \cdot i_2 + 2^{14}, 15))$$

4.3.3 Floating-Point Operations

Floating-point arithmetic follows the IEEE 754[25] 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.

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

Rounding

Rounding always is round-to-nearest ties-to-even, in correspondence with IEEE 754[26] (Section 4.3.1).

An *exact* floating-point number is a rational number that is exactly representable as a floating-point number of given bit width N.

A *limit* number for a given floating-point bit width N is a positive or negative number whose magnitude is the smallest power of 2 that is not exactly representable as a floating-point number of width N (that magnitude is 2^{128} for $N = 32$ and 2^{1024} for $N = 64$).

A *candidate* number is either an exact floating-point number or a positive or negative limit number for the given bit width N.

A *candidate pair* is a pair z_1, z_2 of candidate numbers, such that no candidate number exists that lies between the two.

A real number r is converted to a floating-point value of bit width N as follows:

- If r is 0, then return $+0$.
- Else if r is an exact floating-point number, then return r.
- Else if r greater than or equal to the positive limit, then return $+\infty$.
- Else if r is less than or equal to the negative limit, then return $-\infty$.
- Else if z_1 and z_2 are a candidate pair such that $z_1 < r < z_2$, then:
  - If $|r - z_1| < |r - z_2|$, then let z be z_1.
  - Else if $|r - z_1| > |r - z_2|$, then let z be z_2.
  - Else if $|r - z_1| = |r - z_2|$ and the significand of z_1 is even, then let z be z_1.
  - Else, let z be z_2.
- If z is 0, then:
  - If $r < 0$, then return -0.
  - Else, return $+0$.
- Else if z is a limit number, then:
  - If $r < 0$, then return $-\infty$.
  - Else, return $+\infty$.
- Else, return z.

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

$$\begin{array}{lcll}
\mathrm{rectify}_N(r, \pm\mathrm{limit}_N) & = & \pm\infty & \\
\mathrm{rectify}_N(r, 0) & = & +0 & (r \geq 0) \\
\mathrm{rectify}_N(r, 0) & = & -0 & (r < 0) \\
\mathrm{rectify}_N(r, z) & = & z &
\end{array}$$

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

where:

$$
\begin{array}{lcl}
\mathrm{exact}_N & = & f_N \cap \mathbb{Q} \\
\mathrm{limit}_N & = & 2^{2^{\mathrm{expon}(N)-1}} \\
\mathrm{candidate}_N & = & \mathrm{exact}_N \cup \{+\mathrm{limit}_N, -\mathrm{limit}_N\} \\
\mathrm{candidatepair}_N & = & \{(z_1, z_2) \in \mathrm{candidate}_N^2 \mid z_1 < z_2 \wedge \forall z \in \mathrm{candidate}_N, z \leq z_1 \vee z \geq z_2\} \\
\mathrm{even}_N((d + m \cdot 2^{-M}) \cdot 2^e) & \Leftrightarrow & m \bmod 2 = 0 \\
\mathrm{even}_N(\pm \mathrm{limit}_N) & \Leftrightarrow & \mathrm{true}
\end{array}
$$

NaN Propagation

When the result of a floating-point operator other than fneg, fabs, or fcopysign is a NaN, then its sign is non-deterministic and the payload is computed as follows:

- If the payload of all NaN inputs to the operator is canonical (including the case that there are no NaN inputs), then the payload of the output is canonical as well.
- Otherwise the payload is picked non-deterministically among all arithmetic NaNs; that is, its most significant bit is 1 and all others are unspecified.

This non-deterministic result is expressed by the following auxiliary function producing a set of allowed outputs from a set of inputs:

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

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

- 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 $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return that infinity.
- Else if either z_1 or z_2 is an infinity, then return that infinity.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of equal sign, then return that zero.
- Else if either z_1 or z_2 is a zero, then return the other operand.
- Else if both z_1 and z_2 are values with the same magnitude but opposite signs, then return positive zero.
- Else return the result of adding z_1 and z_2, rounded to the nearest representable value.

$$
\begin{array}{lcl}
\mathrm{fadd}_N(\pm \mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm \mathsf{nan}(n), z_2\} \\
\mathrm{fadd}_N(z_1, \pm \mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm \mathsf{nan}(n), z_1\} \\
\mathrm{fadd}_N(\pm \infty, \mp \infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fadd}_N(\pm \infty, \pm \infty) & = & \pm \infty \\
\mathrm{fadd}_N(z_1, \pm \infty) & = & \pm \infty \\
\mathrm{fadd}_N(\pm \infty, z_2) & = & \pm \infty \\
\mathrm{fadd}_N(\pm 0, \mp 0) & = & +0 \\
\mathrm{fadd}_N(\pm 0, \pm 0) & = & \pm 0 \\
\mathrm{fadd}_N(z_1, \pm 0) & = & z_1 \\
\mathrm{fadd}_N(\pm 0, z_2) & = & z_2 \\
\mathrm{fadd}_N(\pm q, \mp q) & = & +0 \\
\mathrm{fadd}_N(z_1, z_2) & = & \mathrm{float}_N(z_1 + z_2)
\end{array}
$$

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

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities of equal signs, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of opposite sign, then return z_1.
- Else if z_1 is an infinity, then return that infinity.
- Else if z_2 is an infinity, then return that infinity negated.
- Else if both z_1 and z_2 are zeroes of equal sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return z_1.
- Else if z_2 is a zero, then return z_1.
- Else if z_1 is a zero, then return z_2 negated.
- Else if both z_1 and z_2 are the same value, then return positive zero.
- Else return the result of subtracting z_2 from z_1, rounded to the nearest representable value.

$$
\begin{array}{lcl}
\mathrm{fsub}_N(\pm\mathsf{nan}(n), z_2) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fsub}_N(z_1, \pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fsub}_N(\pm\infty, \pm\infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fsub}_N(\pm\infty, \mp\infty) & = & \pm\infty \\
\mathrm{fsub}_N(z_1, \pm\infty) & = & \mp\infty \\
\mathrm{fsub}_N(\pm\infty, z_2) & = & \pm\infty \\
\mathrm{fsub}_N(\pm 0, \pm 0) & = & +0 \\
\mathrm{fsub}_N(\pm 0, \mp 0) & = & \pm 0 \\
\mathrm{fsub}_N(z_1, \pm 0) & = & z_1 \\
\mathrm{fsub}_N(\pm 0, \pm q_2) & = & \mp q_2 \\
\mathrm{fsub}_N(\pm q, \pm q) & = & +0 \\
\mathrm{fsub}_N(z_1, z_2) & = & \mathrm{float}_N(z_1 - z_2)
\end{array}
$$

Note: Up to the non-determinism regarding NaNs, it always holds that $\mathrm{fsub}_N(z_1, z_2) = \mathrm{fadd}_N(z_1, \mathrm{fneg}_N(z_2))$.

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

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if one of z_1 and z_2 is a zero and the other an infinity, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return positive infinity.
- Else if both z_1 and z_2 are infinities of opposite sign, then return negative infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with equal sign, then return positive infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with opposite sign, then return negative infinity.
- Else if both z_1 and z_2 are zeroes of equal sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return negative zero.
- Else return the result of multiplying z_1 and z_2, rounded to the nearest representable value.

$$
\begin{array}{lcl}
\mathrm{fmul}_N(\pm\mathsf{nan}(n), z_2) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fmul}_N(z_1, \pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fmul}_N(\pm\infty, \pm 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm\infty, \mp 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm 0, \pm\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm 0, \mp\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fmul}_N(\pm\infty, \pm\infty) &=& +\infty \\
\mathrm{fmul}_N(\pm\infty, \mp\infty) &=& -\infty \\
\mathrm{fmul}_N(\pm q_1, \pm\infty) &=& +\infty \\
\mathrm{fmul}_N(\pm q_1, \mp\infty) &=& -\infty \\
\mathrm{fmul}_N(\pm\infty, \pm q_2) &=& +\infty \\
\mathrm{fmul}_N(\pm\infty, \mp q_2) &=& -\infty \\
\mathrm{fmul}_N(\pm 0, \pm 0) &=& +0 \\
\mathrm{fmul}_N(\pm 0, \mp 0) &=& -0 \\
\mathrm{fmul}_N(z_1, z_2) &=& \mathrm{float}_N(z_1 \cdot z_2)
\end{array}
$$

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

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if both z_1 and z_2 are infinities, then return an element of $\mathrm{nans}_N\{\}$.
- Else if both z_1 and z_2 are zeroes, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if z_1 is an infinity and z_2 a value with equal sign, then return positive infinity.
- Else if z_1 is an infinity and z_2 a value with opposite sign, then return negative infinity.
- Else if z_2 is an infinity and z_1 a value with equal sign, then return positive zero.
- Else if z_2 is an infinity and z_1 a value with opposite sign, then return negative zero.
- Else if z_1 is a zero and z_2 a value with equal sign, then return positive zero.
- Else if z_1 is a zero and z_2 a value with opposite sign, then return negative zero.
- Else if z_2 is a zero and z_1 a value with equal sign, then return positive infinity.
- Else if z_2 is a zero and z_1 a value with opposite sign, then return negative infinity.
- Else return the result of dividing z_1 by z_2, rounded to the nearest representable value.

$$
\begin{array}{lcl}
\mathrm{fdiv}_N(\pm\mathsf{nan}(n), z_2) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\mathrm{fdiv}_N(z_1, \pm\mathsf{nan}(n)) &=& \mathrm{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\mathrm{fdiv}_N(\pm\infty, \pm\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm\infty, \mp\infty) &=& \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm 0, \pm 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm 0, \mp 0) &=& \mathrm{nans}_N\{\} \\
\mathrm{fdiv}_N(\pm\infty, \pm q_2) &=& +\infty \\
\mathrm{fdiv}_N(\pm\infty, \mp q_2) &=& -\infty \\
\mathrm{fdiv}_N(\pm q_1, \pm\infty) &=& +0 \\
\mathrm{fdiv}_N(\pm q_1, \mp\infty) &=& -0 \\
\mathrm{fdiv}_N(\pm 0, \pm q_2) &=& +0 \\
\mathrm{fdiv}_N(\pm 0, \mp q_2) &=& -0 \\
\mathrm{fdiv}_N(\pm q_1, \pm 0) &=& +\infty \\
\mathrm{fdiv}_N(\pm q_1, \mp 0) &=& -\infty \\
\mathrm{fdiv}_N(z_1, z_2) &=& \mathrm{float}_N(z_1 / z_2)
\end{array}
$$

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

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if either z_1 or z_2 is a negative infinity, then return negative infinity.
- Else if either z_1 or z_2 is a positive infinity, then return the other value.
- Else if both z_1 and z_2 are zeroes of opposite signs, then return negative zero.
- Else return the smaller value of z_1 and z_2.

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

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

- If either z_1 or z_2 is a NaN, then return an element of $\mathrm{nans}_N\{z_1, z_2\}$.
- Else if either z_1 or z_2 is a positive infinity, then return positive infinity.
- Else if either z_1 or z_2 is a negative infinity, then return the other value.
- Else if both z_1 and z_2 are zeroes of opposite signs, then return positive zero.
- Else return the larger value of z_1 and z_2.

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

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

- 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.

$$\begin{array}{lll}
\mathrm{fabs}_N(\pm\mathsf{nan}(n)) & = & +\mathsf{nan}(n) \\
\mathrm{fabs}_N(\pm\infty) & = & +\infty \\
\mathrm{fabs}_N(\pm 0) & = & +0 \\
\mathrm{fabs}_N(\pm q) & = & +q
\end{array}$$

$\mathrm{fneg}_N(z)$

- 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.

$$\begin{array}{lll}
\mathrm{fneg}_N(\pm\mathsf{nan}(n)) & = & \mp\mathsf{nan}(n) \\
\mathrm{fneg}_N(\pm\infty) & = & \mp\infty \\
\mathrm{fneg}_N(\pm 0) & = & \mp 0 \\
\mathrm{fneg}_N(\pm q) & = & \mp q
\end{array}$$

$\mathrm{fsqrt}_N(z)$

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

$$\begin{array}{lll}
\mathrm{fsqrt}_N(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\pm\mathsf{nan}(n)\} \\
\mathrm{fsqrt}_N(-\infty) & = & \mathrm{nans}_N\{\} \\
\mathrm{fsqrt}_N(+\infty) & = & +\infty \\
\mathrm{fsqrt}_N(\pm 0) & = & \pm 0 \\
\mathrm{fsqrt}_N(-q) & = & \mathrm{nans}_N\{\} \\
\mathrm{fsqrt}_N(+q) & = & \mathrm{float}_N\left(\sqrt{q}\right)
\end{array}$$

$\mathrm{fceil}_N(z)$

- 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 1.
- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if both z_1 and z_2 are the same value, then return 0.
- Else return 1.

$$\begin{array}{lcl}
\mathrm{fne}_N(\pm\mathsf{nan}(n), z_2) & = & 1 \\
\mathrm{fne}_N(z_1, \pm\mathsf{nan}(n)) & = & 1 \\
\mathrm{fne}_N(\pm 0, \mp 0) & = & 0 \\
\mathrm{fne}_N(z_1, z_2) & = & \mathrm{bool}(z_1 \neq z_2)
\end{array}$$

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

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 0.
- Else if z_1 is positive infinity, then return 0.
- Else if z_1 is negative infinity, then return 1.
- Else if z_2 is positive infinity, then return 1.
- Else if z_2 is negative infinity, then return 0.

- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if z_1 is smaller than z_2, then return 1.
- Else return 0.

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

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

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 0.
- Else if z_1 is positive infinity, then return 1.
- Else if z_1 is negative infinity, then return 0.
- Else if z_2 is positive infinity, then return 0.
- Else if z_2 is negative infinity, then return 1.
- Else if both z_1 and z_2 are zeroes, then return 0.
- Else if z_1 is larger than z_2, then return 1.
- Else return 0.

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

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

- 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.

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

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

- If either z_1 or z_2 is a NaN, then return 0.
- Else if z_1 and z_2 are the same value, then return 1.
- Else if z_1 is positive infinity, then return 1.
- Else if z_1 is negative infinity, then return 0.
- Else if z_2 is positive infinity, then return 0.
- Else if z_2 is negative infinity, then return 1.
- Else if both z_1 and z_2 are zeroes, then return 1.
- Else if z_1 is smaller than or equal to z_2, then return 1.
- Else return 0.

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

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

- If z_2 is less than z_1 then return z_2.
- Else return z_1.

$$
\begin{array}{lcll}
\mathrm{fpmin}_N(z_1, z_2) & = & z_2 & (\text{if } \mathrm{flt}_N(z_2, z_1) = 1) \\
\mathrm{fpmin}_N(z_1, z_2) & = & z_1 & (\text{otherwise})
\end{array}
$$

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

- If z_1 is less than z_2 then return z_2.
- Else return z_1.

$$
\begin{array}{lcll}
\mathrm{fpmax}_N(z_1, z_2) & = & z_2 & (\text{if } \mathrm{flt}_N(z_1, z_2) = 1) \\
\mathrm{fpmax}_N(z_1, z_2) & = & z_1 & (\text{otherwise})
\end{array}
$$

4.3.4 Conversions

Todo: ext or extend?

$\mathrm{extend}^{\mathsf{u}}{}_{M,N}(i)$

- Return i.

$$\mathrm{extend}^{\mathsf{u}}{}_{M,N}(i) \quad = \quad i$$

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

$\mathrm{extend}^{\mathsf{s}}{}_{M,N}(i)$

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

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

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

- Return i modulo 2^N.

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

$\mathrm{trunc}^{\mathsf{u}}{}_{M,N}(z)$

- If z is a NaN, then the result is undefined.
- Else if z is an infinity, then the result is undefined.
- Else if z is a number and $\mathrm{trunc}(z)$ is a value within range of the target type, then return that value.
- Else the result is undefined.

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

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

$\mathrm{trunc}^{\mathsf{s}}{}_{M,N}(z)$

- If z is a NaN, then the result is undefined.
- Else if z is an infinity, then the result is undefined.
- If z is a number and $\mathrm{trunc}(z)$ is a value within range of the target type, then return that value.
- Else the result is undefined.

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

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

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

- If z is a NaN, then return 0.
- Else if z is negative infinity, then return 0.
- Else if z is positive infinity, then return $2^N - 1$.
- Else, return $\mathrm{sat\_u}_N(\mathrm{trunc}(z))$.

$$\begin{array}{lcl}
\mathrm{trunc_sat_u}_{M,N}(\pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{trunc_sat_u}_{M,N}(-\infty) & = & 0 \\
\mathrm{trunc_sat_u}_{M,N}(+\infty) & = & 2^N - 1 \\
\mathrm{trunc_sat_u}_{M,N}(z) & = & \mathrm{sat_u}_N(\mathrm{trunc}(z))
\end{array}$$

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

- If z is a NaN, then return 0.
- Else if z is negative infinity, then return -2^{N-1}.
- Else if z is positive infinity, then return $2^{N-1} - 1$.
- Else, return $\mathrm{sat\_s}_N(\mathrm{trunc}(z))$.

$$\begin{array}{lcl}
\mathrm{trunc_sat_s}_{M,N}(\pm\mathsf{nan}(n)) & = & 0 \\
\mathrm{trunc_sat_s}_{M,N}(-\infty) & = & -2^{N-1} \\
\mathrm{trunc_sat_s}_{M,N}(+\infty) & = & 2^{N-1} - 1 \\
\mathrm{trunc_sat_s}_{M,N}(z) & = & \mathrm{sat_s}_N(\mathrm{trunc}(z))
\end{array}$$

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

- If z is a canonical NaN, then return an element of $\mathrm{nans}_N\{\}$ (i.e., a canonical NaN of size N).
- Else if z is a NaN, then return an element of $\mathrm{nans}_N\{\pm\mathsf{nan}(1)\}$ (i.e., any arithmetic NaN of size N).
- Else, return z.

$$\begin{array}{lcll}
\mathrm{promote}_{M,N}(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{\} & (\mathrm{if}\ n = \mathrm{canon}_N) \\
\mathrm{promote}_{M,N}(\pm\mathsf{nan}(n)) & = & \mathrm{nans}_N\{+\mathsf{nan}(1)\} & (\mathrm{otherwise}) \\
\mathrm{promote}_{M,N}(z) & = & z &
\end{array}$$

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

- If z is a canonical NaN, then return an element of $\mathrm{nans}_N\{\}$ (i.e., a canonical NaN of size N).
- Else if z is a NaN, then return an element of $\mathrm{nans}_N\{\pm\mathsf{nan}(1)\}$ (i.e., any NaN of size N).
- Else if z is an infinity, then return that infinity.
- Else if z is a zero, then return that zero.
- Else, return $\mathrm{float}_N(z)$.

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

$\mathrm{convert}^{\mathsf{u}}{}_{M,N}(i)$

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

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

$\mathrm{convert}^{\mathsf{s}}{}_{M,N}(i)$

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

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

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

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

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

$\mathrm{narrow}^{\mathsf{s}}{}_{M,N}(i)$

- Let j be the signed interpretation of i of size M.
- Return $\mathrm{sat\_s}_N(j)$.

$$\mathrm{narrow}^{\mathsf{s}}{}_{M,N}(i) \quad = \quad \mathrm{sat\_s}_N(\mathrm{signed}_M(i))$$

$\mathrm{narrow}^{\mathsf{u}}{}_{M,N}(i)$

- Let j be the signed interpretation of i of size M.
- Return $\mathrm{sat\_u}_N(j)$.

$$\mathrm{narrow}^{\mathsf{u}}{}_{M,N}(i) \quad = \quad \mathrm{sat\_u}_N(\mathrm{signed}_M(i))$$

4.4 Types

Execution has to check and compare types in a few places, such as executing call_indirect or instantiating modules.

It is an invariant of the semantics that all types occurring during execution are closed.

Note: Runtime type checks generally involve types from multiple modules or types not defined by a module at all, such that module-local type indices are not meaningful.

4.4.1 Instantiation

Any form of type can be *instantiated* into a closed type inside a module instance by substituting each type index x occurring in it with the corresponding defined type $moduleinst.\mathsf{types}[x]$.

$$\mathrm{clos}_{moduleinst}(t) \quad = \quad t[:= dt^*] \qquad \text{if } dt^* = moduleinst.\mathsf{types}$$

Note: This is the runtime equivalent to type closure.

4.5 Values

4.5.1 Value Typing

For the purpose of checking argument values against the parameter types of exported functions, values are classified by value types. The following auxiliary typing rules specify this typing relation relative to a store S in which possibly referenced addresses live.

Numeric Values $t.\mathsf{const}\ c$

- The value is valid with number type t.

$$\frac{}{S \vdash t.\mathsf{const}\ c : t}$$

Vector Values $t.\mathsf{const}\ c$

- The value is valid with vector type t.

$$\frac{}{S \vdash t.\mathsf{const}\ c : t}$$

Null References $\mathsf{ref.null}\ t$

- The heap type must be valid under the empty context.
- Then the value is valid with reference type $(\mathsf{ref\ null}\ t')$, where the heap type t' is the least type that matches t.

$$\frac{\vdash t : \mathsf{ok} \qquad t' \in \{\mathsf{none}, \mathsf{nofunc}, \mathsf{noextern}\} \qquad \vdash t' \leq t}{S \vdash \mathsf{ref.null}\ t : (\mathsf{ref\ null}\ t')}$$

Note: A null reference is typed with the least type in its respective hierarchy. That ensures that it is compatible with any nullable type in that hierarchy.

Scalar References $\mathsf{ref.i31}\ i$

- The value is valid with reference type $(\mathsf{ref}\ \mathsf{i31})$.

$$\frac{}{S \vdash \mathsf{ref.i31}\ i : \mathsf{ref}\ \mathsf{i31}}$$

Structure References $\mathsf{ref.struct}\ a$

- The structure address a must exist in the store.
- Let $structinst$ be the structure instance $S.\mathsf{structs}[a]$.
- Let $deftype$ be the defined type $structinst.\mathsf{type}$.
- The expansion of $deftype$ must be a struct type.
- Then the value is valid with reference type $(\mathsf{ref}\ deftype)$.

$$\frac{deftype = S.\mathsf{structs}[a].\mathsf{type} \qquad \mathrm{expand}(deftype) = \mathsf{struct}\ structtype}{S \vdash \mathsf{ref.struct}\ a : \mathsf{ref}\ deftype}$$

Array References $\mathsf{ref.array}\ a$

- The array address a must exist in the store.
- Let $arrayinst$ be the array instance $S.\mathsf{arrays}[a]$.
- Let $deftype$ be the defined type $arrayinst.\mathsf{type}$.
- The expansion of $deftype$ must be an array type.
- Then the value is valid with reference type $(\mathsf{ref}\ arraytype)$.

$$\frac{deftype = S.\mathsf{arrays}[a].\mathsf{type} \qquad \mathrm{expand}(deftype) = \mathsf{array}\ arraytype}{S \vdash \mathsf{ref.array}\ a : \mathsf{ref}\ deftype}$$

Function References $\mathsf{ref.func}\ a$

- The function address a must exist in the store.
- Let $funcinst$ be the function instance $S.\mathsf{funcs}[a]$.
- Let $deftype$ be the defined type $funcinst.\mathsf{type}$.
- The expansion of $deftype$ must be a function type.
- Then the value is valid with reference type $(\mathsf{ref}\ functype)$.

$$\frac{deftype = S.\mathsf{funcs}[a].\mathsf{type} \qquad \mathrm{expand}(deftype) = \mathsf{func}\ functype}{S \vdash \mathsf{ref.func}\ a : \mathsf{ref}\ deftype}$$

Host References $\mathsf{ref.host}\ a$

- The value is valid with reference type $(\mathsf{ref}\ \mathsf{any})$.

$$\frac{}{S \vdash \mathsf{ref.host}\ a : \mathsf{ref}\ \mathsf{any}}$$

Note: A host reference is considered internalized by this rule.

External References $\mathsf{ref.extern}\ \mathit{ref}$

- The reference value ref must be valid with some reference type $(\mathsf{ref}\ \mathsf{null}^?\ t)$.
- The heap type t must match the heap type any.
- Then the value is valid with reference type $(\mathsf{ref}\ \mathsf{null}^?\ \mathsf{extern})$.

$$\frac{S \vdash \mathit{ref} : \mathsf{ref}\ \mathsf{null}^?\ t \qquad \vdash t \leq \mathsf{any}}{S \vdash \mathsf{ref.extern}\ \mathit{ref} : \mathsf{ref}\ \mathsf{null}^?\ \mathsf{extern}}$$

Subsumption

- The value must be valid with some value type t.
- The value type t matches another valid type t'.
- Then the value is valid with type t'.

$$\frac{S \vdash \mathit{val} : t \qquad \vdash t' : \mathsf{ok} \qquad \vdash t \leq t'}{S \vdash \mathit{val} : t'}$$

4.5.2 External Typing

For the purpose of checking external values against imports, such values are classified by external types. The following auxiliary typing rules specify this typing relation relative to a store S in which the referenced instances live.

$\mathsf{func}\ a$

- The store entry $S.\mathsf{funcs}[a]$ must exist.
- Then $\mathsf{func}\ a$ is valid with external type $\mathsf{func}\ S.\mathsf{funcs}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{func}\ a : \mathsf{func}\ S.\mathsf{funcs}[a].\mathsf{type}}$$

$\mathsf{table}\ a$

- The store entry $S.\mathsf{tables}[a]$ must exist.
- Then $\mathsf{table}\ a$ is valid with external type $\mathsf{table}\ S.\mathsf{tables}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{table}\ a : \mathsf{table}\ S.\mathsf{tables}[a].\mathsf{type}}$$

$\mathsf{mem}\ a$

- The store entry $S.\mathsf{mems}[a]$ must exist.
- Then $\mathsf{mem}\ a$ is valid with external type $\mathsf{mem}\ S.\mathsf{mems}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{mem}\ a : \mathsf{mem}\ S.\mathsf{mems}[a].\mathsf{type}}$$

global a

- The store entry $S.\mathsf{globals}[a]$ must exist.
- Then global a is valid with external type global $S.\mathsf{globals}[a].\mathsf{type}$.

$$\frac{}{S \vdash \mathsf{global}\ a : \mathsf{global}\ S.\mathsf{globals}[a].\mathsf{type}}$$

Subsumption

- The external value must be valid with some external type et.
- The external type et matches another valid type et'.
- Then the external value is valid with type et'.

$$\frac{S \vdash \mathit{externval} : et \qquad \vdash et' : \mathsf{ok} \qquad \vdash et \leq et'}{S \vdash \mathit{externval} : et'}$$

4.6 Instructions

WebAssembly computation is performed by executing individual instructions.

4.6.1 Parametric Instructions

nop

1. Do nothing.

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

unreachable

1. Trap.

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

drop

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. Do nothing.

Todo:

(1) Remove trailing “Do nothing.”

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

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

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ c)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value val_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop the value val_1 from the stack.
7. If c is not 0, then:
   a. Push the value val_1 to the stack.
8. Else:
   a. Push the value val_2 to the stack.

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

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

4.6.2 Numeric Instructions

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

$$\begin{array}{lcl} op_{\mathsf{i}N}(i_1, \dots, i_k) & = & \mathrm{i}op_N(i_1, \dots, i_k) \\ op_{\mathsf{f}N}(z_1, \dots, z_k) & = & \mathrm{f}op_N(z_1, \dots, z_k) \end{array}$$

And for conversion operators:

$$cvtop^{sx^?}_{t_1,t_2}(c) \quad = \quad cvtop^{sx^?}_{|t_1|,|t_2|}(c)$$

Where the underlying operators are partial, the corresponding instruction will trap when the result is not defined. Where the underlying operators are non-deterministic, because they may return one of multiple possible NaN values, so are the corresponding instructions.

Note: For example, the result of instruction $\mathsf{i32.add}$ applied to operands i_1, i_2 invokes $\mathsf{add}_{\mathsf{i32}}(i_1, i_2)$, which maps to the generic $\mathrm{iadd}_{32}(i_1, i_2)$ via the above definition. Similarly, $\mathsf{i64.trunc\_f32\_s}$ applied to z invokes $\mathsf{trunc}^{\mathsf{s}}_{\mathsf{f32},\mathsf{i64}}(z)$, which maps to the generic $\mathrm{trunc}^{\mathsf{s}}{}_{32,64}(z)$.

$t.\mathsf{const}\ c$

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

Note: No formal reduction rule is required for this instruction, since const instructions already are values.

$nt.unop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\mathsf{const}\ c_1)$ from the stack.
3. If $|unop_{nt}(c_1)|$ is 1, then:
   a. Let c be $unop_{nt}(c_1)$.
   b. Push the value $(nt.\mathsf{const}\ c)$ to the stack.
4. If $unop_{nt}(c_1)$ is ϵ, then:
   a. Trap.

Todo:

(2) SpecTec currently defines the image of unop as a sequence, not a set.

$$\begin{array}{lcll}
(nt.\mathsf{const}\ c_1)\ (nt.unop) & \hookrightarrow & (nt.\mathsf{const}\ c) & \text{if } unop_{nt}(c_1) = c \\
(nt.\mathsf{const}\ c_1)\ (nt.unop) & \hookrightarrow & \mathsf{trap} & \text{if } unop_{nt}(c_1) = \epsilon
\end{array}$$

$nt.binop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\mathsf{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop the value $(nt.\mathsf{const}\ c_1)$ from the stack.
5. If $|binop_{nt}(c_1, c_2)|$ is 1, then:
   a. Let c be $binop_{nt}(c_1, c_2)$.
   b. Push the value $(nt.\mathsf{const}\ c)$ to the stack.
6. If $binop_{nt}(c_1, c_2)$ is ϵ, then:
   a. Trap.

Todo:

(2) SpecTec currently defines the image of binop as a sequence, not a set.

$$\begin{array}{lcll}
(nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.binop) & \hookrightarrow & (nt.\mathsf{const}\ c) & \text{if } binop_{nt}(c_1, c_2) = c \\
(nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.binop) & \hookrightarrow & \mathsf{trap} & \text{if } binop_{nt}(c_1, c_2) = \epsilon
\end{array}$$

$nt.testop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\mathsf{const}\ c_1)$ from the stack.
3. Let c be $testop_{nt}(c_1)$.
4. Push the value $(\mathsf{i32}.\mathsf{const}\ c)$ to the stack.

Todo:

(2) SpecTec currently defines the image of testop as a sequence, not a set.

$$(nt.\mathsf{const}\ c_1)\ (nt.testop) \quad \hookrightarrow \quad (\mathsf{i32}.\mathsf{const}\ c) \qquad \text{if } c = testop_{nt}(c_1)$$

$nt.relop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\mathsf{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop the value $(nt.\mathsf{const}\ c_1)$ from the stack.
5. Let c be $relop_{nt}(c_1, c_2)$.
6. Push the value $(\mathsf{i32}.\mathsf{const}\ c)$ to the stack.

Todo:

(2) SpecTec currently defines the image of testop as a sequence, not a set.

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

$nt_2.cvtop\_nt_1\_sx^?$

1. Assert: Due to validation, a value of value type nt_1 is on the top of the stack.
2. Pop the value $(nt_1.\mathsf{const}\ c_1)$ from the stack.
3. If $|cvtop^{sx^?}_{nt_1,nt_2}(c_1)|$ is 1, then:
   a. Let c be $cvtop^{sx^?}_{nt_1,nt_2}(c_1)$.
   b. Push the value $(nt_2.\mathsf{const}\ c)$ to the stack.
4. If $cvtop^{sx^?}_{nt_1,nt_2}(c_1)$ is ϵ, then:
   a. Trap.

Todo:

(2) SpecTec currently defines the image of testop as a sequence, not a set.

$$\begin{array}{lcll}(nt_1.\mathsf{const}\ c_1)\ (nt_2.cvtop\_nt_1\_sx^?) & \hookrightarrow & (nt_2.\mathsf{const}\ c) & \text{if } cvtop^{sx^?}_{nt_1,nt_2}(c_1) = c \\ (nt_1.\mathsf{const}\ c_1)\ (nt_2.cvtop\_nt_1\_sx^?) & \hookrightarrow & \mathsf{trap} & \text{if } cvtop^{sx^?}_{nt_1,nt_2}(c_1) = \epsilon\end{array}$$

4.6.3 Reference Instructions

$\mathsf{ref.null}\ x$

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Push the value $\mathsf{ref.null}\ deftype$ to the stack.

$$z; (\mathsf{ref.null}\ x) \quad \hookrightarrow \quad (\mathsf{ref.null}\ z.\mathsf{types}[x])$$

Note: No formal reduction rule is required for the case ref.null *absheaptype*, since the instruction form is already a value.

ref.func x

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module.funcs}|$.
3. Push the value $(\mathsf{ref.func}\ z.\mathsf{module.funcs}[x])$ to the stack.

$$z; (\mathsf{ref.func}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ z.\mathsf{module.funcs}[x])$$

ref.is_null

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value *ref* from the stack.
3. If *ref* is of the case ref.null, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
4. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$\begin{array}{lcll} \mathit{ref}\ \mathsf{ref.is\_null} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } \mathit{ref} = (\mathsf{ref.null}\ \mathit{ht}) \\ \mathit{ref}\ \mathsf{ref.is\_null} & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise} \end{array}$$

ref.as_non_null

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value *ref* from the stack.
3. If *ref* is of the case ref.null, then:
   a. Trap.
4. Push the value *ref* to the stack.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$\begin{array}{lcll} \mathit{ref}\ \mathsf{ref.as\_non\_null} & \hookrightarrow & \mathsf{trap} & \text{if } \mathit{ref} = (\mathsf{ref.null}\ \mathit{ht}) \\ \mathit{ref}\ \mathsf{ref.as\_non\_null} & \hookrightarrow & \mathit{ref} & \text{otherwise} \end{array}$$

ref.eq

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value ref_2 from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value ref_1 from the stack.
5. If ref_1 is of the case ref.null and ref_2 is of the case ref.null, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
6. Else if ref_1 is ref_2, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
7. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

Todo:

(3) Introduce if-let instruction instead of “is of the case”.

$$\begin{array}{lcll}
\mathit{ref}_1\ \mathit{ref}_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } \mathit{ref}_1 = (\mathsf{ref.null}\ ht_1) \land \mathit{ref}_2 = (\mathsf{ref.null}\ ht_2) \\
\mathit{ref}_1\ \mathit{ref}_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{otherwise, if } \mathit{ref}_1 = \mathit{ref}_2 \\
\mathit{ref}_1\ \mathit{ref}_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}$$

ref.test rt

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt' be $\mathrm{ref}_{type_{of}}(\mathit{ref})$.
5. If rt' matches $\mathrm{clos}_{f.\mathsf{module}}(rt)$, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
6. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

Todo: Below is the actual prose. (9) Need to handle RulePr s |- ref : rt properly in prose instead of $ref_type_of

---

1. Let F be the current frame.
2. Let rt_1 be the reference type $\mathrm{clos}_{F.\mathsf{module}}(rt)$.
3. Assert: due to validation, rt_1 is closed.
4. Assert: due to validation, a reference value is on the top of the stack.
5. Pop the value ref from the stack.
6. Assert: due to validation, the reference value is valid with some reference type.
7. Let rt_2 be the reference type of ref.
8. If the reference type rt_2 matches rt_1, then:
   a. Push the value $\mathsf{i32.const}\ 1$ to the stack.

9. Else:

   a. Push the value i32.const 0 to the stack.

$$
\begin{array}{lcll}
s; f; \mathit{ref}\ (\mathsf{ref.test}\ \mathit{rt}) & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if}\ s \vdash \mathit{ref} : \mathit{rt}' \\
 & & & \wedge\ \{\} \vdash \mathit{rt}' \leq \mathrm{clos}_{f.\mathsf{module}}(\mathit{rt}) \\
s; f; \mathit{ref}\ (\mathsf{ref.test}\ \mathit{rt}) & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}
$$

ref.cast rt

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt' be $\mathrm{ref}_{type_{of}}(ref)$.
5. If rt' does not match $\mathrm{clos}_{f.\mathsf{module}}(rt)$, then:

   a. Trap.

6. Push the value ref to the stack.

Todo: Below is the actual prose. (9) Need to handle RulePr s |- ref : rt properly in prose instead of $ref_type_of

---

1. Let F be the current frame.
2. Let rt_1 be the reference type $\mathrm{clos}_{F.\mathsf{module}}(rt)$.
3. Assert: due to validation, rt_1 is closed.
4. Assert: due to validation, a reference value is on the top of the stack.
5. Pop the value ref from the stack.
6. Assert: due to validation, the reference value is valid with some reference type.
7. Let rt_2 be the reference type of ref.
8. If the reference type rt_2 matches rt_1, then:

   a. Push the value ref back to the stack.

9. Else:

   a. Trap.

$$
\begin{array}{lcll}
s; f; \mathit{ref}\ (\mathsf{ref.cast}\ \mathit{rt}) & \hookrightarrow & \mathit{ref} & \text{if}\ s \vdash \mathit{ref} : \mathit{rt}' \\
 & & & \wedge\ \{\} \vdash \mathit{rt}' \leq \mathrm{clos}_{f.\mathsf{module}}(\mathit{rt}) \\
s; f; \mathit{ref}\ (\mathsf{ref.cast}\ \mathit{rt}) & \hookrightarrow & \mathsf{trap} & \text{otherwise}
\end{array}
$$

ref.i31

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
3. Push the value $(\mathsf{ref.i31}\ \mathrm{wrap}_{32,31}(i))$ to the stack.

$$
(\mathsf{i32.const}\ i)\ \mathsf{ref.i31} \quad \hookrightarrow \quad (\mathsf{ref.i31}\ \mathrm{wrap}_{32,31}(i))
$$

i31.get_sx

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $instr_{u0}$ from the stack.
3. If $instr_{u0}$ is of the case ref.null, then:
   a. Trap.
4. If $instr_{u0}$ is of the case ref.i31, then:
   a. Let (ref.i31 i) be $instr_{u0}$.
   b. Push the value (i32.const $\mathrm{extend}^{sx}_{31,32}(i)$) to the stack.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (4) Guarantees from validation can help simplify the prose.

---

1. Assert: due to validation, a value of type (ref null i31) is on the top of the stack.
2. Pop the value ref from the stack.
3. If ref is ref.null t, then:
   a. Trap.
4. Assert: due to validation, a ref is a scalar reference.
5. Let ref.i31 i be the reference value ref.
6. Let j be the result of computing $\mathrm{extend}^{sx}_{31,32}(i)$.
7. Push the value i32.const j to the stack.

$$\begin{array}{rcl}
(\mathsf{ref.null}\ ht)\ (\mathsf{i31.get_}sx) & \hookrightarrow & \mathsf{trap} \\
(\mathsf{ref.i31}\ i)\ (\mathsf{i31.get_}sx) & \hookrightarrow & (\mathsf{i32.const}\ \mathrm{extend}^{sx}_{31,32}(i))
\end{array}$$

struct.new x

1. Let z be the current state.
2. Let a be $|z.\mathsf{structs}|$.
3. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
4. Let (struct y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.
5. Let $(\mathsf{mut}^?, zt)^n$ be y_0.
6. Assert: Due to validation, there are at least n values on the top of the stack.
7. Pop the values val^n from the stack.
8. Let si be $\{\mathsf{type}\ z.\mathsf{types}[x],\ \mathsf{fields}\ \mathrm{pack}_{zt}(val)^n\}$.
9. Push the value (ref.struct a) to the stack.
10. Perform $z[.\mathsf{structs} =\oplus si]$.

Todo: Below is the actual prose. (3') Introduce let binding instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module.types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module.types}[x]$.

4. Assert: due to validation, the expansion of $\mathit{deftype}$ is a structure type.
5. Let $\mathsf{struct}~\mathit{ft}^*$ be the expanded structure type of $\mathit{deftype}$.
6. Let n be the length of the field type sequence ft^*.
7. Assert: due to validation, n values are on the top of the stack.
8. Pop the n values val^* from the stack.
9. For every value val_i in val^* and corresponding field type ft_i in ft^*:
   a. Let $\mathit{fieldval}_i$ be the result of computing $\mathrm{pack}_{\mathit{ft}_i}(\mathit{val}_i))$.
10. Let $\mathit{fieldval}^*$ the concatenation of all field values $\mathit{fieldval}_i$.
11. Let si be the structure instance $\{\mathsf{type}~\mathit{deftype}, \mathsf{fields}~\mathit{fieldval}^*\}$.
12. Let a be the length of $S.\mathsf{structs}$.
13. Append si to $S.\mathsf{structs}$.
14. Push the structure reference $\mathsf{ref.struct}~a$ to the stack.

$$z; \mathit{val}^n~(\mathsf{struct.new}~x) \quad\hookrightarrow\quad z[.\mathsf{structs} = \oplus~\mathit{si}]; (\mathsf{ref.struct}~a) \qquad \begin{array}{l} \mathrm{if}~z.\mathsf{types}[x] \approx \mathsf{struct}~(\mathsf{mut}^?~\mathit{zt})^n \\ \wedge~a = |z.\mathsf{structs}| \\ \wedge~\mathit{si} = \{\mathsf{type}~z.\mathsf{types}[x],~\mathsf{fields}~(\mathrm{pack}_{\mathit{zt}}(\mathit{val}))^n\} \end{array}$$

$\mathsf{struct.new\_default}~x$

1. Let z be the current state.
2. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
3. Let $(\mathsf{struct}~y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
4. Let $(\mathsf{mut}^?, \mathit{zt})^*$ be y_0.
5. Assert: Due to validation, $|\mathsf{mut}^{?*}|$ is $|\mathit{zt}^*|$.
6. Assert: Due to validation, for all $(\mathit{zt})^*$, $\mathrm{default}_{\mathrm{unpack}(\mathit{zt})}$ is defined.
7. Let val^* be $\mathrm{default}^*_{\mathrm{unpack}(\mathit{zt})}$.
8. Assert: Due to validation, $|\mathit{val}^*|$ is $|\mathit{zt}^*|$.
9. Push the values val^* to the stack.
10. Execute the instruction $(\mathsf{struct.new}~x)$.

Todo: Below is the actual prose. (3') Introduce let binding instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $\mathit{deftype}$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of $\mathit{deftype}$ is a structure type.
5. Let $\mathsf{struct}~\mathit{ft}^*$ be the expanded structure type of $\mathit{deftype}$.
6. Let n be the length of the field type sequence ft^*.
7. For every field type ft_i in ft^*:
   a. Let t_i be the value type $\mathrm{unpack}(\mathit{ft}_i)$.
   b. Assert: due to validation, $\mathrm{default}_{t_i}$ is defined.

c. Push the value $\mathrm{default}_{t_i}$ to the stack.

8. Execute the instruction $(\mathsf{struct.new}\ x)$.

$$z; (\mathsf{struct.new\_default}\ x) \hookrightarrow val^* \ (\mathsf{struct.new}\ x) \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{struct}\ (\mathsf{mut}^?\ zt)^* \wedge (\mathrm{default}_{\mathrm{unpack}(zt)} = val)^*$$

$\mathsf{struct.get\_}sx^?\ x\ i$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value $instr_{u0}$ from the stack.
4. If $instr_{u0}$ is of the case $\mathsf{ref.null}$, then:
   a. Trap.
5. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
6. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
7. Let $(\mathsf{mut}^?, zt)^*$ be y_0.
8. If $instr_{u0}$ is of the case $\mathsf{ref.struct}$, then:
   a. Let $(\mathsf{ref.struct}\ a)$ be $instr_{u0}$.
   b. If i is less than $|z.\mathsf{structs}[a].\mathsf{fields}|$ and a is less than $|z.\mathsf{structs}|$ and $|\mathsf{mut}^{?*}|$ is $|zt^*|$ and i is less than $|zt^*|$, then:
      1) Push the value $\mathrm{unpack}^{sx^?}_{zt^*[i]}(z.\mathsf{structs}[a].\mathsf{fields}[i])$ to the stack.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is a structure type with at least $y + 1$ fields.
5. Let $\mathsf{struct}\ ft^*$ be the expanded structure type of $deftype$.
6. Let ft_y be the y-th field type of ft^*.
7. Assert: due to validation, a value of type $(\mathsf{ref}\ \mathsf{null}\ x)$ is on the top of the stack.
8. Pop the value ref from the stack.
9. If ref is $\mathsf{ref.null}\ t$, then:
   a. Trap.
10. Assert: due to validation, a ref is a structure reference.
11. Let $\mathsf{ref.struct}\ a$ be the reference value ref.
12. Assert: due to validation, the structure instance $S.\mathsf{structs}[a]$ exists and has at least $y + 1$ fields.
13. Let $fieldval$ be the field value $S.\mathsf{structs}[a].\mathsf{fields}[y]$.
14. Let val be the result of computing $\mathrm{unpack}^{sx^?}_{ft_y}(fieldval)$).
15. Push the value val to the stack.

$$
\begin{array}{lcll}
z; (\mathsf{ref.null}\ ht)\ (\mathsf{struct.get\_}sx^?\ x\ i) & \hookrightarrow & \mathsf{trap} & \\
z; (\mathsf{ref.struct}\ a)\ (\mathsf{struct.get\_}sx^?\ x\ i) & \hookrightarrow & \mathrm{unpack}^{sx^?}_{zt^*[i]}(z.\mathsf{structs}[a].\mathsf{fields}[i]) & \text{if } z.\mathsf{types}[x] \approx \mathsf{struct}\ (\mathsf{mut}^?\ zt)^*
\end{array}
$$

struct.set x i

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value $instr_{u0}$ from the stack.
6. If $instr_{u0}$ is of the case ref.null, then:
   a. Trap.
7. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
8. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
9. Let $(\mathsf{mut}^?, zt)^*$ be y_0.
10. If $instr_{u0}$ is of the case ref.struct, then:
    a. Let $(\mathsf{ref.struct}\ a)$ be $instr_{u0}$.
    b. If $|\mathsf{mut}^{?*}|$ is $|zt^*|$ and i is less than $|zt^*|$, then:
       1) Perform $z[.\mathsf{structs}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt^*[i]}(val)]$.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module.types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module.types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is a structure type with at least $y + 1$ fields.
5. Let struct ft^* be the expanded structure type of $deftype$.
6. Let ft_y be the y-th field type of ft^*.
7. Assert: due to validation, a value is on the top of the stack.
8. Pop the value val from the stack.
9. Assert: due to validation, a value of type $(\mathsf{ref\ null}\ x)$ is on the top of the stack.
10. Pop the value ref from the stack.
11. If ref is $\mathsf{ref.null}\ t$, then:

a. Trap.

12. Assert: due to validation, a ref is a structure reference.
13. Let $\mathsf{ref.struct}\ a$ be the reference value ref.
14. Assert: due to validation, the structure instance $S.\mathsf{structs}[a]$ exists and has at least $y + 1$ fields.
15. Let $fieldval$ be the result of computing $\mathrm{pack}_{ft_y}(val)$).
16. Replace the field value $S.\mathsf{structs}[a].\mathsf{fields}[y]$ with $fieldval$.

$$
\begin{array}{lcll}
z; (\mathsf{ref.null}\ ht)\ val\ (\mathsf{struct.set}\ x\ i) & \hookrightarrow & z; \mathsf{trap} & \\
z; (\mathsf{ref.struct}\ a)\ val\ (\mathsf{struct.set}\ x\ i) & \hookrightarrow & z[.\mathsf{structs}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt^*[i]}(val)]; \epsilon & \text{if } z.\mathsf{types}[x] \approx \mathsf{struct}\ (\mathsf{mut}^?\ zt)^*
\end{array}
$$

array.new x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value (i32.const n) from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value val from the stack.
5. Push the values val^n to the stack.
6. Execute the instruction (array.new_fixed x n).

$$val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new}\ x) \quad \hookrightarrow \quad val^n\ (\mathsf{array.new\_fixed}\ x\ n)$$

array.new_default x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (i32.const n) from the stack.
4. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
5. Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.
6. Let $(\mathsf{mut}^?, zt)$ be y_0.
7. Assert: Due to validation, $\mathrm{default}_{\mathrm{unpack}(zt)}$ is defined.
8. Let val be $\mathrm{default}_{\mathrm{unpack}(zt)}$.
9. Push the values val^n to the stack.
10. Execute the instruction (array.new_fixed x n).

Todo: Below is the actual prose. (3') Introduce let binding instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type F.module.types[x] exists.
3. Let $deftype$ be the defined type F.module.types[x].
4. Assert: due to validation, the expansion of $deftype$ is an array type.
5. Let array ft be the expanded array type of $deftype$.
6. Assert: due to validation, a value of type i32 is on the top of the stack.
7. Pop the value i32.const n from the stack.
8. Let t be the value type $\mathrm{unpack}(ft)$.
9. Assert: due to validation, $\mathrm{default}_t$ is defined.
10. Push the value $\mathrm{default}_t$ to the stack n times.
11. Execute the instruction (array.new_fixed x n).

$$z; (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_default}\ x) \quad \hookrightarrow \quad val^n\ (\mathsf{array.new\_fixed}\ x\ n) \qquad \begin{array}{l} \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \\ \wedge\ \mathrm{default}_{\mathrm{unpack}(zt)} = val \end{array}$$

array.new_fixed x n

1. Let z be the current state.
2. Assert: Due to validation, there are at least n values on the top of the stack.
3. Pop the values val^n from the stack.
4. Let a be $|z.\mathsf{arrays}|$.
5. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
6. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
7. Let $(\mathsf{mut}^?, zt)$ be y_0.
8. Let ai be $\{\mathsf{type}\ z.\mathsf{types}[x],\ \mathsf{fields}\ \mathrm{pack}_{zt}(val)^n\}$.
9. Push the value $(\mathsf{ref.array}\ a)$ to the stack.
10. Perform $z[.\mathsf{arrays} = \oplus\ ai]$.

Todo: Below is the actual prose. (3') Introduce let binding instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is a array type.
5. Let $\mathsf{array}\ ft$ be the expanded array type of $deftype$.
6. Assert: due to validation, n values are on the top of the stack.
7. Pop the n values val^* from the stack.
8. For every value val_i in val^*:
   a. Let $fieldval_i$ be the result of computing $\mathrm{pack}_{ft}(val_i)$).
9. Let $fieldval^*$ be the concatenation of all field values $fieldval_i$.
10. Let ai be the array instance $\{\mathsf{type}\ deftype, \mathsf{fields}\ fieldval^*\}$.
11. Let a be the length of $S.\mathsf{arrays}$.
12. Append ai to $S.\mathsf{arrays}$.
13. Push the array reference $\mathsf{ref.array}\ a$ to the stack.

$$\begin{array}{lcl} z; val^n\ (\mathsf{array.new\_fixed}\ x\ n) & \hookrightarrow & z[.\mathsf{arrays} = \oplus\ ai]; (\mathsf{ref.array}\ a) \\ & & \quad \mathrm{if}\ z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \\ & & \quad \wedge\ a = |z.\mathsf{arrays}| \wedge ai = \{\mathsf{type}\ z.\mathsf{types}[x],\ \mathsf{fields}\ (\mathrm{pack}_{zt}(val))^n\} \end{array}$$

array.new_data x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (const i32 n) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (const i32 i) from the stack.
6. If $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array, then:
   a. Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.
   b. Let $(\mathsf{mut}^?, zt)$ be y_0.
   c. If $i + n \cdot |zt|/8$ is greater than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:
      1) Trap.
   d. Let cnn be $\mathrm{unpack}(zt)$.
   e. Let b^* be $z.\mathsf{datas}[y].\mathsf{bytes}[i : n \cdot |zt|/8]$.
   f. Let gb^* be $\mathrm{group}_{bytes_{by}}(|zt|/8, b^*)$.
   g. Let c^n be $\mathrm{inverse}_{of_{ibytes}}(|zt|, gb)^*$.
   h. Push the values $(\mathsf{const}\ cnn\ c)^n$ to the stack.
   i. Execute the instruction (array.new_fixed x n).

Todo: Below is the actual prose. (7) Render $inverse_ with display hint.

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let *deftype* be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of *deftype* is an array type.
5. Let array ft be the expanded array type of *deftype*.
6. Assert: due to validation, the data address $F.\mathsf{module}.\mathsf{datas}[y]$ exists.
7. Let da be the data address $F.\mathsf{module}.\mathsf{datas}[y]$.
8. Assert: due to validation, the data instance $S.\mathsf{datas}[da]$ exists.
9. Let *datainst* be the data instance $S.\mathsf{datas}[da]$.
10. Assert: due to validation, two values of type i32 are on the top of the stack.
11. Pop the value i32.const n from the stack.
12. Pop the value i32.const s from the stack.
13. Assert: due to validation, the field type ft has a defined bit width.
14. Let z be the bit width of field type ft divided by eight.
15. If the sum of s and n times z is larger than the length of *datainst*.bytes, then:
    a. Trap.
16. Let b^* be the byte sequence $\mathit{datainst}.\mathsf{bytes}[s : n \cdot z]$.
17. Let t be the value type $\mathrm{unpack}(ft)$.
18. For each consecutive subsequence ${b'}^n$ of b^*:

a. Assert: due to validation, bytes_{ft} is defined.

b. Let c_i be the constant for which $\mathrm{bytes}_{ft}(c_i)$ is b'^n.

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

19. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

$$
\begin{array}{lcl}
z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
& & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \\
& & \quad \wedge\ i + n \cdot |zt|/8 > |z.\mathsf{datas}[y].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) & \hookrightarrow & (\mathrm{unpack}(zt).\mathsf{const}\ \mathrm{unpack}_{zt}(c))^n\ (\mathsf{array.new\_fixed}\ x\ n) \\
& & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) \\
& & \quad \wedge\ \bigoplus \mathrm{bytes}_{zt}(c)^n = z.\mathsf{datas}[y].\mathsf{bytes}[i : n \cdot |zt|/8]
\end{array}
$$

$\mathsf{array.new\_elem}\ x\ y$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If $i + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:

   a. Trap.

7. Let ref^n be $z.\mathsf{elems}[y].\mathsf{elem}[i : n]$.
8. Push the values ref^n to the stack.
9. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

$$
\begin{array}{lcl}
z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \qquad \text{if } i + n > |z.\mathsf{elems}[y].\mathsf{elem}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_elem}\ x\ y) & \hookrightarrow & \mathit{ref}^n\ (\mathsf{array.new\_fixed}\ x\ n) \\
& & \quad \text{if } \mathit{ref}^n = z.\mathsf{elems}[y].\mathsf{elem}[i : n]
\end{array}
$$

$\mathsf{array.get\_}\mathit{sx}^?\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value instr_{u0} from the stack.
6. If instr_{u0} is of the case $\mathsf{ref.null}$, then:

   a. Trap.

7. If instr_{u0} is of the case $\mathsf{ref.array}$, then:

   a. Let $(\mathsf{ref.array}\ a)$ be instr_{u0}.

   b. If a is less than $|z.\mathsf{arrays}|$ and i is greater than or equal to $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:

      1) Trap.

8. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
9. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
10. Let $(\mathsf{mut}^?, zt)$ be y_0.

11. If $instr_{u0}$ is of the case ref.array, then:

    a. Let (ref.array a) be $instr_{u0}$.

    b. If i is less than $|z.\mathsf{arrays}[a].\mathsf{fields}|$ and a is less than $|z.\mathsf{arrays}|$, then:

        1) Push the value $\mathrm{unpack}_{zt}^{sx^?}(z.\mathsf{arrays}[a].\mathsf{fields}[i])$ to the stack.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is an array type.
5. Let array ft be the expanded array type of $deftype$.
6. Assert: due to validation, a value of type i32 is on the top of the stack.
7. Pop the value i32.const i from the stack.
8. Assert: due to validation, a value of type (ref null x) is on the top of the stack.
9. Pop the value ref from the stack.
10. If ref is ref.null t, then:

   a. Trap.

11. Assert: due to validation, ref is an array reference.
12. Let ref.array a be the reference value ref.
13. Assert: due to validation, the array instance $S.\mathsf{arrays}[a]$ exists.
14. If n is larger than or equal to the length of $S.\mathsf{arrays}[a].\mathsf{fields}$, then:

    a. Trap.

15. Let $fieldval$ be the field value $S.\mathsf{arrays}[a].\mathsf{fields}[i]$.
16. Let val be the result of computing $\mathrm{unpack}_{ft}^{sx^?}(fieldval)$).
17. Push the value val to the stack.

$$\begin{array}{lcll}
z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get_}sx^?\ x) & \hookrightarrow & \mathsf{trap} & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get_}sx^?\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i \geq |z.\mathsf{arrays}[a].\mathsf{fields}| \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get_}sx^?\ x) & \hookrightarrow & \mathrm{unpack}_{zt}^{sx^?}(z.\mathsf{arrays}[a].\mathsf{fields}[i]) & \\
 & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) &
\end{array}$$

array.set x

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const i) from the stack.
6. Assert: Due to validation, a value is on the top of the stack.
7. Pop the value $instr_{u0}$ from the stack.

8. If $instr_{u0}$ is of the case ref.null, then:
   a. Trap.
9. If $instr_{u0}$ is of the case ref.array, then:
   a. Let (ref.array a) be $instr_{u0}$.
   b. If a is less than $|z.\mathsf{arrays}|$ and i is greater than or equal to $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
      1) Trap.
10. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
11. Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.
12. Let $(\mathsf{mut}^?, zt)$ be y_0.
13. If $instr_{u0}$ is of the case ref.array, then:
    a. Let (ref.array a) be $instr_{u0}$.
    b. Perform $z[.\mathsf{arrays}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt}(val)]$.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module}.\mathsf{types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module}.\mathsf{types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is an array type.
5. Let array ft be the expanded array type of $deftype$.
6. Assert: due to validation, a value is on the top of the stack.
7. Pop the value val from the stack.
8. Assert: due to validation, a value of type i32 is on the top of the stack.
9. Pop the value i32.const i from the stack.
10. Assert: due to validation, a value of type (ref null x) is on the top of the stack.
11. Pop the value ref from the stack.
12. If ref is ref.null t, then:
    a. Trap.
13. Assert: due to validation, ref is an array reference.
14. Let ref.array a be the reference value ref.
15. Assert: due to validation, the array instance $S.\mathsf{arrays}[a]$ exists.
16. If n is larger than or equal to the length of $S.\mathsf{arrays}[a].\mathsf{fields}$, then:
    a. Trap.
17. Let $fieldval$ be the result of computing $\mathrm{pack}_{ft}(val)$).
18. Replace the field value $S.\mathsf{arrays}[a].\mathsf{fields}[i]$ with $fieldval$.

$$\begin{array}{lcll}
z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \text{if } i \geq |z.\mathsf{arrays}[a].\mathsf{fields}| \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z[.\mathsf{arrays}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt}(val)]; \epsilon & \\
 & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) &
\end{array}$$

array.len

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value $instr_{u0}$ from the stack.
4. If $instr_{u0}$ is of the case ref.null, then:
   a. Trap.
5. If $instr_{u0}$ is of the case ref.array, then:
   a. Let (ref.array a) be $instr_{u0}$.
   b. If a is less than $|z.\mathsf{arrays}|$, then:
      1) Push the value (i32.const $|z.\mathsf{arrays}[a].\mathsf{fields}|$) to the stack.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$\begin{array}{rcl} z; (\mathsf{ref.null}\ ht)\ \mathsf{array.len} & \hookrightarrow & \mathsf{trap} \\ z; (\mathsf{ref.array}\ a)\ \mathsf{array.len} & \hookrightarrow & (\mathsf{i32.const}\ |z.\mathsf{arrays}[a].\mathsf{fields}|) \end{array}$$

array.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (i32.const n) from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value (i32.const i) from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value $instr_{u0}$ from the stack.
10. If $instr_{u0}$ is of the case ref.null, then:
    a. Trap.
11. If $instr_{u0}$ is of the case ref.array, then:
    a. Let (ref.array a) be $instr_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
       1) Trap.
    c. If n is 0, then:
       1) Do nothing.
    d. Else:
       1) Let (ref.array a) be $instr_{u0}$.
       2) Push the value (ref.array a) to the stack.
       3) Push the value (i32.const i) to the stack.

4) Push the value val to the stack.
5) Execute the instruction $(\mathsf{array.set}\ x)$.
6) Push the value $(\mathsf{ref.array}\ a)$ to the stack.
7) Push the value $(\mathsf{i32.const}\ i+1)$ to the stack.
8) Push the value val to the stack.
9) Push the value $(\mathsf{i32.const}\ n-1)$ to the stack.
10) Execute the instruction $(\mathsf{array.fill}\ x)$.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$
\begin{array}{lcll}
z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \mathsf{trap} & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i+n > |z.\mathsf{arrays}[a].\mathsf{fields}| \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & & \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & & & \text{otherwise} \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.const}\ n-1)\ (\mathsf{array.fill}\ x) & & &
\end{array}
$$

$\mathsf{array.copy}\ x_1\ x_2$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i_2)$ from the stack.
6. Assert: Due to validation, a value is on the top of the stack.
7. Pop the value $instr_{u1}$ 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 $(\mathsf{i32.const}\ i_1)$ from the stack.
10. Assert: Due to validation, a value is on the top of the stack.
11. Pop the value $instr_{u0}$ from the stack.
12. If $instr_{u0}$ is of the case $\mathsf{ref.null}$ and the type of $instr_{u1}$ is ref, then:
    a. Trap.
13. If $instr_{u1}$ is of the case $\mathsf{ref.null}$ and the type of $instr_{u0}$ is ref, then:
    a. Trap.
14. If $instr_{u0}$ is of the case $\mathsf{ref.array}$, then:
    a. Let $(\mathsf{ref.array}\ a_1)$ be $instr_{u0}$.
    b. If $instr_{u1}$ is of the case $\mathsf{ref.array}$, then:
        1) If a_1 is less than $|z.\mathsf{arrays}|$ and i_1+n is greater than $|z.\mathsf{arrays}[a_1].\mathsf{fields}|$, then:
            a) Trap.
        2) Let $(\mathsf{ref.array}\ a_2)$ be $instr_{u1}$.
        3) If a_2 is less than $|z.\mathsf{arrays}|$ and i_2+n is greater than $|z.\mathsf{arrays}[a_2].\mathsf{fields}|$, then:

a) Trap.

c. If n is 0, then:

1) If $instr_{u1}$ is of the case ref.array, then:

a) Do nothing.

d. Else if i_1 is greater than i_2, then:

1) Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x_2])$ is of the case array.

2) Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x_2])$.

3) Let $(\mathsf{mut}^?, zt_2)$ be y_0.

4) Let $(\mathsf{ref.array}\ a_1)$ be $instr_{u0}$.

5) If $instr_{u1}$ is of the case ref.array, then:

a) Let $(\mathsf{ref.array}\ a_2)$ be $instr_{u1}$.

b) Let $sx^?$ be $sx(zt_2)$.

c) Push the value $(\mathsf{ref.array}\ a_1)$ to the stack.

d) Push the value $(\mathsf{i32.const}\ i_1 + n - 1)$ to the stack.

e) Push the value $(\mathsf{ref.array}\ a_2)$ to the stack.

f) Push the value $(\mathsf{i32.const}\ i_2 + n - 1)$ to the stack.

g) Execute the instruction $(\mathsf{array.get\_}sx^?\ x_2)$.

h) Execute the instruction $(\mathsf{array.set}\ x_1)$.

i) Push the value $(\mathsf{ref.array}\ a_1)$ to the stack.

j) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.

k) Push the value $(\mathsf{ref.array}\ a_2)$ to the stack.

l) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.

m) Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

n) Execute the instruction $(\mathsf{array.copy}\ x_1\ x_2)$.

e. Else:

1) Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x_2])$ is of the case array.

2) Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x_2])$.

3) Let $(\mathsf{mut}^?, zt_2)$ be y_0.

4) Let $(\mathsf{ref.array}\ a_1)$ be $instr_{u0}$.

5) If $instr_{u1}$ is of the case ref.array, then:

a) Let $(\mathsf{ref.array}\ a_2)$ be $instr_{u1}$.

b) Let $sx^?$ be $sx(zt_2)$.

c) Push the value $(\mathsf{ref.array}\ a_1)$ to the stack.

d) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.

e) Push the value $(\mathsf{ref.array}\ a_2)$ to the stack.

f) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.

g) Execute the instruction $(\mathsf{array.get\_}sx^?\ x_2)$.

h) Execute the instruction $(\mathsf{array.set}\ x_1)$.

i) Push the value $(\mathsf{ref.array}\ a_1)$ to the stack.

j) Push the value $(\mathsf{i32.const}\ i_1 + 1)$ to the stack.

k) Push the value $(\mathsf{ref.array}\ a_2)$ to the stack.

l) Push the value $(\mathsf{i32.const}\ i_2 + 1)$ to the stack.

m) Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

n) Execute the instruction $(\mathsf{array.copy}\ x_1\ x_2)$.

Todo: Below is the actual prose. (3) Introduce if-let instruction instead of "is of the case". (5) Use "the expansion of" instead of $expand function application.

---

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module.types}[y]$ exists.
3. Let $\mathit{deftype}$ be the defined type $F.\mathsf{module.types}[y]$.
4. Assert: due to validation, the expansion of $\mathit{deftype}$ is an array type.
5. Let $\mathsf{array}\ \mathit{mut}\ \mathit{st}$ be the expanded array type $\mathit{deftype}$.
6. Assert: due to validation, a value of type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ n$ from the stack.
8. Assert: due to validation, a value of type $\mathsf{i32}$ is on the top of the stack.
9. Pop the value $\mathsf{i32.const}\ s$ from the stack.
10. Assert: due to validation, a value of type $(\mathsf{ref}\ \mathsf{null}\ y)$ is on the top of the stack.
11. Pop the value ref_2 from the stack.
12. Assert: due to validation, a value of type $\mathsf{i32}$ is on the top of the stack.
13. Pop the value $\mathsf{i32.const}\ d$ from the stack.
14. Assert: due to validation, a value of type $(\mathsf{ref}\ \mathsf{null}\ x)$ is on the top of the stack.
15. Pop the value ref_1 from the stack.
16. If ref_1 is $\mathsf{ref.null}\ t$, then:

    a. Trap.
17. Assert: due to validation, ref_1 is an array reference.
18. Let $\mathsf{ref.array}\ a_1$ be the reference value ref_1.
19. If ref_2 is $\mathsf{ref.null}\ t$, then:

    a. Trap.
20. Assert: due to validation, ref_2 is an array reference.
21. Let $\mathsf{ref.array}\ a_2$ be the reference value ref_2.
22. Assert: due to validation, the array instance $S.\mathsf{arrays}[a_1]$ exists.
23. Assert: due to validation, the array instance $S.\mathsf{arrays}[a_2]$ exists.
24. If $d + n$ is larger than the length of $S.\mathsf{arrays}[a_1].\mathsf{fields}$, then:

    a. Trap.
25. If $s + n$ is larger than the length of $S.\mathsf{arrays}[a_2].\mathsf{fields}$, then:

    a. Trap.
26. If $n = 0$, then:

    a. Return.

27. If $d \leq s$, then:
    a. Push the value ref.array a_1 to the stack.
    b. Push the value i32.const d to the stack.
    c. Push the value ref.array a_2 to the stack.
    d. Push the value i32.const s to the stack.
    e. Execute $\mathrm{getfield}(st)$.
    f. Execute the instruction array.set x.
    g. Push the value ref.array a_1 to the stack.
    h. Assert: due to the earlier check against the array size, $d + 1 < 2^{32}$.
    i. Push the value i32.const $(d + 1)$ to the stack.
    j. Push the value ref.array a_2 to the stack.
    k. Assert: due to the earlier check against the array size, $s + 1 < 2^{32}$.
    l. Push the value i32.const $(s + 1)$ to the stack.
28. Else:
    a. Push the value ref.array a_1 to the stack.
    b. Assert: due to the earlier check against the array size, $d + n - 1 < 2^{32}$.
    c. Push the value i32.const $(d + n - 1)$ to the stack.
    d. Push the value ref.array a_2 to the stack.
    e. Assert: due to the earlier check against the array size, $s + n - 1 < 2^{32}$.
    f. Push the value i32.const $(s + n - 1)$ to the stack.
    g. Execute $\mathrm{getfield}(st)$.
    h. Execute the instruction array.set x.
    i. Push the value ref.array a_1 to the stack.
    j. Push the value i32.const d to the stack.
    k. Push the value ref.array a_2 to the stack.
    l. Push the value i32.const s to the stack.
29. Push the value i32.const $(n - 1)$ to the stack.
30. Execute the instruction array.copy x y.

$$
\begin{array}{lcl}
z; (\mathsf{ref.null}\ ht_1)\ (\mathsf{i32.const}\ i_1)\ \mathit{ref}\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} \\
z; \mathit{ref}\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.null}\ ht_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} \\
z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } i_1 + n > |z.\mathsf{arrays}[a_1].\mathsf{fields}| & & \\
z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } i_2 + n > |z.\mathsf{arrays}[a_2].\mathsf{fields}| & & \\
z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \epsilon \\
\quad \text{otherwise, if } n = 0 & & \\
z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \\
\quad (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1) & & \\
\quad (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2) & & \\
\quad (\mathsf{array.get\_}\mathit{sx}^?\ x_2)\ (\mathsf{array.set}\ x_1) & & \\
\quad (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1 + 1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2 + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.copy}\ x_1\ x_2) & & \\
\quad \text{otherwise, if } z.\mathsf{types}[x_2] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt_2) & & \\
\quad \wedge\ i_1 \leq i_2 \wedge \mathit{sx}^? = \mathit{sx}(zt_2) & & \\
z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) & \hookrightarrow & \\
\quad (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1 + n - 1) & & \\
\quad (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2 + n - 1) & & \\
\quad (\mathsf{array.get\_}\mathit{sx}^?\ x_2)\ (\mathsf{array.set}\ x_1) & & \\
\quad (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.copy}\ x_1\ x_2) & & \\
\quad \text{otherwise, if } z.\mathsf{types}[x_2] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt_2) & & \\
\quad \wedge\ \mathit{sx}^? = \mathit{sx}(zt_2) & &
\end{array}
$$

Where:

$$
\begin{array}{lcl}
\mathit{sx}(\mathit{consttype}) & = & \epsilon \\
\mathit{sx}(\mathit{packtype}) & = & \mathsf{s}
\end{array}
$$

array.init_data x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (i32.const n) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const j) from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value (i32.const i) from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value instr_{u0} from the stack.
10. If instr_{u0} is of the case ref.null, then:
    a. Trap.
11. If instr_{u0} is of the case ref.array, then:
    a. Let (ref.array a) be instr_{u0}.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
       1) Trap.
12. If $\mathrm{expand}(z.\mathsf{types}[x])$ is not of the case array, then:
    a. If n is 0 and instr_{u0} is of the case ref.array, then:
       1) Do nothing.

13. Else:
    a. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
    b. Let $(\mathsf{mut}^?, zt)$ be y_0.
    c. If $instr_{u0}$ is of the case $\mathsf{ref.array}$, then:
        1) If $j + n \cdot |zt|/8$ is greater than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:
            a) Trap.
        2) If n is 0, then:
            a) Do nothing.
        3) Else:
            a) Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
            b) Let $(\mathsf{mut}^?, zt)$ be y_0.
            c) Let $(\mathsf{ref.array}\ a)$ be $instr_{u0}$.
            d) Let c be $\mathrm{inverse}_{of_{zbytes}}(zt, z.\mathsf{datas}[y].\mathsf{bytes}[j : |zt|/8])$.
            e) Push the value $(\mathsf{ref.array}\ a)$ to the stack.
            f) Push the value $(\mathsf{i32.const}\ i)$ to the stack.
            g) Push the value $\mathrm{unpack}(zt).\mathsf{const}\ \mathrm{unpack}_{zt}(c)$ to the stack.
            h) Execute the instruction $(\mathsf{array.set}\ x)$.
            i) Push the value $(\mathsf{ref.array}\ a)$ to the stack.
            j) Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
            k) Push the value $(\mathsf{i32.const}\ j + |zt|/8)$ to the stack.
            l) Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
            m) Execute the instruction $(\mathsf{array.init\_data}\ x\ y)$.

Todo: Below is the actual prose. (7) Render $inverse_ with display hint.

1. Let F be the current frame.
2. Assert: due to validation, the defined type $F.\mathsf{module.types}[x]$ exists.
3. Let $deftype$ be the defined type $F.\mathsf{module.types}[x]$.
4. Assert: due to validation, the expansion of $deftype$ is an array type.
5. Let $\mathsf{array}\ ft$ be the expanded array type $deftype$.
6. Assert: due to validation, the data address $F.\mathsf{module.datas}[y]$ exists.
7. Let da be the data address $F.\mathsf{module.datas}[y]$.
8. Assert: due to validation, the data instance $S.\mathsf{datas}[da]$ exists.
9. Let $datainst$ be the data instance $S.\mathsf{datas}[da]$.
10. Assert: due to validation, three values of type $\mathsf{i32}$ are on the top of the stack.
11. Pop the value $\mathsf{i32.const}\ n$ from the stack.
12. Pop the value $\mathsf{i32.const}\ s$ from the stack.
13. Pop the value $\mathsf{i32.const}\ d$ from the stack.
14. Assert: due to validation, a value of type $(\mathsf{ref\ null}\ x)$ is on the top of the stack.
15. Pop the value ref from the stack.

16. If ref is ref.null t, then:
   a. Trap.
17. Assert: due to validation, ref is an array reference.
18. Let ref.array a be the reference value ref.
19. Assert: due to validation, the array instance S.arrays$[a]$ exists.
20. Assert: due to validation, the field type ft has a defined bit width.
21. Let z be the bit width of field type ft divided by eight.
22. If $d + n$ is larger than the length of S.arrays$[a]$.fields, or the sum of s and n times z is larger than the length of $datainst$.bytes, then:
   a. Trap.
23. If $n = 0$, then:
   a. Return.
24. Let b^* be the byte sequence $datainst$.bytes$[s : z]$.
25. Let t be the value type $\mathrm{unpack}(ft)$.
26. Assert: due to validation, bytes_{ft} is defined.
27. Let c be the constant for which $\mathrm{bytes}_{ft}(c)$ is b^*.
28. Push the value ref.array a to the stack.
29. Push the value i32.const d to the stack.
30. Push the value t.const c to the stack.
31. Execute the instruction array.set x.
32. Push the value ref.array a to the stack.
33. Push the value i32.const $(d + 1)$ to the stack.
34. Push the value i32.const $(s + z)$ to the stack.
35. Push the value i32.const $(n - 1)$ to the stack.
36. Execute the instruction array.init_data x y.

$$
\begin{array}{lcl}
z; (\mathsf{ref.null}~ht)~(\mathsf{i32.const}~i)~(\mathsf{i32.const}~j)~(\mathsf{i32.const}~n)~(\mathsf{array.init\_data}~x~y) & \hookrightarrow & \mathsf{trap} \\
z; (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i)~(\mathsf{i32.const}~j)~(\mathsf{i32.const}~n)~(\mathsf{array.init\_data}~x~y) & \hookrightarrow & \mathsf{trap} \\
\qquad \text{if } i + n > |z.\mathsf{arrays}[a].\mathsf{fields}| & & \\
z; (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i)~(\mathsf{i32.const}~j)~(\mathsf{i32.const}~n)~(\mathsf{array.init\_data}~x~y) & \hookrightarrow & \mathsf{trap} \\
\qquad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}~(\mathsf{mut}^?~zt) & & \\
\qquad \wedge\, j + n \cdot |zt|/8 > |z.\mathsf{datas}[y].\mathsf{bytes}| & & \\
z; (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i)~(\mathsf{i32.const}~j)~(\mathsf{i32.const}~n)~(\mathsf{array.init\_data}~x~y) & \hookrightarrow & \epsilon \\
\qquad \text{otherwise, if } n = 0 & & \\
z; (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i)~(\mathsf{i32.const}~j)~(\mathsf{i32.const}~n)~(\mathsf{array.init\_data}~x~y) & \hookrightarrow & \\
\quad (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i)~(\mathrm{unpack}(zt).\mathsf{const}~\mathrm{unpack}_{zt}(c))~(\mathsf{array.set}~x) & & \\
\quad (\mathsf{ref.array}~a)~(\mathsf{i32.const}~i+1)~(\mathsf{i32.const}~j + |zt|/8)~(\mathsf{i32.const}~n-1)~(\mathsf{array.init\_data}~x~y) & & \\
\qquad \text{otherwise, if } z.\mathsf{types}[x] \approx \mathsf{array}~(\mathsf{mut}^?~zt) & & \\
\qquad \wedge\, \mathrm{bytes}_{zt}(c) = z.\mathsf{datas}[y].\mathsf{bytes}[j : |zt|/8] & &
\end{array}
$$

array.init_elem x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (i32.const n) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const j) from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value (i32.const i) from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value $instr_{u0}$ from the stack.
10. If $instr_{u0}$ is of the case ref.null, then:
    a. Trap.
11. If $instr_{u0}$ is of the case ref.array, then:
    a. Let (ref.array a) be $instr_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
        1) Trap.
12. If $j + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:
    a. If $instr_{u0}$ is of the case ref.array, then:
        1) Trap.
    b. If n is 0 and j is less than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:
        1) Let ref be $z.\mathsf{elems}[y].\mathsf{elem}[j]$.
        2) If $instr_{u0}$ is of the case ref.array, then:
            a) Let (ref.array a) be $instr_{u0}$.
            b) Push the value (ref.array a) to the stack.
            c) Push the value (i32.const i) to the stack.
            d) Push the value ref to the stack.
            e) Execute the instruction (array.set x).
            f) Push the value (ref.array a) to the stack.
            g) Push the value (i32.const $i + 1$) to the stack.
            h) Push the value (i32.const $j + 1$) to the stack.
            i) Push the value (i32.const $n - 1$) to the stack.
            j) Execute the instruction (array.init_elem x y).
13. Else if n is 0, then:
    a. If $instr_{u0}$ is of the case ref.array, then:
        1) Do nothing.
14. Else:
    a. If j is less than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:
        1) Let ref be $z.\mathsf{elems}[y].\mathsf{elem}[j]$.
        2) If $instr_{u0}$ is of the case ref.array, then:

a) Let $(\mathsf{ref.array}\ a)$ be $instr_{u0}$.

b) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

c) Push the value $(\mathsf{i32.const}\ i)$ to the stack.

d) Push the value ref to the stack.

e) Execute the instruction $(\mathsf{array.set}\ x)$.

f) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

g) Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.

h) Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.

i) Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

j) Execute the instruction $(\mathsf{array.init\_elem}\ x\ y)$.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$
\begin{array}{lcl}
z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } i + n > |z.\mathsf{arrays}[a].\mathsf{fields}| & & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } j + n > |z.\mathsf{elems}[y].\mathsf{elem}| & & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \epsilon \\
\quad \text{otherwise, if } n = 0 & & \\
z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ ref\ (\mathsf{array.set}\ x) & & \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i + 1)\ (\mathsf{i32.const}\ j + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.init\_elem}\ x\ y) & & \\
\quad \text{otherwise, if } ref = z.\mathsf{elems}[y].\mathsf{elem}[j] & &
\end{array}
$$

any.convert_extern

1. Assert: Due to validation, a value is on the top of the stack.

2. Pop the value $instr_{u0}$ from the stack.

3. If $instr_{u0}$ is of the case $\mathsf{ref.null}$, then:

   a. Push the value $(\mathsf{ref.null}\ \mathsf{any})$ to the stack.

4. If $instr_{u0}$ is of the case $\mathsf{ref.extern}$, then:

   a. Let $(\mathsf{ref.extern}\ addrref)$ be $instr_{u0}$.

   b. Push the value $addrref$ to the stack.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$
\begin{array}{rcl}
(\mathsf{ref.null}\ ht)\ \mathsf{any.convert\_extern} & \hookrightarrow & (\mathsf{ref.null}\ \mathsf{any}) \\
(\mathsf{ref.extern}\ addrref)\ \mathsf{any.convert\_extern} & \hookrightarrow & addrref
\end{array}
$$

extern.convert_any

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $instr_{u0}$ from the stack.
3. If $instr_{u0}$ is of the case ref.null, then:
   a. Push the value (ref.null extern) to the stack.
4. If the type of $instr_{u0}$ is addrref, then:
   a. Let $addrref$ be $instr_{u0}$.
   b. Push the value (ref.extern $addrref$) to the stack.

Todo:

(3) Introduce if-let instruction instead of "is of the case".

$$\begin{array}{rcl}(\mathsf{ref.null}\ ht)\ \mathsf{extern.convert\_any} & \hookrightarrow & (\mathsf{ref.null}\ \mathsf{extern})\\ addrref\ \mathsf{extern.convert\_any} & \hookrightarrow & (\mathsf{ref.extern}\ addrref)\end{array}$$

4.6.4 Vector Instructions

Vector instructions that operate bitwise are handled as integer operations of respective width.

$$op_{\mathsf{v}N}(i_1, \ldots, i_k) \quad = \quad \mathrm{i}op_N(i_1, \ldots, i_k)$$

Most other vector instructions are defined in terms of numeric operators that are applied lane-wise according to the given shape.

$$op_{t\mathsf{x}N}(n_1, \ldots, n_k) \quad = \quad \mathrm{lanes}^{-1}_{t\mathsf{x}N}(op_t(i_1, \ldots, i_k)^*) \qquad (\text{if } i_1^* = \mathrm{lanes}_{t\mathsf{x}N}(n_1) \wedge \cdots \wedge i_k^* = \mathrm{lanes}_{t\mathsf{x}N}(n_k)$$

Note: For example, the result of instruction i32x4.add applied to operands v_1, v_2 invokes $\mathsf{add}_{\mathsf{i32x4}}(v_1, v_2)$, which maps to $\mathrm{lanes}^{-1}_{\mathsf{i32x4}}(\mathsf{add}_{\mathsf{i32}}(i_1, i_2)^*)$, where i_1^* and i_2^* are sequences resulting from invoking $\mathrm{lanes}_{\mathsf{i32x4}}(v_1)$ and $\mathrm{lanes}_{\mathsf{i32x4}}(v_2)$ respectively.

v128.const c

1. Push the value v128.const c to the stack.

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

v128.$vvunop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value (v128.const c_1) from the stack.
3. Let c be $vvunop_{\mathsf{v128}}(c_1)$.
4. Push the value (v128.const c) to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128}.vvunop) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = vvunop_{\mathsf{v128}}(c_1)$$

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

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $\mathit{vvbinop}_{\mathsf{v128}}(c_1, c_2)$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{v128}.\mathit{vvbinop}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = \mathit{vvbinop}_{\mathsf{v128}}(c_1, c_2)$$

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

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_3)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
7. Let c be $\mathit{vvternop}_{\mathsf{v128}}(c_1, c_2, c_3)$.
8. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$\begin{array}{lcl}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{v128.const}\ c_3)\ (\mathsf{v128}.\mathit{vvternop}) & \hookrightarrow & (\mathsf{v128.const}\ c) \\ & & \quad \text{if } c = \mathit{vvternop}_{\mathsf{v128}}(c_1, c_2, c_3)\end{array}$$

$\mathsf{v128.any\_true}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $\mathrm{ine}_{|\mathsf{v128}|}(c_1, 0)$.
4. Push the value $(\mathsf{i32.const}\ c)$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.any\_true}) \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad \text{if } c = \mathrm{ine}_{|\mathsf{v128}|}(c_1, 0)$$

$\mathit{sh}.\mathit{vunop}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $\mathit{vunop}_{\mathit{sh}}(c_1)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathit{sh}.\mathit{vunop}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = \mathit{vunop}_{\mathit{sh}}(c_1)$$

$sh.vbinop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. If $|vbinop_{sh}(c_1, c_2)|$ is 1, then:
   a. Let c be $vbinop_{sh}(c_1, c_2)$.
   b. Push the value $(\mathsf{v128.const}\ c)$ to the stack.
6. If $vbinop_{sh}(c_1, c_2)$ is ϵ, then:
   a. Trap.

Todo:

(2) SpecTec currently defines the image of vbinop as a sequence, not a set.

$$\begin{array}{lcll}
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.vbinop) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } vbinop_{sh}(c_1, c_2) = c \\
(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.vbinop) & \hookrightarrow & \mathsf{trap} & \text{if } vbinop_{sh}(c_1, c_2) = \epsilon
\end{array}$$

$\mathsf{i}N\mathsf{x}M.\mathsf{all\_true}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c)$ from the stack.
3. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c)$.
4. If for all $(ci_1)^*$, ci_1 is not 0, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
5. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

$$\begin{array}{lcll}
(\mathsf{v128.const}\ c)\ (\mathsf{i}N\mathsf{x}M.\mathsf{all_true}) & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c) \\
 & & & \wedge\ (ci_1 \neq 0)^* \\
(\mathsf{v128.const}\ c)\ (\mathsf{i}N\mathsf{x}M.\mathsf{all_true}) & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}$$

$sh.vrelop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $vrelop_{sh}(c_1, c_2)$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.

3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i_1^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_1)$.
5. Let i_2^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_2)$.
6. Let i^* be the result of computing $\mathit{vrelop}_t(i_1^*, i_2^*)$.
7. Let j^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{1,|t|}(i^*)$.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{t\mathsf{x}N}(j^*)$.
9. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.\mathit{vrelop}) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{vrelop}_{sh}(c_1, c_2) = c$$

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ t\mathsf{x}N.\mathit{vrelop} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \\ \quad (\text{if } c = \mathrm{lanes}^{-1}_{t\mathsf{x}N}(\mathrm{extend}^{\mathsf{s}}{}_{1,|t|}(\mathit{vrelop}_t(\mathrm{lanes}_{t\mathsf{x}N}(c_1), \mathrm{lanes}_{t\mathsf{x}N}(c_2)))))\end{array}$$

$\mathsf{i}N\mathsf{x}M.\mathit{vshiftop}$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c'^* be $\mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_1)$.
6. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(\mathit{vshiftop}_{\mathsf{i}N\mathsf{x}M}(c', n)^*)$.
7. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $\mathsf{i32.const}\ s$ from the stack.
3. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
4. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
5. Let i^* be the result of computing $\mathrm{lanes}_{t\mathsf{x}N}(c_1)$.
6. Let j^* be the result of computing $\mathit{vishiftop}_t(i^*, s^N)$.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t\mathsf{x}N}(j^*)$.
8. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{lcll}(\mathsf{v128.const}\ c_1)\ (\mathsf{i32.const}\ n)\ (\mathsf{i}N\mathsf{x}M.\mathit{vshiftop}) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } c'^* = \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_1) \\ & & & \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(\mathit{vshiftop}_{\mathsf{i}N\mathsf{x}M}(c', n)^*)\end{array}$$

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{i32.const}\ s)\ t\mathsf{x}N.\mathit{vishiftop} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \\ \quad (\text{if } i^* = \mathrm{lanes}_{t\mathsf{x}N}(c_1) \\ \quad\ \wedge\ c = \mathrm{lanes}^{-1}_{t\mathsf{x}N}(\mathit{vishiftop}_t(i^*, s^N)))\end{array}$$

iNxM.bitmask

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value (v128.const c) from the stack.
3. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}N\times M}(c)$.
4. Let ci be $\mathrm{inverse}_{of\ _{ibits}}(32, \mathrm{ilt}^{\mathsf{s}}_{|\mathsf{i}N|}(ci_1, 0)^*)$.
5. Push the value (i32.const ci) to the stack.

Todo: Below is the actual prose.

1. Assert: due to validation, a value of value type v128 is on the top of the stack.
2. Pop the value v128.const c from the stack.
3. Let i_1^N be the result of computing $\mathrm{lanes}_{t\times N}(c)$.
4. Let B be the bit width $|t|$ of value type t.
5. Let i_2^N be the result of computing $\mathrm{ilt\_s}_B(i_1^N, 0^N)$.
6. Let j^* be the concatenation of the two sequences i_2^N and 0^{32-N}.
7. Let i be the result of computing $\mathrm{ibits}_{32}^{-1}(j^*)$.
8. Push the value i32.const i onto the stack.

$$(\mathsf{v128.const}\ c)\ (\mathsf{i}N\mathsf{x}M\mathsf{.bitmask}) \quad \hookrightarrow \quad (\mathsf{i32.const}\ ci) \qquad \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}N\times M}(c) \wedge \mathrm{bits}_{\mathsf{i32}}(ci) = \mathrm{ilt}^{\mathsf{s}}_{|\mathsf{i}N|}(ci_1, 0)^*$$

$$(\mathsf{v128.const}\ c)\ t\mathsf{x}N\mathsf{.bitmask} \quad \hookrightarrow \quad (\mathsf{i32.const}\ i) \qquad (\text{if } i = \mathrm{ibits}_{32}^{-1}(\mathrm{ilt\_s}_{|t|}(\mathrm{lanes}_{t\times N}(c), 0^N)))$$

iNxM.swizzle

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value (v128.const c_2) from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value (v128.const c_1) from the stack.
5. Let c^* be $\mathrm{lanes}_{\mathsf{i}N\times M}(c_1)\ 0^{256-M}$.
6. Let ci^* be $\mathrm{lanes}_{\mathsf{i}N\times M}(c_2)$.
7. Assert: Due to validation, for all $(k)^{k<M}$, $ci^*[k]$ is less than $|c^*|$.
8. Assert: Due to validation, for all $(k)^{k<M}$, k is less than $|ci^*|$.
9. Let c' be $\mathrm{lanes}^{-1}_{\mathsf{i}N\times M}(c^*[ci^*[k]]^{k<M})$.
10. Push the value (v128.const c') to the stack.

Todo: Below is the actual prose.

1. Assert: due to validation, two values of value type v128 are on the top of the stack.
2. Pop the value v128.const c_2 from the stack.
3. Let i^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_2)$.
4. Pop the value v128.const c_1 from the stack.
5. Let j^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_1)$.

6. Let c^* be the concatenation of the two sequences j^* and 0^{240}.
7. Let c' be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i8x16}}(c^*[i^*[0]] \dots c^*[i^*[15]])$.
8. Push the value $\mathsf{v128.const}\ c'$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{i}N\mathsf{x}M\mathsf{.swizzle}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c') \qquad \begin{array}{l} \text{if } ci^* = \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_2) \\ \wedge\ c^* = \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_1)\ 0^{256-M} \\ \wedge\ c' = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(c^*[ci^*[k]]^{k<M}) \end{array}$$

$$\begin{array}{l} (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathsf{i8x16.swizzle} \quad \hookrightarrow \quad (\mathsf{v128.const}\ c') \\ \quad (\text{if } i^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_2) \\ \quad\ \wedge\ c^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_1)\ 0^{240} \\ \quad\ \wedge\ c' = \mathrm{lanes}^{-1}_{\mathsf{i8x16}}(c^*[i^*[0]] \dots c^*[i^*[15]])) \end{array}$$

$\mathsf{i}N\mathsf{x}M\mathsf{.shuffle}\ i^*$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Assert: Due to validation, for all $(k)^{k<M}$, k is less than $|i^*|$.
6. Let c'^* be $\mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_1)\ \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_2)$.
7. Assert: Due to validation, for all $(k)^{k<M}$, $i^*[k]$ is less than $|c'^*|$.
8. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(c'^*[i^*[k]]^{k<M})$.
9. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Assert: due to validation, for all x_i in x^* it holds that $x_i < 32$.
3. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
4. Let i_2^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_2)$.
5. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
6. Let i_1^* be the result of computing $\mathrm{lanes}_{\mathsf{i8x16}}(c_1)$.
7. Let i^* be the concatenation of the two sequences i_1^* and i_2^*.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i8x16}}(i^*[x^*[0]] \dots i^*[x^*[15]])$.
9. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{i}N\mathsf{x}M\mathsf{.shuffle}\ i^*) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \begin{array}{l} \text{if } c'^* = \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_1)\ \mathrm{lanes}_{\mathsf{i}N\mathsf{x}M}(c_2) \\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(c'^*[i^*[k]]^{k<M}) \end{array}$$

$$\begin{array}{l} (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{i8x16.shuffle}\ x^*) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \\ \quad (\text{if } i^* = \mathrm{lanes}_{\mathsf{i8x16}}(c_1)\ \mathrm{lanes}_{\mathsf{i8x16}}(c_2) \\ \quad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i8x16}}(i^*[x^*[0]] \dots i^*[x^*[15]])) \end{array}$$

$\mathsf{i}N\mathsf{x}M\mathsf{.splat}$

1. Assert: Due to validation, a value of value type $\mathrm{unpack}(\mathsf{i}N)$ is on the top of the stack.
2. Pop the value $(nt_0\mathsf{.const}\ c_1)$ from the stack.
3. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(\mathrm{pack}_{\mathsf{i}N}(c_1)^M)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Let t be the type $\mathrm{unpack}(\mathit{shape})$.
2. Assert: due to validation, a value of value type t is on the top of the stack.
3. Pop the value $t\mathsf{.const}\ c_1$ from the stack.
4. Let N be the integer $\mathrm{dim}(\mathit{shape})$.
5. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathit{shape}}(c_1^N)$.
6. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$(\mathrm{unpack}(\mathsf{i}N)\mathsf{.const}\ c_1)\ (\mathsf{i}N\mathsf{x}M\mathsf{.splat}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(\mathrm{pack}_{\mathsf{i}N}(c_1)^M)$$

$$(t\mathsf{.const}\ c_1)\ \mathit{shape}\mathsf{.splat} \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad (\text{if } t = \mathrm{unpack}(\mathit{shape}) \wedge c = \mathrm{lanes}^{-1}_{\mathit{shape}}(c_1^{\mathrm{dim}(\mathit{shape})}))$$

$\mathit{lanet}_{u0}\mathsf{x}M\mathsf{.extract\_lane\_}\mathit{sx}^?_{u1}\ i$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. If $\mathit{sx}^?_{u1}$ is not defined and the type of lanet_{u0} is numtype, then:
   a. Let nt be lanet_{u0}.
   b. If i is less than $|\mathrm{lanes}_{\mathit{nt}\mathsf{x}M}(c_1)|$, then:
      1) Let c_2 be $\mathrm{lanes}_{\mathit{nt}\mathsf{x}M}(c_1)[i]$.
      2) Push the value $(\mathit{nt}\mathsf{.const}\ c_2)$ to the stack.
4. If the type of lanet_{u0} is packtype, then:
   a. Let pt be lanet_{u0}.
   b. If $\mathit{sx}^?_{u1}$ is defined, then:
      1) Let sx be $\mathit{sx}^?_{u1}$.
      2) If i is less than $|\mathrm{lanes}_{\mathit{pt}\mathsf{x}M}(c_1)|$, then:
         a) Let c_2 be $\mathrm{extend}^{\mathit{sx}}_{|\mathit{pt}|,32}(\mathrm{lanes}_{\mathit{pt}\mathsf{x}M}(c_1)[i])$.
         b) Push the value $(\mathsf{i32.const}\ c_2)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to validation, $x < N$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}N}(c_1)$.

5. Let t_2 be the type $\mathrm{unpack}(t_1 \times N)$.
6. Let c_2 be the result of computing $\mathrm{extend}^{sx^?}_{t_1,t_2}(i^*[x])$.
7. Push the value $t_2.\mathsf{const}\ c_2$ to the stack.

$$\begin{array}{lcll}
(\mathsf{v128.const}\ c_1)\ (nt \times M.\mathsf{extract_lane}\ i) & \hookrightarrow & (nt.\mathsf{const}\ c_2) & \text{if } c_2 = \mathrm{lanes}_{nt \times M}(c_1)[i] \\
(\mathsf{v128.const}\ c_1)\ (pt \times M.\mathsf{extract_lane_}sx\ i) & \hookrightarrow & (\mathsf{i32.const}\ c_2) & \text{if } c_2 = \mathrm{extend}^{sx}_{|pt|,32}(\mathrm{lanes}_{pt \times M}(c_1)[i])
\end{array}$$

$$\begin{array}{l}
(\mathsf{v128.const}\ c_1)\ (t_1 \times N.\mathsf{extract_lane}\ x) \quad \hookrightarrow \quad (t_2.\mathsf{const}\ c_2) \\
\quad (\text{if } t_2 = \mathrm{unpack}(t_1 \times N) \\
\quad \wedge\ c_2 = \mathrm{extend}^{sx^?}_{t_1,t_2}(\mathrm{lanes}_{t_1 \times N}(c_1)[x]))
\end{array}$$

$\mathsf{i}N \times M.\mathsf{replace\_lane}\ i$

1. Assert: Due to validation, a value of value type $\mathrm{unpack}(\mathsf{i}N)$ is on the top of the stack.
2. Pop the value $(nt_0.\mathsf{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}N \times M}(\mathrm{lanes}_{\mathsf{i}N \times M}(c_1)[[i] = \mathrm{pack}_{\mathsf{i}N}(c_2)])$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to validation, $x < \mathrm{dim}(shape)$.
2. Let t_2 be the type $\mathrm{unpack}(shape)$.
3. Assert: due to validation, a value of value type t_1 is on the top of the stack.
4. Pop the value $t_2.\mathsf{const}\ c_2$ from the stack.
5. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
6. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
7. Let i^* be the result of computing $\mathrm{lanes}_{shape}(c_1)$.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{shape}(i^*\ \mathrm{with}\ [x] = c_2)$.
9. Push $\mathsf{v128.const}\ c$ on the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathrm{unpack}(\mathsf{i}N).\mathsf{const}\ c_2)\ (\mathsf{i}N \times M.\mathsf{replace\_lane}\ i) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = \mathrm{lanes}^{-1}_{\mathsf{i}N \times M}(\mathrm{lanes}_{\mathsf{i}N \times M}(c_1)[[i] = \mathrm{pa}$$

$$\begin{array}{l}
(\mathsf{v128.const}\ c_1)\ (t_2.\mathsf{const}\ c_2)\ (shape.\mathsf{replace_lane}\ x) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \\
\quad (\text{if } i^* = \mathrm{lanes}_{shape}(c_1) \\
\quad \wedge\ c = \mathrm{lanes}^{-1}_{shape}(i^*\ \mathrm{with}\ [x] = c_2))
\end{array}$$

$sh_2.vextunop\_sh_1\_sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $vextunop(sh_1, sh_2, vextunop, sx, c_1)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

---

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, a value of value type v128 is on the top of the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_1)$.
5. Let $(j_1\ j_2)^*$ be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(i^*)$.
6. Let k^* be the result of computing $\mathrm{iadd}_{|t_2|}(j_1, j_2)^*$.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2 \times N}(k^*)$.
8. Push the value v128.const c to the stack.

$$(\mathsf{v128.const}\ c_1)\ (sh_2.vextunop\_sh_1\_sx) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } vextunop(sh_1, sh_2, vextunop, sx, c_1) = c$$

$$\begin{array}{l} (\mathsf{v128.const}\ c_1)\ t_2 \mathsf{x} N.\mathsf{extadd\_pairwise}\_t_1 \mathsf{x} M\_sx \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \\ \quad (\text{if } (i_1\ i_2)^* = \mathrm{extend}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1 \times M}(c_1)) \\ \quad \wedge\ j^* = \mathrm{iadd}_{|t_2|}(i_1, i_2)^* \\ \quad \wedge\ c = \mathrm{lanes}^{-1}_{t_2 \times N}(j^*)) \end{array}$$

$sh_2.vextbinop\_sh_1\_sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value (v128.const c_2) from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value (v128.const c_1) from the stack.
5. Let c be $vextbinop(sh_1, sh_2, vextbinop, sx, c_1, c_2)$.
6. Push the value (v128.const c) to the stack.

Todo: Below is the actual prose.

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, two values of value type v128 are on the top of the stack.
3. Pop the value v128.const c_2 from the stack.
4. Pop the value v128.const c_1 from the stack.
5. Let i_1^* be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_1)$.
6. Let i_2^* be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_2)$.
7. If $half$ is low, then:
   a. Let j_1^* be the sequence $i_1^*[0 : N]$.
   b. Let j_2^* be the sequence $i_2^*[0 : N]$.
8. Else:
   a. Let j_1^* be the sequence $i_1^*[N : N]$.
   b. Let j_2^* be the sequence $i_2^*[N : N]$.
9. Let k_1^* be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(j_1^*)$.
10. Let k_2^* be the result of computing $\mathrm{extend}^{sx}_{|t_1|,|t_2|}(j_2^*)$.
11. Let k^* be the result of computing $\mathrm{imul}_{|t_2|}(k_1^*, k_2^*)$.
12. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2 \times N}(k^*)$.

13. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh_2.vextbinop\_sh_1\_sx) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad \text{if } vextbinop(sh_1, sh_2, vextbinop, sx, c_1, c_2) = c$$

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ t_2\mathsf{x}N.\mathsf{extmul}\_half\_t_1\mathsf{x}M\_sx \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)\\ \quad(\text{if } i^* = \mathrm{lanes}_{t_1\mathsf{x}M}(c_1)[half(0,N) : N]\\ \quad\wedge\ j^* = \mathrm{lanes}_{t_1\mathsf{x}M}(c_2)[half(0,N) : N]\\ \quad\wedge\ c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathrm{imul}_{|t_2|}(\mathrm{extend}^{sx}_{|t_1|,|t_2|}(i^*), \mathrm{extend}^{sx}_{|t_1|,|t_2|}(j^*))))\end{array}$$

where:

$$\begin{array}{lcl}\mathsf{low}(x, y) & = & x\\ \mathsf{high}(x, y) & = & y\end{array}$$

i32x4.dot_i16x8_s

Todo: (*) Prose not spliced, for this seems to be WIP on @Andreas.

1. Assert: due to validation, two values of value type $\mathsf{v128}$ are on the top of the stack.
2. Pop the value $\mathsf{v128.const}\ c_2$ from the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i_1^* be the result of computing $\mathrm{lanes}_{\mathsf{i16x8}}(c_1)$.
5. Let j_1^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{16,32}(i_1^*)$.
6. Let i_2^* be the result of computing $\mathrm{lanes}_{\mathsf{i16x8}}(c_2)$.
7. Let j_2^* be the result of computing $\mathrm{extend}^{\mathsf{s}}{}_{16,32}(i_2^*)$.
8. Let $(k_1\ k_2)^*$ be the result of computing $\mathrm{imul}_{32}(j_1^*, j_2^*)$.
9. Let k^* be the result of computing $\mathrm{iadd}_{32}(k_1, k_2)^*$.
10. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i32x4}}(k^*)$.
11. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$\begin{array}{l}(\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ \mathsf{i32x4.dot\_i16x8\_s} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)\\ \quad(\text{if } (i_1\ i_2)^* = \mathrm{imul}_{32}(\mathrm{extend}^{\mathsf{s}}{}_{16,32}(\mathrm{lanes}_{\mathsf{i16x8}}(c_1)), \mathrm{extend}^{\mathsf{s}}{}_{16,32}(\mathrm{lanes}_{\mathsf{i16x8}}(c_2)))\\ \quad\wedge\ j^* = \mathrm{iadd}_{32}(i_1, i_2)^*\\ \quad\wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i32x4}}(j^*))\end{array}$$

$\mathsf{i}N_2\mathsf{x}M_2.\mathsf{narrow}\_\mathsf{i}N_1\mathsf{x}M_1\_sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}N_1\mathsf{x}M_1}(c_1)$.
6. Let ci_2^* be $\mathrm{lanes}_{\mathsf{i}N_1\mathsf{x}M_1}(c_2)$.
7. Let cj_1^* be $\mathrm{narrow}^{sx}_{|\mathsf{i}N_1|,|\mathsf{i}N_2|}\ {ci_1}^*$.
8. Let cj_2^* be $\mathrm{narrow}^{sx}_{|\mathsf{i}N_1|,|\mathsf{i}N_2|}\ {ci_2}^*$.
9. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}N_2\mathsf{x}M_2}(cj_1^*\ cj_2^*)$.
10. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

Todo: Below is the actual prose.

1. Assert: due to syntax, $N = 2 \cdot M$.
2. Assert: due to validation, two values of value type v128 are on the top of the stack.
3. Pop the value v128.const c_2 from the stack.
4. Let i_2^M be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_2)$.
5. Let d_2^M be the result of computing $\mathrm{narrow}^{sx}_{|t_1|,|t_2|}(i_2^M)$.
6. Pop the value v128.const c_1 from the stack.
7. Let i_1^M be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_1)$.
8. Let d_1^M be the result of computing $\mathrm{narrow}^{sx}_{|t_1|,|t_2|}(i_1^M)$.
9. Let j^N be the concatenation of the two sequences d_1^M and d_2^M.
10. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2 \times N}(j^N)$.
11. Push the value v128.const c onto the stack.

$$
\begin{array}{lcll}
(\mathsf{v128.const}~c_1)~(\mathsf{v128.const}~c_2)~(\mathsf{i}N_2\mathsf{x}M_2.\mathsf{narrow\_i}N_1\mathsf{x}M_1\mathsf{\_}sx) & \hookrightarrow & (\mathsf{v128.const}~c) & \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}N_1\times M_1}(c_1) \\
 & & & \wedge~ ci_2^* = \mathrm{lanes}_{\mathsf{i}N_1\times M_1}(c_2) \\
 & & & \wedge~ cj_1^* = \mathrm{narrow}^{sx}_{|\mathsf{i}N_1|,|\mathsf{i}N_2|}\,{ci_1}^* \\
 & & & \wedge~ cj_2^* = \mathrm{narrow}^{sx}_{|\mathsf{i}N_1|,|\mathsf{i}N_2|}\,{ci_2}^* \\
 & & & \wedge~ c = \mathrm{lanes}^{-1}_{\mathsf{i}N_2\times M_2}(cj_1^*~cj_2^*)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{v128.const}~c_1)~(\mathsf{v128.const}~c_2)~t_2\mathsf{x}N.\mathsf{narrow\_}t_1\mathsf{x}M\mathsf{\_}sx \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \\
\quad (\text{if } d_1^M = \mathrm{narrow}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\times M}(c_1)) \\
\quad \wedge~ d_2^M = \mathrm{narrow}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\times M}(c_2)) \\
\quad \wedge~ c = \mathrm{lanes}^{-1}_{t_2\times N}(d_1^M~d_2^M))
\end{array}
$$

$t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_sx$

Todo: (*) Prose not spliced, for it has merged multiple rules for vcvtop into one algorithm.

1. Assert: due to syntax, $N = M$.
2. Assert: due to validation, a value of value type v128 is on the top of the stack.
3. Pop the value v128.const c_1 from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1 \times M}(c_1)$.
5. Let j^* be the result of computing $\mathit{vcvtop}^{sx}_{|t_1|,|t_2|}(i^*)$.
6. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2 \times N}(j^*)$.
7. Push the value v128.const c onto the stack.

$$
\begin{array}{lcll}
(\mathsf{v128.const}~c_1)~(\mathsf{i}N_2\mathsf{x}M.\mathit{vcvtop}\mathsf{\_i}N_1\mathsf{x}M\mathsf{\_}sx^?) & \hookrightarrow & (\mathsf{v128.const}~c) & \text{if } {c'}^* = \mathrm{lanes}_{\mathsf{i}N_1\times M}(c_1) \\
 & & & \wedge~ c = \mathrm{lanes}^{-1}_{\mathsf{i}N_2\times M}(\mathit{vcvtop}(\mathsf{i}N_1\mathsf{x}M, \mathsf{i}N_2\mathsf{x}M, \mathit{vcvtop}, sx^?,
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{v128.const}~c_1)~t_2\mathsf{x}N.\mathit{vcvtop}\_t_1\mathsf{x}M\_sx \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \\
\quad (\text{if } c = \mathrm{lanes}^{-1}_{t_2\times N}(\mathit{vcvtop}^{sx}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\times M}(c_1))))
\end{array}
$$

$t_2\mathsf{x}N.\mathit{vcvtop\_half\_}t_1\mathsf{x}M\_\mathit{sx}^?$

Todo: (*) Prose not spliced, for it has merged multiple rules for vcvtop into one algorithm.

1. Assert: due to syntax, $N = M/2$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. If half is low, then:
   a. Let j^* be the sequence $i^*[0 : N]$.
6. Else:
   a. Let j^* be the sequence $i^*[N : N]$.
7. Let k^* be the result of computing $\mathit{vcvtop}^{\mathit{sx}^?}_{|t_1|,|t_2|}(j^*)$.
8. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.
9. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathsf{i}N_2\mathsf{x}M_2.\mathit{vcvtop\_sx}^?\mathsf{\_i}N_1\mathsf{x}M_1\mathsf{\_}\mathit{half}) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{ci}^* = \mathrm{lanes}_{\mathsf{i}N_1\mathsf{x}M_1}(c_1)[\mathit{half}(\mathit{half}, 0, M_2) : M_2]$$
$$\wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N_2\mathsf{x}M_2}(\mathit{vcvtop}(\mathsf{i}N_1\mathsf{x}M_1, \mathsf{i}N_2\mathsf{x}M_2, \mathit{vc}$$

$$(\mathsf{v128.const}\ c_1)\ t_2\mathsf{x}N.\mathit{vcvtop\_half\_}t_1\mathsf{x}M\_\mathit{sx}^? \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)$$
$$(\text{if } c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathit{vcvtop}^{\mathit{sx}^?}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)[\mathit{half}(0, N) : N])))$$

where:

$$\begin{array}{lcl} \mathsf{low}(x, y) & = & x \\ \mathsf{high}(x, y) & = & y \end{array}$$

$t_2\mathsf{x}N.\mathit{vcvtop\_}t_1\mathsf{x}M\_\mathit{sx}^?\mathsf{\_zero}$

Todo: (*) Prose not spliced, for it has merged multiple rules for vcvtop into one algorithm.

1. Assert: due to syntax, $N = 2 \cdot M$.
2. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
3. Pop the value $\mathsf{v128.const}\ c_1$ from the stack.
4. Let i^* be the result of computing $\mathrm{lanes}_{t_1\mathsf{x}M}(c_1)$.
5. Let j^* be the result of computing $\mathit{vcvtop}^{\mathit{sx}^?}_{|t_1|,|t_2|}(i^*)$.
6. Let k^* be the concatenation of the two sequences j^* and 0^M.
7. Let c be the result of computing $\mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(k^*)$.
8. Push the value $\mathsf{v128.const}\ c$ onto the stack.

$$(\mathsf{v128.const}\ c_1)\ (\mathit{nt}_2\mathsf{x}M_2.\mathit{vcvtop\_zero\_nt}_1\mathsf{x}M_1\_\mathit{sx}^?) \quad\hookrightarrow\quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{ci}^* = \mathrm{lanes}_{\mathit{nt}_1\mathsf{x}M_1}(c_1)$$
$$\wedge\ c = \mathrm{lanes}^{-1}_{\mathit{nt}_2\mathsf{x}M_2}(\mathit{vcvtop}(\mathit{nt}_1\mathsf{x}M_1, \mathit{nt}_2\mathsf{x}M_2, \mathit{vcvt}$$

$$(\mathsf{v128.const}\ c_1)\ t_2\mathsf{x}N.\mathit{vcvtop\_}t_1\mathsf{x}M\_\mathit{sx}^?\mathsf{\_zero} \quad\hookrightarrow\quad (\mathsf{v128.const}\ c)$$
$$(\text{if } c = \mathrm{lanes}^{-1}_{t_2\mathsf{x}N}(\mathit{vcvtop}^{\mathit{sx}^?}_{|t_1|,|t_2|}(\mathrm{lanes}_{t_1\mathsf{x}M}(c_1))\ 0^M))$$

4.6.5 Variable Instructions

$\mathsf{local.get}\ x$

1. Let z be the current state.
2. Assert: Due to validation, $z.\mathsf{locals}[x]$ is defined.
3. Let val be $z.\mathsf{locals}[x]$.
4. Push the value val to the stack.

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

$\mathsf{local.set}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Perform $z[.\mathsf{locals}[x] = val]$.

$$z; val\ (\mathsf{local.set}\ x) \quad \hookrightarrow \quad z[.\mathsf{locals}[x] = val]; \epsilon$$

$\mathsf{local.tee}\ x$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. Push the value val to the stack.
4. Push the value val to the stack.
5. Execute the instruction $(\mathsf{local.set}\ x)$.

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

$\mathsf{global.get}\ x$

1. Let z be the current state.
2. Let val be $z.\mathsf{globals}[x].\mathsf{value}$.
3. Push the value val to the stack.

$$z; (\mathsf{global.get}\ x) \quad \hookrightarrow \quad val \qquad \text{if } z.\mathsf{globals}[x].\mathsf{value} = val$$

$\mathsf{global.set}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Perform $z[.\mathsf{globals}[x].\mathsf{value} = val]$.

$$z; val\ (\mathsf{global.set}\ x) \quad \hookrightarrow \quad z[.\mathsf{globals}[x].\mathsf{value} = val]; \epsilon$$

4.6.6 Table Instructions

$\mathsf{table.get}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. If i is greater than or equal to $|z.\mathsf{tables}[x].\mathsf{elem}|$, then:
   a. Trap.
5. Push the value $z.\mathsf{tables}[x].\mathsf{elem}[i]$ to the stack.

$$\begin{array}{lcll} z; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i \geq |z.\mathsf{tables}[x].\mathsf{elem}| \\ z; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & z.\mathsf{tables}[x].\mathsf{elem}[i] & \text{if } i < |z.\mathsf{tables}[x].\mathsf{elem}| \end{array}$$

$\mathsf{table.set}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If i is greater than or equal to $|z.\mathsf{tables}[x].\mathsf{elem}|$, then:
   a. Trap.
7. Perform $z[.\mathsf{tables}[x].\mathsf{elem}[i] = \mathit{ref}]$.

$$\begin{array}{lcll} z; (\mathsf{i32.const}\ i)\ \mathit{ref}\ (\mathsf{table.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \text{if } i \geq |z.\mathsf{tables}[x].\mathsf{elem}| \\ z; (\mathsf{i32.const}\ i)\ \mathit{ref}\ (\mathsf{table.set}\ x) & \hookrightarrow & z[.\mathsf{tables}[x].\mathsf{elem}[i] = \mathit{ref}]; \epsilon & \text{if } i < |z.\mathsf{tables}[x].\mathsf{elem}| \end{array}$$

$\mathsf{table.size}\ x$

1. Let z be the current state.
2. Let n be $|z.\mathsf{tables}[x].\mathsf{elem}|$.
3. Push the value $(\mathsf{i32.const}\ n)$ to the stack.

$$z; (\mathsf{table.size}\ x) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \qquad \text{if } |z.\mathsf{tables}[x].\mathsf{elem}| = n$$

$\mathsf{table.grow}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value ref from the stack.
6. Either:
   a. Let ti be $\mathrm{growtable}(z.\mathsf{tables}[x], n, \mathit{ref})$.
   b. Push the value $(\mathsf{i32.const}\ |z.\mathsf{tables}[x].\mathsf{elem}|)$ to the stack.

   c. Perform $z[.\mathsf{tables}[x] = ti]$.

7. Or:

   a. Push the value $(\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))$ to the stack.

Todo:

(6) Maybe we want to be more precise than just saying "Either" in the prose, elaborating that it may succeed or fail non-deterministically.

$$
\begin{array}{lcll}
z; \mathit{ref}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z[.\mathsf{tables}[x] = ti]; (\mathsf{i32.const}\ |z.\mathsf{tables}[x].\mathsf{elem}|) \\
 & & \quad \text{if } ti = \mathrm{growtable}(z.\mathsf{tables}[x], n, \mathit{ref}) \\
z; \mathit{ref}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))
\end{array}
$$

Note: The table.grow instruction is non-deterministic. It may either succeed, returning the old table size sz, or fail, returning -1. Failure *must* occur if the referenced table instance has a maximum size defined that would be exceeded. However, failure *can* occur in other cases as well. In practice, the choice depends on the resources available to the embedder.

---

table.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{tables}[x].\mathsf{elem}|$, then:

   a. Trap.

9. If n is 0, then:

   a. Do nothing.

10. Else:

    a. Push the value $(\mathsf{i32.const}\ i)$ to the stack.

    b. Push the value val to the stack.

    c. Execute the instruction $(\mathsf{table.set}\ x)$.

    d. Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.

    e. Push the value val to the stack.

    f. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

    g. Execute the instruction $(\mathsf{table.fill}\ x)$.

$$
\begin{array}{lcll}
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i + n > |z.\mathsf{tables}[x].\mathsf{elem}| \\
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{table.set}\ x) & & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ i + 1)\ \mathit{val}\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.fill}\ x) & & &
\end{array}
$$

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

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{tables}[y].\mathsf{elem}|$, then:
   a. Trap.
9. If $j + n$ is greater than $|z.\mathsf{tables}[x].\mathsf{elem}|$, then:
   a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else:
    a. If j is less than or equal to i, then:
       1) Push the value $(\mathsf{i32.const}\ j)$ to the stack.
       2) Push the value $(\mathsf{i32.const}\ i)$ to the stack.
       3) Execute the instruction $(\mathsf{table.get}\ y)$.
       4) Execute the instruction $(\mathsf{table.set}\ x)$.
       5) Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.
       6) Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
    b. Else:
       1) Push the value $(\mathsf{i32.const}\ j + n - 1)$ to the stack.
       2) Push the value $(\mathsf{i32.const}\ i + n - 1)$ to the stack.
       3) Execute the instruction $(\mathsf{table.get}\ y)$.
       4) Execute the instruction $(\mathsf{table.set}\ x)$.
       5) Push the value $(\mathsf{i32.const}\ j)$ to the stack.
       6) Push the value $(\mathsf{i32.const}\ i)$ to the stack.
    c. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
    d. Execute the instruction $(\mathsf{table.copy}\ x\ y)$.

$$
\begin{array}{lll}
z;(\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow\ \mathsf{trap} & \\
\quad \text{if } i + n > |z.\mathsf{tables}[y].\mathsf{elem}| \lor j + n > |z.\mathsf{tables}[x].\mathsf{elem}| & & \\
z;(\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow\ \epsilon & \text{otherwise, if } n = 0 \\
z;(\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \\
\quad (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & \text{otherwise, if } j \leq i \\
\quad (\mathsf{i32.const}\ j + 1)\ (\mathsf{i32.const}\ i + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.copy}\ x\ y) & & \\
z;(\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \\
\quad (\mathsf{i32.const}\ j + n - 1)\ (\mathsf{i32.const}\ i + n - 1)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.copy}\ x\ y) & &
\end{array}
$$

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

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:
   a. Trap.
9. If $j + n$ is greater than $|z.\mathsf{tables}[x].\mathsf{elem}|$, then:
   a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else if i is less than $|z.\mathsf{elems}[y].\mathsf{elem}|$, then:
    a. Push the value $(\mathsf{i32.const}\ j)$ to the stack.
    b. Push the value $z.\mathsf{elems}[y].\mathsf{elem}[i]$ to the stack.
    c. Execute the instruction $(\mathsf{table.set}\ x)$.
    d. Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.
    e. Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
    f. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
    g. Execute the instruction $(\mathsf{table.init}\ x\ y)$.

$$
\begin{array}{lcll}
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \mathsf{trap} & \\
\qquad \text{if } i + n > |z.\mathsf{elems}[y].\mathsf{elem}| \lor j + n > |z.\mathsf{tables}[x].\mathsf{elem}| & & & \\
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ j)\ z.\mathsf{elems}[y].\mathsf{elem}[i]\ (\mathsf{table.set}\ x) & & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ j + 1)\ (\mathsf{i32.const}\ i + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.init}\ x\ y) & & &
\end{array}
$$

$\mathsf{elem.drop}\ x$

1. Let z be the current state.
2. Perform $z[.\mathsf{elems}[x].\mathsf{elem} = \epsilon]$.

$$
z; (\mathsf{elem.drop}\ x) \quad \hookrightarrow \quad z[.\mathsf{elems}[x].\mathsf{elem} = \epsilon]; \epsilon
$$

4.6.7 Memory Instructions

Note: The alignment $memarg.\mathsf{align}$ in load and store instructions does not affect the semantics. It is an indication that the offset ea at which the memory is accessed is intended to satisfy the property $ea \bmod 2^{memarg.\mathsf{align}} = 0$. A WebAssembly implementation can use this hint to optimize for the intended use. Unaligned access violating that property is still allowed and must succeed regardless of the annotation. However, it may be substantially slower on some hardware.

$numty_{u0}.\mathsf{load} loado^?_{u2}\ x\ ao$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. If $loado^?_{u2}$ is not defined, then:
   a. Let nt be $numty_{u0}$.
   b. If $i + ao.\mathsf{offset} + |nt|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
      1) Trap.
   c. Let c be $\mathrm{inverse}_{of_{nbytes}}(nt, z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : |nt|/8])$.
   d. Push the value $(nt.\mathsf{const}\ c)$ to the stack.
5. If the type of $numty_{u0}$ is Inn, then:
   a. If $loado^?_{u2}$ is defined, then:
      1) Let y_0 be $loado^?_{u2}$.
      2) Let (n, sx) be y_0.
      3) If $i + ao.\mathsf{offset} + n/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
   b. Let $\mathrm{i}N$ be $numty_{u0}$.
   c. If $loado^?_{u2}$ is defined, then:
      1) Let y_0 be $loado^?_{u2}$.
      2) Let (n, sx) be y_0.
      3) Let c be $\mathrm{inverse}_{of_{ibytes}}(n, z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : n/8])$.
      4) Push the value $(\mathrm{i}N.\mathsf{const}\ \mathsf{extend}^{sx}_{n,|\mathrm{i}N|}(c))$ to the stack.

Todo: Below is the actual prose. (7) Render $inverse_of_nbytes with display hint.

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ i$ from the stack.

8. Let ea be the integer $i + memarg.\mathsf{offset}$.
9. If N is not part of the instruction, then:
   a. Let N be the bit width $|t|$ of number type t.
10. If $ea + N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:
    a. Trap.
11. Let b^* be the byte sequence $mem.\mathsf{bytes}[ea : N/8]$.
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 $t.\mathsf{const}\ c$ to the stack.

$$
\begin{array}{rcl}
z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ ao) & \hookrightarrow & \mathsf{trap} \\
 & & \quad \text{if } i + ao.\mathsf{offset} + |nt|/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ ao) & \hookrightarrow & (nt.\mathsf{const}\ c) \\
 & & \quad \text{if } \mathrm{bytes}_{nt}(c) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : |nt|/8] \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i}N.\mathsf{load}n\ sx\ x\ ao) & \hookrightarrow & \mathsf{trap} \\
 & & \quad \text{if } i + ao.\mathsf{offset} + n/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i}N.\mathsf{load}n\ sx\ x\ ao) & \hookrightarrow & (\mathsf{i}N.\mathsf{const}\ \mathrm{extend}^{sx}_{n,|\mathsf{i}N|}(c)) \\
 & & \quad \text{if } \mathrm{bytes}_{\mathsf{i}n}(c) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : n/8]
\end{array}
$$

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

Todo: (*) Rule and prose both not spliced.

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ i$ from the stack.
8. Let ea be the integer $i + memarg.\mathsf{offset}$.
9. If $ea + M \cdot N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:
   a. Trap.
10. Let b^* be the byte sequence $mem.\mathsf{bytes}[ea : M \cdot N/8]$.
11. Let m_k be the integer for which $\mathrm{bytes}_{iM}(m_k) = b^*[k \cdot M/8 : M/8]$.
12. Let W be the integer $M \cdot 2$.
13. Let n_k be the result of computing $\mathrm{extend}^{sx}_{M,W}(m_k)$.
14. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i}W\mathsf{x}N}(n_0 \dots n_{N-1})$.
15. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ x\ memarg) \quad \hookrightarrow \quad S; F; (\mathsf{v128.const}\ c) \\
\qquad (\text{if } ea = i + memarg.\mathsf{offset} \\
\qquad\ \wedge\ ea + M \cdot N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}| \\
\qquad\ \wedge\ \mathrm{bytes}_{iM}(m_k) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}[ea + k \cdot M/8 : M/8]) \\
\qquad\ \wedge\ W = M \cdot 2 \\
\qquad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}W\mathsf{x}N}(\mathrm{extend}^{sx}_{M,W}(m_0) \ldots \mathrm{extend}^{sx}_{M,W}(m_{N-1}))) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}N\_sx\ x\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}
$$

$$
\begin{array}{lcl}
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}K\_sx\ x\ ao) & \hookrightarrow & \mathsf{trap} \qquad \text{if } i + ao.\mathsf{offset} + M \cdot K/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}M\mathsf{x}K\_sx\ x\ ao) & \hookrightarrow & (\mathsf{v128.const}\ c) \\
\multicolumn{3}{l}{\qquad \text{if } (\mathrm{bytes}_{\mathsf{i}M}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} + k \cdot M/8 : M/8])^{k<K}} \\
\multicolumn{3}{l}{\qquad \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}K}(\mathrm{extend}^{sx}_{M,N}(j)^K) \wedge N = M \cdot 2}
\end{array}
$$

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

Todo: (*) Rule and prose both not spliced.

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ i$ from the stack.
8. Let ea be the integer $i + memarg.\mathsf{offset}$.
9. If $ea + N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:

   a. Trap.
10. Let b^* be the byte sequence $mem.\mathsf{bytes}[ea : N/8]$.
11. Let n be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
12. Let L be the integer $128/N$.
13. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}L}(n^L)$.
14. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{splat}\ x\ memarg) \quad \hookrightarrow \quad S; F; (\mathsf{v128.const}\ c) \\
\qquad (\text{if } ea = i + memarg.\mathsf{offset} \\
\qquad\ \wedge\ ea + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}| \\
\qquad\ \wedge\ \mathrm{bytes}_{iN}(n) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}[ea : N/8] \\
\qquad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}L}(n^L)) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{splat}\ x\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\text{otherwise})
\end{array}
$$

$$
\begin{array}{lcl}
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{splat}\ x\ ao) & \hookrightarrow & \mathsf{trap} \qquad \text{if } i + ao.\mathsf{offset} + N/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{splat}\ x\ ao) & \hookrightarrow & (\mathsf{v128.const}\ c) \\
 & & \quad \text{if } \mathrm{bytes}_{\mathsf{i}N}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : N/8] \\
 & & \quad \wedge\ N = |\mathsf{i}N| \\
 & & \quad \wedge\ M = 128/N \\
 & & \quad \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\mathsf{x}M}(j^M)
\end{array}
$$

$\mathsf{v128.load}N\_\mathsf{zero}\ x\ memarg$

Todo: (*) Rule and prose both not spliced.

---

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $\mathsf{i32.const}\ i$ from the stack.
8. Let ea be the integer $i + memarg.\mathsf{offset}$.
9. If $ea + N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:
   a. Trap.
10. Let b^* be the byte sequence $mem.\mathsf{bytes}[ea : N/8]$.
11. Let n be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
12. Let c be the result of computing $\mathrm{extend}^{\mathsf{u}}{}_{N,128}(n)$.
13. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{zero}\ x\ memarg) \quad \hookrightarrow \quad S; F; (\mathsf{v128.const}\ c) \\
\qquad (\mathrm{if}\ ea = i + memarg.\mathsf{offset} \\
\qquad\ \wedge\ ea + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}| \\
\qquad\ \wedge\ \mathrm{bytes}_{iN}(n) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}[ea : N/8]) \\
\qquad\ \wedge\ c = \mathrm{extend}^{\mathsf{u}}{}_{N,128}(n)) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{zero}\ x\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\qquad (\mathrm{otherwise})
\end{array}
$$

$$
\begin{array}{lcll}
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{zero}\ x\ ao) & \hookrightarrow & \mathsf{trap} & \mathrm{if}\ i + ao.\mathsf{offset} + N/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}N\_\mathsf{zero}\ x\ ao) & \hookrightarrow & (\mathsf{v128.const}\ c) & \\
& & \multicolumn{2}{l}{\quad \mathrm{if}\ \mathrm{bytes}_{\mathrm{i}N}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : N/8]} \\
& & \multicolumn{2}{l}{\quad \wedge\ c = \mathrm{extend}^{\mathsf{u}}_{N,128}(j)}
\end{array}
$$

$\mathsf{v128.load}N\_\mathsf{lane}\ x\ memarg\ y$

Todo: (*) Rule and prose both not spliced.

---

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
7. Pop the value $\mathsf{v128.const}\ v$ 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 the integer $i + memarg.\mathsf{offset}$.
11. If $ea + N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:
    a. Trap.
12. Let b^* be the byte sequence $mem.\mathsf{bytes}[ea : N/8]$.
13. Let r be the constant for which $\mathrm{bytes}_{iN}(r) = b^*$.
14. Let L be $128/N$.
15. Let j^* be the result of computing $\mathrm{lanes}_{\mathsf{i}N\times L}(v)$.
16. Let c be the result of computing $\mathrm{lanes}^{-1}_{\mathsf{i}N\times L}(j^*\ \mathsf{with}\ [y] = r)$.
17. Push the value $\mathsf{v128.const}\ c$ to the stack.

$$\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{_lane}\ x\ memarg\ y) & \hookrightarrow & S; F; (\mathsf{v128.const}\ c) \\
\quad (\mathrm{if}\ ea = i + memarg.\mathsf{offset} & & \\
\quad\ \wedge\ ea + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}| & & \\
\quad\ \wedge\ \mathrm{bytes}_{iN}(r) = S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}[ea : N/8]) & & \\
\quad\ \wedge\ L = 128/N & & \\
\quad\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\times L}(\mathrm{lanes}_{\mathsf{i}N\times L}(v)\ \mathsf{with}\ [y] = r)) & & \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ v)\ (\mathsf{v128.load}N\mathsf{_lane}\ x\ memarg\ y) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\mathrm{otherwise}) & &
\end{array}$$

$$\begin{array}{lcl}
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.load}N\mathsf{_lane}\ x\ ao\ j) & \hookrightarrow & \mathsf{trap} \quad \mathrm{if}\ i + ao.\mathsf{offset} + N/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.load}N\mathsf{_lane}\ x\ ao\ j) & \hookrightarrow & (\mathsf{v128.const}\ c) \\
& & \quad \mathrm{if}\ \mathrm{bytes}_{\mathsf{i}N}(k) = z.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : N/8] \\
& & \quad \wedge\ N = |\mathsf{i}N| \\
& & \quad \wedge\ M = |\mathsf{v128}|/N \\
& & \quad \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}N\times M}(\mathrm{lanes}_{\mathsf{i}N\times M}(c_1)[[j] = k])
\end{array}$$

$nt.\mathsf{store}sz^?_{u1}\ x\ ao$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $numty_{u0}$ is on the top of the stack.
3. Pop the value $(numty_{u0}.\mathsf{const}\ c)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If $numty_{u0}$ is nt, then:
    a. If $i + ao.\mathsf{offset} + |nt|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$ and $sz^?_{u1}$ is not defined, then:
        1) Trap.
    b. If $sz^?_{u1}$ is not defined, then:
        1) Let b^* be $\mathrm{bytes}_{nt}(c)$.
        2) Perform $z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : |nt|/8] = b^*]$.
7. If the type of $numty_{u0}$ is Inn, then:
    a. If $sz^?_{u1}$ is defined, then:
        1) Let n be $sz^?_{u1}$.
        2) If $i + ao.\mathsf{offset} + n/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
            a) Trap.

b. Let $\mathsf{i}N$ be $numty_{u0}$.

c. If $sz^?_{u1}$ is defined, then:

1) Let n be $sz^?_{u1}$.

2) Let b^* be $\mathrm{bytes}_{\mathsf{i}n}(\mathrm{wrap}_{|\mathsf{i}N|,n}(c))$.

3) Perform $z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : n/8] = b^*]$.

$$
\begin{array}{rcl}
z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ ao) & \hookrightarrow & z; \mathsf{trap} \\
& & \quad \text{if } i + ao.\mathsf{offset} + |nt|/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ ao) & \hookrightarrow & z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : |nt|/8] = b^*]; \epsilon \\
& & \quad \text{if } b^* = \mathrm{bytes}_{nt}(c) \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i}N.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ ao) & \hookrightarrow & z; \mathsf{trap} \\
& & \quad \text{if } i + ao.\mathsf{offset} + n/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{i}N.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ ao) & \hookrightarrow & z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : n/8] = b^*]; \epsilon \\
& & \quad \text{if } b^* = \mathrm{bytes}_{\mathsf{i}n}(\mathrm{wrap}_{|\mathsf{i}N|,n}(c)) \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}\ x\ ao) & \hookrightarrow & z; \mathsf{trap} \qquad \text{if } i + ao.\mathsf{offset} + |\mathsf{v12} \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}\ x\ ao) & \hookrightarrow & z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : |\mathsf{v128}|/8] = b^*]; \epsilon \qquad \text{if } b^* = \mathrm{bytes}_{\mathsf{v128}}(c)
\end{array}
$$

$\mathsf{v128.store}N\_\mathsf{lane}\ x\ memarg\ y$

Todo: (*) Rule and prose both not spliced.

---

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{mems}[x]$ exists.
3. Let a be the memory address $F.\mathsf{module}.\mathsf{mems}[x]$.
4. Assert: due to validation, $S.\mathsf{mems}[a]$ exists.
5. Let mem be the memory instance $S.\mathsf{mems}[a]$.
6. Assert: due to validation, a value of value type $\mathsf{v128}$ is on the top of the stack.
7. Pop the value $\mathsf{v128.const}\ c$ 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 the integer $i + memarg.\mathsf{offset}$.
11. If $ea + N/8$ is larger than the length of $mem.\mathsf{bytes}$, then:

    a. Trap.
12. Let L be $128/N$.
13. Let j^* be the result of computing $\mathrm{lanes}_{\mathsf{i}N\times L}(c)$.
14. Let b^* be the result of computing $\mathrm{bytes}_{iN}(j^*[y])$.
15. Replace the bytes $mem.\mathsf{bytes}[ea : N/8]$ with b^*.

$$
\begin{array}{lcl}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\_\mathsf{lane}\ x\ memarg\ y) & \hookrightarrow & S'; F; \epsilon \\
\quad (\text{if } ea = i + memarg.\mathsf{offset} \\
\quad \wedge\ ea + N \leq |S.\mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}| \\
\quad \wedge\ L = 128/N \\
\quad \wedge\ S' = S \text{ with } \mathsf{mems}[F.\mathsf{module}.\mathsf{mems}[x]].\mathsf{bytes}[ea : N/8] = \mathrm{bytes}_{iN}(\mathrm{lanes}_{\mathsf{i}N\times L}(c)[y])) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\_\mathsf{lane}\ x\ memarg\ y) & \hookrightarrow & S; F; \mathsf{trap} \\
\quad (\text{otherwise})
\end{array}
$$

$$
\begin{array}{llll}
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\_\mathsf{lane}\ x\ ao\ j) & \hookrightarrow & z; \mathsf{trap} & \text{if } i + ao.\mathsf{offset} \\
z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\_\mathsf{lane}\ x\ ao\ j) & \hookrightarrow & z[.\mathsf{mems}[x].\mathsf{bytes}[i + ao.\mathsf{offset} : N/8] = b^*]; \epsilon & \text{if } N = |\mathsf{i}N| \\
& & & \wedge\ M = 128/N \\
& & & \wedge\ b^* = \mathrm{bytes}_{\mathsf{i}N}
\end{array}
$$

memory.size x

1. Let z be the current state.
2. Let $n \cdot 64\,\mathrm{Ki}$ be $|z.\mathsf{mems}[x].\mathsf{bytes}|$.
3. Push the value $(\mathsf{i32.const}\ n)$ to the stack.

$$z; (\mathsf{memory.size}\ x) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \qquad \text{if } n \cdot 64\,\mathrm{Ki} = |z.\mathsf{mems}[x].\mathsf{bytes}|$$

memory.grow x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Either:
   a. Let mi be $\mathrm{growmem}(z.\mathsf{mems}[x], n)$.
   b. Push the value $(\mathsf{i32.const}\ |z.\mathsf{mems}[x].\mathsf{bytes}|/64\,\mathrm{Ki})$ to the stack.
   c. Perform $z[.\mathsf{mems}[x] = mi]$.
5. Or:
   a. Push the value $(\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))$ to the stack.

Todo:

(6) Maybe we want to be more precise than just saying "Either" in the prose, elaborating that it may succeed or fail non-deterministically.

$$
\begin{array}{lll}
z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z[.\mathsf{mems}[x] = mi]; (\mathsf{i32.const}\ |z.\mathsf{mems}[x].\mathsf{bytes}|/64\,\mathrm{Ki}) \\
& & \quad \text{if } mi = \mathrm{growmem}(z.\mathsf{mems}[x], n) \\
z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))
\end{array}
$$

Note: The memory.grow 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.

---

memory.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.

7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
    a. Trap.
9. If n is 0, then:
    a. Do nothing.
10. Else:
    a. Push the value $(\mathsf{i32.const}\ i)$ to the stack.
    b. Push the value val to the stack.
    c. Execute the instruction $(\mathsf{i32.store8}\ x)$.
    d. Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
    e. Push the value val to the stack.
    f. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
    g. Execute the instruction $(\mathsf{memory.fill}\ x)$.

$$
\begin{array}{lcll}
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i + n > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ i)\ \mathit{val}\ (\mathsf{i32.store8}\ x) & & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ i + 1)\ \mathit{val}\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{memory.fill}\ x) & & &
\end{array}
$$

$\mathsf{memory.copy}\ x_1\ x_2$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i_2)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i_1)$ from the stack.
8. If $i_1 + n$ is greater than $|z.\mathsf{mems}[x_1].\mathsf{bytes}|$, then:
    a. Trap.
9. If $i_2 + n$ is greater than $|z.\mathsf{mems}[x_2].\mathsf{bytes}|$, then:
    a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else:
    a. If i_1 is less than or equal to i_2, then:
        1) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.
        2) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.
        3) Execute the instruction $(\mathsf{i32.load}(8, \mathsf{u})\ x_2)$.
        4) Execute the instruction $(\mathsf{i32.store8}\ x_1)$.
        5) Push the value $(\mathsf{i32.const}\ i_1 + 1)$ to the stack.

6) Push the value $(\mathsf{i32.const}\ i_2 + 1)$ to the stack.

b. Else:

1) Push the value $(\mathsf{i32.const}\ i_1 + n - 1)$ to the stack.

2) Push the value $(\mathsf{i32.const}\ i_2 + n - 1)$ to the stack.

3) Execute the instruction $(\mathsf{i32.load}(8, \mathsf{u})\ x_2)$.

4) Execute the instruction $(\mathsf{i32.store}8\ x_1)$.

5) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.

6) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.

c. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

d. Execute the instruction $(\mathsf{memory.copy}\ x_1\ x_2)$.

$$
\begin{array}{llll}
z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} & \\
\quad \text{if } i_1 + n > |z.\mathsf{mems}[x_1].\mathsf{bytes}| \vee i_2 + n > |z.\mathsf{mems}[x_2].\mathsf{bytes}| & & & \\
z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.load}8\ \mathsf{u}\ x_2)\ (\mathsf{i32.store}8\ x_1) & & & \text{otherwise, if } i_1 \leq i_2 \\
\quad (\mathsf{i32.const}\ i_1 + 1)\ (\mathsf{i32.const}\ i_2 + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{memory.copy}\ x_1\ x_2) & & & \\
z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ i_1 + n - 1)\ (\mathsf{i32.const}\ i_2 + n - 1)\ (\mathsf{i32.load}8\ \mathsf{u}\ x_2)\ (\mathsf{i32.store}8\ x_1) & & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{memory.copy}\ x_1\ x_2) & & &
\end{array}
$$

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

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:

   a. Trap.

9. If $j + n$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:

   a. Trap.

10. If n is 0, then:

    a. Do nothing.

11. Else if i is less than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:

    a. Push the value $(\mathsf{i32.const}\ j)$ to the stack.

    b. Push the value $(\mathsf{i32.const}\ z.\mathsf{datas}[y].\mathsf{bytes}[i])$ to the stack.

    c. Execute the instruction $(\mathsf{i32.store}8\ x)$.

    d. Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.

    e. Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.

    f. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.

g. Execute the instruction $(\mathsf{memory.init}\ x\ y)$.

$$
\begin{array}{lcll}
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & \mathsf{trap} & \\
\qquad \text{if } i + n > |z.\mathsf{datas}[y].\mathsf{bytes}| \vee j + n > |z.\mathsf{mems}[x].\mathsf{bytes}| & & & \\
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & & \\
\quad (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ z.\mathsf{datas}[y].\mathsf{bytes}[i])\ (\mathsf{i32.store8}\ x) & & & \text{otherwise} \\
\quad (\mathsf{i32.const}\ j+1)\ (\mathsf{i32.const}\ i+1)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{memory.init}\ x\ y) & & &
\end{array}
$$

$\mathsf{data.drop}\ x$

1. Let z be the current state.
2. Perform $z[.\mathsf{datas}[x].\mathsf{bytes} = \epsilon]$.

$$
z; (\mathsf{data.drop}\ x) \quad \hookrightarrow \quad z[.\mathsf{datas}[x].\mathsf{bytes} = \epsilon]; \epsilon
$$

4.6.8 Control Instructions

$\mathsf{block}\ bt\ instr^*$

1. Let z be the current state.
2. Let $t_1^m \rightarrow t_2^n$ be $\mathrm{instrtype}_z(bt)$.
3. Assert: Due to validation, there are at least m values on the top of the stack.
4. Pop the values val^m from the stack.
5. Let L be the label whose arity is n and whose continuation is ϵ.
6. Enter $val^m\ instr^*$ with label L.

$$
z; val^m\ (\mathsf{block}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_n\{\epsilon\}\ val^m\ instr^*) \qquad \text{if } \mathrm{instrtype}_z(bt) = t_1^m \rightarrow t_2^n
$$

$\mathsf{loop}\ bt\ instr^*$

1. Let z be the current state.
2. Let $t_1^m \rightarrow t_2^n$ be $\mathrm{instrtype}_z(bt)$.
3. Assert: Due to validation, there are at least m values on the top of the stack.
4. Pop the values val^m from the stack.
5. Let L be the label whose arity is m and whose continuation is $(\mathsf{loop}\ bt\ instr^*)$.
6. Enter $val^m\ instr^*$ with label L.

$$
z; val^m\ (\mathsf{loop}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_m\{\mathsf{loop}\ bt\ instr^*\}\ val^m\ instr^*) \qquad \text{if } \mathrm{instrtype}_z(bt) = t_1^m \rightarrow t_2^n
$$

$\mathsf{if}\ bt\ instr_1^*\ instr_2^*$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ c)$ from the stack.
3. If c is not 0, then:
   a. Execute the instruction $(\mathsf{block}\ bt\ instr_1^*)$.
4. Else:
   a. Execute the instruction $(\mathsf{block}\ bt\ instr_2^*)$.

$$\begin{array}{lcll}(\mathsf{i32.const}\ c)\ (\mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ bt\ instr_1^*) & \text{if}\ c \neq 0 \\ (\mathsf{i32.const}\ c)\ (\mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ bt\ instr_2^*) & \text{if}\ c = 0\end{array}$$

$\mathsf{br}\ l$

1. Pop all values val^* from the top of the stack.
2. Let L be the current label.
3. Let n be the arity of L.
4. Let $instr'^*$ be the continuation of L.
5. Pop the current label from the stack.
6. Let $instr_{u0}^*$ be val^*.
7. If l is 0 and $|instr_{u0}^*|$ is greater than or equal to n, then:
   a. Let $val'^*\ val^n$ be $instr_{u0}^*$.
   b. Push the values val^n to the stack.
   c. Execute the sequence $instr'^*$.
8. If l is greater than 0, then:
   a. Let val^* be $instr_{u0}^*$.
   b. Push the values val^* to the stack.
   c. Execute the instruction $(\mathsf{br}\ l-1)$.

$$\begin{array}{rcll}(\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^n\ instr'^* & \text{if}\ l = 0 \\ (\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l-1) & \text{if}\ l > 0\end{array}$$

$\mathsf{br\_if}\ l$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ c)$ from the stack.
3. If c is not 0, 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'

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 less than $|l^*|$, then:
   a. Execute the instruction (br $l^*[i]$).
4. Else:
   a. Execute the instruction (br l').

$$\begin{array}{lcll}(\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l^*[i]) & \text{if } i < |l^*| \\ (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l') & \text{if } i \geq |l^*|\end{array}$$

br_on_null l

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. If val is of the case ref.null, then:
   a. Execute the instruction (br l).
4. Else:
   a. Push the value val to the stack.

Todo:

(3) Introduce if-let instruction instead of “is of the case”.

$$\begin{array}{lcll}val\ (\mathsf{br\_on\_null}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } val = \mathsf{ref.null}\ ht \\ val\ (\mathsf{br\_on\_null}\ l) & \hookrightarrow & val & \text{otherwise}\end{array}$$

br_on_non_null l

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. If val is of the case ref.null, then:
   a. Do nothing.
4. Else:
   a. Push the value val to the stack.
   b. Execute the instruction (br l).

Todo:

(3) Introduce if-let instruction instead of “is of the case”.

$$\begin{array}{lcll}val\ (\mathsf{br\_on\_non\_null}\ l) & \hookrightarrow & \epsilon & \text{if } val = \mathsf{ref.null}\ ht \\ val\ (\mathsf{br\_on\_non\_null}\ l) & \hookrightarrow & val\ (\mathsf{br}\ l) & \text{otherwise}\end{array}$$

br_on_cast l rt_1 rt_2

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt be $\mathrm{ref}_{type_{of}}(ref)$.
5. If rt does not match $\mathrm{clos}_{f.\mathsf{module}}(rt_2)$, then:
   a. Push the value ref to the stack.
6. Else:
   a. Push the value ref to the stack.
   b. Execute the instruction $(\mathsf{br}\ l)$.

Todo: Below is the acutal prose. (9) Need to handle RulePr s |- ref : rt properly in prose instead of $ref_type_of

1. Let F be the current frame.
2. Let rt'_2 be the reference type $\mathrm{clos}_{F.\mathsf{module}}(rt_2)$.
3. Assert: due to validation, rt'_2 is closed.
4. Assert: due to validation, a reference value is on the top of the stack.
5. Pop the value ref from the stack.
6. Assert: due to validation, the reference value is valid with some reference type.
7. Let rt be the reference type of ref.
8. Push the value ref back to the stack.
9. If the reference type rt matches rt'_2, then:
   a. Execute the instruction $(\mathsf{br}\ l)$.

$$\begin{array}{lcll} s; f; ref\ (\mathsf{br\_on\_cast}\ l\ rt_1\ rt_2) & \hookrightarrow & ref\ (\mathsf{br}\ l) & \text{if } s \vdash ref : rt \\ & & & \wedge \{\} \vdash rt \leq \mathrm{clos}_{f.\mathsf{module}}(rt_2) \\ s; f; ref\ (\mathsf{br\_on\_cast}\ l\ rt_1\ rt_2) & \hookrightarrow & ref & \text{otherwise} \end{array}$$

br_on_cast_fail l rt_1 rt_2

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt be $\mathrm{ref}_{type_{of}}(ref)$.
5. If rt matches $\mathrm{clos}_{f.\mathsf{module}}(rt_2)$, then:
   a. Push the value ref to the stack.
6. Else:
   a. Push the value ref to the stack.
   b. Execute the instruction $(\mathsf{br}\ l)$.

Todo: Below is the actual prose. (9) Need to handle RulePr s |- ref : rt properly in prose instead of $ref_type_of

1. Let F be the current frame.
2. Let rt_2' be the reference type $\mathrm{clos}_{F.\mathsf{module}}(rt_2)$.
3. Assert: due to validation, rt_2' is closed.
4. Assert: due to validation, a reference value is on the top of the stack.
5. Pop the value ref from the stack.
6. Assert: due to validation, the reference value is valid with some reference type.
7. Let rt be the reference type of ref.
8. Push the value ref back to the stack.
9. If the reference type rt does not match rt_2', then:
   a. Execute the instruction $(\mathsf{br}\ l)$.

$$
\begin{array}{lcll}
s; f; ref\ (\mathsf{br\_on\_cast\_fail}\ l\ rt_1\ rt_2) & \hookrightarrow & ref & \text{if } s \vdash ref : rt \\
 & & & \wedge \{\} \vdash rt \leq \mathrm{clos}_{f.\mathsf{module}}(rt_2) \\
s; f; ref\ (\mathsf{br\_on\_cast\_fail}\ l\ rt_1\ rt_2) & \hookrightarrow & ref\ (\mathsf{br}\ l) & \text{otherwise}
\end{array}
$$

return

1. Pop all values val^* from the top of the stack.
2. If a frame is now on the top of the stack, then:
   a. Let F be the current frame.
   b. Let n be the arity of F.
   c. Pop the current frame from the stack.
   d. Let $val'^*\ val^n$ be val^*.
   e. Push the values val^n to the stack.
3. Else if a label is now on the top of the stack, then:
   a. Pop the current label from the stack.
   b. Push the values val^* to the stack.
   c. Execute the instruction return.

$$
\begin{array}{rcl}
(\mathsf{frame}_n\{f\}\ val'^*\ val^n\ \mathsf{return}\ instr^*) & \hookrightarrow & val^n \\
(\mathsf{label}_n\{instr'^*\}\ val^*\ \mathsf{return}\ instr^*) & \hookrightarrow & val^*\ \mathsf{return}
\end{array}
$$

call x

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module}.\mathsf{funcs}|$.
3. Let a be $z.\mathsf{module}.\mathsf{funcs}[x]$.
4. Assert: Due to validation, a is less than $|z.\mathsf{funcs}|$.
5. Push the value $(\mathsf{ref.func}\ a)$ to the stack.
6. Execute the instruction $(\mathsf{call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type})$.

$$
z; (\mathsf{call}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ a)\ (\mathsf{call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type}) \qquad \text{if } z.\mathsf{module}.\mathsf{funcs}[x] = a
$$

call_ref x

Todo: (*) Prose not spliced, for the prose merges the two cases of null and non-null references.

---

1. Assert: due to validation, a null or function reference is on the top of the stack.
2. Pop the reference value r from the stack.
3. If r is ref.null ht, then:
   a. Trap.
4. Assert: due to validation, r is a function reference.
5. Let ref.func a be the reference r.
6. Invoke the function instance at address a.

$$z; (\mathsf{ref.null}\ ht)\ (\mathsf{call\_ref}\ y) \quad \hookrightarrow \quad \mathsf{trap}$$

Note: The formal rule for calling a non-null function reference is described below.

---

call_indirect x y

1. Execute the instruction (table.get x).
2. Execute the instruction (ref.cast (ref (null ()) y)).
3. Execute the instruction (call_ref y).

Todo: Below is the actual prose. (11) ref.cast (ref (null ()) y) is rendered differently. Need to handle TERMINAL? case in AL-to-EL-expr phase.

---

1. Let F be the current frame.
2. Assert: due to validation, F.module.tables[x] exists.
3. Let ta be the table address F.module.tables[x].
4. Assert: due to validation, S.tables[ta] exists.
5. Let tab be the table instance S.tables[ta].
6. Assert: due to validation, F.module.types[y] is defined.
7. Let dt_{expect} be the defined type F.module.types[y].
8. Assert: due to validation, a value with value type i32 is on the top of the stack.
9. Pop the value i32.const i from the stack.
10. If i is not smaller than the length of tab.elem, then:
    a. Trap.
11. Let r be the reference tab.elem[i].
12. If r is ref.null ht, then:
    a. Trap.
13. Assert: due to validation of table mutation, r is a function reference.
14. Let ref.func a be the function reference r.

15. Assert: due to validation of table mutation, $S.\mathsf{funcs}[a]$ exists.
16. Let f be the function instance $S.\mathsf{funcs}[a]$.
17. Let dt_{actual} be the defined type $f.\mathsf{type}$.
18. If dt_{actual} does not match dt_{expect}, then:
    a. Trap.
19. Invoke the function instance at address a.

$$(\mathsf{call\_indirect}\ x\ y) \quad \hookrightarrow \quad (\mathsf{table.get}\ x)\ (\mathsf{ref.cast}\ (\mathsf{ref}\ \mathsf{null}\ y))\ (\mathsf{call\_ref}\ y)$$

return_call x

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module.funcs}|$.
3. Let a be $z.\mathsf{module.funcs}[x]$.
4. Assert: Due to validation, a is less than $|z.\mathsf{funcs}|$.
5. Push the value $(\mathsf{ref.func}\ a)$ to the stack.
6. Execute the instruction $(\mathsf{return\_call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type})$.

$$z; (\mathsf{return\_call}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ a)\ (\mathsf{return\_call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type}) \qquad \text{if } z.\mathsf{module.funcs}[x] = a$$

return_call_ref x

Todo: (*) Prose not spliced, Sphinx cannot build the document with deeply nested ordered list. (mainly caused by spurious conditions that should be assertions)

1. Assert: due to validation, a function reference is on the top of the stack.
2. Pop the reference value r from the stack.
3. If r is $\mathsf{ref.null}\ ht$, then:
    a. Trap.
4. Assert: due to validation, r is a function reference.
5. Let $\mathsf{ref.func}\ a$ be the reference r.
6. Tail-invoke the function instance at address a.

$$\begin{array}{rcl}
z; (\mathsf{label}_k\{instr'^*\}\ val^*\ (\mathsf{return_call_ref}\ y)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{return_call_ref}\ y) \\
z; (\mathsf{frame}_k\{f\}\ val^*\ (\mathsf{ref.null}\ ht)\ (\mathsf{return_call_ref}\ y)\ instr^*) & \hookrightarrow & \mathsf{trap} \\
z; (\mathsf{frame}_k\{f\}\ val'^*\ val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{return_call_ref}\ y)\ instr^*) & \hookrightarrow & val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{call_ref}\ y) \\
 & & \quad \text{if } z.\mathsf{funcs}[a].\mathsf{type} \approx \mathsf{func}\ (t_1^n \to t_2^m)
\end{array}$$

$\mathsf{return\_call\_indirect}\ x\ y$

1. Execute the instruction $(\mathsf{table.get}\ x)$.
2. Execute the instruction $(\mathsf{ref.cast}\ (\mathsf{ref}\ (\mathsf{null}\ ())\ y))$.
3. Execute the instruction $(\mathsf{return\_call\_ref}\ y)$.

Todo: Below is the actual prose. (11) ref.cast (ref (null ()) y) is rendered differently. Need to handle TERMINAL? case in AL-to-EL-expr phase.

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module.tables}[x]$ exists.
3. Let ta be the table address $F.\mathsf{module.tables}[x]$.
4. Assert: due to validation, $S.\mathsf{tables}[ta]$ exists.
5. Let tab be the table instance $S.\mathsf{tables}[ta]$.
6. Assert: due to validation, $F.\mathsf{module.types}[y]$ exists.
7. Let dt_{expect} be the defined type $F.\mathsf{module.types}[y]$.
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 dt_{actual} be the defined type $f.\mathsf{type}$.
16. If dt_{actual} does not match dt_{expect}, then:
    a. Trap.
17. Tail-invoke the function instance at address a.

$$(\mathsf{return\_call\_indirect}\ x\ y) \quad \hookrightarrow \quad (\mathsf{table.get}\ x)\ (\mathsf{ref.cast}\ (\mathsf{ref}\ \mathsf{null}\ y))\ (\mathsf{return\_call\_ref}\ y)$$

4.6.9 Blocks

The following auxiliary rules define the semantics of executing an instruction sequence that forms a block.

Entering $instr^*$ with label L

1. Push L to the stack.
2. Jump to the start of the instruction sequence $instr^*$.

Note: No formal reduction rule is needed for entering an instruction sequence, because the label L is embedded in the administrative instruction that structured control instructions reduce to directly.

Exiting $instr^*$ with label L

When the end of a block is reached without a jump or trap aborting it, then the following steps are performed.

1. Pop all values val^* from the top of the stack.
2. Assert: due to validation, the label L is now on the top of the stack.
3. Pop the label from the stack.
4. Push val^* back to the stack.
5. Jump to the position after the end of the structured control instruction associated with the label L.

$$(\mathsf{label}_n\{instr^*\}\ val^*) \quad \hookrightarrow \quad val^*$$

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

4.6.10 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 reference $(\mathsf{ref.func}\ a)$

1. Assert: due to validation, $S.\mathsf{funcs}[a]$ exists.
2. Let f be the function instance, $S.\mathsf{funcs}[a]$.
3. Let $\mathsf{func}\ [t_1^n] \rightarrow [t_2^m]$ be the composite type $\mathrm{expand}(f.\mathsf{type})$.
4. Let $local^*$ be the list of locals $f.\mathsf{code}.\mathsf{locals}$.
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 F be the frame $\{\mathsf{module}\ f.\mathsf{module}, \mathsf{locals}\ val^n\ (\mathrm{default}_t)^*\}$.
9. Push the activation of F with arity m to the stack.
10. Let L be the label whose arity is m and whose continuation is the end of the function.
11. Enter the instruction sequence $instr^*$ with label L.

$$
\begin{array}{lcl}
z; \mathit{val}^n~(\mathsf{ref.func}~a)~(\mathsf{call\_ref}~y) & \hookrightarrow & (\mathsf{frame}_m\{f\}~(\mathsf{label}_m\{\epsilon\}~\mathit{instr}^*)) \\
 & & \quad \text{if}~z.\mathsf{funcs}[a] = \mathit{fi} \\
 & & \quad \wedge~\mathit{fi}.\mathsf{type} \approx \mathsf{func}~(t_1^n \rightarrow t_2^m) \\
 & & \quad \wedge~\mathit{fi}.\mathsf{code} = \mathsf{func}~x~(\mathsf{local}~t)^*~(\mathit{instr}^*) \\
 & & \quad \wedge~f = \{\mathsf{locals}~\mathit{val}^n~(\mathrm{default}_t)^*,~\mathsf{module}~\mathit{fi}.\mathsf{module}\}
\end{array}
$$

Note: For non-defaultable types, the respective local is left uninitialized by these rules.

Returning from a function

When the end of a function is reached without a jump (i.e., return) or trap aborting it, then the following steps are performed.

1. Let F be the current frame.
2. Let n be the arity of the activation of F.
3. Assert: due to validation, there are n values on the top of the stack.
4. Pop the results val^n from the stack.
5. Assert: due to validation, the frame F is now on the top of the stack.
6. Pop the frame from the stack.
7. Push val^n back to the stack.
8. Jump to the instruction after the original call.

$$
(\mathsf{frame}_n\{f\}~\mathit{val}^n) \quad \hookrightarrow \quad \mathit{val}^n
$$

Host Functions

Invoking a host function has non-deterministic behavior. It may either terminate with a trap or return regularly. However, in the latter case, it must consume and produce the right number and types of WebAssembly values on the stack, according to its function type.

A host function may also modify the store. However, all store modifications must result in an extension of the original store, i.e., they must only modify mutable contents and must not have instances removed. Furthermore, the resulting store must be valid, i.e., all data and code in it is well-typed.

$$
\begin{array}{l}
S; \mathit{val}^n~(\mathsf{invoke}~a) \quad \hookrightarrow \quad S'; \mathit{result} \\
\quad (\text{if}~S.\mathsf{funcs}[a] = \{\mathsf{type}~\mathit{deftype}, \mathsf{hostfunc}~\mathit{hf}\} \\
\quad \wedge~\mathrm{expand}(\mathit{deftype}) = \mathsf{func}~[t_1^n] \rightarrow [t_2^m] \\
\quad \wedge~(S'; \mathit{result}) \in \mathit{hf}(S; \mathit{val}^n)) \\
S; \mathit{val}^n~(\mathsf{invoke}~a) \quad \hookrightarrow \quad S; \mathit{val}^n~(\mathsf{invoke}~a) \\
\quad (\text{if}~S.\mathsf{funcs}[a] = \{\mathsf{type}~\mathit{deftype}, \mathsf{hostfunc}~\mathit{hf}\} \\
\quad \wedge~\mathrm{expand}(\mathit{deftype}) = \mathsf{func}~[t_1^n] \rightarrow [t_2^m] \\
\quad \wedge~\bot \in \mathit{hf}(S; \mathit{val}^n))
\end{array}
$$

Here, $\mathit{hf}(S; \mathit{val}^n)$ denotes the implementation-defined execution of host function hf in current store S with arguments val^n. It yields a set of possible outcomes, where each element is either a pair of a modified store S' and a result or the special value $\bot$ indicating divergence. A host function is non-deterministic if there is at least one argument for which the set of outcomes is not singular.

For a WebAssembly implementation to be sound in the presence of host functions, every host function instance must be valid, which means that it adheres to suitable pre- and post-conditions: under a valid store S, and given arguments val^n matching the ascribed parameter types t_1^n, executing the host function must yield a non-empty set of possible outcomes each of which is either divergence or consists of a valid store S' that is an extension of S and a result matching the ascribed return types t_2^m. All these notions are made precise in the Appendix.

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.

4.6.11 Expressions

An expression is *evaluated* relative to a current frame pointing to its containing module instance.

1. Jump to the start of the instruction sequence $instr^*$ of the expression.
2. Execute the instruction sequence.
3. Assert: due to validation, the top of the stack contains a value.
4. Pop the value val from the stack.

The value val is the result of the evaluation.

$$z; instr^* \quad \hookrightarrow^* \quad z'; val^* \qquad \text{if } z; instr^* \hookrightarrow^* z'; val^*$$

$$S; F; instr^* \hookrightarrow S'; F'; instr'^* \qquad (\text{if } S; F; instr^* \ \mathsf{end} \hookrightarrow S'; F'; instr'^* \ \mathsf{end})$$

Note: Evaluation iterates this reduction rule until reaching a value. Expressions constituting function bodies are executed during function invocation.

4.7 Modules

For modules, the execution semantics primarily defines instantiation, which allocates instances for a module and its contained definitions, initializes tables and memories from contained element and data segments, and invokes the start function if present. It also includes invocation of exported functions.

4.7.1 Allocation

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

Functions

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

$$\begin{array}{rcl} \mathrm{allocfunc}(s, \mathit{deftype}, \mathit{code}, \mathit{moduleinst}) &=& (s \oplus \{\mathsf{funcs}~\mathit{funcinst}\}, |s.\mathsf{funcs}|) \\ && \mathrm{if}~\mathit{funcinst} = \{\mathsf{type}~\mathit{deftype},~\mathsf{module}~\mathit{moduleinst},~\mathsf{code}~\mathit{code}\} \end{array}$$

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

Tables

1. Let $\mathit{tabletype}$ be the table type of the table to allocate and ref the initialization value.
2. Let $(\{\mathsf{min}~n, \mathsf{max}~m^?\}~\mathit{reftype})$ be the structure of table type $\mathit{tabletype}$.
3. Let a be the first free table address in S.
4. Let $\mathit{tableinst}$ be the table instance $\{\mathsf{type}~\mathit{tabletype}', \mathsf{elem}~\mathit{ref}^n\}$ with n elements set to ref.
5. Append $\mathit{tableinst}$ to the tables of S.
6. Return a.

$$\begin{array}{rcl} \mathrm{alloctable}(s, [i\mathrel{..}j]~\mathit{rt}, \mathit{ref}) &=& (s \oplus \{\mathsf{tables}~\mathit{tableinst}\}, |s.\mathsf{tables}|) \\ && \mathrm{if}~\mathit{tableinst} = \{\mathsf{type}~([i\mathrel{..}j]~\mathit{rt}),~\mathsf{elem}~\mathit{ref}^i\} \end{array}$$

Memories

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

$$\begin{array}{rcl} \mathrm{allocmem}(s, [i\mathrel{..}j]~\mathsf{page}) &=& (s \oplus \{\mathsf{mems}~\mathit{meminst}\}, |s.\mathsf{mems}|) \\ && \mathrm{if}~\mathit{meminst} = \{\mathsf{type}~([i\mathrel{..}j]~\mathsf{page}),~\mathsf{bytes}~(\mathtt{0x00})^{i\cdot 64\,\mathrm{Ki}}\} \end{array}$$

Globals

1. Let $\mathit{globaltype}$ be the global type of the global to allocate and val its initialization value.
2. Let a be the first free global address in S.
3. Let $\mathit{globalinst}$ be the global instance $\{\mathsf{type}~\mathit{globaltype}, \mathsf{value}~\mathit{val}\}$.
4. Append $\mathit{globalinst}$ to the globals of S.
5. Return a.

$$\begin{array}{rcl} \mathrm{allocglobal}(s, \mathit{globaltype}, \mathit{val}) &=& (s \oplus \{\mathsf{globals}~\mathit{globalinst}\}, |s.\mathsf{globals}|) \\ && \mathrm{if}~\mathit{globalinst} = \{\mathsf{type}~\mathit{globaltype},~\mathsf{value}~\mathit{val}\} \end{array}$$

Element segments

1. Let $\mathit{reftype}$ be the elements' type and ref^* the list of references to allocate.
2. Let a be the first free element address in S.
3. Let $\mathit{eleminst}$ be the element instance $\{\mathsf{type}~\mathit{reftype}, \mathsf{elem}~\mathit{ref}^*\}$.
4. Append $\mathit{eleminst}$ to the elems of S.
5. Return a.

$$\begin{array}{rcl} \mathrm{allocelem}(s, \mathit{elemtype}, \mathit{ref}^*) & = & (s \oplus \{\mathsf{elems}~\mathit{eleminst}\}, |s.\mathsf{elems}|) \\ & & \text{if } \mathit{eleminst} = \{\mathsf{type}~\mathit{elemtype},~\mathsf{elem}~\mathit{ref}^*\} \end{array}$$

Data segments

1. Let b^* be the list of bytes to allocate.
2. Let a be the first free data address in S.
3. Let $\mathit{datainst}$ be the data instance $\{\mathsf{bytes}~b^*\}$.
4. Append $\mathit{datainst}$ to the datas of S.
5. Return a.

$$\begin{array}{rcl} \mathrm{allocdata}(s, \mathsf{ok}, \mathit{byte}^*) & = & (s \oplus \{\mathsf{datas}~\mathit{datainst}\}, |s.\mathsf{datas}|) \\ & & \text{if } \mathit{datainst} = \{\mathsf{bytes}~\mathit{byte}^*\} \end{array}$$

Growing tables

1. Let $\mathit{tableinst}$ be the table instance to grow, n the number of elements by which to grow it, and ref the initialization value.
2. Let len be n added to the length of $\mathit{tableinst}.\mathsf{elem}$.
3. If len is larger than or equal to 2^{32}, then fail.
4. Let $\mathit{limits}~t$ be the structure of table type $\mathit{tableinst}.\mathsf{type}$.
5. Let limits' be limits with min updated to len.
6. If limits' is not valid, then fail.
7. Append ref^n to $\mathit{tableinst}.\mathsf{elem}$.
8. Set $\mathit{tableinst}.\mathsf{type}$ to the table type $\mathit{limits}'~t$.

$$\begin{array}{rcll} \mathrm{growtable}(\mathit{tableinst}, n, r) & = & \mathit{tableinst}' & \text{if } \mathit{tableinst} = \{\mathsf{type}~([i\,..\,j]~rt),~\mathsf{elem}~r'^*\} \\ & & & \wedge~\mathit{tableinst}' = \{\mathsf{type}~([i'\,..\,j]~rt),~\mathsf{elem}~r'^*~r^n\} \\ & & & \wedge~i' = |r'^*| + n \leq j \end{array}$$

Growing memories

1. Let *meminst* be the memory instance to grow and n the number of pages by which to grow it.
2. Assert: The length of $meminst$.bytes is divisible by the page size $64\,\mathrm{Ki}$.
3. Let *len* be n added to the length of $meminst$.bytes divided by the page size $64\,\mathrm{Ki}$.
4. If *len* is larger than 2^{16}, then fail.
5. Let *limits* be the structure of memory type $meminst$.type.
6. Let $limits'$ be *limits* with min updated to *len*.
7. If $limits'$ is not valid, then fail.
8. Append n times $64\,\mathrm{Ki}$ bytes with value `0x00` to $meminst$.bytes.
9. Set $meminst$.type to the memory type $limits'$.

$$\begin{array}{lcll}
\mathrm{growmem}(meminst, n) & = & meminst' & \text{if } meminst = \{\mathsf{type}~([i\,..\,j]~\mathsf{page}),~\mathsf{bytes}~b^*\} \\
&&& \wedge~meminst' = \{\mathsf{type}~([i'\,..\,j]~\mathsf{page}),~\mathsf{bytes}~b^*~(\mathtt{0x00})^{n\cdot 64\,\mathrm{Ki}}\} \\
&&& \wedge~i' = |b^*|/(64\,\mathrm{Ki}) + n \leq j
\end{array}$$

Modules

Todo: update prose for types

The allocation function for modules requires a suitable list of external values that are assumed to match the import list of the module, a list of initialization values for the module's globals, and list of reference lists for the module's element segments.

1. Let *module* be the module to allocate and $externval^*_{\mathrm{im}}$ the list of external values providing the module's imports, val^*_{g} the initialization values of the module's globals, ref^*_{t} the initializer reference of the module's tables, and $(ref^*_{\mathrm{e}})^*$ the reference lists of the module's element segments.
2. For each defined type $deftype'_i$ in $module$.types, do:
   a. Let $deftype_i$ be the instantiation $deftype'_i$ in *moduleinst* defined below.
3. For each function $func_i$ in $module$.funcs, do:
   a. Let $funcaddr_i$ be the function address resulting from allocating $func_i$ for the module instance *moduleinst* defined below.
4. For each table $table_i$ in $module$.tables, do:
   a. Let $limits_i~t_i$ be the table type obtained by instantiating $table_i$.type in *moduleinst* defined below.
   b. Let $tableaddr_i$ be the table address resulting from allocating $table_i$.type with initialization value $ref^*_{\mathrm{t}}[i]$.
5. For each memory mem_i in $module$.mems, do:
   a. Let $memtype_i$ be the memory type obtained by insantiating mem_i.type in *moduleinst* defined below.
   b. Let $memaddr_i$ be the memory address resulting from allocating $memtype_i$.
6. For each global $global_i$ in $module$.globals, do:
   a. Let $globaltype_i$ be the global type obtained by instantiating $global_i$.type in *moduleinst* defined below.
   b. Let $globaladdr_i$ be the global address resulting from allocating $globaltype_i$ with initializer value $val^*_{\mathrm{g}}[i]$.
7. For each element segment $elem_i$ in $module$.elems, do:
   a. Let $reftype_i$ be the element reference type obtained by *instantiating <type-inst>* $elem_i$.type in *moduleinst* defined below.

b. Let $elemaddr_i$ be the element address resulting from allocating a element instance of reference type $reftype_i$ with contents $(ref^*_{\mathrm{e}})^*[i]$.

8. For each data segment $data_i$ in $module$.datas, do:

   a. Let $dataaddr_i$ be the data address resulting from allocating a data instance with contents $data_i$.init.

9. Let $deftype^*$ be the concatenation of the defined types $deftype_i$ in index order.
10. Let $funcaddr^*$ be the concatenation of the function addresses $funcaddr_i$ in index order.
11. Let $tableaddr^*$ be the concatenation of the table addresses $tableaddr_i$ in index order.
12. Let $memaddr^*$ be the concatenation of the memory addresses $memaddr_i$ in index order.
13. Let $globaladdr^*$ be the concatenation of the global addresses $globaladdr_i$ in index order.
14. Let $elemaddr^*$ be the concatenation of the element addresses $elemaddr_i$ in index order.
15. Let $dataaddr^*$ be the concatenation of the data addresses $dataaddr_i$ in index order.
16. Let $funcaddr^*_{\mathrm{mod}}$ be the list of function addresses extracted from $externval^*_{\mathrm{im}}$, concatenated with $funcaddr^*$.
17. Let $tableaddr^*_{\mathrm{mod}}$ be the list of table addresses extracted from $externval^*_{\mathrm{im}}$, concatenated with $tableaddr^*$.
18. Let $memaddr^*_{\mathrm{mod}}$ be the list of memory addresses extracted from $externval^*_{\mathrm{im}}$, concatenated with $memaddr^*$.
19. Let $globaladdr^*_{\mathrm{mod}}$ be the list of global addresses extracted from $externval^*_{\mathrm{im}}$, concatenated with $globaladdr^*$.
20. For each export $export_i$ in $module$.exports, do:

   a. If $export_i$ is a function export for function index x, then let $externval_i$ be the external value 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 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 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 global $(globaladdr^*_{\mathrm{mod}}[x])$.

   e. Let $exportinst_i$ be the export instance {name $(export_i$.name$)$, value $externval_i$}.

21. Let $exportinst^*$ be the concatenation of the export instances $exportinst_i$ in index order.
22. Let $moduleinst$ be the module instance {types $deftype^*$, funcs $funcaddr^*_{\mathrm{mod}}$, tables $tableaddr^*_{\mathrm{mod}}$, mems $memaddr^*_{\mathrm{mod}}$, globals $globaladdr^*_{\mathrm{mod}}$, exports $exportinst^*$}.
23. Return $moduleinst$.

$$
\begin{array}{l}
\mathrm{allocmodule}(s, \mathit{module}, \mathit{externval}^*, \mathit{val}_{\mathsf{g}}^*, \mathit{ref}_{\mathsf{t}}^*, (\mathit{ref}_{\mathsf{e}}^*)^*) \quad = \quad (s_6, \mathit{moduleinst}) \\
\qquad \text{if } \mathit{module} = \mathsf{module}\ \mathit{type}^*\ \mathit{import}^*\ \mathit{func}^*\ \mathit{global}^*\ \mathit{table}^*\ \mathit{mem}^*\ \mathit{elem}^*\ \mathit{data}^*\ \mathit{start}^?\ \mathit{export}^* \\
\qquad \wedge\ \mathit{func}^* = (\mathsf{func}\ x\ \mathit{local}^*\ \mathit{expr}_{\mathsf{f}})^* \\
\qquad \wedge\ \mathit{global}^* = (\mathsf{global}\ \mathit{globaltype}\ \mathit{expr}_{\mathsf{g}})^* \\
\qquad \wedge\ \mathit{table}^* = (\mathsf{table}\ \mathit{tabletype}\ \mathit{expr}_{\mathsf{t}})^* \\
\qquad \wedge\ \mathit{mem}^* = (\mathsf{memory}\ \mathit{memtype})^* \\
\qquad \wedge\ \mathit{elem}^* = (\mathsf{elem}\ \mathit{elemtype}\ \mathit{expr}_{\mathsf{e}}^*\ \mathit{elemmode})^* \\
\qquad \wedge\ \mathit{data}^* = (\mathsf{data}\ \mathit{byte}^*\ \mathit{datamode})^* \\
\qquad \wedge\ \mathit{fa}_{\mathsf{i}}^* = \mathrm{funcs}(\mathit{externval}^*) \\
\qquad \wedge\ \mathit{ga}_{\mathsf{i}}^* = \mathrm{globals}(\mathit{externval}^*) \\
\qquad \wedge\ \mathit{ta}_{\mathsf{i}}^* = \mathrm{tables}(\mathit{externval}^*) \\
\qquad \wedge\ \mathit{ma}_{\mathsf{i}}^* = \mathrm{mems}(\mathit{externval}^*) \\
\qquad \wedge\ \mathit{fa}^* = (|s.\mathsf{funcs}| + i_{\mathsf{f}})^{i_{\mathsf{f}} < |\mathit{func}^*|} \\
\qquad \wedge\ \mathit{ga}^* = (|s.\mathsf{globals}| + i_{\mathsf{g}})^{i_{\mathsf{g}} < |\mathit{global}^*|} \\
\qquad \wedge\ \mathit{ta}^* = (|s.\mathsf{tables}| + i_{\mathsf{t}})^{i_{\mathsf{t}} < |\mathit{table}^*|} \\
\qquad \wedge\ \mathit{ma}^* = (|s.\mathsf{mems}| + i_{\mathsf{m}})^{i_{\mathsf{m}} < |\mathit{mem}^*|} \\
\qquad \wedge\ \mathit{ea}^* = (|s.\mathsf{elems}| + i_{\mathsf{e}})^{i_{\mathsf{e}} < |\mathit{elem}^*|} \\
\qquad \wedge\ \mathit{da}^* = (|s.\mathsf{datas}| + i_{\mathsf{d}})^{i_{\mathsf{d}} < |\mathit{data}^*|} \\
\qquad \wedge\ \mathit{dt}^* = \mathrm{alloctype}^*(\mathit{type}^*) \\
\qquad \wedge\ (s_1, \mathit{fa}^*) = \mathrm{allocfunc}^*(s, \mathit{dt}^*[x]^*, (\mathsf{func}\ x\ \mathit{local}^*\ \mathit{expr}_{\mathsf{f}})^*, \mathit{moduleinst}^{|\mathit{func}^*|}) \\
\qquad \wedge\ (s_2, \mathit{ga}^*) = \mathrm{allocglobal}^*(s_1, \mathit{globaltype}^*, \mathit{val}_{\mathsf{g}}^*) \\
\qquad \wedge\ (s_3, \mathit{ta}^*) = \mathrm{alloctable}^*(s_2, \mathit{tabletype}^*, \mathit{ref}_{\mathsf{t}}^*) \\
\qquad \wedge\ (s_4, \mathit{ma}^*) = \mathrm{allocmem}^*(s_3, \mathit{memtype}^*) \\
\qquad \wedge\ (s_5, \mathit{ea}^*) = \mathrm{allocelem}^*(s_4, \mathit{elemtype}^*, (\mathit{ref}_{\mathsf{e}}^*)^*) \\
\qquad \wedge\ (s_6, \mathit{da}^*) = \mathrm{allocdata}^*(s_5, \mathsf{ok}^{|\mathit{data}^*|}, (\mathit{byte}^*)^*) \\
\qquad \wedge\ \mathit{xi}^* = \mathrm{allocexport}^*(\{\mathsf{funcs}\ \mathit{fa}_{\mathsf{i}}^*\ \mathit{fa}^*,\ \mathsf{globals}\ \mathit{ga}_{\mathsf{i}}^*\ \mathit{ga}^*,\ \mathsf{tables}\ \mathit{ta}_{\mathsf{i}}^*\ \mathit{ta}^*,\ \mathsf{mems}\ \mathit{ma}_{\mathsf{i}}^*\ \mathit{ma}^*\}, \mathit{export}^*) \\
\qquad \wedge\ \mathit{moduleinst} = \{\mathsf{types}\ \mathit{dt}^*, \\
\qquad\qquad\qquad\qquad \mathsf{funcs}\ \mathit{fa}_{\mathsf{i}}^*\ \mathit{fa}^*,\ \mathsf{globals}\ \mathit{ga}_{\mathsf{i}}^*\ \mathit{ga}^*, \\
\qquad\qquad\qquad\qquad \mathsf{tables}\ \mathit{ta}_{\mathsf{i}}^*\ \mathit{ta}^*,\ \mathsf{mems}\ \mathit{ma}_{\mathsf{i}}^*\ \mathit{ma}^*, \\
\qquad\qquad\qquad\qquad \mathsf{elems}\ \mathit{ea}^*,\ \mathsf{datas}\ \mathit{da}^*, \\
\qquad\qquad\qquad\qquad \mathsf{exports}\ \mathit{xi}^*\}
\end{array}
$$

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

$$
\begin{array}{lcll}
\mathrm{allocX}^*(s, \epsilon, \epsilon) & = & (s, \epsilon) & \\
\mathrm{allocX}^*(s, X\ X'^*, Y\ Y'^*) & = & (s_2, a\ a'^*) & \text{if } (s_1, a) = \mathrm{allocX}(X, Y, s, X, Y) \\
& & & \wedge\ (s_2, a'^*) = \mathrm{allocX}^*(s_1, X'^*, Y'^*)
\end{array}
$$

For types, however, allocation is defined in terms of rolling and substitution of all preceding types to produce a list of closed defined types:

$$
\begin{array}{lcll}
\mathrm{alloctype}^*(\epsilon) & = & \epsilon & \\
\mathrm{alloctype}^*(\mathit{type}'^*\ \mathit{type}) & = & \mathit{deftype}'^*\ \mathit{deftype}^* & \text{if } \mathit{deftype}'^* = \mathrm{alloctype}^*(\mathit{type}'^*) \\
& & & \wedge\ \mathit{type} = \mathsf{type}\ \mathit{rectype} \\
& & & \wedge\ \mathit{deftype}^* = \mathrm{roll}_x^*(\mathit{rectype})[:= \mathit{deftype}'^*] \\
& & & \wedge\ x = |\mathit{deftype}'^*|
\end{array}
$$

Finally, export instances are produced with the help of the following definition:

$$
\begin{array}{lcl}
\mathrm{allocexport}^*(\mathit{moduleinst}, \mathit{export}^*) & = & \mathrm{allocexport}(\mathit{moduleinst}, \mathit{export})^* \\
\mathrm{allocexport}(\mathit{moduleinst}, \mathsf{export}\ \mathit{name}\ (\mathsf{func}\ x)) & = & \{\mathsf{name}\ \mathit{name},\ \mathsf{value}\ (\mathsf{func}\ \mathit{moduleinst}.\mathsf{funcs}[x])\} \\
\mathrm{allocexport}(\mathit{moduleinst}, \mathsf{export}\ \mathit{name}\ (\mathsf{global}\ x)) & = & \{\mathsf{name}\ \mathit{name},\ \mathsf{value}\ (\mathsf{global}\ \mathit{moduleinst}.\mathsf{globals}[x])\} \\
\mathrm{allocexport}(\mathit{moduleinst}, \mathsf{export}\ \mathit{name}\ (\mathsf{table}\ x)) & = & \{\mathsf{name}\ \mathit{name},\ \mathsf{value}\ (\mathsf{table}\ \mathit{moduleinst}.\mathsf{tables}[x])\} \\
\mathrm{allocexport}(\mathit{moduleinst}, \mathsf{export}\ \mathit{name}\ (\mathsf{memory}\ x)) & = & \{\mathsf{name}\ \mathit{name},\ \mathsf{value}\ (\mathsf{mem}\ \mathit{moduleinst}.\mathsf{mems}[x])\}
\end{array}
$$

Note: The definition of module allocation is mutually recursive with the allocation of its associated functions, because the resulting module instance is passed to the allocators 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.

4.7.2 Instantiation

Given a store s, a $module$ is instantiated with a list of external values $externval^*$ supplying the required imports as follows.

Instantiation checks that the module is valid and the provided imports match the declared types, and may *fail* with an error otherwise. Instantiation can also result in a trap from initializing a table or memory from an active segment or from executing the start function. It is up to the embedder to define how such conditions are reported.

1. If $module$ is not valid, then:
   a. Fail.
2. Assert: $module$ is valid with external types $externtype_{\mathrm{im}}^m$ 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 $externval_i$ in $externval^n$ and external type $externtype'_i$ in $externtype_{\mathrm{im}}^n$, do:
   a. If $externval_i$ is not valid with an external type $externtype_i$ in store S, then:
      i. Fail.
   b. Let $externtype''_i$ be the external type obtained by instantiating $externtype'_i$ in $moduleinst$ defined below.
   c. If $externtype_i$ does not match $externtype''_i$, then:
      i. Fail.
6. Let F be the auxiliary frame $\{\mathsf{module}\ moduleinst, \mathsf{locals}\ \epsilon\}$, that consists of the final module instance $moduleinst$, defined below.
7. Push the frame F to the stack.
8. Let val_{g}^* be the list of global initialization values determined by $module$ and $externval^n$. These may be calculated as follows.
   a. For each global $global_i$ in $module.\mathsf{globals}$, do:
      i. Let $val_{\mathrm{g}i}$ be the result of evaluating the initializer expression $global_i.\mathsf{init}$.
   b. Assert: due to validation, the frame F is now on the top of the stack.
   c. Let val_{g}^* be the concatenation of $val_{\mathrm{g}i}$ in index order.
9. Let ref_{t}^* be the list of table initialization references determined by $module$ and $externval^n$. These may be calculated as follows.
   a. For each table $table_i$ in $module.\mathsf{tables}$, do:
      i. Let $val_{\mathrm{t}i}$ be the result of evaluating the initializer expression $table_i.\mathsf{init}$.
      ii. Assert: due to validation, $val_{\mathrm{t}i}$ is a reference.
      iii. Let $ref_{\mathrm{t}i}$ be the reference $val_{\mathrm{t}i}$.
   b. Assert: due to validation, the frame F is now on the top of the stack.
   c. Let ref_{t}^* be the concatenation of $ref_{\mathrm{t}i}$ in index order.
10. Let $(ref_{\mathrm{e}}^*)^*$ be the list of reference lists determined by the element segments in $module$. These may be calculated as follows.
    a. For each element segment $elem_i$ in $module.\mathsf{elems}$, and for each element expression $expr_{ij}$ in $elem_i.\mathsf{init}$, do:
       i. Let ref_{ij} be the result of evaluating the initializer expression $expr_{ij}$.
    b. Let ref_i^* be the concatenation of function elements ref_{ij} in order of index j.
    c. Let $(ref_{\mathrm{e}}^*)^*$ be the concatenation of function element lists ref_i^* in order of index i.

11. Let $moduleinst$ be a new module instance allocated from $module$ in store S with imports $externval^n$, global initializer values val_{g}^*, table initializer values ref_{t}^*, and element segment contents $(ref_{\mathsf{e}}^*)^*$, and let S' be the extended store produced by module allocation.

12. For each element segment $elem_i$ in $module.\mathsf{elems}$ whose mode is of the form $\mathsf{active}\ \{\mathsf{table}\ tableidx_i, \mathsf{offset}\ einstr_i^*\ \mathsf{end}\}$, do:

    a. Let n be the length of the list $elem_i.\mathsf{init}$.

    b. Execute the instruction sequence $einstr_i^*$.

    c. Execute the instruction $\mathsf{i32.const}\ 0$.

    d. Execute the instruction $\mathsf{i32.const}\ n$.

    e. Execute the instruction $\mathsf{table.init}\ tableidx_i\ i$.

    f. Execute the instruction $\mathsf{elem.drop}\ i$.

13. For each element segment $elem_i$ in $module.\mathsf{elems}$ whose mode is of the form $\mathsf{declare}$, do:

    a. Execute the instruction $\mathsf{elem.drop}\ i$.

14. For each data segment $data_i$ in $module.\mathsf{datas}$ whose mode is of the form $\mathsf{active}\ \{\mathsf{memory}\ memidx_i, \mathsf{offset}\ dinstr_i^*\ \mathsf{end}\}$, do:

    a. Assert: $memidx_i$ is 0.

    b. Let n be the length of the list $data_i.\mathsf{init}$.

    c. Execute the instruction sequence $dinstr_i^*$.

    d. Execute the instruction $\mathsf{i32.const}\ 0$.

    e. Execute the instruction $\mathsf{i32.const}\ n$.

    f. Execute the instruction $\mathsf{memory.init}\ i$.

    g. Execute the instruction $\mathsf{data.drop}\ i$.

15. If the start function $module.\mathsf{start}$ is not empty, then:

    a. Let $start$ be the start function $module.\mathsf{start}$.

    b. Execute the instruction $\mathsf{call}\ start.\mathsf{func}$.

16. Assert: due to validation, the frame F is now on the top of the stack.

17. Pop the frame F from the stack.

$$
\begin{array}{l}
\mathrm{instantiate}(s, module, externval^*) \quad = \quad s'; f; instr_{\mathsf{e}}^*\ instr_{\mathsf{d}}^*\ instr_{\mathsf{s}}^? \\
\qquad \text{if } module = \mathsf{module}\ type^*\ import^*\ func^*\ global^*\ table^*\ mem^*\ elem^*\ data^*\ start^?\ export^* \\
\qquad \wedge\ global^* = (\mathsf{global}\ globaltype\ expr_{\mathsf{g}})^* \\
\qquad \wedge\ table^* = (\mathsf{table}\ tabletype\ expr_{\mathsf{t}})^* \\
\qquad \wedge\ elem^* = (\mathsf{elem}\ reftype\ expr_{\mathsf{e}}^*\ elemmode)^* \\
\qquad \wedge\ data^* = (\mathsf{data}\ byte^*\ datamode)^* \\
\qquad \wedge\ start^? = (\mathsf{start}\ x)^? \\
\qquad \wedge\ moduleinst_0 = \{\mathsf{types}\ \mathrm{alloctype}^*(type^*), \\
\qquad\qquad\qquad\qquad \mathsf{funcs}\ \mathrm{funcs}(externval^*)\ (|s.\mathsf{funcs}| + i_{\mathsf{f}})^{i_{\mathsf{f}}<|func^*|}, \\
\qquad\qquad\qquad\qquad \mathsf{globals}\ \mathrm{globals}(externval^*)\} \\
\qquad \wedge\ z = s; \{\mathsf{module}\ moduleinst_0\} \\
\qquad \wedge\ (z; expr_{\mathsf{g}} \hookrightarrow^* z; val_{\mathsf{g}})^* \\
\qquad \wedge\ (z; expr_{\mathsf{t}} \hookrightarrow^* z; ref_{\mathsf{t}})^* \\
\qquad \wedge\ (z; expr_{\mathsf{e}} \hookrightarrow^* z; ref_{\mathsf{e}})^{**} \\
\qquad \wedge\ (s', moduleinst) = \mathrm{allocmodule}(s, module, externval^*, val_{\mathsf{g}}^*, ref_{\mathsf{t}}^*, (ref_{\mathsf{e}}^*)^*) \\
\qquad \wedge\ f = \{\mathsf{module}\ moduleinst\} \\
\qquad \wedge\ instr_{\mathsf{e}}^* = \bigoplus \mathrm{runelem}_{i_{\mathsf{e}}}(elem^*[i_{\mathsf{e}}])^{i_{\mathsf{e}}<|elem^*|} \\
\qquad \wedge\ instr_{\mathsf{d}}^* = \bigoplus \mathrm{rundata}_{i_{\mathsf{d}}}(data^*[i_{\mathsf{d}}])^{i_{\mathsf{d}}<|data^*|} \\
\qquad \wedge\ instr_{\mathsf{s}}^? = (\mathsf{call}\ x)^?
\end{array}
$$

where:

$$
\begin{array}{lcl}
\mathrm{runelem}_x(\mathsf{elem}\ rt\ e^n\ (\mathsf{passive})) & = & \epsilon \\
\mathrm{runelem}_x(\mathsf{elem}\ rt\ e^n\ (\mathsf{declare})) & = & (\mathsf{elem.drop}\ x) \\
\mathrm{runelem}_x(\mathsf{elem}\ rt\ e^n\ (\mathsf{active}\ y\ instr^*)) & = & instr^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ y\ x)\ (\mathsf{elem.drop}\ x) \\
\mathrm{rundata}_x(\mathsf{data}\ b^n\ (\mathsf{passive})) & = & \epsilon \\
\mathrm{rundata}_x(\mathsf{data}\ b^n\ (\mathsf{active}\ y\ instr^*)) & = & instr^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ y\ x)\ (\mathsf{data.drop}\ x)
\end{array}
$$

Note: Checking import types assumes that the module instance has already been allocated to compute the respective closed defined types. However, this forward reference merely is a way to simplify the specification. In practice, implementations will likely allocate or canonicalize types beforehand, when *compiling* a module, in a stage before instantiation and before imports are checked.

Similarly, module allocation and the evaluation of global and table initializers as well as element segments are mutually recursive because the global initialization values val_{g}^*, ref_{t}, and element segment contents ref_{e}^{**} are passed to the module allocator while depending on the module instance $moduleinst$ and store s' returned by allocation. Again, this recursion is just a specification device. In practice, the initialization values can be determined beforehand by staging module allocation such that first, the module's own function instances are pre-allocated in the store, then the initializer expressions are evaluated in order, allocating globals on the way, then the rest of the module instance is allocated, and finally the new function instances' module fields are set to that module instance. This is possible because validation ensures that initialization expressions cannot actually call a function, only take their reference.

All failure conditions are checked before any observable mutation of the store takes place. Store mutation is not atomic; it happens in individual steps that may be interleaved with other threads.

Evaluation of constant expressions does not affect the store.

4.7.3 Invocation

Once a module has been instantiated, any exported function can be *invoked* externally via its function address $funcaddr$ in the store s and an appropriate list val^* of argument values.

Invocation may *fail* with an error if the arguments do not fit the function type. Invocation can also result in a trap. It is up to the embedder to define how such conditions are reported.

Note: If the embedder API performs type checks itself, either statically or dynamically, before performing an invocation, then no failure other than traps can occur.

The following steps are performed:

1. Assert: $S.\mathsf{funcs}[funcaddr]$ exists.
2. Let $funcinst$ be the function instance $S.\mathsf{funcs}[funcaddr]$.
3. Let $\mathsf{func}\ [t_1^n] \to [t_2^m]$ be the composite type $\mathrm{expand}(funcinst.\mathsf{type})$.
4. If the length $|val^*|$ of the provided argument values is different from the number n of expected arguments, then:
   a. Fail.
5. For each value type t_i in t_1^n and corresponding value val_i in val^*, do:
   a. If val_i is not valid with value type t_i, then:
      i. Fail.
6. Let F be the dummy frame $\{\mathsf{module}\ \{\}, \mathsf{locals}\ \epsilon\}$.

7. Push the frame F to the stack.
8. Push the values val^* to the stack.
9. Invoke the function instance at address $funcaddr$.

Once the function has returned, the following steps are executed:

1. Assert: due to validation, m values are on the top of the stack.
2. Pop val_{res}^m from the stack.
3. Assert: due to validation, the frame F is now on the top of the stack.
4. Pop the frame F from the stack.

The values val_{res}^m are returned as the results of the invocation.

$$\begin{array}{lcl} \mathrm{invoke}(s, funcaddr, val^*) &=& s; f; val^*\ (\mathsf{ref.func}\ funcaddr)\ (\mathsf{call\_ref}\ s.\mathsf{funcs}[funcaddr].\mathsf{type}) \\ \qquad \text{if } s.\mathsf{funcs}[funcaddr].\mathsf{type} \approx \mathsf{func}\ (t_1^* \to t_2^*) \\ \qquad \wedge\ f = \{\mathsf{module}\ \{\}\} \end{array}$$

CHAPTER 5

Binary Format

5.1 Conventions

The binary format for WebAssembly modules is a dense linear *encoding* of their abstract syntax.[28]

The format is defined by an *attribute grammar* whose only terminal symbols are bytes. A byte sequence is a well-formed encoding of a module if and only if it is generated by the grammar.

Each production of this grammar has exactly one synthesized attribute: the abstract syntax that the respective byte sequence encodes. Thus, the attribute grammar implicitly defines a *decoding* function (i.e., a parsing function for the binary format).

Except for a few exceptions, the binary grammar closely mirrors the grammar of the abstract syntax.

Note: Some phrases of abstract syntax have multiple possible encodings in the binary format. For example, numbers may be encoded as if they had optional leading zeros. Implementations of decoders must support all possible alternatives; implementations of encoders can pick any allowed encoding.

The recommended extension for files containing WebAssembly modules in binary format is "`.wasm`" and the recommended Media Type[27] is "`application/wasm`".

5.1.1 Grammar

The following conventions are adopted in defining grammar rules for the binary format. They mirror the conventions used for abstract syntax. In order to distinguish symbols of the binary syntax from symbols of the abstract syntax, `typewriter` font is adopted for the former.

- Terminal symbols are bytes expressed in hexadecimal notation: `0x0F`.
- Nonterminal symbols are written in typewriter font: `valtype`, `instr`.
- B^n is a sequence of $n \geq 0$ iterations of B.
- B^* is a possibly empty sequence of iterations of B. (This is a shorthand for B^n used where n is not relevant.)

[28] 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.

[27] https://www.iana.org/assignments/media-types/media-types.xhtml

- $B^?$ is an optional occurrence of B. (This is a shorthand for B^n where $n \leq 1$.)
- $x{:}B$ denotes the same language as the nonterminal B, but also binds the variable x to the attribute synthesized for B. A pattern may also be used instead of a variable, e.g., $7{:}B$.
- Productions are written $\mathtt{sym} ::= B_1 \Rightarrow A_1 \mid \dots \mid B_n \Rightarrow A_n$, where each A_i is the attribute that is synthesized for $\mathtt{sym}$ in the given case, usually from attribute variables bound in B_i.
- Some productions are augmented by side conditions in parentheses, which restrict the applicability of the production. They provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production (in the syntax or in an attribute), then all those occurrences must have the same instantiation. (This is a shorthand for a side condition requiring multiple different variables to be equal.)

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

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

Consequently, the byte 0x7F encodes the type $\mathsf{i32}$, 0x7E encodes the type $\mathsf{i64}$, and so forth. No other byte value is allowed as the encoding of a number type.

The binary grammar for limits is defined as follows:

$$\begin{array}{llll}
\mathtt{limits} & ::= & \mathtt{0x00}~~n{:}\mathtt{u32} & \Rightarrow & [n \mathrel{..} 2^{32}-1] \\
& | & \mathtt{0x01}~~n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & [n \mathrel{..} m]
\end{array}$$

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.
- $||\mathtt{B}||$ is the length of the byte sequence generated from the production B in a derivation.

5.1.3 Lists

Lists are encoded with their u32 length followed by the encoding of their element sequence.

$$\mathtt{list}(\mathtt{X}) \quad ::= \quad n{:}\mathtt{u32}~~(el{:}\mathtt{X})^n \quad \Rightarrow \quad el^n$$

5.2 Values

5.2.1 Bytes

Bytes encode themselves.

$$\mathtt{byte} \quad ::= \quad b{:}\mathtt{0x00} \mid \ldots \mid b{:}\mathtt{0xFF} \quad \Rightarrow \quad b$$

5.2.2 Integers

All integers are encoded using the LEB128[29] variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in unsigned LEB128[30] format. As an additional constraint, the total number of bytes encoding a uN value must not exceed $\mathrm{ceil}(N/7)$ bytes.

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

Signed integers are encoded in signed LEB128[31] format, which uses a two's complement representation. As an additional constraint, the total number of bytes encoding an sN value must not exceed $\mathrm{ceil}(N/7)$ bytes.

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

Uninterpreted integers are encoded as signed integers.

$$\mathtt{i}N \quad ::= \quad i{:}\mathtt{s}N \quad \Rightarrow \quad \mathrm{signed}_N^{-1}(i)$$

Note: The side conditions $N > 7$ in the productions for non-terminal bytes of the uN and sN 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 an $s16$.

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[32] (Section 3.4) bit pattern in little endian[33] byte order:

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

[29] https://en.wikipedia.org/wiki/LEB128
[30] https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128
[31] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128
[32] https://ieeexplore.ieee.org/document/8766229
[33] https://en.wikipedia.org/wiki/Endianness#Little-endian

5.2.4 Names

Names are encoded as a list of bytes containing the Unicode[34] (Section 3.9) UTF-8 encoding of the name's character sequence.

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

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

$$\begin{array}{lcll}
\mathrm{utf8}(ch^*) &=& \bigoplus \mathrm{utf8}(ch)^* & \\
\mathrm{utf8}(ch) &=& b & \text{if } ch < \mathrm{U{+}80} \\
&&& \wedge\ ch = b \\
\mathrm{utf8}(ch) &=& b_1\ b_2 & \text{if } \mathrm{U{+}80} \leq ch < \mathrm{U{+}0800} \\
&&& \wedge\ ch = 2^6 \cdot (b_1 - \mathtt{0xC0}) + \mathrm{cont}(b_2) \\
\mathrm{utf8}(ch) &=& b_1\ b_2\ b_3 & \text{if } \mathrm{U{+}0800} \leq ch < \mathrm{U{+}D800} \vee \mathrm{U{+}E000} \leq ch < \mathrm{U{+}10000} \\
&&& \wedge\ ch = 2^{12} \cdot (b_1 - \mathtt{0xE0}) + 2^6 \cdot \mathrm{cont}(b_2) + \mathrm{cont}(b_3) \\
\mathrm{utf8}(ch) &=& b_1\ b_2\ b_3\ b_4 & \text{if } \mathrm{U{+}10000} \leq ch < \mathrm{U{+}11000} \\
&&& \wedge\ ch = 2^{18} \cdot (b_1 - \mathtt{0xF0}) + 2^{12} \cdot \mathrm{cont}(b_2) + 2^6 \cdot \mathrm{cont}(b_3) + \mathrm{cont}(b_4)
\end{array}$$

where $\mathrm{cont}(b) = b - \mathtt{0x80} \qquad \text{if } (\mathtt{0x80} < b < \mathtt{0xC0})$

Note: Unlike in some other formats, name strings are not 0-terminated.

5.3 Types

Note: In some places, possible types include both type constructors or types denoted by type indices. Thus, the binary format for type constructors corresponds to the encodings of small negative sN values, such that they can unambiguously occur in the same place as (positive) type indices.

5.3.1 Number Types

Number types are encoded by a single byte.

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

5.3.2 Vector Types

Vector types are also encoded by a single byte.

$$\mathtt{vectype} \ ::= \ \mathtt{0x7B} \ \Rightarrow \ \mathsf{v128}$$

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

5.3.3 Heap Types

Heap types are encoded as either a single byte, or as a type index encoded as a positive signed integer.

$$
\begin{array}{llclll}
\mathtt{absheaptype} & ::= & \mathtt{0x6A} & \Rightarrow & \mathsf{array} \\
& | & \mathtt{0x6B} & \Rightarrow & \mathsf{struct} \\
& | & \mathtt{0x6C} & \Rightarrow & \mathsf{i31} \\
& | & \mathtt{0x6D} & \Rightarrow & \mathsf{eq} \\
& | & \mathtt{0x6E} & \Rightarrow & \mathsf{any} \\
& | & \mathtt{0x6F} & \Rightarrow & \mathsf{extern} \\
& | & \mathtt{0x70} & \Rightarrow & \mathsf{func} \\
& | & \mathtt{0x71} & \Rightarrow & \mathsf{none} \\
& | & \mathtt{0x72} & \Rightarrow & \mathsf{noextern} \\
& | & \mathtt{0x73} & \Rightarrow & \mathsf{nofunc} \\
\mathtt{heaptype} & ::= & ht{:}\mathtt{absheaptype} & \Rightarrow & ht \\
& | & x{:}\mathtt{s33} & \Rightarrow & x & \text{if } x \geq 0
\end{array}
$$

Note: The heap type bot cannot occur in a module.

5.3.4 Reference Types

Reference types are either encoded by a single byte followed by a heap type, or, as a short form, directly as an abstract heap type.

$$
\begin{array}{llcll}
\mathtt{reftype} & ::= & \mathtt{0x63}\ ht{:}\mathtt{heaptype} & \Rightarrow & \mathsf{ref}\ \mathsf{null}\ ht \\
& | & \mathtt{0x64}\ ht{:}\mathtt{heaptype} & \Rightarrow & \mathsf{ref}\ ht \\
& | & ht{:}\mathtt{absheaptype} & \Rightarrow & \mathsf{ref}\ \mathsf{null}\ ht
\end{array}
$$

5.3.5 Value Types

Value types are encoded with their respective encoding as a number type, vector type, or reference type.

$$
\begin{array}{llcll}
\mathtt{valtype} & ::= & nt{:}\mathtt{numtype} & \Rightarrow & nt \\
& | & vt{:}\mathtt{vectype} & \Rightarrow & vt \\
& | & rt{:}\mathtt{reftype} & \Rightarrow & rt
\end{array}
$$

Note: The value type bot cannot occur in a module.

Value types can occur in contexts where type indices are also allowed, such as in the case of block types. Thus, the binary format for types corresponds to the signed LEB128[35] encoding of small negative sN values, so that they can coexist with (positive) type indices in the future.

5.3.6 Result Types

Result types are encoded by the respective lists of value types.

$$
\mathtt{resulttype} \ ::= \ t^*{:}\mathtt{list}(\mathtt{valtype}) \ \Rightarrow \ t^*
$$

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

5.3.7 Composite Types

Composite types are encoded by a distinct byte followed by a type encoding of the respective form.

$$
\begin{array}{llclll}
\mathtt{mut} & ::= & \mathtt{0x00} & & \Rightarrow & \epsilon \\
& | & \mathtt{0x01} & & \Rightarrow & \mathsf{mut} \\
\mathtt{comptype} & ::= & \mathtt{0x5E} & yt{:}\mathtt{fieldtype} & \Rightarrow & \mathsf{array}~yt \\
& | & \mathtt{0x5F} & yt^*{:}\mathtt{list}(\mathtt{fieldtype}) & \Rightarrow & \mathsf{struct}~yt^* \\
& | & \mathtt{0x60} & t_1^*{:}\mathtt{resulttype}~t_2^*{:}\mathtt{resulttype} & \Rightarrow & \mathsf{func}~(t_1^* \rightarrow t_2^*) \\
\mathtt{fieldtype} & ::= & zt{:}\mathtt{storagetype}~\mathtt{mut}^?{:}\mathtt{mut} & & \Rightarrow & \mathsf{mut}^?~zt \\
\mathtt{storagetype} & ::= & t{:}\mathtt{valtype} & & \Rightarrow & t \\
& | & pt{:}\mathtt{packtype} & & \Rightarrow & pt \\
\mathtt{packtype} & ::= & \mathtt{0x77} & & \Rightarrow & \mathsf{i16} \\
& | & \mathtt{0x78} & & \Rightarrow & \mathsf{i8}
\end{array}
$$

5.3.8 Recursive Types

Recursive types are encoded by the byte 0x4E followed by a list of sub types. Additional shorthands are recognized for unary recursions and sub types without super types.

$$
\begin{array}{llclll}
\mathtt{rectype} & ::= & \mathtt{0x4E}~st^*{:}\mathtt{list}(\mathtt{subtype}) & & \Rightarrow & \mathsf{rec}~st^* \\
& | & st{:}\mathtt{subtype} & & \Rightarrow & \mathsf{rec}~st \\
\mathtt{subtype} & ::= & \mathtt{0x4F}~x^*{:}\mathtt{list}(\mathtt{typeidx}) & ct{:}\mathtt{comptype} & \Rightarrow & \mathsf{sub}~\mathsf{final}~x^*~ct \\
& | & \mathtt{0x50}~x^*{:}\mathtt{list}(\mathtt{typeidx}) & ct{:}\mathtt{comptype} & \Rightarrow & \mathsf{sub}~x^*~ct \\
& | & ct{:}\mathtt{comptype} & & \Rightarrow & \mathsf{sub}~\mathsf{final}~\epsilon~ct
\end{array}
$$

5.3.9 Limits

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

$$
\begin{array}{llclll}
\mathtt{limits} & ::= & \mathtt{0x00}~~n{:}\mathtt{u32} & \Rightarrow & [n \mathrel{..} 2^{32}-1] \\
& | & \mathtt{0x01}~~n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & [n \mathrel{..} m]
\end{array}
$$

5.3.10 Memory Types

Memory types are encoded with their limits.

$$
\begin{array}{llclll}
\mathtt{memtype} & ::= & lim{:}\mathtt{limits} & \Rightarrow & lim~\mathsf{page}
\end{array}
$$

5.3.11 Table Types

Table types are encoded with their limits and the encoding of their element reference type.

$$
\begin{array}{llclll}
\mathtt{tabletype} & ::= & rt{:}\mathtt{reftype}~~lim{:}\mathtt{limits} & \Rightarrow & lim~rt
\end{array}
$$

5.3.12 Global Types

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

$$\mathtt{globaltype} \quad ::= \quad t{:}\mathtt{valtype}\ \ \mathsf{mut}^?{:}\mathtt{mut} \quad \Rightarrow \quad \mathsf{mut}^?\ t$$

5.3.13 External Types

External types are encoded by a distiguishing byte followed by an encoding of the respective form of type.

$$\begin{array}{llcll}
\mathtt{externtype} & ::= & \mathtt{0x00}\ \ x{:}\mathtt{typeidx} & \Rightarrow & \mathsf{func}\ x \\
& | & \mathtt{0x01}\ \ tt{:}\mathtt{tabletype} & \Rightarrow & \mathsf{table}\ tt \\
& | & \mathtt{0x02}\ \ mt{:}\mathtt{memtype} & \Rightarrow & \mathsf{mem}\ mt \\
& | & \mathtt{0x03}\ \ gt{:}\mathtt{globaltype} & \Rightarrow & \mathsf{global}\ gt
\end{array}$$

5.4 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.

$$\begin{array}{llclll}
\mathtt{blocktype} & ::= & \mathtt{0x40} & \Rightarrow & \epsilon & \\
& | & t{:}\mathtt{valtype} & \Rightarrow & t & \\
& | & i{:}\mathtt{s33} & \Rightarrow & i & \text{if } i \geq 0 \\
\mathtt{instr} & ::= & \mathtt{0x00} & \Rightarrow & \mathsf{unreachable} & \\
& | & \mathtt{0x01} & \Rightarrow & \mathsf{nop} & \\
& | & \mathtt{0x02}\ \ bt{:}\mathtt{blocktype}\ \ (in{:}\mathtt{instr})^*\ \ \mathtt{0x0B} & \Rightarrow & \mathsf{block}\ bt\ in^* & \\
& | & \mathtt{0x03}\ \ bt{:}\mathtt{blocktype}\ \ (in{:}\mathtt{instr})^*\ \ \mathtt{0x0B} & \Rightarrow & \mathsf{loop}\ bt\ in^* & \\
& | & \mathtt{0x04}\ \ bt{:}\mathtt{blocktype}\ \ (in{:}\mathtt{instr})^*\ \ \mathtt{0x0B} & \Rightarrow & \mathsf{if}\ bt\ in^*\ \mathsf{else}\ \epsilon & \\
& | & \mathtt{0x04}\ \ bt{:}\mathtt{blocktype}\ \ (in_1{:}\mathtt{instr})^* & & & \\
& & \quad \mathtt{0x05}\ \ (in_2{:}\mathtt{instr})^*\ \ \mathtt{0x0B} & \Rightarrow & \mathsf{if}\ bt\ in_1^*\ \mathsf{else}\ in_2^* & \\
& | & \mathtt{0x0C}\ \ l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br}\ l & \\
& | & \mathtt{0x0D}\ \ l{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br_if}\ l & \\
& | & \mathtt{0x0E}\ \ l^*{:}\mathtt{list}(\mathtt{labelidx})\ \ l_n{:}\mathtt{labelidx} & \Rightarrow & \mathsf{br_table}\ l^*\ l_n & \\
& | & \mathtt{0x0F} & \Rightarrow & \mathsf{return} & \\
& | & \mathtt{0x10}\ \ x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{call}\ x & \\
& | & \mathtt{0x11}\ \ y{:}\mathtt{typeidx}\ \ x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{call_indirect}\ x\ y & \\
& | & \mathtt{0x12}\ \ x{:}\mathtt{funcidx} & \Rightarrow & \mathsf{return_call}\ x & \\
& | & \mathtt{0x13}\ \ y{:}\mathtt{typeidx}\ \ x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{return_call_indirect}\ x\ y & \\
& | & \dots & & &
\end{array}$$

Note: The else opcode 0x05 in the encoding of an if instruction can be omitted if the following instruction sequence is empty.

Unlike any other occurrence, the type index in a block type is encoded as a positive signed integer, so that its signed LEB128 bit pattern cannot collide with the encoding of value types or the special code 0x40, which correspond to the LEB128 encoding of negative integers. To avoid any loss in the range of allowed indices, it is treated as a 33 bit signed integer.

5.4.2 Reference Instructions

Generic reference instructions are represented by single byte codes, others use prefixes and type operands.

$$
\begin{array}{llll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xD0}~~ht{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.null}~ht \\
& | & \mathtt{0xD1} & \Rightarrow \mathsf{ref.is\_null} \\
& | & \mathtt{0xD2}~~x{:}\mathtt{funcidx} & \Rightarrow \mathsf{ref.func}~x \\
& | & \mathtt{0xD3} & \Rightarrow \mathsf{ref.eq} \\
& | & \mathtt{0xD4} & \Rightarrow \mathsf{ref.as\_non\_null} \\
& | & \mathtt{0xD5}~~l{:}\mathtt{labelidx} & \Rightarrow \mathsf{br\_on\_null}~l \\
& | & \mathtt{0xD6}~~l{:}\mathtt{labelidx} & \Rightarrow \mathsf{br\_on\_non\_null}~l \\
& | & \mathtt{0xFB}~~0{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{struct.new}~x \\
& | & \mathtt{0xFB}~~1{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{struct.new\_default}~x \\
& | & \mathtt{0xFB}~~2{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~i{:}\mathtt{u32} & \Rightarrow \mathsf{struct.get}~x~i \\
& | & \mathtt{0xFB}~~3{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~i{:}\mathtt{u32} & \Rightarrow \mathsf{struct.get\_s}~x~i \\
& | & \mathtt{0xFB}~~4{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~i{:}\mathtt{u32} & \Rightarrow \mathsf{struct.get\_u}~x~i \\
& | & \mathtt{0xFB}~~5{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~i{:}\mathtt{u32} & \Rightarrow \mathsf{struct.set}~x~i \\
& | & \mathtt{0xFB}~~6{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.new}~x \\
& | & \mathtt{0xFB}~~7{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.new\_default}~x \\
& | & \mathtt{0xFB}~~8{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~n{:}\mathtt{u32} & \Rightarrow \mathsf{array.new\_fixed}~x~n \\
& | & \mathtt{0xFB}~~9{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~y{:}\mathtt{dataidx} & \Rightarrow \mathsf{array.new\_data}~x~y \\
& | & \mathtt{0xFB}~~10{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~y{:}\mathtt{elemidx} & \Rightarrow \mathsf{array.new\_elem}~x~y \\
& | & \mathtt{0xFB}~~11{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.get}~x \\
& | & \mathtt{0xFB}~~12{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.get\_s}~x \\
& | & \mathtt{0xFB}~~13{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.get\_u}~x \\
& | & \mathtt{0xFB}~~14{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.set}~x \\
& | & \mathtt{0xFB}~~15{:}\mathtt{u32} & \Rightarrow \mathsf{array.len} \\
& | & \mathtt{0xFB}~~16{:}\mathtt{u32}~~x{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.fill}~x \\
& | & \mathtt{0xFB}~~17{:}\mathtt{u32}~~x_1{:}\mathtt{typeidx}~~x_2{:}\mathtt{typeidx} & \Rightarrow \mathsf{array.copy}~x_1~x_2 \\
& | & \mathtt{0xFB}~~18{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~y{:}\mathtt{dataidx} & \Rightarrow \mathsf{array.init\_data}~x~y \\
& | & \mathtt{0xFB}~~19{:}\mathtt{u32}~~x{:}\mathtt{typeidx}~~y{:}\mathtt{elemidx} & \Rightarrow \mathsf{array.init\_elem}~x~y \\
& | & \mathtt{0xFB}~~20{:}\mathtt{u32}~~ht{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.test}~(\mathsf{ref}~ht) \\
& | & \mathtt{0xFB}~~21{:}\mathtt{u32}~~ht{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.test}~(\mathsf{ref}~\mathsf{null}~ht) \\
& | & \mathtt{0xFB}~~22{:}\mathtt{u32}~~ht{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.cast}~(\mathsf{ref}~ht) \\
& | & \mathtt{0xFB}~~23{:}\mathtt{u32}~~ht{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.cast}~(\mathsf{ref}~\mathsf{null}~ht) \\
& | & \mathtt{0xFB}~~24{:}\mathtt{u32}~~(\mathsf{null}_1^?, \mathsf{null}_2^?){:}\mathtt{castop} \\
& & \quad l{:}\mathtt{labelidx}~~ht_1{:}\mathtt{heaptype}~~ht_2{:}\mathtt{heaptype} & \Rightarrow \mathsf{br\_on\_cast}~l~(\mathsf{ref}~\mathsf{null}_1^?~ht_1)~(\mathsf{ref}~\mathsf{null}_2^?~ht_2) \\
& | & \mathtt{0xFB}~~25{:}\mathtt{u32}~~(\mathsf{null}_1^?, \mathsf{null}_2^?){:}\mathtt{castop} \\
& & \quad l{:}\mathtt{labelidx}~~ht_1{:}\mathtt{heaptype}~~ht_2{:}\mathtt{heaptype} & \Rightarrow \mathsf{br\_on\_cast\_fail}~l~(\mathsf{ref}~\mathsf{null}_1^?~ht_1)~(\mathsf{ref}~\mathsf{null}_2^?~ht_2) \\
& | & \mathtt{0xFB}~~26{:}\mathtt{u32} & \Rightarrow \mathsf{any.convert\_extern} \\
& | & \mathtt{0xFB}~~27{:}\mathtt{u32} & \Rightarrow \mathsf{extern.convert\_any} \\
& | & \mathtt{0xFB}~~28{:}\mathtt{u32} & \Rightarrow \mathsf{ref.i31} \\
& | & \mathtt{0xFB}~~29{:}\mathtt{u32} & \Rightarrow \mathsf{i31.get\_s} \\
& | & \mathtt{0xFB}~~30{:}\mathtt{u32} & \Rightarrow \mathsf{i31.get\_u} \\
& | & \dots \\
\mathtt{castop} & ::= & \mathtt{0x00} & \Rightarrow (\epsilon, \epsilon) \\
& | & \mathtt{0x01} & \Rightarrow (\mathsf{null}, \epsilon) \\
& | & \mathtt{0x02} & \Rightarrow (\epsilon, \mathsf{null}) \\
& | & \mathtt{0x03} & \Rightarrow (\mathsf{null}, \mathsf{null}) \\
\end{array}
$$

5.4.3 Parametric Instructions

Parametric instructions are represented by single byte codes, possibly followed by a type annotation.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x1A} & \Rightarrow & \mathsf{drop} \\
& | & \mathtt{0x1B} & \Rightarrow & \mathsf{select} \\
& | & \mathtt{0x1C}~~ts{:}\mathtt{list}(\mathtt{valtype}) & \Rightarrow & \mathsf{select}~ts \\
& | & \dots
\end{array}
$$

5.4.4 Variable Instructions

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

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

5.4.5 Table Instructions

Table instructions are represented either by a single byte or a one byte prefix followed by a variable-length unsigned integer.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x25}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.get}~x \\
& | & \mathtt{0x26}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.set}~x \\
& | & \mathtt{0xFC}~~12{:}\mathtt{u32}~~y{:}\mathtt{elemidx}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.init}~x~y \\
& | & \mathtt{0xFC}~~13{:}\mathtt{u32}~~x{:}\mathtt{elemidx} & \Rightarrow & \mathsf{elem.drop}~x \\
& | & \mathtt{0xFC}~~14{:}\mathtt{u32}~~x_1{:}\mathtt{tableidx}~~x_2{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.copy}~x_1~x_2 \\
& | & \mathtt{0xFC}~~15{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.grow}~x \\
& | & \mathtt{0xFC}~~16{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.size}~x \\
& | & \mathtt{0xFC}~~17{:}\mathtt{u32}~~x{:}\mathtt{tableidx} & \Rightarrow & \mathsf{table.fill}~x \\
& | & \dots
\end{array}
$$

5.4.6 Memory Instructions

Each variant of memory instruction is encoded with a different byte code. Loads and stores are followed by the encoding of their *memarg* immediate, which includes the memory index if bit 6 of the flags field containing alignment is set; the memory index defaults to 0 otherwise.

$$
\begin{array}{llcll}
\mathtt{memarg} & ::= & n{:}\mathtt{u32}~~m{:}\mathtt{u32} & \Rightarrow & (0, \{\mathsf{align}~n,~\mathsf{offset}~m\}) & \text{if } n < 2^6 \\
& | & n{:}\mathtt{u32}~~x{:}\mathtt{memidx}~~m{:}\mathtt{u32} & \Rightarrow & (x, \{\mathsf{align}~(n-2^6),~\mathsf{offset}~m\}) & \text{if } 2^6 \leq n < 2^7 \\
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x28}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load}~x~ao \\
& | & \mathtt{0x29}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load}~x~ao \\
& | & \mathtt{0x2A}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{f32.load}~x~ao \\
& | & \mathtt{0x2B}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{f64.load}~x~ao \\
& | & \mathtt{0x2C}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load8\_s}~x~ao \\
& | & \mathtt{0x2D}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load8\_u}~x~ao \\
& | & \mathtt{0x2E}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load16\_s}~x~ao \\
& | & \mathtt{0x2F}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.load16\_u}~x~ao \\
& | & \mathtt{0x30}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load8\_s}~x~ao \\
& | & \mathtt{0x31}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load8\_u}~x~ao \\
& | & \mathtt{0x32}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load16\_s}~x~ao \\
& | & \mathtt{0x33}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load16\_u}~x~ao \\
& | & \mathtt{0x34}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load32\_s}~x~ao \\
& | & \mathtt{0x35}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.load32\_u}~x~ao \\
& | & \mathtt{0x36}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store}~x~ao \\
& | & \mathtt{0x37}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store}~x~ao \\
& | & \mathtt{0x38}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{f32.store}~x~ao \\
& | & \mathtt{0x39}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{f64.store}~x~ao \\
& | & \mathtt{0x3A}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store8}~x~ao \\
& | & \mathtt{0x3B}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i32.store16}~x~ao \\
& | & \mathtt{0x3C}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store8}~x~ao \\
& | & \mathtt{0x3D}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store16}~x~ao \\
& | & \mathtt{0x3E}~~(x, ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{i64.store32}~x~ao \\
& | & \mathtt{0x3F}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{memory.size}~x \\
& | & \mathtt{0x40}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{memory.grow}~x \\
& | & \mathtt{0xFC}~~8{:}\mathtt{u32}~~y{:}\mathtt{dataidx}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{memory.init}~x~y \\
& | & \mathtt{0xFC}~~9{:}\mathtt{u32}~~x{:}\mathtt{dataidx} & \Rightarrow & \mathsf{data.drop}~x \\
& | & \mathtt{0xFC}~~10{:}\mathtt{u32}~~x_1{:}\mathtt{memidx}~~x_2{:}\mathtt{memidx} & \Rightarrow & \mathsf{memory.copy}~x_1~x_2 \\
& | & \mathtt{0xFC}~~11{:}\mathtt{u32}~~x{:}\mathtt{memidx} & \Rightarrow & \mathsf{memory.fill}~x \\
& | & \dots
\end{array}
$$

5.4.7 Numeric Instructions

All variants of numeric instructions are represented by separate byte codes.

The const instructions are followed by the respective literal.

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x41}~~n{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.const}~n \\
& | & \mathtt{0x42}~~n{:}\mathtt{u64} & \Rightarrow & \mathsf{i64.const}~n \\
& | & \mathtt{0x43}~~p{:}\mathtt{f32} & \Rightarrow & \mathsf{f32.const}~p \\
& | & \mathtt{0x44}~~p{:}\mathtt{f64} & \Rightarrow & \mathsf{f64.const}~p \\
& | & \dots
\end{array}
$$

All other numeric instructions are plain opcodes without any immediates.

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x45} & \Rightarrow & \mathsf{i32.eqz} \\
& | & \mathtt{0x46} & \Rightarrow & \mathsf{i32.eq} \\
& | & \mathtt{0x47} & \Rightarrow & \mathsf{i32.ne} \\
& | & \mathtt{0x48} & \Rightarrow & \mathsf{i32.lt\_s} \\
& | & \mathtt{0x49} & \Rightarrow & \mathsf{i32.lt\_u} \\
& | & \mathtt{0x4A} & \Rightarrow & \mathsf{i32.gt\_s} \\
& | & \mathtt{0x4B} & \Rightarrow & \mathsf{i32.gt\_u} \\
& | & \mathtt{0x4C} & \Rightarrow & \mathsf{i32.le\_s} \\
& | & \mathtt{0x4D} & \Rightarrow & \mathsf{i32.le\_u} \\
& | & \mathtt{0x4E} & \Rightarrow & \mathsf{i32.ge\_s} \\
& | & \mathtt{0x4F} & \Rightarrow & \mathsf{i32.ge\_u} \\
& | & \mathtt{0x50} & \Rightarrow & \mathsf{i64.eqz} \\
& | & \mathtt{0x51} & \Rightarrow & \mathsf{i64.eq} \\
& | & \mathtt{0x52} & \Rightarrow & \mathsf{i64.ne} \\
& | & \mathtt{0x53} & \Rightarrow & \mathsf{i64.lt\_s} \\
& | & \mathtt{0x54} & \Rightarrow & \mathsf{i64.lt\_u} \\
& | & \mathtt{0x55} & \Rightarrow & \mathsf{i64.gt\_s} \\
& | & \mathtt{0x56} & \Rightarrow & \mathsf{i64.gt\_u} \\
& | & \mathtt{0x57} & \Rightarrow & \mathsf{i64.le\_s} \\
& | & \mathtt{0x58} & \Rightarrow & \mathsf{i64.le\_u} \\
& | & \mathtt{0x59} & \Rightarrow & \mathsf{i64.ge\_s} \\
& | & \mathtt{0x5A} & \Rightarrow & \mathsf{i64.ge\_u} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x5B} & \Rightarrow & \mathsf{f32.eq} \\
& | & \mathtt{0x5C} & \Rightarrow & \mathsf{f32.ne} \\
& | & \mathtt{0x5D} & \Rightarrow & \mathsf{f32.lt} \\
& | & \mathtt{0x5E} & \Rightarrow & \mathsf{f32.gt} \\
& | & \mathtt{0x5F} & \Rightarrow & \mathsf{f32.le} \\
& | & \mathtt{0x60} & \Rightarrow & \mathsf{f32.ge} \\
& | & \mathtt{0x61} & \Rightarrow & \mathsf{f64.eq} \\
& | & \mathtt{0x62} & \Rightarrow & \mathsf{f64.ne} \\
& | & \mathtt{0x63} & \Rightarrow & \mathsf{f64.lt} \\
& | & \mathtt{0x64} & \Rightarrow & \mathsf{f64.gt} \\
& | & \mathtt{0x65} & \Rightarrow & \mathsf{f64.le} \\
& | & \mathtt{0x66} & \Rightarrow & \mathsf{f64.ge} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots & & \\
& | & \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} \\
& | & \dots & & \\
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0x8B} & \Rightarrow & \mathsf{f32.abs} \\
& | & \mathtt{0x8C} & \Rightarrow & \mathsf{f32.neg} \\
& | & \mathtt{0x8D} & \Rightarrow & \mathsf{f32.ceil} \\
& | & \mathtt{0x8E} & \Rightarrow & \mathsf{f32.floor} \\
& | & \mathtt{0x8F} & \Rightarrow & \mathsf{f32.trunc} \\
& | & \mathtt{0x90} & \Rightarrow & \mathsf{f32.nearest} \\
& | & \mathtt{0x91} & \Rightarrow & \mathsf{f32.sqrt} \\
& | & \mathtt{0x92} & \Rightarrow & \mathsf{f32.add} \\
& | & \mathtt{0x93} & \Rightarrow & \mathsf{f32.sub} \\
& | & \mathtt{0x94} & \Rightarrow & \mathsf{f32.mul} \\
& | & \mathtt{0x95} & \Rightarrow & \mathsf{f32.div} \\
& | & \mathtt{0x96} & \Rightarrow & \mathsf{f32.min} \\
& | & \mathtt{0x97} & \Rightarrow & \mathsf{f32.max} \\
& | & \mathtt{0x98} & \Rightarrow & \mathsf{f32.copysign} \\
& | & \mathtt{0x99} & \Rightarrow & \mathsf{f64.abs} \\
& | & \mathtt{0x9A} & \Rightarrow & \mathsf{f64.neg} \\
& | & \mathtt{0x9B} & \Rightarrow & \mathsf{f64.ceil} \\
& | & \mathtt{0x9C} & \Rightarrow & \mathsf{f64.floor} \\
& | & \mathtt{0x9D} & \Rightarrow & \mathsf{f64.trunc} \\
& | & \mathtt{0x9E} & \Rightarrow & \mathsf{f64.nearest} \\
& | & \mathtt{0x9F} & \Rightarrow & \mathsf{f64.sqrt} \\
& | & \mathtt{0xA0} & \Rightarrow & \mathsf{f64.add} \\
& | & \mathtt{0xA1} & \Rightarrow & \mathsf{f64.sub} \\
& | & \mathtt{0xA2} & \Rightarrow & \mathsf{f64.mul} \\
& | & \mathtt{0xA3} & \Rightarrow & \mathsf{f64.div} \\
& | & \mathtt{0xA4} & \Rightarrow & \mathsf{f64.min} \\
& | & \mathtt{0xA5} & \Rightarrow & \mathsf{f64.max} \\
& | & \mathtt{0xA6} & \Rightarrow & \mathsf{f64.copysign} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xA7} & \Rightarrow & \mathsf{i32.convert\_i64} \\
& | & \mathtt{0xA8} & \Rightarrow & \mathsf{i32.convert\_f32\_s} \\
& | & \mathtt{0xA9} & \Rightarrow & \mathsf{i32.convert\_f32\_u} \\
& | & \mathtt{0xAA} & \Rightarrow & \mathsf{i32.convert\_f64\_s} \\
& | & \mathtt{0xAB} & \Rightarrow & \mathsf{i32.convert\_f64\_u} \\
& | & \mathtt{0xAC} & \Rightarrow & \mathsf{i64.convert\_i32\_s} \\
& | & \mathtt{0xAD} & \Rightarrow & \mathsf{i64.convert\_i32\_u} \\
& | & \mathtt{0xAE} & \Rightarrow & \mathsf{i64.convert\_f32\_s} \\
& | & \mathtt{0xAF} & \Rightarrow & \mathsf{i64.convert\_f32\_u} \\
& | & \mathtt{0xB0} & \Rightarrow & \mathsf{i64.convert\_f64\_s} \\
& | & \mathtt{0xB1} & \Rightarrow & \mathsf{i64.convert\_f64\_u} \\
& | & \mathtt{0xB2} & \Rightarrow & \mathsf{f32.convert\_i32\_s} \\
& | & \mathtt{0xB3} & \Rightarrow & \mathsf{f32.convert\_i32\_u} \\
& | & \mathtt{0xB4} & \Rightarrow & \mathsf{f32.convert\_i64\_s} \\
& | & \mathtt{0xB5} & \Rightarrow & \mathsf{f32.convert\_i64\_u} \\
& | & \mathtt{0xB6} & \Rightarrow & \mathsf{f32.convert\_f64} \\
& | & \mathtt{0xB7} & \Rightarrow & \mathsf{f64.convert\_i32\_s} \\
& | & \mathtt{0xB8} & \Rightarrow & \mathsf{f64.convert\_i32\_u} \\
& | & \mathtt{0xB9} & \Rightarrow & \mathsf{f64.convert\_i64\_s} \\
& | & \mathtt{0xBA} & \Rightarrow & \mathsf{f64.convert\_i64\_u} \\
& | & \mathtt{0xBB} & \Rightarrow & \mathsf{f64.convert\_f32} \\
& | & \mathtt{0xBC} & \Rightarrow & \mathsf{i32.reinterpret\_f32} \\
& | & \mathtt{0xBD} & \Rightarrow & \mathsf{i64.reinterpret\_f64} \\
& | & \mathtt{0xBE} & \Rightarrow & \mathsf{f32.reinterpret\_i32} \\
& | & \mathtt{0xBF} & \Rightarrow & \mathsf{f64.reinterpret\_i64} \\
& | & \dots
\end{array}
$$

$$\begin{array}{llcll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xC0} & \Rightarrow & \mathsf{i32.extend8_s} \\
& | & \mathtt{0xC1} & \Rightarrow & \mathsf{i32.extend16_s} \\
& | & \mathtt{0xC2} & \Rightarrow & \mathsf{i64.extend8_s} \\
& | & \mathtt{0xC3} & \Rightarrow & \mathsf{i64.extend16_s} \\
& | & \mathtt{0xC4} & \Rightarrow & \mathsf{i64.extend32_s} \\
& | & \dots
\end{array}$$

The saturating truncation instructions all have a one byte prefix, whereas the actual opcode is encoded by a variable-length unsigned integer.

$$\begin{array}{llcll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFC}~~0{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.convert_sat_f32_s} \\
& | & \mathtt{0xFC}~~1{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.convert_sat_f32_u} \\
& | & \mathtt{0xFC}~~2{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.convert_sat_f64_s} \\
& | & \mathtt{0xFC}~~3{:}\mathtt{u32} & \Rightarrow & \mathsf{i32.convert_sat_f64_u} \\
& | & \mathtt{0xFC}~~4{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.convert_sat_f32_s} \\
& | & \mathtt{0xFC}~~5{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.convert_sat_f32_u} \\
& | & \mathtt{0xFC}~~6{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.convert_sat_f64_s} \\
& | & \mathtt{0xFC}~~7{:}\mathtt{u32} & \Rightarrow & \mathsf{i64.convert_sat_f64_u} \\
& | & \dots
\end{array}$$

5.4.8 Vector Instructions

All variants of vector instructions are represented by separate byte codes. They all have a one byte prefix, whereas the actual opcode is encoded by a variable-length unsigned integer.

Vector loads and stores are followed by the encoding of their *memarg* immediate.

$$\begin{array}{llcll}
\mathtt{laneidx} & ::= & l{:}\mathtt{byte} & \Rightarrow & l \\
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~0{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load}~x~ao \\
& | & \mathtt{0xFD}~~1{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8x8_s}~x~ao \\
& | & \mathtt{0xFD}~~2{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8x8_u}~x~ao \\
& | & \mathtt{0xFD}~~3{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16x4_s}~x~ao \\
& | & \mathtt{0xFD}~~4{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16x4_u}~x~ao \\
& | & \mathtt{0xFD}~~5{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32x2_s}~x~ao \\
& | & \mathtt{0xFD}~~6{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32x2_u}~x~ao \\
& | & \mathtt{0xFD}~~7{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load8_splat}~x~ao \\
& | & \mathtt{0xFD}~~8{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load16_splat}~x~ao \\
& | & \mathtt{0xFD}~~9{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32_splat}~x~ao \\
& | & \mathtt{0xFD}~~10{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load64_splat}~x~ao \\
& | & \mathtt{0xFD}~~11{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.store}~x~ao \\
& | & \mathtt{0xFD}~~84{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load8_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~85{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load16_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~86{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load32_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~87{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.load64_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~88{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store8_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~89{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store16_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~90{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store32_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~91{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{v128.store64_lane}~x~ao~l \\
& | & \mathtt{0xFD}~~92{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load32_zero}~x~ao \\
& | & \mathtt{0xFD}~~93{:}\mathtt{u32}~~(x,ao){:}\mathtt{memarg} & \Rightarrow & \mathsf{v128.load64_zero}~x~ao \\
& | & \dots
\end{array}$$

The const instruction for vectors is followed by 16 immediate bytes, which are converted into an $i128$ in littleendian byte order:

$$\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~12{:}\mathtt{u32}~~(b{:}\mathtt{byte})^{16} & \Rightarrow & \mathsf{v128.const}~\mathrm{bytes}_{\mathsf{i128}}^{-1}((b)^{16}) \\
& | & \dots
\end{array}$$

The shuffle instruction is also followed by the encoding of 16 *laneidx* immediates.

$$\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~13{:}\mathtt{u32}~~(l{:}\mathtt{laneidx})^{16} & \Rightarrow & \mathsf{i8x16.shuffle}~l \\
& | & \mathtt{0xFD}~~14{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.swizzle} \\
& | & \dots
\end{array}$$

Lane instructions are followed by the encoding of a *laneidx* immediate.

$$\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~21{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.extract_lane_s}~l \\
& | & \mathtt{0xFD}~~22{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.extract_lane_u}~l \\
& | & \mathtt{0xFD}~~23{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i8x16.replace_lane}~l \\
& | & \mathtt{0xFD}~~24{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.extract_lane_s}~l \\
& | & \mathtt{0xFD}~~25{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.extract_lane_u}~l \\
& | & \mathtt{0xFD}~~26{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i16x8.replace_lane}~l \\
& | & \mathtt{0xFD}~~27{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i32x4.extract_lane}~l \\
& | & \mathtt{0xFD}~~28{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i32x4.replace_lane}~l \\
& | & \mathtt{0xFD}~~29{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i64x2.extract_lane}~l \\
& | & \mathtt{0xFD}~~30{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{i64x2.replace_lane}~l \\
& | & \mathtt{0xFD}~~31{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f32x4.extract_lane}~l \\
& | & \mathtt{0xFD}~~32{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f32x4.replace_lane}~l \\
& | & \mathtt{0xFD}~~33{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f64x2.extract_lane}~l \\
& | & \mathtt{0xFD}~~34{:}\mathtt{u32}~~l{:}\mathtt{laneidx} & \Rightarrow & \mathsf{f64x2.replace_lane}~l \\
& | & \dots
\end{array}$$

All other vector instructions are plain opcodes without any immediates.

$$\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~15{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.splat} \\
& | & \mathtt{0xFD}~~16{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.splat} \\
& | & \mathtt{0xFD}~~17{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.splat} \\
& | & \mathtt{0xFD}~~18{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.splat} \\
& | & \mathtt{0xFD}~~19{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.splat} \\
& | & \mathtt{0xFD}~~20{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.splat} \\
& | & \dots
\end{array}$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~35{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.eq} \\
& | & \mathtt{0xFD}~~36{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ne} \\
& | & \mathtt{0xFD}~~37{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.lt\_s} \\
& | & \mathtt{0xFD}~~38{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.lt\_u} \\
& | & \mathtt{0xFD}~~39{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.gt\_s} \\
& | & \mathtt{0xFD}~~40{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.gt\_u} \\
& | & \mathtt{0xFD}~~41{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.le\_s} \\
& | & \mathtt{0xFD}~~42{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.le\_u} \\
& | & \mathtt{0xFD}~~43{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ge\_s} \\
& | & \mathtt{0xFD}~~44{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.ge\_u} \\
& | & \mathtt{0xFD}~~45{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.eq} \\
& | & \mathtt{0xFD}~~46{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ne} \\
& | & \mathtt{0xFD}~~47{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.lt\_s} \\
& | & \mathtt{0xFD}~~48{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.lt\_u} \\
& | & \mathtt{0xFD}~~49{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.gt\_s} \\
& | & \mathtt{0xFD}~~50{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.gt\_u} \\
& | & \mathtt{0xFD}~~51{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.le\_s} \\
& | & \mathtt{0xFD}~~52{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.le\_u} \\
& | & \mathtt{0xFD}~~53{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ge\_s} \\
& | & \mathtt{0xFD}~~54{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.ge\_u} \\
& | & \mathtt{0xFD}~~55{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.eq} \\
& | & \mathtt{0xFD}~~56{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.ne} \\
& | & \mathtt{0xFD}~~57{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.lt\_s} \\
& | & \mathtt{0xFD}~~58{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.lt\_u} \\
& | & \mathtt{0xFD}~~59{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.gt\_s} \\
& | & \mathtt{0xFD}~~60{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.gt\_u} \\
& | & \mathtt{0xFD}~~61{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.le\_s} \\
& | & \mathtt{0xFD}~~62{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.le\_u} \\
& | & \mathtt{0xFD}~~63{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.ge\_s} \\
& | & \mathtt{0xFD}~~64{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.ge\_u} \\
& | & \mathtt{0xFD}~~214{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.eq} \\
& | & \mathtt{0xFD}~~215{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.ne} \\
& | & \mathtt{0xFD}~~216{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.lt\_s} \\
& | & \mathtt{0xFD}~~217{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.gt\_s} \\
& | & \mathtt{0xFD}~~218{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.le\_s} \\
& | & \mathtt{0xFD}~~219{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.ge\_s} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~65{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.eq} \\
& | & \mathtt{0xFD}~~66{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.ne} \\
& | & \mathtt{0xFD}~~67{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.lt} \\
& | & \mathtt{0xFD}~~68{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.gt} \\
& | & \mathtt{0xFD}~~69{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.le} \\
& | & \mathtt{0xFD}~~70{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.ge} \\
& | & \mathtt{0xFD}~~71{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.eq} \\
& | & \mathtt{0xFD}~~72{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.ne} \\
& | & \mathtt{0xFD}~~73{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.lt} \\
& | & \mathtt{0xFD}~~74{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.gt} \\
& | & \mathtt{0xFD}~~75{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.le} \\
& | & \mathtt{0xFD}~~76{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.ge} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~77{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.not} \\
& | & \mathtt{0xFD}~~78{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.and} \\
& | & \mathtt{0xFD}~~79{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.andnot} \\
& | & \mathtt{0xFD}~~80{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.or} \\
& | & \mathtt{0xFD}~~81{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.xor} \\
& | & \mathtt{0xFD}~~82{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.bitselect} \\
& | & \mathtt{0xFD}~~83{:}\mathtt{u32} & \Rightarrow & \mathsf{v128.any\_true} \\
& | & \dots \\
\end{array}
$$

$$
\begin{array}{llcll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~96{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.abs} \\
& | & \mathtt{0xFD}~~97{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.neg} \\
& | & \mathtt{0xFD}~~98{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.popcnt} \\
& | & \mathtt{0xFD}~~99{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.all\_true} \\
& | & \mathtt{0xFD}~~100{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.bitmask} \\
& | & \mathtt{0xFD}~~101{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.narrow\_i16x8\_s} \\
& | & \mathtt{0xFD}~~102{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.narrow\_i16x8\_u} \\
& | & \mathtt{0xFD}~~107{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.shl} \\
& | & \mathtt{0xFD}~~108{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.shr\_s} \\
& | & \mathtt{0xFD}~~109{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.shr\_u} \\
& | & \mathtt{0xFD}~~110{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.add} \\
& | & \mathtt{0xFD}~~111{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.add\_sat\_s} \\
& | & \mathtt{0xFD}~~112{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.add\_sat\_u} \\
& | & \mathtt{0xFD}~~113{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.sub} \\
& | & \mathtt{0xFD}~~114{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.sub\_sat\_s} \\
& | & \mathtt{0xFD}~~115{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.sub\_sat\_u} \\
& | & \mathtt{0xFD}~~118{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.min\_s} \\
& | & \mathtt{0xFD}~~119{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.min\_u} \\
& | & \mathtt{0xFD}~~120{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.max\_s} \\
& | & \mathtt{0xFD}~~121{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.max\_u} \\
& | & \mathtt{0xFD}~~123{:}\mathtt{u32} & \Rightarrow & \mathsf{i8x16.avgr\_u} \\
& | & \dots \\
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~124{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extadd\_pairwise\_i8x16\_s} \\
& | & \mathtt{0xFD}~~125{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extadd\_pairwise\_i8x16\_u} \\
& | & \mathtt{0xFD}~~128{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.abs} \\
& | & \mathtt{0xFD}~~129{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.neg} \\
& | & \mathtt{0xFD}~~131{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.all\_true} \\
& | & \mathtt{0xFD}~~132{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.bitmask} \\
& | & \mathtt{0xFD}~~133{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.narrow\_i32x4\_s} \\
& | & \mathtt{0xFD}~~134{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.narrow\_i32x4\_u} \\
& | & \mathtt{0xFD}~~135{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extend\_s\_i8x16\_low} \\
& | & \mathtt{0xFD}~~136{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extend\_s\_i8x16\_high} \\
& | & \mathtt{0xFD}~~137{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extend\_u\_i8x16\_low} \\
& | & \mathtt{0xFD}~~138{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extend\_u\_i8x16\_high} \\
& | & \mathtt{0xFD}~~139{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.shl} \\
& | & \mathtt{0xFD}~~140{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.shr\_s} \\
& | & \mathtt{0xFD}~~141{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.shr\_u} \\
& | & \mathtt{0xFD}~~130{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.q15mulr\_sat\_s} \\
& | & \mathtt{0xFD}~~142{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.add} \\
& | & \mathtt{0xFD}~~143{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.add\_sat\_s} \\
& | & \mathtt{0xFD}~~144{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.add\_sat\_u} \\
& | & \mathtt{0xFD}~~145{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.sub} \\
& | & \mathtt{0xFD}~~146{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.sub\_sat\_s} \\
& | & \mathtt{0xFD}~~147{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.sub\_sat\_u} \\
& | & \mathtt{0xFD}~~149{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.mul} \\
& | & \mathtt{0xFD}~~150{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.min\_s} \\
& | & \mathtt{0xFD}~~151{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.min\_u} \\
& | & \mathtt{0xFD}~~152{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.max\_s} \\
& | & \mathtt{0xFD}~~153{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.max\_u} \\
& | & \mathtt{0xFD}~~155{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.avgr\_u} \\
& | & \mathtt{0xFD}~~156{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extmul\_low\_i8x16\_s} \\
& | & \mathtt{0xFD}~~157{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extmul\_high\_i8x16\_s} \\
& | & \mathtt{0xFD}~~158{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extmul\_low\_i8x16\_u} \\
& | & \mathtt{0xFD}~~159{:}\mathtt{u32} & \Rightarrow & \mathsf{i16x8.extmul\_high\_i8x16\_u} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~126{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extadd\_pairwise\_i16x8\_s} \\
& | & \mathtt{0xFD}~~127{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extadd\_pairwise\_i16x8\_u} \\
& | & \mathtt{0xFD}~~160{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.abs} \\
& | & \mathtt{0xFD}~~161{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.neg} \\
& | & \mathtt{0xFD}~~163{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.all\_true} \\
& | & \mathtt{0xFD}~~164{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.bitmask} \\
& | & \mathtt{0xFD}~~167{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extend\_s\_i16x8\_low} \\
& | & \mathtt{0xFD}~~168{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extend\_s\_i16x8\_high} \\
& | & \mathtt{0xFD}~~169{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extend\_u\_i16x8\_low} \\
& | & \mathtt{0xFD}~~170{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extend\_u\_i16x8\_high} \\
& | & \mathtt{0xFD}~~171{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.shl} \\
& | & \mathtt{0xFD}~~172{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.shr\_s} \\
& | & \mathtt{0xFD}~~173{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.shr\_u} \\
& | & \mathtt{0xFD}~~174{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.add} \\
& | & \mathtt{0xFD}~~177{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.sub} \\
& | & \mathtt{0xFD}~~181{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.mul} \\
& | & \mathtt{0xFD}~~182{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.min\_s} \\
& | & \mathtt{0xFD}~~183{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.min\_u} \\
& | & \mathtt{0xFD}~~184{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.max\_s} \\
& | & \mathtt{0xFD}~~185{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.max\_u} \\
& | & \mathtt{0xFD}~~186{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.dot\_i16x8\_s} \\
& | & \mathtt{0xFD}~~188{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extmul\_low\_i16x8\_s} \\
& | & \mathtt{0xFD}~~189{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extmul\_high\_i16x8\_s} \\
& | & \mathtt{0xFD}~~190{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extmul\_low\_i16x8\_u} \\
& | & \mathtt{0xFD}~~191{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.extmul\_high\_i16x8\_u} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
& | & \mathtt{0xFD}~~192{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.abs} \\
& | & \mathtt{0xFD}~~193{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.neg} \\
& | & \mathtt{0xFD}~~195{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.all\_true} \\
& | & \mathtt{0xFD}~~196{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.bitmask} \\
& | & \mathtt{0xFD}~~199{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extend\_s\_i32x4\_low} \\
& | & \mathtt{0xFD}~~200{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extend\_s\_i32x4\_high} \\
& | & \mathtt{0xFD}~~201{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extend\_u\_i32x4\_low} \\
& | & \mathtt{0xFD}~~202{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extend\_u\_i32x4\_high} \\
& | & \mathtt{0xFD}~~203{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.shl} \\
& | & \mathtt{0xFD}~~204{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.shr\_s} \\
& | & \mathtt{0xFD}~~205{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.shr\_u} \\
& | & \mathtt{0xFD}~~206{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.add} \\
& | & \mathtt{0xFD}~~209{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.sub} \\
& | & \mathtt{0xFD}~~213{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.mul} \\
& | & \mathtt{0xFD}~~220{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extmul\_low\_i32x4\_s} \\
& | & \mathtt{0xFD}~~221{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extmul\_high\_i32x4\_s} \\
& | & \mathtt{0xFD}~~222{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extmul\_low\_i32x4\_u} \\
& | & \mathtt{0xFD}~~223{:}\mathtt{u32} & \Rightarrow & \mathsf{i64x2.extmul\_high\_i32x4\_u} \\
& | & \dots
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
 & | & \mathtt{0xFD}~~103{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.ceil} \\
 & | & \mathtt{0xFD}~~104{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.floor} \\
 & | & \mathtt{0xFD}~~105{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.trunc} \\
 & | & \mathtt{0xFD}~~106{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.nearest} \\
 & | & \mathtt{0xFD}~~224{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.abs} \\
 & | & \mathtt{0xFD}~~225{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.neg} \\
 & | & \mathtt{0xFD}~~227{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.sqrt} \\
 & | & \mathtt{0xFD}~~228{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.add} \\
 & | & \mathtt{0xFD}~~229{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.sub} \\
 & | & \mathtt{0xFD}~~230{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.mul} \\
 & | & \mathtt{0xFD}~~231{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.div} \\
 & | & \mathtt{0xFD}~~232{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.min} \\
 & | & \mathtt{0xFD}~~233{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.max} \\
 & | & \mathtt{0xFD}~~234{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.pmin} \\
 & | & \mathtt{0xFD}~~235{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.pmax} \\
 & | & \dots \\
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
 & | & \mathtt{0xFD}~~116{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.ceil} \\
 & | & \mathtt{0xFD}~~117{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.floor} \\
 & | & \mathtt{0xFD}~~122{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.trunc} \\
 & | & \mathtt{0xFD}~~148{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.nearest} \\
 & | & \mathtt{0xFD}~~236{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.abs} \\
 & | & \mathtt{0xFD}~~237{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.neg} \\
 & | & \mathtt{0xFD}~~239{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.sqrt} \\
 & | & \mathtt{0xFD}~~240{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.add} \\
 & | & \mathtt{0xFD}~~241{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.sub} \\
 & | & \mathtt{0xFD}~~242{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.mul} \\
 & | & \mathtt{0xFD}~~243{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.div} \\
 & | & \mathtt{0xFD}~~244{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.min} \\
 & | & \mathtt{0xFD}~~245{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.max} \\
 & | & \mathtt{0xFD}~~246{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.pmin} \\
 & | & \mathtt{0xFD}~~247{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.pmax} \\
 & | & \dots \\
\end{array}
$$

$$
\begin{array}{llclll}
\mathtt{instr} & ::= & \dots \\
 & | & \mathtt{0xFD}~~94{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.demote\_f64x2\_zero} \\
 & | & \mathtt{0xFD}~~95{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.promote\_f32x4\_low} \\
 & | & \mathtt{0xFD}~~248{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_s} \\
 & | & \mathtt{0xFD}~~249{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_u} \\
 & | & \mathtt{0xFD}~~250{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_s} \\
 & | & \mathtt{0xFD}~~251{:}\mathtt{u32} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_u} \\
 & | & \mathtt{0xFD}~~252{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_zero\_f64x2\_s} \\
 & | & \mathtt{0xFD}~~253{:}\mathtt{u32} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_zero\_f64x2\_u} \\
 & | & \mathtt{0xFD}~~254{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.convert\_s\_i32x4\_low} \\
 & | & \mathtt{0xFD}~~255{:}\mathtt{u32} & \Rightarrow & \mathsf{f64x2.convert\_u\_i32x4\_low} \\
\end{array}
$$

5.4.9 Expressions

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

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

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 basic indices are encoded with their respective value.

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

External indices are encoded by a distiguishing byte followed by an encoding of their respective value.

$$\begin{array}{rcll}
\mathtt{externidx} & ::= & \mathtt{0x00}\ x{:}\mathtt{funcidx} & \Rightarrow \quad \mathsf{func}\ x \\
& | & \mathtt{0x01}\ x{:}\mathtt{tableidx} & \Rightarrow \quad \mathsf{table}\ x \\
& | & \mathtt{0x02}\ x{:}\mathtt{memidx} & \Rightarrow \quad \mathsf{memory}\ x \\
& | & \mathtt{0x03}\ x{:}\mathtt{globalidx} & \Rightarrow \quad \mathsf{global}\ x
\end{array}$$

5.5.2 Sections

Each section consists of

- a one-byte section *id*,
- the $\mathit{u32}$ *length* of the contents, in bytes,
- the actual *contents*, whose structure is dependent on the section id.

Every section is optional; an omitted section is equivalent to the section being present with empty contents.

The following parameterized grammar rule defines the generic structure of a section with id N and contents described by the grammar `X`.

$$\begin{array}{rclll}
\mathtt{section}_N(\mathtt{X}) & ::= & N{:}\mathtt{byte}\ \ len{:}\mathtt{u32}\ \ en^*{:}\mathtt{X} & \Rightarrow \quad en^* & \text{if } len = ||\mathtt{X}|| \\
& | & \epsilon & \Rightarrow \quad \epsilon &
\end{array}$$

For most sections, the contents `X` encodes a list. In these cases, the empty result ϵ is interpreted as the empty list.

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 X.

The following section ids are used:

Id	Section
0	custom section
1	type section
2	import section
3	function section
4	table section
5	memory section
6	global section
7	export section
8	start section
9	element section
10	code section
11	data section
12	data count section

Note: Section ids do not always correspond to the order of sections in the encoding of a module.

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.

$$\begin{array}{rcl} \mathtt{customsec} & ::= & \mathtt{section}_0(\mathtt{custom}) \\ \mathtt{custom} & ::= & \mathtt{name}\ \mathtt{byte}^* \end{array}$$

Note: If an implementation interprets the data of a custom section, then errors in that data, or the placement of the section, must not invalidate the module.

5.5.4 Type Section

The *type section* has the id 1. It decodes into the list of recursive types of a module.

$$\begin{array}{rclcl} \mathtt{typesec} & ::= & ty^*{:}\mathtt{section}_1(\mathtt{list}(\mathtt{type})) & \Rightarrow & ty^* \\ \mathtt{type} & ::= & qt{:}\mathtt{rectype} & \Rightarrow & \mathtt{type}\ qt \end{array}$$

5.5.5 Import Section

The *import section* has the id 2. It decodes into the list of imports of a module.

$$\begin{array}{rcllll} \mathtt{importsec} & ::= & \mathit{im}^*{:}\mathtt{section}_2(\mathtt{list}(\mathtt{import})) & \Rightarrow & \mathit{im}^* \\ \mathtt{import} & ::= & \mathit{nm}_1{:}\mathtt{name}~~\mathit{nm}_2{:}\mathtt{name}~~\mathit{xt}{:}\mathtt{externtype} & \Rightarrow & \mathsf{import}~\mathit{nm}_1~\mathit{nm}_2~\mathit{xt} \end{array}$$

5.5.6 Function Section

The *function section* has the id 3. It decodes into a list of type indices that classify the functions of a module. The bodies of the respective functions are encoded separately in the code section.

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

5.5.7 Table Section

The *table section* has the id 4. It decodes into the list of tables of a module.

$$\begin{array}{rclll} \mathtt{tablesec} & ::= & \mathit{tab}^*{:}\mathtt{section}_4(\mathtt{list}(\mathtt{table})) & \Rightarrow & \mathit{tab}^* \\ \mathtt{table} & ::= & \mathit{tt}{:}\mathtt{tabletype} & \Rightarrow & \mathsf{table}~\mathit{tt}~(\mathsf{ref.null}~\mathit{ht}) \qquad \text{if } \mathit{tt} = \mathit{lim}~(\mathsf{ref}~\mathsf{null}^?~\mathit{ht}) \\ & | & \mathtt{0x40}~~\mathtt{0x00}~~\mathit{tt}{:}\mathtt{tabletype}~~e{:}\mathtt{expr} & \Rightarrow & \mathsf{table}~\mathit{tt}~e \end{array}$$

Note: The encoding of a table type cannot start with byte `0x40`, hence decoding is unambiguous. The zero byte following it is reserved for future extensions.

5.5.8 Memory Section

The *memory section* has the id 5. It decodes into the list of memories of a module.

$$\begin{array}{rclll} \mathtt{memsec} & ::= & \mathit{mem}^*{:}\mathtt{section}_5(\mathtt{list}(\mathtt{mem})) & \Rightarrow & \mathit{mem}^* \\ \mathtt{mem} & ::= & \mathit{mt}{:}\mathtt{memtype} & \Rightarrow & \mathsf{memory}~\mathit{mt} \end{array}$$

5.5.9 Global Section

The *global section* has the id 6. It decodes into the list of globals of a module.

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

5.5.10 Export Section

The *export section* has the id 7. It decodes into the list of exports of a module.

$$\begin{array}{rclll} \mathtt{exportsec} & ::= & \mathit{ex}^*{:}\mathtt{section}_7(\mathtt{list}(\mathtt{export})) & \Rightarrow & \mathit{ex}^* \\ \mathtt{export} & ::= & \mathit{nm}{:}\mathtt{name}~~\mathit{xx}{:}\mathtt{externidx} & \Rightarrow & \mathsf{export}~\mathit{nm}~\mathit{xx} \end{array}$$

5.5.11 Start Section

The *start section* has the id 8. It decodes into the optional start function of a module.

$$
\begin{array}{llll}
\mathtt{startsec} & ::= & \mathit{start}^*{:}\mathtt{section}_8(\mathtt{start}) & \Rightarrow \mathit{start}^* \\
\mathtt{start} & ::= & x{:}\mathtt{funcidx} & \Rightarrow (\mathsf{start}\ x)
\end{array}
$$

5.5.12 Element Section

The *element section* has the id 9. It decodes into the list of element segments of a module.

$$
\begin{array}{llll}
\mathtt{elemsec} & ::= & \mathit{elem}^*{:}\mathtt{section}_9(\mathtt{list}(\mathtt{elem})) & \Rightarrow \mathit{elem}^* \\
\mathtt{elemkind} & ::= & \mathtt{0x00} & \Rightarrow \mathsf{ref\ null\ func} \\
\mathtt{elem} & ::= & 0{:}\mathtt{u32}\ e_o{:}\mathtt{expr}\ y^*{:}\mathtt{list}(\mathtt{funcidx}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ (\mathsf{ref\ func})\ (\mathsf{ref.func}\ y)^*\ (\mathsf{active}\ 0\ e_o) & \\
 & | & 1{:}\mathtt{u32}\ \mathit{rt}{:}\mathtt{elemkind}\ y^*{:}\mathtt{list}(\mathtt{funcidx}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ \mathit{rt}\ (\mathsf{ref.func}\ y)^*\ \mathsf{passive} & \\
 & | & 2{:}\mathtt{u32}\ x{:}\mathtt{tableidx}\ e{:}\mathtt{expr}\ \mathit{rt}{:}\mathtt{elemkind}\ y^*{:}\mathtt{list}(\mathtt{funcidx}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ \mathit{rt}\ (\mathsf{ref.func}\ y)^*\ (\mathsf{active}\ x\ e) & \\
 & | & 3{:}\mathtt{u32}\ \mathit{rt}{:}\mathtt{elemkind}\ y^*{:}\mathtt{list}(\mathtt{funcidx}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ \mathit{rt}\ (\mathsf{ref.func}\ y)^*\ \mathsf{declare} & \\
 & | & 4{:}\mathtt{u32}\ e_o{:}\mathtt{expr}\ e^*{:}\mathtt{list}(\mathtt{expr}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ (\mathsf{ref\ null\ func})\ e^*\ (\mathsf{active}\ 0\ e_o) & \\
 & | & 5{:}\mathtt{u32}\ \mathit{rt}{:}\mathtt{reftype}\ e^*{:}\mathtt{list}(\mathtt{expr}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ \mathit{rt}\ e^*\ \mathsf{passive} & \\
 & | & 6{:}\mathtt{u32}\ x{:}\mathtt{tableidx}\ e_o{:}\mathtt{expr}\ e^*{:}\mathtt{list}(\mathtt{expr}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ (\mathsf{ref\ null\ func})\ e^*\ (\mathsf{active}\ x\ e_o) & \\
 & | & 7{:}\mathtt{u32}\ \mathit{rt}{:}\mathtt{reftype}\ e^*{:}\mathtt{list}(\mathtt{expr}) & \Rightarrow \\
 & & \quad \mathsf{elem}\ \mathit{rt}\ e^*\ \mathsf{declare} &
\end{array}
$$

Note: The initial integer can be interpreted as a bitfield. Bit 0 distinguishes a passive or declarative segment from an active segment, bit 1 indicates the presence of an explicit table index for an active segment and otherwise distinguishes passive from declarative segments, bit 2 indicates the use of element type and element expressions instead of element kind and element indices.

Additional element kinds may be added in future versions of WebAssembly.

5.5.13 Code Section

The *code section* has the id 10. It decodes into the list of *code* entries that are pairs of lists of locals and expressions. They represent the body of the functions of a module. The types of the respective functions are encoded separately in the function section.

The encoding of each code entry consists of

- the $\mathit{u32}$ *length* 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 list whose entries consist of

- a $\mathit{u32}$ *count*,
- a value type,

denoting *count* locals of the same value type.

$$
\begin{array}{llcllll}
\mathtt{codesec} & ::= & \mathit{code}^*{:}\mathtt{section}_{10}(\mathtt{list}(\mathtt{code})) & \Rightarrow & \mathit{code}^* & \\
\mathtt{code} & ::= & \mathit{len}{:}\mathtt{u32}~~\mathit{code}{:}\mathtt{func} & \Rightarrow & \mathit{code} & \text{if } \mathit{len} = ||\mathtt{func}|| \\
\mathtt{func} & ::= & \mathit{loc}^{**}{:}\mathtt{list}(\mathtt{locals})~~e{:}\mathtt{expr} & \Rightarrow & (\bigoplus \mathit{loc}^{**}, e) & \text{if } |\bigoplus \mathit{loc}^{**}| < 2^{32} \\
\mathtt{locals} & ::= & n{:}\mathtt{u32}~~t{:}\mathtt{valtype} & \Rightarrow & (\mathsf{local}~t)^n &
\end{array}
$$

Here, *code* ranges over pairs $(\mathit{local}^*, \mathit{expr})$. Any code for which the length of the resulting sequence is out of bounds of the maximum size of a list 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 the list of data segments of a module.

$$
\begin{array}{llcll}
\mathtt{datasec} & ::= & \mathit{data}^*{:}\mathtt{section}_{11}(\mathtt{list}(\mathtt{data})) & \Rightarrow & \mathit{data}^* \\
\mathtt{data} & ::= & 0{:}\mathtt{u32}~~e{:}\mathtt{expr}~~b^*{:}\mathtt{list}(\mathtt{byte}) & \Rightarrow & \mathsf{data}~b^*~(\mathsf{active}~0~e) \\
& | & 1{:}\mathtt{u32}~~b^*{:}\mathtt{list}(\mathtt{byte}) & \Rightarrow & \mathsf{data}~b^*~\mathsf{passive} \\
& | & 2{:}\mathtt{u32}~~x{:}\mathtt{memidx}~~e{:}\mathtt{expr}~~b^*{:}\mathtt{list}(\mathtt{byte}) & \Rightarrow & \mathsf{data}~b^*~(\mathsf{active}~x~e)
\end{array}
$$

Note: The initial integer can be interpreted as a bitfield. Bit 0 indicates a passive segment, bit 1 indicates the presence of an explicit memory index for an active segment.

5.5.15 Data Count Section

The *data count section* has the id 12. It decodes into an optional $\mathit{u32}$ count that represents the number of data segments in the data section. If this count does not match the length of the data segment list, the module is malformed.

$$
\begin{array}{llcll}
\mathtt{datacntsec} & ::= & n^*{:}\mathtt{section}_{12}(\mathtt{datacnt}) & \Rightarrow & n^* \\
\mathtt{datacnt} & ::= & n{:}\mathtt{u32} & \Rightarrow & n
\end{array}
$$

Note: The data count section is used to simplify single-pass validation. Since the data section occurs after the code section, the memory.init and data.drop instructions would not be able to check whether the data segment index is valid until the data section is read. The data count section occurs before the code section, so a single-pass validator can use this count instead of deferring validation.

5.5.16 Modules

The encoding of a module starts with a preamble containing a 4-byte magic number (the string '`\0asm`') and a version field. The current version of the WebAssembly binary format is 1.

The preamble is followed by a sequence of sections. Custom sections may be inserted at any place in this sequence, while other sections must occur at most once and in the prescribed order. All sections can be empty.

The lengths of lists produced by the (possibly empty) function and code section must match up.

Similarly, the optional data count must match the length of the data segment list. Furthermore, it must be present if any data index occurs in the code section.

$$
\begin{array}{rcl}
\mathtt{magic} & ::= & \mathtt{0x00~0x61~0x73~0x6D} \\
\mathtt{version} & ::= & \mathtt{0x01~0x00~0x00~0x00} \\
\mathtt{module} & ::= & \mathtt{magic~version} \\
& & \mathtt{customsec}^*~\mathit{type}^*{:}\mathtt{typesec} \\
& & \mathtt{customsec}^*~\mathit{import}^*{:}\mathtt{importsec} \\
& & \mathtt{customsec}^*~\mathit{typeidx}^n{:}\mathtt{funcsec} \\
& & \mathtt{customsec}^*~\mathit{table}^*{:}\mathtt{tablesec} \\
& & \mathtt{customsec}^*~\mathit{mem}^*{:}\mathtt{memsec} \\
& & \mathtt{customsec}^*~\mathit{global}^*{:}\mathtt{globalsec} \\
& & \mathtt{customsec}^*~\mathit{export}^*{:}\mathtt{exportsec} \\
& & \mathtt{customsec}^*~\mathit{start}^*{:}\mathtt{startsec} \\
& & \mathtt{customsec}^*~\mathit{elem}^*{:}\mathtt{elemsec} \\
& & \mathtt{customsec}^*~m'^*{:}\mathtt{datacntsec} \\
& & \mathtt{customsec}^*~(\mathit{local}^*, \mathit{expr})^n{:}\mathtt{codesec} \\
& & \mathtt{customsec}^*~\mathit{data}^m{:}\mathtt{datasec} \\
& & \mathtt{customsec}^* \qquad\qquad \Rightarrow \\
& & \quad \mathsf{module}~\mathit{type}^*~\mathit{import}^*~\mathit{func}^n~\mathit{global}^*~\mathit{table}^*~\mathit{mem}^*~\mathit{elem}^*~\mathit{data}^m~\mathit{start}^*~\mathit{export}^* \\
& & \qquad \text{if } m'^* \neq \epsilon \vee \mathrm{dataidx}(\mathit{func}^n) = \epsilon \\
& & \qquad \wedge\, m = \Sigma\, m'^* \\
& & \qquad \wedge\, (\mathit{func} = \mathsf{func}~\mathit{typeidx}~\mathit{local}^*~\mathit{expr})^n
\end{array}
$$

Note: The version of the WebAssembly binary format may increase in the future if backward-incompatible changes have to be made to the format. However, such changes are expected to occur very infrequently, if ever. The binary format is intended to be forward-compatible, such that future extensions can be made without incrementing its version.

CHAPTER 6

Text Format

6.1 Conventions

The textual format for WebAssembly modules is a rendering of their abstract syntax into S-expressions[36].

Like the binary format, the text format is defined by an *attribute grammar*. A text string is a well-formed description of a module if and only if it is generated by the grammar. Each production of this grammar has at most one synthesized attribute: the abstract syntax that the respective character sequence expresses. Thus, the attribute grammar implicitly defines a *parsing* function. Some productions also take a context as an inherited attribute that records bound identifiers.

Except for a few exceptions, the core of the text grammar closely mirrors the grammar of the abstract syntax. However, it also defines a number of *abbreviations* that are "syntactic sugar" over the core syntax.

The recommended extension for files containing WebAssembly modules in text format is "`.wat`". Files with this extension are assumed to be encoded in UTF-8, as per Unicode[37] (Section 2.5).

6.1.1 Grammar

The following conventions are adopted in defining grammar rules of the text format. They mirror the conventions used for abstract syntax and for the binary format. In order to distinguish symbols of the textual syntax from symbols of the abstract syntax, `typewriter` font is adopted for the former.

- Terminal symbols are either literal strings of characters enclosed in quotes or expressed as Unicode[38] scalar values: '`module`', U+0A. (All characters written literally are unambiguously drawn from the 7-bit ASCII[39] 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 $n \leq 1$.)

[36] https://en.wikipedia.org/wiki/S-expression
[37] https://www.unicode.org/versions/latest/
[38] https://www.unicode.org/versions/latest/
[39] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d

- $x{:}T$ denotes the same language as the nonterminal T, but also binds the variable x to the attribute synthesized for T. A pattern may also be used instead of a variable, e.g., $(x, y){:}T$.
- Productions are written $\mathtt{sym} ::= T_1 \Rightarrow A_1 \mid \dots \mid T_n \Rightarrow A_n$, where each A_i is the attribute that is synthesized for $\mathtt{sym}$ in the given case, usually from attribute variables bound in T_i.
- Some productions are augmented by side conditions in parentheses, which restrict the applicability of the production. They provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production (in the syntax or in an attribute), then all those occurrences must have the same instantiation.
- A distinction is made between *lexical* and *syntactic* productions. For the latter, arbitrary white space is allowed in any place where the grammar contains spaces. The productions defining lexical syntax and the syntax of values are considered lexical, all others are syntactic.

Note: For example, the textual grammar for number types is given as follows:

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

The textual grammar for limits is defined as follows:

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

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:

$$\mathit{abbreviation\ syntax} \quad \equiv \quad \mathit{expanded\ syntax}$$

These expansions are assumed to be applied, recursively and in order of appearance, before applying the core grammar rules to construct the abstract syntax.

6.1.3 Contexts

The text format allows the use of symbolic identifiers in place of indices. To resolve these identifiers into concrete indices, some grammar productions are indexed by an *identifier context* I as a synthesized attribute that records the declared identifiers in each index space. In addition, the context records the types defined in the module, so that parameter indices can be computed for functions.

It is convenient to define identifier contexts as records I with abstract syntax as follows:

$$\begin{array}{llll} I & ::= & \{ & \mathsf{types} & (name^?)^*, \\ &&& \mathsf{funcs} & (name^?)^*, \\ &&& \mathsf{tables} & (name^?)^*, \\ &&& \mathsf{mems} & (name^?)^*, \\ &&& \mathsf{globals} & (name^?)^*, \\ &&& \mathsf{elem} & (name^?)^*, \\ &&& \mathsf{data} & (name^?)^*, \\ &&& \mathsf{locals} & (name^?)^*, \\ &&& \mathsf{labels} & (name^?)^*, \\ &&& \mathsf{fields} & ((name^?)^*)^* \} \\ &&& \mathsf{typedefs} & subtype^* \} \end{array}$$

For each index space, such a context contains the list of names assigned to the defined indices, which were denoted by the corresponding identifiers. Unnamed indices are associated with empty (ϵ) entries in these lists. Fields have *dependent* name spaces, and hence a separate list of field identifiers per type.

An identifier context is *well-formed* if no index space contains duplicate identifiers. For fields, names need only be unique within a single type.

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 Lists

Lists are written as plain sequences, but with a restriction on the length of these sequence.

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

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[40] (Section 2.4) *scalar values*.

$$\begin{array}{lll} \mathtt{source} & ::= & \mathtt{char}^* \\ \mathtt{char} & ::= & \mathrm{U{+}00} \mid \dots \mid \mathrm{U{+}D7FF} \mid \mathrm{U{+}E000} \mid \dots \mid \mathrm{U{+}10FFFF} \end{array}$$

Note: While source text may contain any Unicode character in comments or string literals, the rest of the grammar is formed exclusively from the characters supported by the 7-bit ASCII[41] subset of Unicode.

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

6.2.2 Tokens

The character stream in the source text is divided, from left to right, into a sequence of *tokens*, as defined by the following grammar.

$$\begin{array}{lll} \mathtt{token} & ::= & \mathtt{keyword} \mid \mathtt{u}N \mid \mathtt{s}N \mid \mathtt{f}N \mid \mathtt{string} \mid \mathtt{id} \mid \text{`('} \mid \text{`)'} \mid \mathtt{reserved} \\ \mathtt{keyword} & ::= & (\text{`}\mathtt{a}\text{'} \mid \dots \mid \text{`}\mathtt{z}\text{'})\ \mathtt{idchar}^* \qquad (\text{if occurring as a literal terminal in the grammar}) \\ \mathtt{reserved} & ::= & (\mathtt{idchar} \mid \mathtt{string} \mid \text{`,'} \mid \text{`;'} \mid \text{`['} \mid \text{`]'} \mid \text{`\{'} \mid \text{`\}'})^+ \end{array}$$

Tokens are formed from the input character stream according to the *longest match* rule. That is, the next token always consists of the longest possible sequence of characters that is recognized by the above lexical grammar. Tokens can be separated by white space, but except for strings, they cannot themselves contain whitespace.

Keyword tokens are defined either implicitly by an occurrence of a terminal symbol in literal form, such as '`keyword`', in a syntactic production of this chapter, or explicitly where they arise in this chapter.

Any token that does not fall into any of the other categories is considered *reserved*, and cannot occur in source text.

Note: The effect of defining the set of reserved tokens is that all tokens must be separated by either parentheses, white space, or comments. For example, '`0$x`' is a single reserved token, as is '`"a""b"`'. Consequently, they are not recognized as two separate tokens '`0`' and '`$x`', or "a" and "b", respectively, but instead disallowed. This property of tokenization is not affected by the fact that the definition of reserved tokens overlaps with other token classes.

6.2.3 White Space

White space is any sequence of literal space characters, formatting characters, comments, or annotations. The allowed formatting characters correspond to a subset of the ASCII[42] *format effectors*, namely, *horizontal tabulation* (U+09), *line feed* (U+0A), and *carriage return* (U+0D).

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

The only relevance of white space is to separate tokens. It is otherwise ignored.

6.2.4 Comments

A *comment* can either be a *line comment*, started with a double semicolon ';;' and extending to the end of the line, or a *block comment*, enclosed in delimiters '(;' … ';)'. Block comments can be nested.

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

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.

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

6.2.5 Annotations

An *annotation* is a bracketed token sequence headed by an *annotation id* of the form '@id' or '@"…"'. No space is allowed between the opening parenthesis and this id. Annotations are intended to be used for third-party extensions; they can appear anywhere in a program but are ignored by the WebAssembly semantics itself, which treats them as white space.

Annotations can contain other parenthesized token sequences (including nested annotations), as long as they are well-nested. String literals and comments occurring in an annotation must also be properly nested and closed.

$$\begin{array}{llll} \mathtt{annot} & ::= & \text{`(@'}~\mathtt{annotid}~(\mathtt{space} \mid \mathtt{token})^*~\text{`)'} \\ \mathtt{annotid} & ::= & \mathtt{idchar}^+ \mid \mathtt{name} \end{array}$$

Note: The annotation id is meant to be an identifier categorising the extension, and plays a role similar to the name of a custom section. By convention, annotations corresponding to a custom section should use the custom section's name as an id.

Implementations are expected to ignore annotations with ids that they do not recognize. On the other hand, they may impose restrictions on annotations that they do recognize, e.g., requiring a specific structure by superimposing a more concrete grammar. It is up to an implementation how it deals with errors in such annotations.

6.3 Values

The grammar productions in this section define *lexical syntax*, hence no white space is allowed.

6.3.1 Integers

All integers can be written in either decimal or hexadecimal notation. In both cases, digits can optionally be separated by underscores.

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

$$\begin{array}{llllll} \mathtt{num} & ::= & d{:}\mathtt{digit} & \Rightarrow & d \\ & \mid & n{:}\mathtt{num}~\text{`\_'}^?~d{:}\mathtt{digit} & \Rightarrow & 10 \cdot n + d \\ \mathtt{hexnum} & ::= & h{:}\mathtt{hexdigit} & \Rightarrow & h \\ & \mid & n{:}\mathtt{hexnum}~\text{`\_'}^?~h{:}\mathtt{hexdigit} & \Rightarrow & 16 \cdot n + h \end{array}$$

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

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

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

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

6.3.2 Floating-Point

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

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

The value of a literal must not lie outside the representable range of the corresponding IEEE 754[43] 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.

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

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

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

Each character in a string literal represents the byte sequence corresponding to its UTF-8 Unicode[45] (Section 2.5) encoding, except for hexadecimal escape sequences ‘$\backslash hh$’, which represent raw bytes of the respective value.

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

---

[43] https://ieeexplore.ieee.org/document/8766229
[44] https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS+4-1986%5bR2012%5d
[45] https://www.unicode.org/versions/latest/

6.3.4 Names

Names are strings denoting a literal character sequence. A name string must form a valid UTF-8 encoding as defined by Unicode[46] (Section 2.5) and is interpreted as a string of Unicode scalar values.

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

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 eiter a sequence of printable ASCII[47] characters that does not contain a space, quotation mark, comma, semicolon, or bracket, or by a quoted name.

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

Note: The value of an identifier character is the Unicode codepoint denoting it.

Conventions

The expansion rules of some abbreviations require insertion of a *fresh* identifier. That may be any syntactically valid identifier that does not already occur in the given source text.

6.4 Types

6.4.1 Number Types

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

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

6.4.2 Vector Types

$$\mathtt{vectype}_I ::= \text{‘}\mathtt{v128}\text{’} \Rightarrow \mathsf{V128}$$

6.4.3 Heap Types

$$\begin{array}{llll}
\mathtt{absheaptype} & ::= & \text{‘}\mathtt{any}\text{’} & \Rightarrow & \mathsf{any} \\
& | & \text{‘}\mathtt{eq}\text{’} & \Rightarrow & \mathsf{eq} \\
& | & \text{‘}\mathtt{i31}\text{’} & \Rightarrow & \mathsf{i31} \\
& | & \text{‘}\mathtt{struct}\text{’} & \Rightarrow & \mathsf{struct} \\
& | & \text{‘}\mathtt{array}\text{’} & \Rightarrow & \mathsf{array} \\
& | & \text{‘}\mathtt{none}\text{’} & \Rightarrow & \mathsf{none} \\
& | & \text{‘}\mathtt{func}\text{’} & \Rightarrow & \mathsf{func} \\
& | & \text{‘}\mathtt{nofunc}\text{’} & \Rightarrow & \mathsf{nofunc} \\
& | & \text{‘}\mathtt{extern}\text{’} & \Rightarrow & \mathsf{extern} \\
& | & \text{‘}\mathtt{noextern}\text{’} & \Rightarrow & \mathsf{noextern} \\
\mathtt{heaptype}_I & ::= & t{:}\mathtt{absheaptype} & \Rightarrow & y \\
& | & x{:}\mathtt{typeidx}_I & \Rightarrow & x
\end{array}$$

6.4.4 Reference Types

$$\begin{array}{llll}
\mathtt{reftype}_I & ::= & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ ht{:}\mathtt{heaptype}\ \text{‘})\text{’} & \Rightarrow & \mathsf{ref}\ ht \\
& | & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ ht{:}\mathtt{heaptype}\ \text{‘})\text{’} & \Rightarrow & \mathsf{ref}\ \mathsf{null}\ ht
\end{array}$$

Abbreviations

There are shorthands for references to abstract heap types.

$$\begin{array}{lll}
\text{‘}\mathtt{anyref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{any}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{eqref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{eq}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{i31ref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{i31}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{structref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{struct}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{arrayref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{array}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{nullref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{none}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{funcref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{func}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{nullfuncref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{nofunc}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{externref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{extern}\text{’}\ \text{‘})\text{’} \\
\text{‘}\mathtt{nullexternref}\text{’} & \equiv & \text{‘}(\text{’}\ \text{‘}\mathtt{ref}\text{’}\ \text{‘}\mathtt{null}\text{’}\ \text{‘}\mathtt{noextern}\text{’}\ \text{‘})\text{’}
\end{array}$$

6.4.5 Value Types

$$\begin{array}{llll}
\mathtt{valtype}_I & ::= & t{:}\mathtt{numtype}_I & \Rightarrow & t \\
& | & t{:}\mathtt{vectype}_I & \Rightarrow & t \\
& | & t{:}\mathtt{reftype}_I & \Rightarrow & t
\end{array}$$

6.4.6 Function Types

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

Note: The optional identifier names for parameters in a function type only have documentation purpose. They cannot be referenced from anywhere.

Abbreviations

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

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

6.4.7 Aggregate Types

$$
\begin{array}{llll}
\mathtt{arraytype}_I & ::= & \text{`('}~\text{`array'}~\mathit{ft}{:}\mathtt{fieldtype}_I~\text{`)'} & \Rightarrow \mathit{ft} \\
\mathtt{structtype}_I & ::= & \text{`('}~\text{`struct'}~\mathit{ft}^*{:}\mathtt{list}(\mathtt{field}_I)~\text{`)'} & \Rightarrow \mathit{ft}^* \\
\mathtt{field}_I & ::= & \text{`('}~\text{`field'}~\mathtt{id}^?~\mathit{ft}{:}\mathtt{fieldtype}_I~\text{`)'} & \Rightarrow \mathit{ft} \\
\mathtt{fieldtype}_I & ::= & \mathit{st}{:}\mathtt{storagetype} & \Rightarrow \mathsf{const}~\mathit{st} \\
 & | & \text{`('}~\text{`mut'}~\mathit{st}{:}\mathtt{storagetype}~\text{`)'} & \Rightarrow \mathsf{var}~\mathit{st} \\
\mathtt{storagetype}_I & ::= & t{:}\mathtt{valtype}_I & \Rightarrow t \\
 & | & t{:}\mathtt{packtype} & \Rightarrow t \\
\mathtt{packtype} & ::= & \text{`i8'} & \Rightarrow \mathsf{i8} \\
 & | & \text{`i16'} & \Rightarrow \mathsf{i16}
\end{array}
$$

Abbreviations

Multiple anonymous structure fields may be combined into a single declaration:

$$
\text{`('}~\text{`field'}~\mathtt{fieldtype}^*~\text{`)'} \equiv (\text{`('}~\text{`field'}~\mathtt{fieldtype}~\text{`)'})^*
$$

6.4.8 Composite Types

$$
\begin{array}{llll}
\mathtt{comptype}_I & ::= & \mathit{at}{:}\mathtt{arraytype}_I & \Rightarrow \mathsf{array}~\mathit{at} \\
 & | & \mathit{st}{:}\mathtt{structtype}_I & \Rightarrow \mathsf{struct}~\mathit{at} \\
 & | & \mathit{ft}{:}\mathtt{functype}_I & \Rightarrow \mathsf{func}~\mathit{ft}
\end{array}
$$

6.4.9 Recursive Types

$$
\begin{array}{llll}
\mathtt{rectype}_I & ::= & \text{`('}~\text{`rec'}~\mathit{st}^*{:}\mathtt{list}(\mathtt{typedef}_I)~\text{`)'} & \Rightarrow \mathsf{rec}~\mathit{st}^* \\
\mathtt{typedef}_I & ::= & \text{`('}~\text{`type'}~\mathtt{id}^?~\mathit{st}{:}\mathtt{subtype}_I~\text{`)'} & \Rightarrow \mathit{st} \\
\mathtt{subtype}_I & ::= & \text{`('}~\text{`sub'}~\text{`final'}^?~x^*{:}\mathtt{list}(\mathtt{typeidx}_I)~\mathit{ct}{:}\mathtt{comptype}_I~\text{`)'} & \Rightarrow \mathsf{sub}~\mathsf{final}^?~x^*~\mathit{ct}
\end{array}
$$

Abbreviations

Singular recursive types can omit the ‘`rec`’ keyword:

$$\mathtt{typedef} \equiv \text{‘('}\ \text{‘}\mathtt{rec}\text{'}\ \mathtt{typedef}\ \text{‘)'}$$

Similarly, final sub types with no super-types can omit the sub keyword and arguments:

$$\mathtt{comptype} \equiv \text{‘('}\ \text{‘}\mathtt{sub}\text{'}\ \text{‘}\mathtt{final}\text{'}\ \epsilon\ \mathtt{comptype}\ \text{‘)'}$$

6.4.10 Limits

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

6.4.11 Memory Types

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

6.4.12 Table Types

$$\mathtt{tabletype}_I ::= \mathit{lim}{:}\mathtt{limits}\ \ \mathit{et}{:}\mathtt{reftype}_I \Rightarrow \mathit{lim}\ \mathit{et}$$

6.4.13 Global Types

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

6.5 Instructions

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

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

In addition, as a syntactic abbreviation, instructions can be written as S-expressions in folded form, to group them visually.

6.5.1 Labels

Structured control instructions can be annotated with a symbolic label identifier. They are the only symbolic identifiers that can be bound locally in an instruction sequence. The following grammar handles the corresponding update to the identifier context by composing the context with an additional label entry.

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

Note: The new label entry is inserted at the *beginning* of the label list in the identifier context. This effectively shifts all existing labels up by one, mirroring the fact that control instructions are indexed relatively not absolutely.

If a label with the same name already exists, then it is shadowed and the earlier label becomes inaccessible.

6.5.2 Control Instructions

Structured control instructions can bind an optional symbolic label identifier. The same label identifier may optionally be repeated after the corresponding `end` and `else` pseudo instructions, to indicate the matching delimiters.

Their block type is given as a type use, analogous to the type of functions. However, the special case of a type use that is syntactically empty or consists of only a single result is not regarded as an abbreviation for an inline function type, but is parsed directly into an optional value type.

$$
\begin{array}{llll}
\mathtt{blocktype}_I & ::= & (t{:}\mathtt{result}_I)^? \quad \Rightarrow \quad t^? \\
& | & x, I'{:}\mathtt{typeuse}_I \quad \Rightarrow \quad x \qquad (\text{if } I' = \{\mathsf{locals}~(\epsilon)^*\}) \\
\mathtt{blockinstr}_I & ::= & \text{`}\mathtt{block}\text{'}~(v^?, I'){:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in{:}\mathtt{instr}_{I'})^*~~\text{`}\mathtt{end}\text{'}~~v'^?{:}\mathtt{id}^? \\
& & \quad \Rightarrow \quad \mathsf{block}~bt~in^*~\mathsf{end} \qquad (\text{if } v'^? = \epsilon \vee v'^? = v^?) \\
& | & \text{`}\mathtt{loop}\text{'}~(v^?, I'){:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in{:}\mathtt{instr}_{I'})^*~~\text{`}\mathtt{end}\text{'}~~v'^?{:}\mathtt{id}^? \\
& & \quad \Rightarrow \quad \mathsf{loop}~bt~in^*~\mathsf{end} \qquad (\text{if } v'^? = \epsilon \vee v'^? = v^?) \\
& | & \text{`}\mathtt{if}\text{'}~(v^?, I'){:}\mathtt{label}_I~~bt{:}\mathtt{blocktype}_I~~(in_1{:}\mathtt{instr}_{I'})^*~~\text{`}\mathtt{else}\text{'}~~v_1^?{:}\mathtt{id}_1^?~~(in_2{:}\mathtt{instr}_{I'})^*~~\text{`}\mathtt{end}\text{'}~~v_2^?{:}\mathtt{id}_2^? \\
& & \quad \Rightarrow \quad \mathsf{if}~bt~in_1^*~\mathsf{else}~in_2^*~\mathsf{end} \qquad (\text{if } v_1^? = \epsilon \vee v_1^? = v^?, v_2^? = \epsilon \vee v_2^? = v^?)
\end{array}
$$

Note: The side condition stating that the identifier context I' must only contain unnamed entries in the rule for `typeuse` block types enforces that no identifier can be bound in any `param` declaration for a block type.

All other control instruction are represented verbatim.

$$
\begin{array}{llllll}
\mathtt{plaininstr}_I & ::= & \text{`}\mathtt{unreachable}\text{'} & \Rightarrow & \mathsf{unreachable} \\
& | & \text{`}\mathtt{nop}\text{'} & \Rightarrow & \mathsf{nop} \\
& | & \text{`}\mathtt{br}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow & \mathsf{br}~l \\
& | & \text{`}\mathtt{br\_if}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow & \mathsf{br\_if}~l \\
& | & \text{`}\mathtt{br\_table}\text{'}~~l^*{:}\mathtt{list}(\mathtt{labelidx}_I)~~l_N{:}\mathtt{labelidx}_I & \Rightarrow & \mathsf{br\_table}~l^*~l_N \\
& | & \text{`}\mathtt{br\_on\_null}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow & \mathsf{br\_on\_null}~l \\
& | & \text{`}\mathtt{br\_on\_non\_null}\text{'}~~l{:}\mathtt{labelidx}_I & \Rightarrow & \mathsf{br\_on\_non\_null}~l \\
& | & \text{`}\mathtt{br\_on\_cast}\text{'}~~l{:}\mathtt{labelidx}_I~~t_1{:}\mathtt{reftype}~~t_2{:}\mathtt{reftype} & \Rightarrow & \mathsf{br\_on\_cast}~l~t_1~t_2 \\
& | & \text{`}\mathtt{br\_on\_cast\_fail}\text{'}~~l{:}\mathtt{labelidx}_I~~t_1{:}\mathtt{reftype}~~t_2{:}\mathtt{reftype} & \Rightarrow & \mathsf{br\_on\_cast\_fail}~l~t_1~t_2 \\
& | & \text{`}\mathtt{return}\text{'} & \Rightarrow & \mathsf{return} \\
& | & \text{`}\mathtt{call}\text{'}~~x{:}\mathtt{funcidx}_I & \Rightarrow & \mathsf{call}~x \\
& | & \text{`}\mathtt{call\_ref}\text{'}~~x{:}\mathtt{typeidx} & \Rightarrow & \mathsf{call\_ref}~x \\
& | & \text{`}\mathtt{call\_indirect}\text{'}~~x{:}\mathtt{tableidx}~~y, I'{:}\mathtt{typeuse}_I & \Rightarrow & \mathsf{call\_indirect}~x~y & (\text{if } I' = \\
& | & \text{`}\mathtt{return\_call}\text{'}~~x{:}\mathtt{funcidx}_I & \Rightarrow & \mathsf{return\_call}~x \\
& | & \text{`}\mathtt{return\_call\_ref}\text{'}~~x{:}\mathtt{typeidx} & \Rightarrow & \mathsf{return\_call\_ref}~x \\
& | & \text{`}\mathtt{return\_call\_indirect}\text{'}~~x{:}\mathtt{tableidx}~~y, I'{:}\mathtt{typeuse}_I & \Rightarrow & \mathsf{return\_call\_indirect}~x~y & (\text{if } I' =
\end{array}
$$

Note: The side condition stating that the identifier context I' must only contain unnamed entries in the rule for call_indirect enforces that no identifier can be bound in any `param` declaration appearing in the type annotation.

Abbreviations

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

$$\text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}_I~\mathtt{instr}^*~\text{‘}\mathtt{end}\text{’} \quad\equiv\quad \text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}_I~\mathtt{instr}^*~\text{‘}\mathtt{else}\text{’}~\text{‘}\mathtt{end}\text{’}$$

Also, for backwards compatibility, the table index to ‘`call_indirect`’ and ‘`return_call_indirect`’ can be omitted, defaulting to 0.

$$\text{‘}\mathtt{call\_indirect}\text{’}~\mathtt{typeuse} \quad\equiv\quad \text{‘}\mathtt{call\_indire}$$
$$\text{‘}\mathtt{return\_call\_indirect}\text{’}~\mathtt{typeuse} \quad\equiv\quad \text{‘}\mathtt{return\_call\_indirect}\text{’}~0~\mathtt{typeuse}$$

6.5.3 Reference Instructions

$$
\begin{array}{llll}
\mathtt{plaininstr}_I & ::= & \dots & \\
& | & \text{‘}\mathtt{ref.null}\text{’}~t{:}\mathtt{heaptype} & \Rightarrow \mathsf{ref.null}~t \\
& | & \text{‘}\mathtt{ref.func}\text{’}~x{:}\mathtt{funcidx} & \Rightarrow \mathsf{ref.func}~x \\
& | & \text{‘}\mathtt{ref.is\_null}\text{’} & \Rightarrow \mathsf{ref.is\_null} \\
& | & \text{‘}\mathtt{ref.as\_non\_null}\text{’} & \Rightarrow \mathsf{ref.as\_non\_null} \\
& | & \text{‘}\mathtt{ref.eq}\text{’} & \Rightarrow \mathsf{ref.eq} \\
& | & \text{‘}\mathtt{ref.test}\text{’}~t{:}\mathtt{reftype} & \Rightarrow \mathsf{ref.test}~t \\
& | & \text{‘}\mathtt{ref.cast}\text{’}~t{:}\mathtt{reftype} & \Rightarrow \mathsf{ref.cast}~t \\
& | & \text{‘}\mathtt{struct.new}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{struct.new}~x \\
& | & \text{‘}\mathtt{struct.new\_default}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{struct.new\_default}~x \\
& | & \text{‘}\mathtt{struct.get}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{fieldidx}_{I,x} & \Rightarrow \mathsf{struct.get}~x~y \\
& | & \text{‘}\mathtt{struct.get\_u}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{fieldidx}_{I,x} & \Rightarrow \mathsf{struct.get\_u}~x~y \\
& | & \text{‘}\mathtt{struct.get\_s}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{fieldidx}_{I,x} & \Rightarrow \mathsf{struct.get\_s}~x~y \\
& | & \text{‘}\mathtt{struct.set}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{fieldidx}_{I,x} & \Rightarrow \mathsf{struct.set}~x~y \\
& | & \text{‘}\mathtt{array.new}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.new}~x \\
& | & \text{‘}\mathtt{array.new\_default}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.new\_default}~x \\
& | & \text{‘}\mathtt{array.new\_fixed}\text{’}~x{:}\mathtt{typeidx}_I~n{:}\mathtt{u32} & \Rightarrow \mathsf{array.new\_fixed}~x~n \\
& | & \text{‘}\mathtt{array.new\_data}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{dataidx}_I & \Rightarrow \mathsf{array.new\_data}~x~y \\
& | & \text{‘}\mathtt{array.new\_elem}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{elemidx}_I & \Rightarrow \mathsf{array.new\_elem}~x~y \\
& | & \text{‘}\mathtt{array.get}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.get}~x \\
& | & \text{‘}\mathtt{array.get\_u}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.get\_u}~x \\
& | & \text{‘}\mathtt{array.get\_s}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.get\_s}~x \\
& | & \text{‘}\mathtt{array.set}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.set}~x \\
& | & \text{‘}\mathtt{array.len}\text{’} & \Rightarrow \mathsf{array.len} \\
& | & \text{‘}\mathtt{array.fill}\text{’}~x{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.fill}~x \\
& | & \text{‘}\mathtt{array.copy}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{typeidx}_I & \Rightarrow \mathsf{array.copy}~x~y \\
& | & \text{‘}\mathtt{array.init\_data}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{dataidx}_I & \Rightarrow \mathsf{array.init\_data}~x~y \\
& | & \text{‘}\mathtt{array.init\_elem}\text{’}~x{:}\mathtt{typeidx}_I~y{:}\mathtt{elemidx}_I & \Rightarrow \mathsf{array.init\_elem}~x~y \\
& | & \text{‘}\mathtt{ref.i31}\text{’} & \Rightarrow \mathsf{ref.i31} \\
& | & \text{‘}\mathtt{i31.get\_u}\text{’} & \Rightarrow \mathsf{i31.get\_u} \\
& | & \text{‘}\mathtt{i31.get\_s}\text{’} & \Rightarrow \mathsf{i31.get\_s} \\
& | & \text{‘}\mathtt{any.convert\_extern}\text{’} & \Rightarrow \mathsf{any.convert\_extern} \\
& | & \text{‘}\mathtt{extern.convert\_any}\text{’} & \Rightarrow \mathsf{extern.convert\_any}
\end{array}
$$

6.5.4 Parametric Instructions

$$
\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{`}\mathtt{drop}\text{'} & \Rightarrow & \mathsf{drop} \\
& | & \text{`}\mathtt{select}\text{'}\ ((t{:}\mathtt{result}_I)^*)^? & \Rightarrow & \mathsf{select}\ (t^*)^?
\end{array}
$$

6.5.5 Variable Instructions

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

6.5.6 Table Instructions

$$
\begin{array}{llcll}
\mathtt{plaininstr}_I & ::= & \dots \\
& | & \text{`}\mathtt{table.get}\text{'}\ x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.get}\ x \\
& | & \text{`}\mathtt{table.set}\text{'}\ x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.set}\ x \\
& | & \text{`}\mathtt{table.size}\text{'}\ x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.size}\ x \\
& | & \text{`}\mathtt{table.grow}\text{'}\ x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.grow}\ x \\
& | & \text{`}\mathtt{table.fill}\text{'}\ x{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.fill}\ x \\
& | & \text{`}\mathtt{table.copy}\text{'}\ x{:}\mathtt{tableidx}_I\ \ y{:}\mathtt{tableidx}_I & \Rightarrow & \mathsf{table.copy}\ x\ y \\
& | & \text{`}\mathtt{table.init}\text{'}\ x{:}\mathtt{tableidx}_I\ \ y{:}\mathtt{elemidx}_I & \Rightarrow & \mathsf{table.init}\ x\ y \\
& | & \text{`}\mathtt{elem.drop}\text{'}\ x{:}\mathtt{elemidx}_I & \Rightarrow & \mathsf{elem.drop}\ x
\end{array}
$$

Abbreviations

For backwards compatibility, all table indices may be omitted from table instructions, defaulting to 0.

$$
\begin{array}{lll}
\text{`}\mathtt{table.get}\text{'} & \equiv & \text{`}\mathtt{table.get}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.set}\text{'} & \equiv & \text{`}\mathtt{table.set}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.size}\text{'} & \equiv & \text{`}\mathtt{table.size}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.grow}\text{'} & \equiv & \text{`}\mathtt{table.grow}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.fill}\text{'} & \equiv & \text{`}\mathtt{table.fill}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.copy}\text{'} & \equiv & \text{`}\mathtt{table.copy}\text{'}\ \text{`}\mathtt{0}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{table.init}\text{'}\ x{:}\mathtt{elemidx}_I & \equiv & \text{`}\mathtt{table.init}\text{'}\ \text{`}\mathtt{0}\text{'}\ x{:}\mathtt{elemidx}_I
\end{array}
$$

6.5.7 Memory Instructions

The offset and alignment immediates to memory instructions are optional. The offset defaults to $\mathtt{0}$, the alignment to the storage size of the respective memory access, which is its *natural alignment*. Lexically, an `offset` or `align` phrase is considered a single keyword token, so no white space is allowed around the '='.

memarg$_N$	::=	o:offset a:align$_N$	⇒	{align
offset	::=	'offset='o:u32	⇒	o
	|	ϵ	⇒	0
align$_N$	::=	'align='a:u32	⇒	a
	|	ϵ	⇒	N
plaininstr$_I$	::=	...		
	|	'i32.load' x:memidx m:memarg$_4$	⇒	i32.loa
	|	'i64.load' x:memidx m:memarg$_8$	⇒	i64.loa
	|	'f32.load' x:memidx m:memarg$_4$	⇒	f32.loa
	|	'f64.load' x:memidx m:memarg$_8$	⇒	f64.loa
	|	'v128.load' x:memidx m:memarg$_{16}$	⇒	v128.l
	|	'i32.load8_s' x:memidx m:memarg$_1$	⇒	i32.loa
	|	'i32.load8_u' x:memidx m:memarg$_1$	⇒	i32.loa
	|	'i32.load16_s' x:memidx m:memarg$_2$	⇒	i32.loa
	|	'i32.load16_u' x:memidx m:memarg$_2$	⇒	i32.loa
	|	'i64.load8_s' x:memidx m:memarg$_1$	⇒	i64.loa
	|	'i64.load8_u' x:memidx m:memarg$_1$	⇒	i64.loa
	|	'i64.load16_s' x:memidx m:memarg$_2$	⇒	i64.loa
	|	'i64.load16_u' x:memidx m:memarg$_2$	⇒	i64.loa
	|	'i64.load32_s' x:memidx m:memarg$_4$	⇒	i64.loa
	|	'i64.load32_u' x:memidx m:memarg$_4$	⇒	i64.loa
	|	'v128.load8x8_s' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load8x8_u' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load16x4_s' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load16x4_u' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load32x2_s' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load32x2_u' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load8_splat' x:memidx m:memarg$_1$	⇒	v128.l
	|	'v128.load16_splat' x:memidx m:memarg$_2$	⇒	v128.l
	|	'v128.load32_splat' x:memidx m:memarg$_4$	⇒	v128.l
	|	'v128.load64_splat' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load32_zero' x:memidx m:memarg$_4$	⇒	v128.l
	|	'v128.load64_zero' x:memidx m:memarg$_8$	⇒	v128.l
	|	'v128.load8_lane' x:memidx m:memarg$_1$ y:u8	⇒	v128.l
	|	'v128.load16_lane' x:memidx m:memarg$_2$ y:u8	⇒	v128.l
	|	'v128.load32_lane' x:memidx m:memarg$_4$ y:u8	⇒	v128.l
	|	'v128.load64_lane' x:memidx m:memarg$_8$ y:u8	⇒	v128.l
	|	'i32.store' x:memidx m:memarg$_4$	⇒	i32.sto
	|	'i64.store' x:memidx m:memarg$_8$	⇒	i64.sto
	|	'f32.store' x:memidx m:memarg$_4$	⇒	f32.sto
	|	'f64.store' x:memidx m:memarg$_8$	⇒	f64.sto
	|	'v128.store' x:memidx m:memarg$_{16}$	⇒	v128.s
	|	'i32.store8' x:memidx m:memarg$_1$	⇒	i32.sto
	|	'i32.store16' x:memidx m:memarg$_2$	⇒	i32.sto
	|	'i64.store8' x:memidx m:memarg$_1$	⇒	i64.sto
	|	'i64.store16' x:memidx m:memarg$_2$	⇒	i64.sto
	|	'i64.store32' x:memidx m:memarg$_4$	⇒	i64.sto
	|	'v128.store8_lane' x:memidx m:memarg$_1$ y:u8	⇒	v128.s
	|	'v128.store16_lane' x:memidx m:memarg$_2$ y:u8	⇒	v128.s
	|	'v128.store32_lane' x:memidx m:memarg$_4$ y:u8	⇒	v128.s
	|	'v128.store64_lane' x:memidx m:memarg$_8$ y:u8	⇒	v128.s
'memory.size' x:memidx ⇒	memory.size x			
	|	'memory.grow' x:memidx	⇒	memo
	|	'memory.fill' x:memidx	⇒	memo
	|	'memory.copy' x:memidx y:memidx	⇒	memo
	|	'memory.init' x:memidx y:dataidx$_I$	⇒	memo
	|	'data.drop' x:dataidx$_I$	⇒	data.

Abbreviations

As an abbreviation, the memory index can be omitted in all memory instructions, defaulting to 0.

$$
\begin{array}{lcl}
\mathtt{numtype}\text{`}\mathtt{.load}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{numtype}\text{`}\mathtt{.load}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.load}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{vectype}\text{`}\mathtt{.load}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{numtype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_}\text{'}sx\ \mathtt{memarg} & \equiv & \mathtt{numtype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_}\text{'}sx\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\mathtt{x}M\text{`}\mathtt{\_}\text{'}sx\ \mathtt{memarg} & \equiv & \mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\mathtt{x}M\text{`}\mathtt{\_}\text{'}sx\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_splat}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_splat}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_zero}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_zero}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_lane}\text{'}\ \mathtt{memarg}\ \mathtt{u8} & \equiv & \mathtt{vectype}\text{`}\mathtt{.load}\text{'}N\text{`}\mathtt{\_lane}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg}\ \mathtt{u8} \\
\mathtt{numtype}\text{`}\mathtt{.store}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{numtype}\text{`}\mathtt{.store}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.store}\text{'}\ \mathtt{memarg} & \equiv & \mathtt{vectype}\text{`}\mathtt{.store}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{numtype}\text{`}\mathtt{.store}\text{'}N\ \mathtt{memarg} & \equiv & \mathtt{numtype}\text{`}\mathtt{.store}\text{'}N\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg} \\
\mathtt{vectype}\text{`}\mathtt{.store}\text{'}N\text{`}\mathtt{\_lane}\text{'}\ \mathtt{memarg}\ \mathtt{u8} & \equiv & \mathtt{vectype}\text{`}\mathtt{.store}\text{'}N\text{`}\mathtt{\_lane}\text{'}\ \text{`}\mathtt{0}\text{'}\ \mathtt{memarg}\ \mathtt{u8} \\
\text{`}\mathtt{memory.size}\text{'} & \equiv & \text{`}\mathtt{memory.size}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{memory.grow}\text{'} & \equiv & \text{`}\mathtt{memory.grow}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{memory.fill}\text{'} & \equiv & \text{`}\mathtt{memory.fill}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{memory.copy}\text{'} & \equiv & \text{`}\mathtt{memory.copy}\text{'}\ \text{`}\mathtt{0}\text{'}\ \text{`}\mathtt{0}\text{'} \\
\text{`}\mathtt{memory.init}\text{'}\ x{:}\mathtt{elemidx}_I & \equiv & \text{`}\mathtt{memory.init}\text{'}\ \text{`}\mathtt{0}\text{'}\ x{:}\mathtt{elemidx}_I
\end{array}
$$

6.5.8 Numeric Instructions

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

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

|	‘i64.clz’	⇒	i64.clz
|	‘i64.ctz’	⇒	i64.ctz
|	‘i64.popcnt’	⇒	i64.popcnt
|	‘i64.add’	⇒	i64.add
|	‘i64.sub’	⇒	i64.sub
|	‘i64.mul’	⇒	i64.mul
|	‘i64.div_s’	⇒	i64.div_s
|	‘i64.div_u’	⇒	i64.div_u
|	‘i64.rem_s’	⇒	i64.rem_s
|	‘i64.rem_u’	⇒	i64.rem_u
|	‘i64.and’	⇒	i64.and
|	‘i64.or’	⇒	i64.or
|	‘i64.xor’	⇒	i64.xor
|	‘i64.shl’	⇒	i64.shl
|	‘i64.shr_s’	⇒	i64.shr_s
|	‘i64.shr_u’	⇒	i64.shr_u
|	‘i64.rotl’	⇒	i64.rotl
|	‘i64.rotr’	⇒	i64.rotr

|	‘f32.abs’	⇒	f32.abs
|	‘f32.neg’	⇒	f32.neg
|	‘f32.ceil’	⇒	f32.ceil
|	‘f32.floor’	⇒	f32.floor
|	‘f32.trunc’	⇒	f32.trunc
|	‘f32.nearest’	⇒	f32.nearest
|	‘f32.sqrt’	⇒	f32.sqrt
|	‘f32.add’	⇒	f32.add
|	‘f32.sub’	⇒	f32.sub
|	‘f32.mul’	⇒	f32.mul
|	‘f32.div’	⇒	f32.div
|	‘f32.min’	⇒	f32.min
|	‘f32.max’	⇒	f32.max
|	‘f32.copysign’	⇒	f32.copysign

|	‘f64.abs’	⇒	f64.abs
|	‘f64.neg’	⇒	f64.neg
|	‘f64.ceil’	⇒	f64.ceil
|	‘f64.floor’	⇒	f64.floor
|	‘f64.trunc’	⇒	f64.trunc
|	‘f64.nearest’	⇒	f64.nearest
|	‘f64.sqrt’	⇒	f64.sqrt
|	‘f64.add’	⇒	f64.add
|	‘f64.sub’	⇒	f64.sub
|	‘f64.mul’	⇒	f64.mul
|	‘f64.div’	⇒	f64.div
|	‘f64.min’	⇒	f64.min
|	‘f64.max’	⇒	f64.max
|	‘f64.copysign’	⇒	f64.copysign

$$
\begin{array}{llcl}
| & \text{‘i32.eqz’} & \Rightarrow & \mathsf{i32.eqz} \\
| & \text{‘i32.eq’} & \Rightarrow & \mathsf{i32.eq} \\
| & \text{‘i32.ne’} & \Rightarrow & \mathsf{i32.ne} \\
| & \text{‘i32.lt\_s’} & \Rightarrow & \mathsf{i32.lt\_s} \\
| & \text{‘i32.lt\_u’} & \Rightarrow & \mathsf{i32.lt\_u} \\
| & \text{‘i32.gt\_s’} & \Rightarrow & \mathsf{i32.gt\_s} \\
| & \text{‘i32.gt\_u’} & \Rightarrow & \mathsf{i32.gt\_u} \\
| & \text{‘i32.le\_s’} & \Rightarrow & \mathsf{i32.le\_s} \\
| & \text{‘i32.le\_u’} & \Rightarrow & \mathsf{i32.le\_u} \\
| & \text{‘i32.ge\_s’} & \Rightarrow & \mathsf{i32.ge\_s} \\
| & \text{‘i32.ge\_u’} & \Rightarrow & \mathsf{i32.ge\_u} \\
\\
| & \text{‘i64.eqz’} & \Rightarrow & \mathsf{i64.eqz} \\
| & \text{‘i64.eq’} & \Rightarrow & \mathsf{i64.eq} \\
| & \text{‘i64.ne’} & \Rightarrow & \mathsf{i64.ne} \\
| & \text{‘i64.lt\_s’} & \Rightarrow & \mathsf{i64.lt\_s} \\
| & \text{‘i64.lt\_u’} & \Rightarrow & \mathsf{i64.lt\_u} \\
| & \text{‘i64.gt\_s’} & \Rightarrow & \mathsf{i64.gt\_s} \\
| & \text{‘i64.gt\_u’} & \Rightarrow & \mathsf{i64.gt\_u} \\
| & \text{‘i64.le\_s’} & \Rightarrow & \mathsf{i64.le\_s} \\
| & \text{‘i64.le\_u’} & \Rightarrow & \mathsf{i64.le\_u} \\
| & \text{‘i64.ge\_s’} & \Rightarrow & \mathsf{i64.ge\_s} \\
| & \text{‘i64.ge\_u’} & \Rightarrow & \mathsf{i64.ge\_u} \\
\\
| & \text{‘f32.eq’} & \Rightarrow & \mathsf{f32.eq} \\
| & \text{‘f32.ne’} & \Rightarrow & \mathsf{f32.ne} \\
| & \text{‘f32.lt’} & \Rightarrow & \mathsf{f32.lt} \\
| & \text{‘f32.gt’} & \Rightarrow & \mathsf{f32.gt} \\
| & \text{‘f32.le’} & \Rightarrow & \mathsf{f32.le} \\
| & \text{‘f32.ge’} & \Rightarrow & \mathsf{f32.ge} \\
\\
| & \text{‘f64.eq’} & \Rightarrow & \mathsf{f64.eq} \\
| & \text{‘f64.ne’} & \Rightarrow & \mathsf{f64.ne} \\
| & \text{‘f64.lt’} & \Rightarrow & \mathsf{f64.lt} \\
| & \text{‘f64.gt’} & \Rightarrow & \mathsf{f64.gt} \\
| & \text{‘f64.le’} & \Rightarrow & \mathsf{f64.le} \\
| & \text{‘f64.ge’} & \Rightarrow & \mathsf{f64.ge}
\end{array}
$$

$\mid$ ‘i32.wrap_i64’ $\Rightarrow$ i32.wrap_i64
$\mid$ ‘i32.trunc_f32_s’ $\Rightarrow$ i32.trunc_f32_s
$\mid$ ‘i32.trunc_f32_u’ $\Rightarrow$ i32.trunc_f32_u
$\mid$ ‘i32.trunc_f64_s’ $\Rightarrow$ i32.trunc_f64_s
$\mid$ ‘i32.trunc_f64_u’ $\Rightarrow$ i32.trunc_f64_u
$\mid$ ‘i32.trunc_sat_f32_s’ $\Rightarrow$ i32.trunc_sat_f32_s
$\mid$ ‘i32.trunc_sat_f32_u’ $\Rightarrow$ i32.trunc_sat_f32_u
$\mid$ ‘i32.trunc_sat_f64_s’ $\Rightarrow$ i32.trunc_sat_f64_s
$\mid$ ‘i32.trunc_sat_f64_u’ $\Rightarrow$ i32.trunc_sat_f64_u
$\mid$ ‘i64.extend_i32_s’ $\Rightarrow$ i64.extend_i32_s
$\mid$ ‘i64.extend_i32_u’ $\Rightarrow$ i64.extend_i32_u
$\mid$ ‘i64.trunc_f32_s’ $\Rightarrow$ i64.trunc_f32_s
$\mid$ ‘i64.trunc_f32_u’ $\Rightarrow$ i64.trunc_f32_u
$\mid$ ‘i64.trunc_f64_s’ $\Rightarrow$ i64.trunc_f64_s
$\mid$ ‘i64.trunc_f64_u’ $\Rightarrow$ i64.trunc_f64_u
$\mid$ ‘i64.trunc_sat_f32_s’ $\Rightarrow$ i64.trunc_sat_f32_s
$\mid$ ‘i64.trunc_sat_f32_u’ $\Rightarrow$ i64.trunc_sat_f32_u
$\mid$ ‘i64.trunc_sat_f64_s’ $\Rightarrow$ i64.trunc_sat_f64_s
$\mid$ ‘i64.trunc_sat_f64_u’ $\Rightarrow$ i64.trunc_sat_f64_u
$\mid$ ‘f32.convert_i32_s’ $\Rightarrow$ f32.convert_i32_s
$\mid$ ‘f32.convert_i32_u’ $\Rightarrow$ f32.convert_i32_u
$\mid$ ‘f32.convert_i64_s’ $\Rightarrow$ f32.convert_i64_s
$\mid$ ‘f32.convert_i64_u’ $\Rightarrow$ f32.convert_i64_u
$\mid$ ‘f32.demote_f64’ $\Rightarrow$ f32.demote_f64
$\mid$ ‘f64.convert_i32_s’ $\Rightarrow$ f64.convert_i32_s
$\mid$ ‘f64.convert_i32_u’ $\Rightarrow$ f64.convert_i32_u
$\mid$ ‘f64.convert_i64_s’ $\Rightarrow$ f64.convert_i64_s
$\mid$ ‘f64.convert_i64_u’ $\Rightarrow$ f64.convert_i64_u
$\mid$ ‘f64.promote_f32’ $\Rightarrow$ f64.promote_f32
$\mid$ ‘i32.reinterpret_f32’ $\Rightarrow$ i32.reinterpret_f32
$\mid$ ‘i64.reinterpret_f64’ $\Rightarrow$ i64.reinterpret_f64
$\mid$ ‘f32.reinterpret_i32’ $\Rightarrow$ f32.reinterpret_i32
$\mid$ ‘f64.reinterpret_i64’ $\Rightarrow$ f64.reinterpret_i64

$\mid$ ‘i32.extend8_s’ $\Rightarrow$ i32.extend8_s
$\mid$ ‘i32.extend16_s’ $\Rightarrow$ i32.extend16_s
$\mid$ ‘i64.extend8_s’ $\Rightarrow$ i64.extend8_s
$\mid$ ‘i64.extend16_s’ $\Rightarrow$ i64.extend16_s
$\mid$ ‘i64.extend32_s’ $\Rightarrow$ i64.extend32_s

6.5.9 Vector Instructions

Vector constant instructions have a mandatory shape descriptor, which determines how the following values are parsed.

$\mid$ ‘v128.const’ ‘i8x16’ $(n{:}\mathrm{i8})^{16}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i8}(n)^{16})$
$\mid$ ‘v128.const’ ‘i16x8’ $(n{:}\mathrm{i16})^{8}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i16}(n)^{8})$
$\mid$ ‘v128.const’ ‘i32x4’ $(n{:}\mathrm{i32})^{4}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i32}(n)^{4})$
$\mid$ ‘v128.const’ ‘i64x2’ $(n{:}\mathrm{i64})^{2}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{i64}(n)^{2})$
$\mid$ ‘v128.const’ ‘f32x4’ $(z{:}\mathrm{f32})^{4}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{f32}(z)^{4})$
$\mid$ ‘v128.const’ ‘f64x2’ $(z{:}\mathrm{f64})^{2}$ $\Rightarrow$ v128.const $\mathrm{bytes}_{i128}^{-1}(\mathrm{bytes}_{f64}(z)^{2})$

$\mid$ ‘i8x16.shuffle’ $(laneidx{:}\mathrm{u8})^{16}$ $\Rightarrow$ i8x16.shuffle $laneidx^{16}$
$\mid$ ‘i8x16.swizzle’ $\Rightarrow$ i8x16.swizzle

| ‘i8x16.splat’ ⇒ i8x16.splat
| ‘i16x8.splat’ ⇒ i16x8.splat
| ‘i32x4.splat’ ⇒ i32x4.splat
| ‘i64x2.splat’ ⇒ i64x2.splat
| ‘f32x4.splat’ ⇒ f32x4.splat
| ‘f64x2.splat’ ⇒ f64x2.splat

| ‘i8x16.extract_lane_s’ *laneidx*:u8 ⇒ i8x16.extract_lane_s *laneidx*
| ‘i8x16.extract_lane_u’ *laneidx*:u8 ⇒ i8x16.extract_lane_u *laneidx*
| ‘i8x16.replace_lane’ *laneidx*:u8 ⇒ i8x16.replace_lane *laneidx*
| ‘i16x8.extract_lane_s’ *laneidx*:u8 ⇒ i16x8.extract_lane_s *laneidx*
| ‘i16x8.extract_lane_u’ *laneidx*:u8 ⇒ i16x8.extract_lane_u *laneidx*
| ‘i16x8.replace_lane’ *laneidx*:u8 ⇒ i16x8.replace_lane *laneidx*
| ‘i32x4.extract_lane’ *laneidx*:u8 ⇒ i32x4.extract_lane *laneidx*
| ‘i32x4.replace_lane’ *laneidx*:u8 ⇒ i32x4.replace_lane *laneidx*
| ‘i64x2.extract_lane’ *laneidx*:u8 ⇒ i64x2.extract_lane *laneidx*
| ‘i64x2.replace_lane’ *laneidx*:u8 ⇒ i64x2.replace_lane *laneidx*
| ‘f32x4.extract_lane’ *laneidx*:u8 ⇒ f32x4.extract_lane *laneidx*
| ‘f32x4.replace_lane’ *laneidx*:u8 ⇒ f32x4.replace_lane *laneidx*
| ‘f64x2.extract_lane’ *laneidx*:u8 ⇒ f64x2.extract_lane *laneidx*
| ‘f64x2.replace_lane’ *laneidx*:u8 ⇒ f64x2.replace_lane *laneidx*

| ‘i8x16.eq’ ⇒ i8x16.eq
| ‘i8x16.ne’ ⇒ i8x16.ne
| ‘i8x16.lt_s’ ⇒ i8x16.lt_s
| ‘i8x16.lt_u’ ⇒ i8x16.lt_u
| ‘i8x16.gt_s’ ⇒ i8x16.gt_s
| ‘i8x16.gt_u’ ⇒ i8x16.gt_u
| ‘i8x16.le_s’ ⇒ i8x16.le_s
| ‘i8x16.le_u’ ⇒ i8x16.le_u
| ‘i8x16.ge_s’ ⇒ i8x16.ge_s
| ‘i8x16.ge_u’ ⇒ i8x16.ge_u

| ‘i16x8.eq’ ⇒ i16x8.eq
| ‘i16x8.ne’ ⇒ i16x8.ne
| ‘i16x8.lt_s’ ⇒ i16x8.lt_s
| ‘i16x8.lt_u’ ⇒ i16x8.lt_u
| ‘i16x8.gt_s’ ⇒ i16x8.gt_s
| ‘i16x8.gt_u’ ⇒ i16x8.gt_u
| ‘i16x8.le_s’ ⇒ i16x8.le_s
| ‘i16x8.le_u’ ⇒ i16x8.le_u
| ‘i16x8.ge_s’ ⇒ i16x8.ge_s
| ‘i16x8.ge_u’ ⇒ i16x8.ge_u

| ‘i32x4.eq’ ⇒ i32x4.eq
| ‘i32x4.ne’ ⇒ i32x4.ne
| ‘i32x4.lt_s’ ⇒ i32x4.lt_s
| ‘i32x4.lt_u’ ⇒ i32x4.lt_u
| ‘i32x4.gt_s’ ⇒ i32x4.gt_s
| ‘i32x4.gt_u’ ⇒ i32x4.gt_u
| ‘i32x4.le_s’ ⇒ i32x4.le_s
| ‘i32x4.le_u’ ⇒ i32x4.le_u
| ‘i32x4.ge_s’ ⇒ i32x4.ge_s
| ‘i32x4.ge_u’ ⇒ i32x4.ge_u

```
 | 'i64x2.eq'                 ⇒ i64x2.eq
 | 'i64x2.ne'                 ⇒ i64x2.ne
 | 'i64x2.lt_s'               ⇒ i64x2.lt_s
 | 'i64x2.gt_s'               ⇒ i64x2.gt_s
 | 'i64x2.le_s'               ⇒ i64x2.le_s
 | 'i64x2.ge_s'               ⇒ i64x2.ge_s
 |

 | 'f32x4.eq'                 ⇒ f32x4.eq
 | 'f32x4.ne'                 ⇒ f32x4.ne
 | 'f32x4.lt'                 ⇒ f32x4.lt
 | 'f32x4.gt'                 ⇒ f32x4.gt
 | 'f32x4.le'                 ⇒ f32x4.le
 | 'f32x4.ge'                 ⇒ f32x4.ge

 | 'f64x2.eq'                 ⇒ f64x2.eq
 | 'f64x2.ne'                 ⇒ f64x2.ne
 | 'f64x2.lt'                 ⇒ f64x2.lt
 | 'f64x2.gt'                 ⇒ f64x2.gt
 | 'f64x2.le'                 ⇒ f64x2.le
 | 'f64x2.ge'                 ⇒ f64x2.ge

 | 'v128.not'                 ⇒ v128.not
 | 'v128.and'                 ⇒ v128.and
 | 'v128.andnot'              ⇒ v128.andnot
 | 'v128.or'                  ⇒ v128.or
 | 'v128.xor'                 ⇒ v128.xor
 | 'v128.bitselect'           ⇒ v128.bitselect
 | 'v128.any_true'            ⇒ v128.any_true

 | 'i8x16.abs'                ⇒ i8x16.abs
 | 'i8x16.neg'                ⇒ i8x16.neg
 | 'i8x16.all_true'           ⇒ i8x16.all_true
 | 'i8x16.bitmask'            ⇒ i8x16.bitmask
 | 'i8x16.narrow_i16x8_s'     ⇒ i8x16.narrow_i16x8_s
 | 'i8x16.narrow_i16x8_u'     ⇒ i8x16.narrow_i16x8_u
 | 'i8x16.shl'                ⇒ i8x16.shl
 | 'i8x16.shr_s'              ⇒ i8x16.shr_s
 | 'i8x16.shr_u'              ⇒ i8x16.shr_u
 | 'i8x16.add'                ⇒ i8x16.add
 | 'i8x16.add_sat_s'          ⇒ i8x16.add_sat_s
 | 'i8x16.add_sat_u'          ⇒ i8x16.add_sat_u
 | 'i8x16.sub'                ⇒ i8x16.sub
 | 'i8x16.sub_sat_s'          ⇒ i8x16.sub_sat_s
 | 'i8x16.sub_sat_u'          ⇒ i8x16.sub_sat_u
 | 'i8x16.min_s'              ⇒ i8x16.min_s
 | 'i8x16.min_u'              ⇒ i8x16.min_u
 | 'i8x16.max_s'              ⇒ i8x16.max_s
 | 'i8x16.max_u'              ⇒ i8x16.max_u
 | 'i8x16.avgr_u'             ⇒ i8x16.avgr_u
 | 'i8x16.popcnt'             ⇒ i8x16.popcnt
```

```
| 'i16x8.abs'                      ⇒ i16x8.abs
| 'i16x8.neg'                      ⇒ i16x8.neg
| 'i16x8.all_true'                 ⇒ i16x8.all_true
| 'i16x8.bitmask'                  ⇒ i16x8.bitmask
| 'i16x8.narrow_i32x4_s'           ⇒ i16x8.narrow_i32x4_s
| 'i16x8.narrow_i32x4_u'           ⇒ i16x8.narrow_i32x4_u
| 'i16x8.extend_low_i8x16_s'       ⇒ i16x8.extend_low_i8x16_s
| 'i16x8.extend_high_i8x16_s'      ⇒ i16x8.extend_high_i8x16_s
| 'i16x8.extend_low_i8x16_u'       ⇒ i16x8.extend_low_i8x16_u
| 'i16x8.extend_high_i8x16_u'      ⇒ i16x8.extend_high_i8x16_u
| 'i16x8.shl'                      ⇒ i16x8.shl
| 'i16x8.shr_s'                    ⇒ i16x8.shr_s
| 'i16x8.shr_u'                    ⇒ i16x8.shr_u
| 'i16x8.add'                      ⇒ i16x8.add
| 'i16x8.add_sat_s'                ⇒ i16x8.add_sat_s
| 'i16x8.add_sat_u'                ⇒ i16x8.add_sat_u
| 'i16x8.sub'                      ⇒ i16x8.sub
| 'i16x8.sub_sat_s'                ⇒ i16x8.sub_sat_s
| 'i16x8.sub_sat_u'                ⇒ i16x8.sub_sat_u
| 'i16x8.mul'                      ⇒ i16x8.mul
| 'i16x8.min_s'                    ⇒ i16x8.min_s
| 'i16x8.min_u'                    ⇒ i16x8.min_u
| 'i16x8.max_s'                    ⇒ i16x8.max_s
| 'i16x8.max_u'                    ⇒ i16x8.max_u
| 'i16x8.avgr_u'                   ⇒ i16x8.avgr_u
| 'i16x8.q15mulr_sat_s'            ⇒ i16x8.q15mulr_sat_s
| 'i16x8.extmul_low_i8x16_s'       ⇒ i16x8.extmul_low_i8x16_s
| 'i16x8.extmul_high_i8x16_s'      ⇒ i16x8.extmul_high_i8x16_s
| 'i16x8.extmul_low_i8x16_u'       ⇒ i16x8.extmul_low_i8x16_u
| 'i16x8.extmul_high_i8x16_u'      ⇒ i16x8.extmul_high_i8x16_u
| 'i16x8.extadd_pairwise_i8x16_s'  ⇒ i16x8.extadd_pairwise_i8x16_s
| 'i16x8.extadd_pairwise_i8x16_u'  ⇒ i16x8.extadd_pairwise_i8x16_u

| 'i32x4.abs'                      ⇒ i32x4.abs
| 'i32x4.neg'                      ⇒ i32x4.neg
| 'i32x4.all_true'                 ⇒ i32x4.all_true
| 'i32x4.bitmask'                  ⇒ i32x4.bitmask
| 'i32x4.extadd_pairwise_i16x8_s'  ⇒ i32x4.extadd_pairwise_i16x8_s
| 'i32x4.extend_low_i16x8_s'       ⇒ i32x4.extend_low_i16x8_s
| 'i32x4.extend_high_i16x8_s'      ⇒ i32x4.extend_high_i16x8_s
| 'i32x4.extend_low_i16x8_u'       ⇒ i32x4.extend_low_i16x8_u
| 'i32x4.extend_high_i16x8_u'      ⇒ i32x4.extend_high_i16x8_u
| 'i32x4.shl'                      ⇒ i32x4.shl
| 'i32x4.shr_s'                    ⇒ i32x4.shr_s
| 'i32x4.shr_u'                    ⇒ i32x4.shr_u
| 'i32x4.add'                      ⇒ i32x4.add
| 'i32x4.sub'                      ⇒ i32x4.sub
| 'i32x4.mul'                      ⇒ i32x4.mul
| 'i32x4.min_s'                    ⇒ i32x4.min_s
| 'i32x4.min_u'                    ⇒ i32x4.min_u
| 'i32x4.max_s'                    ⇒ i32x4.max_s
| 'i32x4.max_u'                    ⇒ i32x4.max_u
| 'i32x4.dot_i16x8_s'              ⇒ i32x4.dot_i16x8_s
| 'i32x4.extmul_low_i16x8_s'       ⇒ i32x4.extmul_low_i16x8_s
| 'i32x4.extmul_high_i16x8_s'      ⇒ i32x4.extmul_high_i16x8_s
| 'i32x4.extmul_low_i16x8_u'       ⇒ i32x4.extmul_low_i16x8_u
| 'i32x4.extmul_high_i16x8_u'      ⇒ i32x4.extmul_high_i16x8_u
```

```
| 'i64x2.abs'                    ⇒ i64x2.abs
| 'i64x2.neg'                    ⇒ i64x2.neg
| 'i64x2.all_true'               ⇒ i64x2.all_true
| 'i64x2.bitmask'                ⇒ i64x2.bitmask
| 'i64x2.extend_low_i32x4_s'     ⇒ i64x2.extend_low_i32x4_s
| 'i64x2.extend_high_i32x4_s'    ⇒ i64x2.extend_high_i32x4_s
| 'i64x2.extend_low_i32x4_u'     ⇒ i64x2.extend_low_i32x4_u
| 'i64x2.extend_high_i32x4_u'    ⇒ i64x2.extend_high_i32x4_u
| 'i64x2.shl'                    ⇒ i64x2.shl
| 'i64x2.shr_s'                  ⇒ i64x2.shr_s
| 'i64x2.shr_u'                  ⇒ i64x2.shr_u
| 'i64x2.add'                    ⇒ i64x2.add
| 'i64x2.sub'                    ⇒ i64x2.sub
| 'i64x2.mul'                    ⇒ i64x2.mul
| 'i64x2.extmul_low_i32x4_s'     ⇒ i64x2.extmul_low_i32x4_s
| 'i64x2.extmul_high_i32x4_s'    ⇒ i64x2.extmul_high_i32x4_s
| 'i64x2.extmul_low_i32x4_u'     ⇒ i64x2.extmul_low_i32x4_u
| 'i64x2.extmul_high_i32x4_u'    ⇒ i64x2.extmul_high_i32x4_u

| 'f32x4.abs'                    ⇒ f32x4.abs
| 'f32x4.neg'                    ⇒ f32x4.neg
| 'f32x4.sqrt'                   ⇒ f32x4.sqrt
| 'f32x4.ceil'                   ⇒ f32x4.ceil
| 'f32x4.floor'                  ⇒ f32x4.floor
| 'f32x4.trunc'                  ⇒ f32x4.trunc
| 'f32x4.nearest'                ⇒ f32x4.nearest
| 'f32x4.add'                    ⇒ f32x4.add
| 'f32x4.sub'                    ⇒ f32x4.sub
| 'f32x4.mul'                    ⇒ f32x4.mul
| 'f32x4.div'                    ⇒ f32x4.div
| 'f32x4.min'                    ⇒ f32x4.min
| 'f32x4.max'                    ⇒ f32x4.max
| 'f32x4.pmin'                   ⇒ f32x4.pmin
| 'f32x4.pmax'                   ⇒ f32x4.pmax

| 'f64x2.abs'                    ⇒ f64x2.abs
| 'f64x2.neg'                    ⇒ f64x2.neg
| 'f64x2.sqrt'                   ⇒ f64x2.sqrt
| 'f64x2.ceil'                   ⇒ f64x2.ceil
| 'f64x2.floor'                  ⇒ f64x2.floor
| 'f64x2.trunc'                  ⇒ f64x2.trunc
| 'f64x2.nearest'                ⇒ f64x2.nearest
| 'f64x2.add'                    ⇒ f64x2.add
| 'f64x2.sub'                    ⇒ f64x2.sub
| 'f64x2.mul'                    ⇒ f64x2.mul
| 'f64x2.div'                    ⇒ f64x2.div
| 'f64x2.min'                    ⇒ f64x2.min
| 'f64x2.max'                    ⇒ f64x2.max
| 'f64x2.pmin'                   ⇒ f64x2.pmin
| 'f64x2.pmax'                   ⇒ f64x2.pmax
```

$$
\begin{array}{llll}
| & \text{‘}\mathtt{i32x4.trunc\_sat\_f32x4\_s}\text{’} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_s} \\
| & \text{‘}\mathtt{i32x4.trunc\_sat\_f32x4\_u}\text{’} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f32x4\_u} \\
| & \text{‘}\mathtt{i32x4.trunc\_sat\_f64x2\_s\_zero}\text{’} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f64x2\_s\_zero} \\
| & \text{‘}\mathtt{i32x4.trunc\_sat\_f64x2\_u\_zero}\text{’} & \Rightarrow & \mathsf{i32x4.trunc\_sat\_f64x2\_u\_zero} \\
| & \text{‘}\mathtt{f32x4.convert\_i32x4\_s}\text{’} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_s} \\
| & \text{‘}\mathtt{f32x4.convert\_i32x4\_u}\text{’} & \Rightarrow & \mathsf{f32x4.convert\_i32x4\_u} \\
| & \text{‘}\mathtt{f64x2.convert\_low\_i32x4\_s}\text{’} & \Rightarrow & \mathsf{f64x2.convert\_low\_i32x4\_s} \\
| & \text{‘}\mathtt{f64x2.convert\_low\_i32x4\_u}\text{’} & \Rightarrow & \mathsf{f64x2.convert\_low\_i32x4\_u} \\
| & \text{‘}\mathtt{f32x4.demote\_f64x2\_zero}\text{’} & \Rightarrow & \mathsf{f32x4.demote\_f64x2\_zero} \\
| & \text{‘}\mathtt{f64x2.promote\_low\_f32x4}\text{’} & \Rightarrow & \mathsf{f64x2.promote\_low\_f32x4}
\end{array}
$$

6.5.10 Folded Instructions

Instructions can be written as S-expressions by grouping them into *folded* form. In that notation, an instruction is wrapped in parentheses and optionally includes nested folded instructions to indicate its operands.

In the case of block instructions, the folded form omits the ‘end’ delimiter. For if instructions, both branches have to be wrapped into nested S-expressions, headed by the keywords ‘then’ and ‘else’.

The set of all phrases defined by the following abbreviations recursively forms the auxiliary syntactic class foldedinstr. Such a folded instruction can appear anywhere a regular instruction can.

$$
\begin{array}{lll}
\text{‘(’}~\mathtt{plaininstr}~\mathtt{foldedinstr}^*~\text{‘)’} & \equiv & \mathtt{foldedinstr}^*~\mathtt{plaininstr} \\
\text{‘(’}~\text{‘}\mathtt{block}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘)’} & \equiv & \text{‘}\mathtt{block}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘}\mathtt{end}\text{’} \\
\text{‘(’}~\text{‘}\mathtt{loop}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘)’} & \equiv & \text{‘}\mathtt{loop}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}^*~\text{‘}\mathtt{end}\text{’} \\
\multicolumn{3}{l}{\text{‘(’}~\text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{foldedinstr}^*~\text{‘(’}~\text{‘}\mathtt{then}\text{’}~\mathtt{instr}_1^*~\text{‘)’}~(\text{‘(’}~\text{‘}\mathtt{else}\text{’}~\mathtt{instr}_2^*~\text{‘)’})^?~\text{‘)’} \quad \equiv} \\
\multicolumn{3}{l}{\quad \mathtt{foldedinstr}^*~\text{‘}\mathtt{if}\text{’}~\mathtt{label}~\mathtt{blocktype}~\mathtt{instr}_1^*~\text{‘}\mathtt{else}\text{’}~(\mathtt{instr}_2^*)^?~\text{‘}\mathtt{end}\text{’}}
\end{array}
$$

Note: For example, the instruction sequence

$$(\mathtt{local.get}~\$\mathtt{x})~(\mathtt{i32.const}~2)~\mathtt{i32.add}~(\mathtt{i32.const}~3)~\mathtt{i32.mul}$$

can be folded into

$$(\mathtt{i32.mul}~(\mathtt{i32.add}~(\mathtt{local.get}~\$\mathtt{x})~(\mathtt{i32.const}~2))~(\mathtt{i32.const}~3))$$

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

6.5.11 Expressions

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

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

6.6 Modules

6.6.1 Indices

Indices can be given either in raw numeric form or as symbolic identifiers when bound by a respective construct. Such identifiers are looked up in the suitable space of the identifier context I.

$$
\begin{array}{llcllll}
\mathtt{typeidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{types}[x] = v) \\
\mathtt{funcidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{funcs}[x] = v) \\
\mathtt{tableidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{tables}[x] = v) \\
\mathtt{memidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{mems}[x] = v) \\
\mathtt{globalidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{globals}[x] = v) \\
\mathtt{elemidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{elem}[x] = v) \\
\mathtt{dataidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{data}[x] = v) \\
\mathtt{localidx}_I & ::= & x{:}\mathtt{u32} & \Rightarrow & x \\
& | & v{:}\mathtt{id} & \Rightarrow & x & (\text{if } I.\mathsf{locals}[x] = v) \\
\mathtt{labelidx}_I & ::= & l{:}\mathtt{u32} & \Rightarrow & l \\
& | & v{:}\mathtt{id} & \Rightarrow & l & (\text{if } I.\mathsf{labels}[l] = v) \\
\mathtt{fieldidx}_{I,x} & ::= & i{:}\mathtt{u32} & \Rightarrow & i \\
& | & v{:}\mathtt{id} & \Rightarrow & i & (\text{if } I.\mathsf{fields}[x][i] = v)
\end{array}
$$

6.6.2 Type Uses

A *type use* is a reference to a function type definition. It may optionally be augmented by explicit inlined parameter and result declarations. That allows binding symbolic identifiers to name the local indices of parameters. If inline declarations are given, then their types must match the referenced function type.

$$
\begin{array}{llll}
\mathtt{typeuse}_I & ::= & \text{‘('}~\text{‘}\mathtt{type}\text{'}~x{:}\mathtt{typeidx}_I~\text{‘)'} \quad \Rightarrow \quad x, I' \\
& & \quad (\text{if } I.\mathsf{typedefs}[x] = \mathsf{sub~final~(func}~[t_1^n] \to [t_2^*]) \wedge I' = \{\mathsf{locals}~(\epsilon)^n\}) \\
& | & \text{‘('}~\text{‘}\mathtt{type}\text{'}~x{:}\mathtt{typeidx}_I~\text{‘)'}~(t_1{:}\mathtt{param})^*~(t_2{:}\mathtt{result})^* \quad \Rightarrow \quad x, I' \\
& & \quad (\text{if } I.\mathsf{typedefs}[x] = \mathsf{sub~final~(func}~[t_1^*] \to [t_2^*]) \wedge I' = \{\mathsf{locals}~\mathrm{id}(\mathtt{param})^*\}~\text{well-formed})
\end{array}
$$

Note: If inline declarations are given, their types must be *syntactically* equal to the types from the indexed definition; possible type substitutions from other definitions that might make them equal are not taken into account. This is to simplify syntactic pre-processing.

The synthesized attribute of a `typeuse` is a pair consisting of both the used type index and the local identifier context containing possible parameter identifiers. The following auxiliary function extracts optional identifiers from parameters:

$$
\mathrm{id}(\text{‘('}~\text{‘}\mathtt{param}\text{'}~\mathtt{id}^?~\dots~\text{‘)'}) \quad = \quad \mathtt{id}^?
$$

Note: Both productions overlap for the case that the function type is $[] \to []$. However, in that case, they also produce the same results, so that the choice is immaterial.

The well-formedness condition on I' ensures that the parameters do not contain duplicate identifiers.

Abbreviations

A typeuse may also be replaced entirely by inline parameter and result declarations. In that case, a type index is automatically inserted:

$$(t_1{:}\mathtt{param})^* \ (t_2{:}\mathtt{result})^* \quad \equiv \quad \text{‘('}\ \text{‘}\mathtt{type}\text{'}\ x\ \text{‘)'}\ \mathtt{param}^*\ \mathtt{result}^*$$

where x is the smallest existing type index whose recursive type definition in the current module is of the form

$$\text{‘('}\ \text{‘}\mathtt{rec}\text{'}\ \text{‘('}\ \text{‘}\mathtt{type}\text{'}\ \text{‘('}\ \text{‘}\mathtt{sub}\text{'}\ \text{‘}\mathtt{final}\text{'}\ \text{‘('}\ \text{‘}\mathtt{func}\text{'}\ \mathtt{param}^*\ \mathtt{result}^*\ \text{‘)'}\ \text{‘)'}\ \text{‘)'}\ \text{‘)'}$$

If no such index exists, then a new recursive type of the same form 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.

6.6.3 Imports

The descriptors in imports can bind a symbolic function, table, memory, or global identifier.

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

Abbreviations

As an abbreviation, imports may also be specified inline with function, table, memory, or global definitions; see the respective sections.

6.6.4 Functions

Function definitions can bind a symbolic function identifier, and local identifiers for its parameters and locals.

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

The definition of the local identifier context I'' uses the following auxiliary function to extract optional identifiers from locals:

$$\mathrm{id}(\text{‘('}\ \text{‘}\mathtt{local}\text{'}\ \mathtt{id}^?\ \dots\ \text{‘)'}) \quad = \quad \mathtt{id}^?$$

Note: The well-formedness condition on I'' ensures that parameters and locals do not contain duplicate identifiers.

Abbreviations

Multiple anonymous locals may be combined into a single declaration:

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

Functions can be defined as imports or exports inline:

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

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a function declaration can contain any number of exports, possibly followed by an import.

6.6.5 Tables

Table definitions can bind a symbolic table identifier.

$$\mathtt{table}_I\ ::=\ \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ tt{:}\mathtt{tabletype}_I\ e{:}\mathtt{expr}_I\ \text{`)'}\ \Rightarrow\ \{\mathsf{type}\ tt, \mathsf{init}\ e\}$$

Abbreviations

A table's initialization expression can be omitted, in which case it defaults to ref.null:

$$\begin{array}{l}\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathtt{tabletype}\ \text{`)'}\quad\equiv\quad\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathtt{tabletype}\ \text{`('}\ \mathsf{ref.null}\ ht\ \text{`)'}\ \text{`)'}\\ \qquad\qquad\qquad(\text{if } \mathtt{tabletype} = \mathtt{limits}\ \text{`('}\ \text{`}\mathtt{ref}\text{'}\ \text{`}\mathtt{null}\text{'}^?\ ht\ \text{`)'})\end{array}$$

An element segment can be given inline with a table definition, in which case its offset is 0 and the limits of the table type are inferred from the length of the given segment:

$$\begin{array}{l}\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}^?\ \mathtt{reftype}\ \text{`('}\ \text{`}\mathtt{elem}\text{'}\ \mathit{expr}^n{:}\mathtt{list}(\mathtt{elemexpr})\ \text{`)'}\ \text{`)'}\quad\equiv\\ \quad\text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}'\ n\ n\ \mathtt{reftype}\ \text{`)'}\\ \quad\text{`('}\ \text{`}\mathtt{elem}\text{'}\ \text{`('}\ \text{`}\mathtt{table}\text{'}\ \mathtt{id}'\ \text{`)'}\ \text{`('}\ \text{`}\mathtt{i32.const}\text{'}\ \text{`}\mathtt{0}\text{'}\ \text{`)'}\ \mathtt{reftype}\ \mathtt{list}(\mathtt{elemexpr})\ \text{`)'}\\ \qquad(\text{if } \mathtt{id}^? \neq \epsilon \wedge \mathtt{id}' = \mathtt{id}^? \vee \mathtt{id}^? = \epsilon \wedge \mathtt{id}' \text{ fresh})\end{array}$$

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

Tables can be defined as imports or exports inline:

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

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a table declaration can contain any number of exports, possibly followed by an import.

6.6.6 Memories

Memory definitions can bind a symbolic memory identifier.

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

Abbreviations

A data segment can be given inline with a memory definition, in which case its offset is 0 and the limits of the memory type are inferred from the length of the data, rounded up to page size:

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

Memories can be defined as imports or exports inline:

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

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a memory declaration can contain any number of exports, possibly followed by an import.

6.6.7 Globals

Global definitions can bind a symbolic global identifier.

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

Abbreviations

Globals can be defined as imports or exports inline:

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

Note: The latter abbreviation can be applied repeatedly, if "…" contains additional export clauses. Consequently, a global declaration can contain any number of exports, possibly followed by an import.

6.6.8 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.

6.6.9 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.

6.6.10 Element Segments

Element segments allow for an optional table index to identify the table to initialize.

$$
\begin{array}{lll}
\mathtt{elem}_I & ::= & \text{`('}~\text{`}\mathtt{elem}\text{'}~~\mathtt{id}^?~~(et, y^*){:}\mathtt{elemlist}_I~\text{`)'} \\
& & \quad\Rightarrow~~ \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{passive}\} \\
& | & \text{`('}~\text{`}\mathtt{elem}\text{'}~~\mathtt{id}^?~~x{:}\mathtt{tableuse}_I~~\text{`('}~\text{`}\mathtt{offset}\text{'}~~e{:}\mathtt{expr}_I~\text{`)'}~~(et, y^*){:}\mathtt{elemlist}_I~\text{`)'} \\
& & \quad\Rightarrow~~ \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{table}~x, \mathsf{offset}~e\}\} \\
& & \text{`('}~\text{`}\mathtt{elem}\text{'}~~\mathtt{id}^?~~\text{`}\mathtt{declare}\text{'}~~(et, y^*){:}\mathtt{elemlist}_I~\text{`)'} \\
& & \quad\Rightarrow~~ \{\mathsf{type}~et, \mathsf{init}~y^*, \mathsf{mode}~\mathsf{declare}\} \\
\mathtt{elemlist}_I & ::= & t{:}\mathtt{reftype}_I~~y^*{:}\mathtt{list}(\mathtt{elemexpr}_I) \quad\Rightarrow\quad (\mathsf{type}~t, \mathsf{init}~y^*) \\
\mathtt{elemexpr}_I & ::= & \text{`('}~\text{`}\mathtt{item}\text{'}~~e{:}\mathtt{expr}_I~\text{`)'} \quad\Rightarrow\quad e \\
\mathtt{tableuse}_I & ::= & \text{`('}~\text{`}\mathtt{table}\text{'}~~x{:}\mathtt{tableidx}_I~\text{`)'} \quad\Rightarrow\quad x
\end{array}
$$

Abbreviations

As an abbreviation, a single instruction may occur in place of the offset of an active element segment or as an element expression:

$$
\begin{array}{lcl}
\text{`('}~\mathtt{instr}~\text{`)'} & \equiv & \text{`('}~\text{`}\mathtt{offset}\text{'}~~\mathtt{instr}~\text{`)'} \\
\text{`('}~\mathtt{instr}~\text{`)'} & \equiv & \text{`('}~\text{`}\mathtt{item}\text{'}~~\mathtt{instr}~\text{`)'}
\end{array}
$$

Also, the element list may be written as just a sequence of function indices:

$$
\text{`}\mathtt{func}\text{'}~~\mathtt{list}(\mathtt{funcidx}_I) ~~\equiv~~ \text{`(}\mathtt{ref}\text{'}~\text{`}\mathtt{func}\text{)'}~~\mathtt{list}(\text{`('}~\text{`}\mathtt{ref.func}\text{'}~~\mathtt{funcidx}_I~\text{`)'})
$$

A table use can be omitted, defaulting to `0`. Furthermore, for backwards compatibility with earlier versions of WebAssembly, if the table use is omitted, the ‘`func`’ keyword can be omitted as well.

$$
\begin{array}{lcl}
\epsilon & \equiv & \text{`('}~\text{`}\mathtt{table}\text{'}~~\text{`}0\text{'}~\text{`)'} \\
\text{`('}~\text{`}\mathtt{elem}\text{'}~~\mathtt{id}^?~~\text{`('}~\text{`}\mathtt{offset}\text{'}~~\mathtt{expr}_I~\text{`)'}~~\mathtt{list}(\mathtt{funcidx}_I)~\text{`)'} & \equiv & \text{`('}~\text{`}\mathtt{elem}\text{'}~~\mathtt{id}^?~~\text{`('}~\text{`}\mathtt{table}\text{'}~~\text{`}0\text{'}~\text{`)'}~~\text{`('}~\text{`}\mathtt{offset}\text{'}~~\mathtt{expr}_I
\end{array}
$$

As another abbreviation, element segments may also be specified inline with table definitions; see the respective section.

6.6.11 Data Segments

Data segments allow for an optional memory index to identify the memory to initialize. The data is written as a string, which may be split up into a possibly empty sequence of individual string literals.

$$
\begin{array}{llll}
\mathtt{data}_I & ::= & \text{`('}~\text{`}\mathtt{data}\text{'}~\mathtt{id}^?~b^*{:}\mathtt{datastring}~\text{`)'} \\
 & & \quad \Rightarrow \quad \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{passive}\} \\
 & | & \text{`('}~\text{`}\mathtt{data}\text{'}~\mathtt{id}^?~x{:}\mathtt{memuse}_I~\text{`('}~\text{`}\mathtt{offset}\text{'}~e{:}\mathtt{expr}_I~\text{`)'}~b^*{:}\mathtt{datastring}~\text{`)'} \\
 & & \quad \Rightarrow \quad \{\mathsf{init}~b^*, \mathsf{mode}~\mathsf{active}~\{\mathsf{memory}~x', \mathsf{offset}~e\}\} \\
\mathtt{datastring} & ::= & (b^*{:}\mathtt{string})^* \quad \Rightarrow \quad \bigoplus((b^*)^*) \\
\mathtt{memuse}_I & ::= & \text{`('}~\text{`}\mathtt{memory}\text{'}~x{:}\mathtt{memidx}_I~\text{`)'} \quad \Rightarrow \quad x
\end{array}
$$

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

Abbreviations

As an abbreviation, a single instruction may occur in place of the offset of an active data segment:

$$\text{`('}~\mathtt{instr}~\text{`)'} \quad \equiv \quad \text{`('}~\text{`}\mathtt{offset}\text{'}~\mathtt{instr}~\text{`)'}$$

Also, a memory use can be omitted, defaulting to 0.

$$\epsilon \quad \equiv \quad \text{`('}~\text{`}\mathtt{memory}\text{'}~\text{`}\mathtt{0}\text{'}~\text{`)'}$$

As another abbreviation, data segments may also be specified inline with memory definitions; see the respective section.

6.6.12 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.

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

The following restrictions are imposed on the composition of modules: $m_1 \oplus m_2$ is defined if and only if

- $m_1.\mathsf{start} = \epsilon \lor m_2.\mathsf{start} = \epsilon$
- $m_1.\mathsf{funcs} = m_1.\mathsf{tables} = m_1.\mathsf{mems} = m_1.\mathsf{globals} = \epsilon \lor m_2.\mathsf{imports} = \epsilon$

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:

$$
\begin{array}{lcl}
\mathrm{idc}(\text{`('}~\text{`}\mathtt{rec}\text{'}~\mathtt{typedef}^*~\text{`)'}) &=& \bigoplus \mathrm{idc}(\mathtt{typedef})^* \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{type}\text{'}~v^?{:}\mathtt{id}^?~\mathtt{subtype}~\text{`)'}) &=& \{\mathsf{types}~(v^?), \mathsf{fields}~\mathrm{idf}(\mathtt{subtype}), \mathsf{typedefs}~st\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{func}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{funcs}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{table}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{tables}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{memory}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{mems}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{global}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{globals}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{elem}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{elem}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{data}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& \{\mathsf{data}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{func}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) &=& \{\mathsf{funcs}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{table}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) &=& \{\mathsf{tables}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{memory}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) &=& \{\mathsf{mems}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\text{`}\mathtt{import}\text{'}~\dots~\text{`('}~\text{`}\mathtt{global}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}~\text{`)'}) &=& \{\mathsf{globals}~(v^?)\} \\
\mathrm{idc}(\text{`('}~\dots~\text{`)'}) &=& \{\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{idf}(\text{`('}~\text{`}\mathtt{sub}\text{'}~\dots~\mathtt{comptype}~\text{`)'}) &=& \mathrm{idf}(\mathtt{comptype}) \\
\mathrm{idf}(\text{`('}~\text{`}\mathtt{struct}\text{'}~\mathit{Tfield}^*~\text{`)'}) &=& \bigoplus \mathrm{idf}(\mathtt{field})^* \\
\mathrm{idf}(\text{`('}~\text{`}\mathtt{array}\text{'}~\dots~\text{`)'}) &=& \epsilon \\
\mathrm{idf}(\text{`('}~\text{`}\mathtt{func}\text{'}~\dots~\text{`)'}) &=& \epsilon \\
\mathrm{idf}(\text{`('}~\text{`}\mathtt{field}\text{'}~v^?{:}\mathtt{id}^?~\dots~\text{`)'}) &=& v^?
\end{array}
$$

Abbreviations

In a source file, the toplevel $(\mathtt{module}~\dots)$ surrounding the module body may be omitted.

$$
\mathtt{modulefield}^* \quad \equiv \quad \text{`('}~\text{`}\mathtt{module}\text{'}~\mathtt{modulefield}^*~\text{`)'}
$$

CHAPTER 7

Appendix

7.1 Embedding

A WebAssembly implementation will typically be *embedded* into a *host* environment. An *embedder* implements the connection between such a host environment and the WebAssembly semantics as defined in the main body of this specification. An embedder is expected to interact with the semantics in well-defined ways.

This section defines a suitable interface to the WebAssembly semantics in the form of entry points through which an embedder can access it. The interface is intended to be complete, in the sense that an embedder does not need to reference other functional parts of the WebAssembly specification directly.

Note: On the other hand, an embedder does not need to provide the host environment with access to all functionality defined in this interface. For example, an implementation may not support parsing of the text format.

7.1.1 Types

In the description of the embedder interface, syntactic classes from the abstract syntax and the runtime's abstract machine are used as names for variables that range over the possible objects from that class. Hence, these syntactic classes can also be interpreted as types.

For numeric parameters, notation like $n : \mathit{u32}$ is used to specify a symbolic name in addition to the respective value range.

7.1.2 Booleans

Interface operation that are predicates return Boolean values:

$$\mathit{bool} \quad ::= \quad \mathit{false} \mid \mathit{true}$$

7.1.3 Errors

Failure of an interface operation is indicated by an auxiliary syntactic class:

$$error \quad ::= \quad \mathsf{error}$$

In addition to the error conditions specified explicitly in this section, implementations may also return errors when specific implementation limitations are reached.

Note: Errors are abstract and unspecific with this definition. Implementations can refine it to carry suitable classifications and diagnostic messages.

7.1.4 Pre- and Post-Conditions

Some operations state *pre-conditions* about their arguments or *post-conditions* about their results. It is the embedder's responsibility to meet the pre-conditions. If it does, the post conditions are guaranteed by the semantics.

In addition to pre- and post-conditions explicitly stated with each operation, the specification adopts the following conventions for runtime objects ($store$, $moduleinst$, $externval$, addresses):

- Every runtime object passed as a parameter must be valid per an implicit pre-condition.
- Every runtime object returned as a result is valid per an implicit post-condition.

Note: As long as an embedder treats runtime objects as abstract and only creates and manipulates them through the interface defined here, all implicit pre-conditions are automatically met.

7.1.5 Store

$\mathrm{store\_init}() : store$

1. Return the empty store.

$$\mathrm{store\_init}() \quad = \quad \{\mathsf{funcs}~\epsilon,~\mathsf{mems}~\epsilon,~\mathsf{tables}~\epsilon,~\mathsf{globals}~\epsilon\}$$

7.1.6 Modules

$\mathrm{module\_decode}(byte^*) : module \mid error$

1. If there exists a derivation for the byte sequence $byte^*$ as a `module` according to the binary grammar for modules, yielding a module m, then return m.
2. Else, return error.

$$\begin{array}{lcll} \mathrm{module\_decode}(b^*) & = & m & (\text{if } \mathtt{module} \stackrel{*}{\Longrightarrow} m{:}b^*) \\ \mathrm{module\_decode}(b^*) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{module\_parse}(char^*) : module \mid error$

1. If there exists a derivation for the source $char^*$ as a `module` according to the text grammar for modules, yielding a module m, then return m.
2. Else, return error.

$$\begin{array}{lcll} \mathrm{module\_parse}(c^*) & = & m & (\text{if } \mathtt{module} \stackrel{*}{\Longrightarrow} m{:}c^*) \\ \mathrm{module\_parse}(c^*) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{module\_validate}(module) : error^?$

1. If $module$ is valid, then return nothing.
2. Else, return error.

$$\begin{array}{lcll} \mathrm{module\_validate}(m) & = & \epsilon & (\text{if } \vdash m : externtype^* \rightarrow externtype'^*) \\ \mathrm{module\_validate}(m) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{module\_instantiate}(store, module, externval^*) : (store, moduleinst \mid error)$

1. Try instantiating $module$ in $store$ with external values $externval^*$ as imports:
   a. If it succeeds with a module instance $moduleinst$, then let $result$ be $moduleinst$.
   b. Else, let $result$ be error.
2. Return the new store paired with $result$.

$$\begin{array}{lcll} \mathrm{module\_instantiate}(S, m, ev^*) & = & (S', F.\mathsf{module}) & (\text{if } \mathrm{instantiate}(S, m, ev^*) \hookrightarrow^* S'; F; \epsilon) \\ \mathrm{module\_instantiate}(S, m, ev^*) & = & (S', \mathsf{error}) & (\text{if } \mathrm{instantiate}(S, m, ev^*) \hookrightarrow^* S'; F; \mathsf{trap}) \end{array}$$

Note: The store may be modified even in case of an error.

$\mathrm{module\_imports}(module) : (name, name, externtype)^*$

1. Pre-condition: $module$ is valid with the external import types $externtype^*$ and external export types $externtype'^*$.
2. Let $import^*$ be the imports $module.\mathsf{imports}$.
3. Assert: the length of $import^*$ equals the length of $externtype^*$.
4. For each $import_i$ in $import^*$ and corresponding $externtype_i$ in $externtype^*$, do:
   a. Let $result_i$ be the triple $(import_i.\mathsf{module}, import_i.\mathsf{name}, externtype_i)$.
5. Return the concatenation of all $result_i$, in index order.
6. Post-condition: each $externtype_i$ is valid under the empty context.

$$\begin{array}{lcl} \mathrm{module\_imports}(m) & = & (im.\mathsf{module}, im.\mathsf{name}, externtype)^* \\ & & \quad (\text{if } im^* = m.\mathsf{imports} \wedge \vdash m : externtype^* \rightarrow externtype'^*) \end{array}$$

$\mathrm{module\_exports}(module) : (name, externtype)^*$

1. Pre-condition: $module$ is valid with the external import types $externtype^*$ and external export types $externtype'^*$.
2. Let $export^*$ be the exports $module$.exports.
3. Assert: the length of $export^*$ equals the length of $externtype'^*$.
4. For each $export_i$ in $export^*$ and corresponding $externtype'_i$ in $externtype'^*$, do:
a. Let $result_i$ be the pair $(export_i.\mathsf{name}, externtype'_i)$.
5. Return the concatenation of all $result_i$, in index order.
6. Post-condition: each $externtype'_i$ is valid under the empty context.

$$\begin{array}{lcl} \mathrm{module\_exports}(m) & = & (ex.\mathsf{name}, externtype')^* \\ & & \quad (\text{if } ex^* = m.\mathsf{exports} \wedge \vdash m : externtype^* \to externtype'^*) \end{array}$$

7.1.7 Module Instances

$\mathrm{instance\_export}(moduleinst, name) : externval \mid error$

1. Assert: due to validity of the module instance $moduleinst$, all its export names are different.
2. If there exists an $exportinst_i$ in $moduleinst$.exports such that name $exportinst_i$.name equals $name$, then:
   a. Return the external value $exportinst_i$.value.
3. Else, return error.

$$\begin{array}{lclr} \mathrm{instance\_export}(m, name) & = & m.\mathsf{exports}[i].\mathsf{value} & (\text{if } m.\mathsf{exports}[i].\mathsf{name} = name) \\ \mathrm{instance\_export}(m, name) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

7.1.8 Functions

$\mathrm{func\_alloc}(store, functype, hostfunc) : (store, funcaddr)$

1. Pre-condition: the $functype$ is valid under the empty context.
2. Let $funcaddr$ be the result of allocating a host function in $store$ with function type $functype$ and host function code $hostfunc$.
3. Return the new store paired with $funcaddr$.

$$\mathrm{func\_alloc}(S, ta, code) \quad = \quad (S', a) \qquad (\text{if } \mathrm{allocfunc}(S, \{\}, ta, code) = S', a)$$

Note: This operation assumes that $hostfunc$ satisfies the pre- and post-conditions required for a function instance with type $functype$.

Regular (non-host) function instances can only be created indirectly through module instantiation.

$\mathrm{func\_type}(store, funcaddr) : functype$

1. Let $functype$ be the function type $S.\mathsf{funcs}[a].\mathsf{type}$.
2. Return $functype$.
3. Post-condition: the returned function type is valid.

$$\mathrm{func\_type}(S, a) \quad = \quad S.\mathsf{funcs}[a].\mathsf{type}$$

$\mathrm{func\_invoke}(store, funcaddr, val^*) : (store, val^* \mid error)$

1. Try invoking the function $funcaddr$ in $store$ with values val^* as arguments:
   a. If it succeeds with values val'^* as results, then let $result$ be val'^*.
   b. Else it has trapped, hence let $result$ be error.
2. Return the new store paired with $result$.

$$\begin{array}{lcll} \mathrm{func\_invoke}(S, a, v^*) & = & (S', v'^*) & \quad (\text{if } \mathrm{invoke}(S, a, v^*) \hookrightarrow^* S'; F; v'^*) \\ \mathrm{func\_invoke}(S, a, v^*) & = & (S', \mathsf{error}) & \quad (\text{if } \mathrm{invoke}(S, a, v^*) \hookrightarrow^* S'; F; \mathsf{trap}) \end{array}$$

Note: The store may be modified even in case of an error.

7.1.9 Tables

$\mathrm{table\_alloc}(store, tabletype, ref) : (store, tableaddr)$

1. Pre-condition: the $tabletype$ is valid under the empty context.
2. Let $tableaddr$ be the result of allocating a table in $store$ with table type $tabletype$ and initialization value ref.
3. Return the new store paired with $tableaddr$.

$$\mathrm{table\_alloc}(S, tt, r) \quad = \quad (S', a) \qquad (\text{if } \mathrm{alloctable}(S, tt, r) = S', a)$$

$\mathrm{table\_type}(store, tableaddr) : tabletype$

1. Return $S.\mathsf{tables}[a].\mathsf{type}$.
2. Post-condition: the returned table type is valid under the empty context.

$$\mathrm{table\_type}(S, a) \quad = \quad S.\mathsf{tables}[a].\mathsf{type}$$

$\mathrm{table\_read}(store, tableaddr, i : u32) : ref \mid error$

1. Let ti be the table instance $store.\mathsf{tables}[tableaddr]$.
2. If i is larger than or equal to the length of $ti.\mathsf{elem}$, then return error.
3. Else, return the reference value $ti.\mathsf{elem}[i]$.

$$\begin{array}{lcll} \mathrm{table\_read}(S, a, i) & = & r & (\text{if } S.\mathsf{tables}[a].\mathsf{elem}[i] = r) \\ \mathrm{table\_read}(S, a, i) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{table\_write}(store, tableaddr, i : u32, ref) : store \mid error$

1. Let ti be the table instance $store.\mathsf{tables}[tableaddr]$.
2. If i is larger than or equal to the length of $ti.\mathsf{elem}$, then return error.
3. Replace $ti.\mathsf{elem}[i]$ with the reference value ref.
4. Return the updated store.

$$\begin{array}{lcll} \mathrm{table\_write}(S, a, i, r) & = & S' & (\text{if } S' = S \mathrel{\mathsf{with}} \mathsf{tables}[a].\mathsf{elem}[i] = r) \\ \mathrm{table\_write}(S, a, i, r) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{table\_size}(store, tableaddr) : u32$

1. Return the length of $store.\mathsf{tables}[tableaddr].\mathsf{elem}$.

$$\mathrm{table\_size}(S, a) = n \qquad (\text{if } |S.\mathsf{tables}[a].\mathsf{elem}| = n)$$

$\mathrm{table\_grow}(store, tableaddr, n : u32, ref) : store \mid error$

1. Try growing the table instance $store.\mathsf{tables}[tableaddr]$ by n elements with initialization value ref:
   a. If it succeeds, return the updated store.
   b. Else, return error.

$$\begin{array}{lcll} \mathrm{table\_grow}(S, a, n, r) & = & S' & (\text{if } S' = S \mathrel{\mathsf{with}} \mathsf{tables}[a] = \mathrm{growtable}(S.\mathsf{tables}[a], n, r)) \\ \mathrm{table\_grow}(S, a, n, r) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

7.1.10 Memories

$\mathrm{mem\_alloc}(store, memtype) : (store, memaddr)$

1. Pre-condition: the $memtype$ is valid under the empty context.
2. Let $memaddr$ be the result of allocating a memory in $store$ with memory type $memtype$.
3. Return the new store paired with $memaddr$.

$$\mathrm{mem\_alloc}(S, mt) = (S', a) \qquad (\text{if } \mathrm{allocmem}(S, mt) = S', a)$$

$\mathrm{mem\_type}(\mathit{store}, \mathit{memaddr}) : \mathit{memtype}$

1. Return $S.\mathsf{mems}[a].\mathsf{type}$.
2. Post-condition: the returned memory type is valid under the empty context.

$$\mathrm{mem\_type}(S, a) \quad = \quad S.\mathsf{mems}[a].\mathsf{type}$$

$\mathrm{mem\_read}(\mathit{store}, \mathit{memaddr}, i : \mathit{u32}) : \mathit{byte} \mid \mathit{error}$

1. Let mi be the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$.
2. If i is larger than or equal to the length of $mi.\mathsf{bytes}$, then return error.
3. Else, return the byte $mi.\mathsf{bytes}[i]$.

$$\begin{array}{lcll} \mathrm{mem\_read}(S, a, i) & = & b & (\text{if } S.\mathsf{mems}[a].\mathsf{bytes}[i] = b) \\ \mathrm{mem\_read}(S, a, i) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{mem\_write}(\mathit{store}, \mathit{memaddr}, i : \mathit{u32}, \mathit{byte}) : \mathit{store} \mid \mathit{error}$

1. Let mi be the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$.
2. If $\mathit{u32}$ is larger than or equal to the length of $mi.\mathsf{bytes}$, then return error.
3. Replace $mi.\mathsf{bytes}[i]$ with byte.
4. Return the updated store.

$$\begin{array}{lcll} \mathrm{mem\_write}(S, a, i, b) & = & S' & (\text{if } S' = S \text{ with } \mathsf{mems}[a].\mathsf{bytes}[i] = b) \\ \mathrm{mem\_write}(S, a, i, b) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

$\mathrm{mem\_size}(\mathit{store}, \mathit{memaddr}) : \mathit{u32}$

1. Return the length of $\mathit{store}.\mathsf{mems}[\mathit{memaddr}].\mathsf{bytes}$ divided by the page size.

$$\mathrm{mem\_size}(S, a) \quad = \quad n \qquad (\text{if } |S.\mathsf{mems}[a].\mathsf{bytes}| = n \cdot 64\,\mathrm{Ki})$$

$\mathrm{mem\_grow}(\mathit{store}, \mathit{memaddr}, n : \mathit{u32}) : \mathit{store} \mid \mathit{error}$

1. Try growing the memory instance $\mathit{store}.\mathsf{mems}[\mathit{memaddr}]$ by n pages:
   a. If it succeeds, return the updated store.
   b. Else, return error.

$$\begin{array}{lcll} \mathrm{mem\_grow}(S, a, n) & = & S' & (\text{if } S' = S \text{ with } \mathsf{mems}[a] = \mathrm{growmem}(S.\mathsf{mems}[a], n)) \\ \mathrm{mem\_grow}(S, a, n) & = & \mathsf{error} & (\text{otherwise}) \end{array}$$

7.1.11 Globals

$\mathrm{global\_alloc}(\mathit{store}, \mathit{globaltype}, \mathit{val}) : (\mathit{store}, \mathit{globaladdr})$

1. Pre-condition: the $\mathit{globaltype}$ is valid under the empty context.
2. Let $\mathit{globaladdr}$ be the result of allocating a global in store with global type $\mathit{globaltype}$ and initialization value val.
3. Return the new store paired with $\mathit{globaladdr}$.

$$\mathrm{global\_alloc}(S, gt, v) \quad = \quad (S', a) \qquad (\text{if } \mathrm{allocglobal}(S, gt, v) = S', a)$$

$\mathrm{global\_type}(\mathit{store}, \mathit{globaladdr}) : \mathit{globaltype}$

1. Return $S.\mathsf{globals}[a].\mathsf{type}$.
2. Post-condition: the returned global type is valid under the empty context.

$$\mathrm{global\_type}(S, a) \quad = \quad S.\mathsf{globals}[a].\mathsf{type}$$

$\mathrm{global\_read}(\mathit{store}, \mathit{globaladdr}) : \mathit{val}$

1. Let gi be the global instance $\mathit{store}.\mathsf{globals}[\mathit{globaladdr}]$.
2. Return the value $gi.\mathsf{value}$.

$$\mathrm{global\_read}(S, a) \quad = \quad v \qquad (\text{if } S.\mathsf{globals}[a].\mathsf{value} = v)$$

$\mathrm{global\_write}(\mathit{store}, \mathit{globaladdr}, \mathit{val}) : \mathit{store} \mid \mathit{error}$

1. Let gi be the global instance $\mathit{store}.\mathsf{globals}[\mathit{globaladdr}]$.
2. Let $\mathit{mut}\ t$ be the structure of the global type $gi.\mathsf{type}$.
3. If mut is not var, then return error.
4. Replace $gi.\mathsf{value}$ with the value val.
5. Return the updated store.

$$\begin{array}{lcll}
\mathrm{global_write}(S, a, v) & = & S' & (\text{if } S.\mathsf{globals}[a].\mathsf{type} = \mathsf{var}\ t \wedge S' = S \text{ with } \mathsf{globals}[a].\mathsf{value} = v) \\
\mathrm{global_write}(S, a, v) & = & \mathsf{error} & (\text{otherwise})
\end{array}$$

7.1.12 Values

$\mathrm{ref\_type}(\mathit{store}, \mathit{ref}) : \mathit{reftype}$

1. Pre-condition: the reference ref is valid under store S.
2. Return the reference type t with which ref is valid.
3. Post-condition: the returned reference type is valid under the empty context.

$$\mathrm{ref\_type}(S, r) \quad = \quad t \qquad (\text{if } S \vdash r : t)$$

Note: In future versions of WebAssembly, not all references may carry precise type information at run time. In such cases, this function may return a less precise supertype.

$\mathrm{val\_default}(valtype) : val$

1. If $\mathrm{default}_{valtype}$ is not defined, then return error.
1. Else, return the value $\mathrm{default}_{valtype}$.

$$\begin{array}{lcll} \mathrm{val\_default}(t) & = & v & (\text{if } \mathrm{default}_t = v) \\ \mathrm{val\_default}(t) & = & \mathsf{error} & (\text{if } \mathrm{default}_t = \epsilon) \end{array}$$

7.1.13 Matching

$\mathrm{match\_valtype}(valtype_1, valtype_2) : bool$

1. Pre-condition: the value types $valtype_1$ and $valtype_2$ are valid under the empty context.
2. If $valtype_1$ matches $valtype_2$, then return $true$.
3. Else, return $false$.

$$\begin{array}{lcll} \mathrm{match\_reftype}(t_1, t_2) & = & true & (\text{if } \vdash t_1 \leq t_2) \\ \mathrm{match\_reftype}(t_1, t_2) & = & false & (\text{otherwise}) \end{array}$$

$\mathrm{match\_externtype}(externtype_1, externtype_2) : bool$

1. Pre-condition: the extern types $externtype_1$ and $externtype_2$ are valid under the empty context.
2. If $externtype_1$ matches $externtype_2$, then return $true$.
3. Else, return $false$.

$$\begin{array}{lcll} \mathrm{match\_externtype}(et_1, et_2) & = & true & (\text{if } \vdash et_1 \leq et_2) \\ \mathrm{match\_externtype}(et_1, et_2) & = & false & (\text{otherwise}) \end{array}$$

7.2 Implementation Limitations

Implementations typically impose additional restrictions on a number of aspects of a WebAssembly module or execution. These may stem from:

- physical resource limits,
- constraints imposed by the embedder or its environment,
- limitations of selected implementation strategies.

This section lists allowed limitations. Where restrictions take the form of numeric limits, no minimum requirements are given, nor are the limits assumed to be concrete, fixed numbers. However, it is expected that all implementations have “reasonably” large limits to enable common applications.

Note: A conforming implementation is not allowed to leave out individual *features*. However, designated subsets of WebAssembly may be specified in the future.

7.2.1 Syntactic Limits

Structure

An implementation may impose restrictions on the following dimensions of a module:

- the number of types in a module
- the number of functions in a module, including imports
- the number of tables in a module, including imports
- the number of memories in a module, including imports
- the number of globals in a module, including imports
- the number of element segments in a module
- the number of data segments in a module
- the number of imports to a module
- the number of exports from a module
- the number of sub types in a recursive type
- the subtyping depth of a sub type
- the number of fields in a structure type
- the number of parameters in a function type
- the number of results in a function type
- the number of parameters in a block type
- the number of results in a block type
- the number of locals in a function
- the number of instructions in a function body
- the number of instructions in 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 number of instructions in a constant expression
- the length of the array in a array.new_fixed instruction
- the length of an element segment
- the length of a data segment
- the length of a name
- the range of characters in a name

If the limits of an implementation are exceeded for a given module, then the implementation may reject the validation, compilation, or instantiation of that module with an embedder-specific error.

Note: The last item allows embedders that operate in limited environments without support for Unicode[48] to limit the names of imports and exports to common subsets like ASCII[49].

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

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 size of a structured control instruction
- the size of an individual constant expression's instruction sequence
- the number of sections

Text Format

For a module given in text format, additional limitations may be imposed on the following dimensions:

- the size of the source text
- the size of any syntactic element
- the size of an individual token
- the nesting depth of folded instructions
- the length of symbolic identifiers
- the range of literal characters allowed in the source text

7.2.2 Validation

An implementation may defer validation of individual functions until they are first invoked.

If a function turns out to be invalid, then the invocation, and every consecutive call to the same function, results in a trap.

Note: This is to allow implementations to use interpretation or just-in-time compilation for functions. The function must still be fully validated before execution of its body begins.

7.2.3 Execution

Restrictions on the following dimensions may be imposed during execution of a WebAssembly program:

- the number of allocated module instances
- the number of allocated function instances
- the number of allocated table instances
- the number of allocated memory instances
- the number of allocated global instances
- the number of allocated structure instances
- the number of allocated array instances
- the size of a table instance
- the size of a memory instance
- the size of an array 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.

7.3 Type Soundness

The type system of WebAssembly is *sound*, implying both *type safety* and *memory safety* with respect to the WebAssembly semantics. For example:

- All types declared and derived during validation are respected at run time; e.g., every local or global variable will only contain type-correct values, every instruction will only be applied to operands of the expected type, and every function invocation always evaluates to a result of the right type (if it does not trap or diverge).
- No memory location will be read or written except those explicitly defined by the program, i.e., as a local, a global, an element in a table, or a location within a linear memory.
- There is no undefined behavior, i.e., the execution rules cover all possible cases that can occur in a valid program, and the rules are mutually consistent.

Soundness also is instrumental in ensuring additional properties, most notably, *encapsulation* of function and module scopes: no locals can be accessed outside their own function and no module components can be accessed outside their own module unless they are explicitly exported or imported.

The typing rules defining WebAssembly validation only cover the *static* components of a WebAssembly program. In order to state and prove soundness precisely, the typing rules must be extended to the *dynamic* components of the abstract runtime, that is, the store, configurations, and administrative instructions.[50]

7.3.1 Contexts

In order to check rolled up recursive types, the context is locally extended with an additional component that records the sub type corresponding to each recursive type index within the current recursive type:

$$C \quad ::= \quad \{ \ldots, \mathsf{recs}\ subtype^* \}$$

[50] The formalization and theorems are derived from the following article: Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman. Bringing the Web up to Speed with WebAssembly[Page 258, 51]. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM 2017.

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

7.3.2 Types

Well-formedness for extended type forms is defined as follows.

Heap Type bot

- The heap type is valid.

$$\frac{}{C \vdash \mathsf{bot} : \mathsf{ok}}$$

Heap Type rec i

- The recursive type index i must exist in $C.\mathsf{recs}$.
- Then the heap type is valid.

$$\frac{C.\mathsf{recs}[i] = \mathit{subtype}}{C \vdash \mathsf{rec}\ i : \mathsf{ok}}$$

Value Type bot

- The value type is valid.

$$\frac{}{C \vdash \mathsf{bot} : \mathsf{ok}}$$

Recursive Types rec $\mathit{subtype}^*$

- Let C' be the current context C, but where recs is $\mathit{subtype}^*$.
- There must be a type index x, such that for each sub type $\mathit{subtype}_i$ in $\mathit{subtype}^*$:
  - Under the context C', the sub type $\mathit{subtype}_i$ must be valid for type index $x + i$ and recursive type index i.
- Then the recursive type is valid for the type index x.

$$\frac{C, \mathsf{recs}\ \mathit{subtype}^* \vdash \mathsf{rec}\ \mathit{subtype}^* : \mathsf{ok}(x, 0)}{C \vdash \mathsf{rec}\ \mathit{subtype}^* : \mathsf{ok}(x)}$$

$$\frac{}{C \vdash \mathsf{rec}\ \epsilon : \mathsf{ok}(x, i)} \qquad \frac{C \vdash \mathit{subtype} : \mathsf{ok}(x, i) \qquad C \vdash \mathsf{rec}\ \mathit{subtype}'^* : \mathsf{ok}(x + 1, i + 1)}{C \vdash \mathsf{rec}\ \mathit{subtype}\ \mathit{subtype}'^* : \mathsf{ok}(x, i)}$$

Note: These rules are a generalisation of the ones previously given.

Sub types sub final$^?$ ht^* $\mathit{comptype}$

- The composite type $\mathit{comptype}$ must be valid.
- The sequence ht^* may be no longer than 1.
- For every heap type ht_k in ht^*:
  - The heap type ht_k must be ordered before a type index x and recursive type index a i, meaning:
    - Either ht_k is a defined type.
    - Or ht_k is a type index y_k that is smaller than x.
    - Or ht_k is a recursive type index rec j_k where j_k is smaller than i.

  - Let sub type $subtype_k$ be the unrolling of the heap type ht_k, meaning:
    * Either ht_k is a defined type $deftype_k$, then $subtype_k$ must be the unrolling of $deftype_k$.
    * Or ht_k is a type index y_k, then $subtype_k$ must be the unrolling of the defined type $C.\mathsf{types}[y_k]$.
    * Or ht_k is a recursive type index $\mathsf{rec}\ j_k$, then $subtype_k$ must be $C.\mathsf{recs}[j_k]$.
  - The sub type $subtype_k$ must not contain final.
  - Let $comptype'_k$ be the composite type in $subtype_k$.
  - The composite type $comptype$ must match $comptype'_k$.
- Then the sub type is valid for the type index x and recursive type index i.

$$\frac{|ht^*| \leq 1 \qquad (ht \prec x, i)^* \qquad (\mathrm{unroll}_C(ht) = \mathsf{sub}\ ht'^*\ comptype')^* \qquad C \vdash comptype : \mathsf{ok} \qquad (C \vdash comptype \leq comptype')^*}{C \vdash \mathsf{sub}\ \mathsf{final}^?\ ht^*\ comptype : \mathsf{ok}(x, i)}$$

where:

$$\begin{array}{lcl} (deftype \prec x, i) & = & \mathrm{true} \\ (y \prec x, i) & = & y < x \\ (\mathsf{rec}\ j \prec x, i) & = & j < i \\ \\ \mathrm{unroll}_C(deftype) & = & \mathrm{unroll}(deftype) \\ \mathrm{unroll}_C(y) & = & \mathrm{unroll}(C.\mathsf{types}[y]) \\ \mathrm{unroll}_C(\mathsf{rec}\ j) & = & C.\mathsf{recs}[j] \end{array}$$

Note: This rule is a generalisation of the ones previously given, which only allowed type indices as supertypes.

7.3.3 Subtyping

In a rolled-up recursive type, a recursive type indices $\mathsf{rec}\ i$ matches another heap type ht if:

- Let $\mathsf{sub}\ \mathsf{final}^?\ ht'^*\ comptype$ be the sub type $C.\mathsf{recs}[i]$.
- The heap type ht is contained in ht'^*.

$$\frac{C.\mathsf{recs}[i] = \mathsf{sub}\ \mathsf{final}^?\ (ht_1^*\ ht\ ht_2^*)\ comptype}{C \vdash \mathsf{rec}\ i \leq ht}$$

Note: This rule is only invoked when checking validity of rolled-up recursive types.

7.3.4 Results

Results can be classified by result types as follows.

Results val^*

- For each value val_i in val^*:
  - The value val_i is valid with some value type t_i.
- Let t^* be the concatenation of all t_i.
- Then the result is valid with result type $[t^*]$.

$$\frac{(S \vdash val : t)^*}{S \vdash val^* : [t^*]}$$

Results trap

- The result is valid with result type $[t^*]$, for any valid closed result types.

$$\frac{\vdash [t^*] : \mathsf{ok}}{S \vdash \mathsf{trap} : [t^*]}$$

7.3.5 Store Validity

The following typing rules specify when a runtime store S is *valid*. A valid store must consist of function, table, memory, global, and module instances that are themselves valid, relative to S.

To that end, each kind of instance is classified by a respective function, table, memory, or global type. Module instances are classified by *module contexts*, which are regular contexts repurposed as module types describing the index spaces defined by a module.

Store S

- Each function instance $funcinst_i$ in $S.\mathsf{funcs}$ must be valid with some function type $functype_i$.
- Each table instance $tableinst_i$ in $S.\mathsf{tables}$ must be valid with some table type $tabletype_i$.
- Each memory instance $meminst_i$ in $S.\mathsf{mems}$ must be valid with some memory type $memtype_i$.
- Each global instance $globalinst_i$ in $S.\mathsf{globals}$ must be valid with some global type $globaltype_i$.
- Each element instance $eleminst_i$ in $S.\mathsf{elems}$ must be valid with some reference type $reftype_i$.
- Each data instance $datainst_i$ in $S.\mathsf{datas}$ must be valid.
- Each structure instance $structinst_i$ in $S.\mathsf{structs}$ must be valid.
- Each array instance $arrayinst_i$ in $S.\mathsf{arrays}$ must be valid.
- No reference to a bound structure address must be reachable from itself through a path consisting only of indirections through immutable structure or array fields.
- No reference to a bound array address must be reachable from itself through a path consisting only of indirections through immutable structure or array fields.
- Then the store is valid.

$$\frac{\begin{array}{c} (S \vdash funcinst : deftype)^* \qquad (S \vdash tableinst : tabletype)^* \\ (S \vdash meminst : memtype)^* \qquad (S \vdash globalinst : globaltype)^* \\ (S \vdash eleminst : reftype)^* \qquad (S \vdash datainst : \mathsf{ok})^* \\ (S \vdash structinst : \mathsf{ok})^* \qquad (S \vdash arrayinst : \mathsf{ok})^* \\ S = \{\mathsf{funcs}\ funcinst^*, \mathsf{globals}\ globalinst^*, \mathsf{tables}\ tableinst^*, \mathsf{mems}\ meminst^*, \\ \quad \mathsf{elems}\ eleminst^*, \mathsf{datas}\ datainst^*, \mathsf{structs}\ structinst^*, \mathsf{arrays}\ arrayinst^*\} \\ (S.\mathsf{structs}[a_{\mathrm{s}}] = structinst)^* \qquad ((\mathsf{ref.struct}\ a_{\mathrm{s}}) \not\gg^+_S (\mathsf{ref.struct}\ a_{\mathrm{s}}))^* \\ (S.\mathsf{arrays}[a_{\mathrm{a}}] = arrayinst)^* \qquad ((\mathsf{ref.array}\ a_{\mathrm{a}}) \not\gg^+_S (\mathsf{ref.array}\ a_{\mathrm{a}}))^* \end{array}}{\vdash S : \mathsf{ok}}$$

where $val_1 \gg^+_S val_2$ denotes the transitive closure of the following *reachability* relation on values:

$$\begin{array}{llll} (\mathsf{ref.struct}\ a) & \gg_S & S.\mathsf{structs}[a].\mathsf{fields}[i] & \text{if } \mathrm{expand}(S.\mathsf{structs}[a].\mathsf{type}) = \mathsf{struct}\ ft_1^i\ (\mathsf{const}\ st)\ ft_2^* \\ (\mathsf{ref.array}\ a) & \gg_S & S.\mathsf{arrays}[a].\mathsf{fields}[i] & \text{if } \mathrm{expand}(S.\mathsf{arrays}[a].\mathsf{type}) = \mathsf{array}\ (\mathsf{const}\ st) \\ (\mathsf{ref.extern}\ ref) & \gg_S & ref & \end{array}$$

Note: The constraint on reachability through immutable fields prevents the presence of cyclic data structures that can not be constructed in the language. Cycles can only be formed using mutation.

Function Instances $\{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\}$

- The function type $\mathit{functype}$ must be valid under an empty context.
- The module instance $\mathit{moduleinst}$ must be valid with some context C.
- Under context C:
  - The function func must be valid with some function type $\mathit{functype}'$.
  - The function type $\mathit{functype}'$ must match $\mathit{functype}$.
- Then the function instance is valid with function type $\mathit{functype}$.

$$\frac{\vdash \mathit{functype} : \mathsf{ok} \qquad S \vdash \mathit{moduleinst} : C \qquad C \vdash \mathit{func} : \mathit{functype}' \qquad C \vdash \mathit{functype}' \leq \mathit{functype}}{S \vdash \{\mathsf{type}~\mathit{functype}, \mathsf{module}~\mathit{moduleinst}, \mathsf{code}~\mathit{func}\} : \mathit{functype}}$$

Host Function Instances $\{\mathsf{type}~\mathit{functype}, \mathsf{hostfunc}~\mathit{hf}\}$

- The function type $\mathit{functype}$ must be valid under an empty context.
- Let $[t_1^*] \to [t_2^*]$ be the function type $\mathit{functype}$.
- For every valid store S_1 extending S and every sequence val^* of values whose types coincide with t_1^*:
  - Executing hf in store S_1 with arguments val^* has a non-empty set of possible outcomes.
  - For every element R of this set:
    * Either R must be $\bot$ (i.e., divergence).
    * Or R consists of a valid store S_2 extending S_1 and a result result whose type coincides with $[t_2^*]$.
- Then the function instance is valid with function type $\mathit{functype}$.

$$\frac{\vdash [t_1^*] \to [t_2^*] : \mathsf{ok} \qquad \begin{array}{l} \forall S_1, \mathit{val}^*, \vdash S_1 : \mathsf{ok} \wedge \vdash S \preceq S_1 \wedge S_1 \vdash \mathit{val}^* : [t_1^*] \Longrightarrow \\ \quad \mathit{hf}(S_1; \mathit{val}^*) \supset \emptyset \wedge \\ \quad \forall R \in \mathit{hf}(S_1; \mathit{val}^*),~ R = \bot \vee \\ \qquad \exists S_2, \mathit{result}, \vdash S_2 : \mathsf{ok} \wedge \vdash S_1 \preceq S_2 \wedge S_2 \vdash \mathit{result} : [t_2^*] \wedge R = (S_2; \mathit{result}) \end{array}}{S \vdash \{\mathsf{type}~[t_1^*] \to [t_2^*], \mathsf{hostfunc}~\mathit{hf}\} : [t_1^*] \to [t_2^*]}$$

Note: This rule states that, if appropriate pre-conditions about store and arguments are satisfied, then executing the host function must satisfy appropriate post-conditions about store and results. The post-conditions match the ones in the execution rule for invoking host functions.

Any store under which the function is invoked is assumed to be an extension of the current store. That way, the function itself is able to make sufficient assumptions about future stores.

Table Instances $\{\mathsf{type}~(\mathit{limits}~t), \mathsf{elem}~\mathit{ref}^*\}$

- The table type $\mathit{limits}~t$ must be valid under the empty context.
- The length of ref^* must equal $\mathit{limits}.\mathsf{min}$.
- For each reference ref_i in the table's elements ref^n:
  - The reference ref_i must be valid with some reference type t_i'.
  - The reference type t_i' must match the reference type t.
- Then the table instance is valid with table type $\mathit{limits}~t$.

$$\frac{\vdash \mathit{limits}\ t : \mathsf{ok} \qquad n = \mathit{limits}.\mathsf{min} \qquad (S \vdash \mathit{ref} : t')^n \qquad (\vdash t' \leq t)^n}{S \vdash \{\mathsf{type}\ (\mathit{limits}\ t), \mathsf{elem}\ \mathit{ref}^n\} : \mathit{limits}\ t}$$

Memory Instances $\{\mathsf{type}\ \mathit{limits}, \mathsf{bytes}\ b^*\}$

- The memory type limits must be valid under the empty context.
- The length of b^* must equal $\mathit{limits}.\mathsf{min}$ multiplied by the page size $64\,\mathrm{Ki}$.
- Then the memory instance is valid with memory type limits.

$$\frac{\vdash \mathit{limits} : \mathsf{ok} \qquad n = \mathit{limits}.\mathsf{min} \cdot 64\,\mathrm{Ki}}{S \vdash \{\mathsf{type}\ \mathit{limits}, \mathsf{bytes}\ b^n\} : \mathit{limits}}$$

Global Instances $\{\mathsf{type}\ (\mathit{mut}\ t), \mathsf{value}\ \mathit{val}\}$

- The global type $\mathit{mut}\ t$ must be valid under the empty context.
- The value val must be valid with some value type t'.
- The value type t' must match the value type t.
- Then the global instance is valid with global type $\mathit{mut}\ t$.

$$\frac{\vdash \mathit{mut}\ t : \mathsf{ok} \qquad S \vdash \mathit{val} : t' \qquad \vdash t' \leq t}{S \vdash \{\mathsf{type}\ (\mathit{mut}\ t), \mathsf{value}\ \mathit{val}\} : \mathit{mut}\ t}$$

Element Instances $\{\mathsf{type}\ t, \mathsf{elem}\ \mathit{ref}^*\}$

- The reference type t must be valid under the empty context.
- For each reference ref_i in the elements ref^n:
  - The reference ref_i must be valid with some reference type t'_i.
  - The reference type t'_i must match the reference type t.
- Then the element instance is valid with reference type t.

$$\frac{\vdash t : \mathsf{ok} \qquad (S \vdash \mathit{ref} : t')^* \qquad (\vdash t' \leq t)^*}{S \vdash \{\mathsf{type}\ t, \mathsf{elem}\ \mathit{ref}^*\} : t}$$

Data Instances $\{\mathsf{bytes}\ b^*\}$

- The data instance is valid.

$$\frac{}{S \vdash \{\mathsf{bytes}\ b^*\} : \mathsf{ok}}$$

Structure Instances $\{\mathsf{type}\ \mathit{deftype}, \mathsf{fields}\ \mathit{fieldval}^*\}$

- The defined type $\mathit{deftype}$ must be valid.
- The expansion of $\mathit{deftype}$ must be a structure type $\mathsf{struct}\ \mathit{fieldtype}^*$.
- The length of the sequence of field values $\mathit{fieldval}^*$ must be the same as the length of the sequence of field types $\mathit{fieldtype}^*$.
- For each field value $\mathit{fieldval}_i$ in $\mathit{fieldval}^*$ and corresponding field type $\mathit{fieldtype}_i$ in $\mathit{fieldtype}^*$:
  - Let $\mathit{fieldtype}_i$ be $\mathit{mut}\ \mathit{storagetype}_i$.
  - The field value $\mathit{fieldval}_i$ must be valid with storage type $\mathit{storagetype}_i$.
- Then the structure instance is valid.

$$\frac{\vdash \mathit{dt} : \mathsf{ok} \qquad \mathrm{expand}(\mathit{dt}) = \mathsf{struct}\ (\mathit{mut}\ \mathit{st})^* \qquad (S \vdash \mathit{fv} : \mathit{st})^*}{S \vdash \{\mathsf{type}\ \mathit{dt}, \mathsf{fields}\ \mathit{fv}^*\} : \mathsf{ok}}$$

Array Instances $\{\mathsf{type}\ \mathit{deftype}, \mathsf{fields}\ \mathit{fieldval}^*\}$

- The defined type $\mathit{deftype}$ must be valid.
- The expansion of $\mathit{deftype}$ must be an array type $\mathsf{array}\ \mathit{fieldtype}$.
- Let $\mathit{fieldtype}$ be $\mathit{mut}\ \mathit{storagetype}$.
- For each field value $\mathit{fieldval}_i$ in $\mathit{fieldval}^*$:
  - The field value $\mathit{fieldval}_i$ must be valid with storage type $\mathit{storagetype}$.
- Then the array instance is valid.

$$\frac{\vdash \mathit{dt} : \mathsf{ok} \qquad \mathrm{expand}(\mathit{dt}) = \mathsf{array}\ (\mathit{mut}\ \mathit{st}) \qquad (S \vdash \mathit{fv} : \mathit{st})^*}{S \vdash \{\mathsf{type}\ \mathit{dt}, \mathsf{fields}\ \mathit{fv}^*\} : \mathsf{ok}}$$

Field Values $\mathit{fieldval}$

- If $\mathit{fieldval}$ is a value val, then:
  - The value val must be valid with value type t.
  - Then the field value is valid with value type t.
- Else, $\mathit{fieldval}$ is a packed value $\mathit{packval}$:
  - Let $\mathit{packtype}.\mathsf{pack}\ i$ be the field value $\mathit{fieldval}$.
  - Then the field value is valid with packed type $\mathit{packtype}$.

$$\frac{}{S \vdash \mathit{pt}.\mathsf{pack}\ i : \mathit{pt}}$$

Export Instances $\{\mathsf{name}~\mathit{name}, \mathsf{value}~\mathit{externval}\}$

- The external value $\mathit{externval}$ must be valid with some external type $\mathit{externtype}$.
- Then the export instance is valid.

$$\frac{S \vdash \mathit{externval} : \mathit{externtype}}{S \vdash \{\mathsf{name}~\mathit{name}, \mathsf{value}~\mathit{externval}\} : \mathsf{ok}}$$

Module Instances $\mathit{moduleinst}$

- Each defined type $\mathit{deftype}_i$ in $\mathit{moduleinst}.\mathsf{types}$ must be valid under the empty context.
- For each function address $\mathit{funcaddr}_i$ in $\mathit{moduleinst}.\mathsf{funcs}$, the external value $\mathsf{func}~\mathit{funcaddr}_i$ must be valid with some external type $\mathsf{func}~\mathit{functype}_i$.
- For each table address $\mathit{tableaddr}_i$ in $\mathit{moduleinst}.\mathsf{tables}$, the external value $\mathsf{table}~\mathit{tableaddr}_i$ must be valid with some external type $\mathsf{table}~\mathit{tabletype}_i$.
- For each memory address $\mathit{memaddr}_i$ in $\mathit{moduleinst}.\mathsf{mems}$, the external value $\mathsf{mem}~\mathit{memaddr}_i$ must be valid with some external type $\mathsf{mem}~\mathit{memtype}_i$.
- For each global address $\mathit{globaladdr}_i$ in $\mathit{moduleinst}.\mathsf{globals}$, the external value $\mathsf{global}~\mathit{globaladdr}_i$ must be valid with some external type $\mathsf{global}~\mathit{globaltype}_i$.
- For each element address $\mathit{elemaddr}_i$ in $\mathit{moduleinst}.\mathsf{elems}$, the element instance $S.\mathsf{elems}[\mathit{elemaddr}_i]$ must be valid with some reference type $\mathit{reftype}_i$.
- For each data address $\mathit{dataaddr}_i$ in $\mathit{moduleinst}.\mathsf{datas}$, the data instance $S.\mathsf{datas}[\mathit{dataaddr}_i]$ must be valid with ok_i.
- Each export instance $\mathit{exportinst}_i$ in $\mathit{moduleinst}.\mathsf{exports}$ must be valid.
- For each export instance $\mathit{exportinst}_i$ in $\mathit{moduleinst}.\mathsf{exports}$, the name $\mathit{exportinst}_i.\mathsf{name}$ must be different from any other name occurring in $\mathit{moduleinst}.\mathsf{exports}$.
- Let $\mathit{deftype}^*$ be the concatenation of all $\mathit{deftype}_i$ in order.
- Let $\mathit{functype}^*$ be the concatenation of all $\mathit{functype}_i$ in order.
- Let $\mathit{tabletype}^*$ be the concatenation of all $\mathit{tabletype}_i$ in order.
- Let $\mathit{memtype}^*$ be the concatenation of all $\mathit{memtype}_i$ in order.
- Let $\mathit{globaltype}^*$ be the concatenation of all $\mathit{globaltype}_i$ in order.
- Let $\mathit{reftype}^*$ be the concatenation of all $\mathit{reftype}_i$ in order.
- Let ok^* be the concatenation of all ok_i in order.
- Let m be the length of $\mathit{moduleinst}.\mathsf{funcs}$.
- Let x^* be the sequence of function indices from 0 to $m - 1$.
- Then the module instance is valid with context $\{\mathsf{types}~\mathit{deftype}^*, \mathsf{funcs}~\mathit{functype}^*, \mathsf{tables}~\mathit{tabletype}^*, \mathsf{mems}~\mathit{memtype}^*, \mathsf{globals}~\mathit{globaltype}^*, \mathsf{elems}~\mathit{reftype}^*, \mathsf{datas}~\mathit{ok}^*, \mathsf{refs}~x^*\}$.

$$\frac{\begin{array}{c}(\vdash \mathit{deftype} : \mathsf{ok})^* \\ (S \vdash \mathsf{func}\ \mathit{funcaddr} : \mathsf{func}\ \mathit{functype})^* \qquad (S \vdash \mathsf{table}\ \mathit{tableaddr} : \mathsf{table}\ \mathit{tabletype})^* \\ (S \vdash \mathsf{mem}\ \mathit{memaddr} : \mathsf{mem}\ \mathit{memtype})^* \qquad (S \vdash \mathsf{global}\ \mathit{globaladdr} : \mathsf{global}\ \mathit{globaltype})^* \\ (S \vdash S.\mathsf{elems}[\mathit{elemaddr}] : \mathit{reftype})^* \qquad (S \vdash S.\mathsf{datas}[\mathit{dataaddr}] : \mathit{ok})^* \\ (S \vdash \mathit{exportinst} : \mathsf{ok})^* \qquad (\mathit{exportinst}.\mathsf{name})^*\ \text{disjoint}\end{array}}{\begin{array}{l}S \vdash \{\begin{array}[t]{ll}\mathsf{types} & \mathit{deftype}^*, \\ \mathsf{funcs} & \mathit{funcaddr}^*, \\ \mathsf{tables} & \mathit{tableaddr}^*, \\ \mathsf{mems} & \mathit{memaddr}^*, \\ \mathsf{globals} & \mathit{globaladdr}^*, \\ \mathsf{elems} & \mathit{elemaddr}^*, \\ \mathsf{datas} & \mathit{dataaddr}^*, \\ \mathsf{exports} & \mathit{exportinst}^*\ \} : \{\begin{array}[t]{ll}\mathsf{types} & \mathit{deftype}^*, \\ \mathsf{funcs} & \mathit{functype}^*, \\ \mathsf{tables} & \mathit{tabletype}^*, \\ \mathsf{mems} & \mathit{memtype}^*, \\ \mathsf{globals} & \mathit{globaltype}^*, \\ \mathsf{elems} & \mathit{reftype}^*, \\ \mathsf{datas} & \mathit{ok}^*, \\ \mathsf{refs} & 0 \dots (|\mathit{funcaddr}^*| - 1)\ \}\end{array}\end{array}\end{array}}$$

7.3.6 Configuration Validity

To relate the WebAssembly type system to its execution semantics, the typing rules for instructions must be extended to configurations $S; T$, which relates the store to execution threads.

Configurations and threads are classified by their result type. In addition to the store S, threads are typed under a *return type* $\mathit{resulttype}^?$, which controls whether and with which type a return instruction is allowed. This type is absent (ϵ) except for instruction sequences inside an administrative frame instruction.

Finally, frames are classified with *frame contexts*, which extend the module contexts of a frame's associated module instance with the locals that the frame contains.

Configurations $S; T$

- The store S must be valid.
- Under no allowed return type, the thread T must be valid with some result type $[t^*]$.
- Then the configuration is valid with the result type $[t^*]$.

$$\frac{\vdash S : \mathsf{ok} \qquad S; \epsilon \vdash T : [t^*]}{\vdash S; T : [t^*]}$$

Threads $F; \mathit{instr}^*$

- Let $\mathit{resulttype}^?$ be the current allowed return type.
- The frame F must be valid with a context C.
- Let C' be the same context as C, but with return set to $\mathit{resulttype}^?$.
- Under context C', the instruction sequence instr^* must be valid with some type $[] \to [t^*]$.
- Then the thread is valid with the result type $[t^*]$.

$$\frac{S \vdash F : C \qquad S; C, \mathsf{return}\ \mathit{resulttype}^? \vdash \mathit{instr}^* : [] \to [t^*]}{S; \mathit{resulttype}^? \vdash F; \mathit{instr}^* : [t^*]}$$

Frames $\{\mathsf{locals}~\mathit{val}^*, \mathsf{module}~\mathit{moduleinst}\}$

- The module instance $\mathit{moduleinst}$ must be valid with some module context C.
- Each value val_i in val^* must be valid with some value type t_i.
- Let t^* be the concatenation of all t_i in order.
- Let C' be the same context as C, but with the value types t^* prepended to the locals list.
- Then the frame is valid with frame context C'.

$$\frac{S \vdash \mathit{moduleinst} : C \qquad (S \vdash \mathit{val} : t)^*}{S \vdash \{\mathsf{locals}~\mathit{val}^*, \mathsf{module}~\mathit{moduleinst}\} : (C, \mathsf{locals}~t^*)}$$

7.3.7 Administrative Instructions

Typing rules for administrative instructions are specified as follows. In addition to the context C, typing of these instructions is defined under a given store S.

To that end, all previous typing judgements $C \vdash \mathit{prop}$ are generalized to include the store, as in $S; C \vdash \mathit{prop}$, by implicitly adding S to all rules – S is never modified by the pre-existing rules, but it is accessed in the extra rules for administrative instructions given below.

trap

- The instruction is valid with any valid instruction type of the form $[t_1^*] \to [t_2^*]$.

$$\frac{C \vdash [t_1^*] \to [t_2^*] : \mathsf{ok}}{S; C \vdash \mathsf{trap} : [t_1^*] \to [t_2^*]}$$

val

- The value val must be valid with value type t.
- Then it is valid as an instruction with type $[] \to [t]$.

$$\frac{S \vdash \mathit{val} : t}{S; C \vdash \mathit{val} : [] \to [t]}$$

$\mathsf{invoke}~\mathit{funcaddr}$

- The external function value $\mathsf{func}~\mathit{funcaddr}$ must be valid with external function type $\mathsf{func} \mathit{functype}'$.
- Let $[t_1^*] \to [t_2^*])$ be the function type $\mathit{functype}$.
- Then the instruction is valid with type $[t_1^*] \to [t_2^*]$.

$$\frac{S \vdash \mathsf{func}~\mathit{funcaddr} : \mathsf{func}~[t_1^*] \to [t_2^*]}{S; C \vdash \mathsf{invoke}~\mathit{funcaddr} : [t_1^*] \to [t_2^*]}$$

$\mathsf{label}_n\{instr_0^*\}\ instr^*\ \mathsf{end}$

- The instruction sequence $instr_0^*$ must be valid with some type $[t_1^n] \rightarrow_{x^*} [t_2^*]$.
- Let C' be the same context as C, but with the result type $[t_1^n]$ prepended to the labels list.
- Under context C', the instruction sequence $instr^*$ must be valid with type $[] \rightarrow_{x'^*} [t_2^*]$.
- Then the compound instruction is valid with type $[] \rightarrow [t_2^*]$.

$$\frac{S; C \vdash instr_0^* : [t_1^n] \rightarrow_{x^*} [t_2^*] \qquad S; C, \mathsf{labels}\,[t_1^n] \vdash instr^* : [] \rightarrow_{x'^*} [t_2^*]}{S; C \vdash \mathsf{label}_n\{instr_0^*\}\ instr^*\ \mathsf{end} : [] \rightarrow [t_2^*]}$$

$\mathsf{frame}_n\{F\}\ instr^*\ \mathsf{end}$

- Under the valid return type $[t^n]$, the thread $F; instr^*$ must be valid with result type $[t^n]$.
- Then the compound instruction is valid with type $[] \rightarrow [t^n]$.

$$\frac{C \vdash [t^n] : \mathsf{ok} \qquad S; [t^n] \vdash F; instr^* : [t^n]}{S; C \vdash \mathsf{frame}_n\{F\}\ instr^*\ \mathsf{end} : [] \rightarrow [t^n]}$$

7.3.8 Store Extension

Programs can mutate the store and its contained instances. Any such modification must respect certain invariants, such as not removing allocated instances or changing immutable definitions. While these invariants are inherent to the execution semantics of WebAssembly instructions and modules, host functions do not automatically adhere to them. Consequently, the required invariants must be stated as explicit constraints on the invocation of host functions. Soundness only holds when the embedder ensures these constraints.

The necessary constraints are codified by the notion of store *extension*: a store state S' extends state S, written $S \preceq S'$, when the following rules hold.

Note: Extension does not imply that the new store is valid, which is defined separately above.

Store S

- The length of $S.\mathsf{funcs}$ must not shrink.
- The length of $S.\mathsf{tables}$ must not shrink.
- The length of $S.\mathsf{mems}$ must not shrink.
- The length of $S.\mathsf{globals}$ must not shrink.
- The length of $S.\mathsf{elems}$ must not shrink.
- The length of $S.\mathsf{datas}$ must not shrink.
- The length of $S.\mathsf{structs}$ must not shrink.
- The length of $S.\mathsf{arrays}$ must not shrink.
- For each function instance $funcinst_i$ in the original $S.\mathsf{funcs}$, the new function instance must be an extension of the old.
- For each table instance $tableinst_i$ in the original $S.\mathsf{tables}$, the new table instance must be an extension of the old.
- For each memory instance $meminst_i$ in the original $S.\mathsf{mems}$, the new memory instance must be an extension of the old.

- For each global instance $globalinst_i$ in the original S.globals, the new global instance must be an extension of the old.
- For each element instance $eleminst_i$ in the original S.elems, the new element instance must be an extension of the old.
- For each data instance $datainst_i$ in the original S.datas, the new data instance must be an extension of the old.
- For each structure instance $structinst_i$ in the original S.structs, the new structure instance must be an extension of the old.
- For each array instance $arrayinst_i$ in the original S.arrays, the new array instance must be an extension of the old.

$$\frac{\begin{array}{ccc}
S_1.\mathsf{funcs} = funcinst_1^* & S_2.\mathsf{funcs} = {funcinst'_1}^*\ funcinst_2^* & (\vdash funcinst_1 \preceq funcinst'_1)^* \\
S_1.\mathsf{tables} = tableinst_1^* & S_2.\mathsf{tables} = {tableinst'_1}^*\ tableinst_2^* & (\vdash tableinst_1 \preceq tableinst'_1)^* \\
S_1.\mathsf{mems} = meminst_1^* & S_2.\mathsf{mems} = {meminst'_1}^*\ meminst_2^* & (\vdash meminst_1 \preceq meminst'_1)^* \\
S_1.\mathsf{globals} = globalinst_1^* & S_2.\mathsf{globals} = {globalinst'_1}^*\ globalinst_2^* & (\vdash globalinst_1 \preceq globalinst'_1)^* \\
S_1.\mathsf{elems} = eleminst_1^* & S_2.\mathsf{elems} = {eleminst'_1}^*\ eleminst_2^* & (\vdash eleminst_1 \preceq eleminst'_1)^* \\
S_1.\mathsf{datas} = datainst_1^* & S_2.\mathsf{datas} = {datainst'_1}^*\ datainst_2^* & (\vdash datainst_1 \preceq datainst'_1)^* \\
S_1.\mathsf{structs} = structinst_1^* & S_2.\mathsf{structs} = {structinst'_1}^*\ structinst_2^* & (\vdash structinst_1 \preceq structinst'_1)^* \\
S_1.\mathsf{arrays} = arrayinst_1^* & S_2.\mathsf{arrays} = {arrayinst'_1}^*\ arrayinst_2^* & (\vdash arrayinst_1 \preceq arrayinst'_1)^*
\end{array}}{\vdash S_1 \preceq S_2}$$

Function Instance $funcinst$

- A function instance must remain unchanged.

$$\frac{}{\vdash funcinst \preceq funcinst}$$

Table Instance $tableinst$

- The table type $tableinst$.type must remain unchanged.
- The length of $tableinst$.elem must not shrink.

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}\ tt, \mathsf{elem}\ (fa_1^?)^{n_1}\} \preceq \{\mathsf{type}\ tt, \mathsf{elem}\ (fa_2^?)^{n_2}\}}$$

Memory Instance $meminst$

- The memory type $meminst$.type must remain unchanged.
- The length of $meminst$.bytes must not shrink.

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}\ mt, \mathsf{bytes}\ b_1^{n_1}\} \preceq \{\mathsf{type}\ mt, \mathsf{bytes}\ b_2^{n_2}\}}$$

Global Instance *globalinst*

- The global type $globalinst.\mathsf{type}$ must remain unchanged.
- Let $mut\ t$ be the structure of $globalinst.\mathsf{type}$.
- If mut is const, then the value $globalinst.\mathsf{value}$ must remain unchanged.

$$\frac{mut = \mathsf{var} \lor val_1 = val_2}{\vdash \{\mathsf{type}\ (mut\ t), \mathsf{value}\ val_1\} \preceq \{\mathsf{type}\ (mut\ t), \mathsf{value}\ val_2\}}$$

Element Instance *eleminst*

- The reference type $eleminst.\mathsf{type}$ must remain unchanged.
- The list $eleminst.\mathsf{elem}$ must:
  - either remain unchanged,
  - or shrink to length 0.

$$\frac{}{\vdash \{\mathsf{type}\ t, \mathsf{elem}\ a^*\} \preceq \{\mathsf{type}\ t, \mathsf{elem}\ a^*\}}$$

$$\frac{}{\vdash \{\mathsf{type}\ t, \mathsf{elem}\ a^*\} \preceq \{\mathsf{type}\ t, \mathsf{elem}\ \epsilon\}}$$

Data Instance *datainst*

- The list $datainst.\mathsf{bytes}$ must:
  - either remain unchanged,
  - or shrink to length 0.

$$\frac{}{\vdash \{\mathsf{bytes}\ b^*\} \preceq \{\mathsf{bytes}\ b^*\}}$$

$$\frac{}{\vdash \{\mathsf{bytes}\ b^*\} \preceq \{\mathsf{bytes}\ \epsilon\}}$$

Structure Instance *structinst*

- The defined type $structinst.\mathsf{type}$ must remain unchanged.
- Assert: due to store well-formedness, the expansion of $structinst.\mathsf{type}$ is a structure type.
- Let $\mathsf{struct}\ fieldtype^*$ be the expansion of $structinst.\mathsf{type}$.
- The length of the list $structinst.\mathsf{fields}$ must remain unchanged.
- Assert: due to store well-formedness, the length of $structinst.\mathsf{fields}$ is the same as the length of $fieldtype^*$.
- For each field value $fieldval_i$ in $structinst.\mathsf{fields}$ and corresponding field type $fieldtype_i$ in $fieldtype^*$:
  - Let $mut_i\ st_i$ be the structure of $fieldtype_i$.
  - If mut_i is const, then the field value $fieldval_i$ must remain unchanged.

$$\frac{(mut = \mathsf{var} \lor fieldval_1 = fieldval_2)^*}{\vdash \{\mathsf{type}\ (mut\ st)^*, \mathsf{fields}\ fieldval_1^*\} \preceq \{\mathsf{type}\ (mut\ st)^*, \mathsf{fields}\ fieldval_2^*\}}$$

Array Instance $arrayinst$

- The defined type $arrayinst.\mathsf{type}$ must remain unchanged.
- Assert: due to store well-formedness, the expansion of $arrayinst.\mathsf{type}$ is an array type.
- Let $\mathsf{array}\ fieldtype$ be the expansion of $arrayinst.\mathsf{type}$.
- The length of the list $arrayinst.\mathsf{fields}$ must remain unchanged.
- Let $mut\ st$ be the structure of $fieldtype$.
- If mut is const, then the sequence of field values $arrayinst.\mathsf{fields}$ must remain unchanged.

$$\frac{mut = \mathsf{var} \lor fieldval_1^* = fieldval_2^*}{\vdash \{\mathsf{type}\ (mut\ st), \mathsf{fields}\ fieldval_1^*\} \preceq \{\mathsf{type}\ (mut\ st), \mathsf{fields}\ fieldval_2^*\}}$$

7.3.9 Theorems

Given the definition of valid configurations, the standard soundness theorems hold.[52][54]

Theorem (Preservation). If a configuration $S;T$ is valid with result type $[t^*]$ (i.e., $\vdash S;T : [t^*]$), and steps to $S';T'$ (i.e., $S;T \hookrightarrow S';T'$), then $S';T'$ is a valid configuration with the same result type (i.e., $\vdash S';T' : [t^*]$). Furthermore, S' is an extension of S (i.e., $\vdash S \preceq S'$).

A *terminal* thread is one whose sequence of instructions is a result. A terminal configuration is a configuration whose thread is terminal.

Theorem (Progress). If a configuration $S;T$ is valid (i.e., $\vdash S;T : [t^*]$ for some result type $[t^*]$), then either it is terminal, or it can step to some configuration $S';T'$ (i.e., $S;T \hookrightarrow S';T'$).

From Preservation and Progress the soundness of the WebAssembly type system follows directly.

Corollary (Soundness). If a configuration $S;T$ is valid (i.e., $\vdash S;T : [t^*]$ for some result type $[t^*]$), then it either diverges or takes a finite number of steps to reach a terminal configuration $S';T'$ (i.e., $S;T \hookrightarrow {}^* S';T'$) that is valid with the same result type (i.e., $\vdash S';T' : [t^*]$) and where S' is an extension of S (i.e., $\vdash S \preceq S'$).

In other words, every thread in a valid configuration either runs forever, traps, or terminates with a result that has the expected type. Consequently, given a valid store, no computation defined by instantiation or invocation of a valid module can "crash" or otherwise (mis)behave in ways not covered by the execution semantics given in this specification.

7.4 Type System Properties

7.4.1 Principal Types

The type system of WebAssembly features both subtyping and simple forms of polymorphism for instruction types. That has the effect that every instruction or instruction sequence can be classified with multiple different instruction types.

However, the typing rules still allow deriving *principal types* for instruction sequences. That is, every valid instruction sequence has one particular type scheme, possibly containing some unconstrained place holder *type variables*, that is a subtype of all its valid instruction types, after substituting its type variables with suitable specific types.

[52] A machine-verified version of the formalization and soundness proof of the PLDI 2017 paper is described in the following article: Conrad Watt. Mechanising and Verifying the WebAssembly Specification[Page 271, 53]. Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2018). ACM 2018.
[53] https://dl.acm.org/citation.cfm?id=3167082
[54] Machine-verified formalizations and soundness proofs of the semantics from the official specification are described in the following article: Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, Philippa Gardner. Two Mechanisations of WebAssembly 1.0[55]. Proceedings of the 24th International Symposium on Formal Methods (FM 2021). Springer 2021.
[55] https://link.springer.com/chapter/10.1007/978-3-030-90870-6_4

Moreover, when deriving an instruction type in a "forward" manner, i.e., the *input* of the instruction sequence is already fixed to specific types, then it has a principal *output* type expressible without type variables, up to a possibly polymorphic stack bottom representable with one single variable. In other words, "forward" principal types are effectively *closed*.

Note: For example, in isolation, the instruction $\mathsf{ref.as\_non\_null}$ has the type $[(\mathsf{ref}\ \mathsf{null}\ ht)] \to [(\mathsf{ref}\ ht)]$ for any choice of valid heap type ht. Moreover, if the input type $[(\mathsf{ref}\ \mathsf{null}\ ht)]$ is already determined, i.e., a specific ht is given, then the output type $[(\mathsf{ref}\ ht)]$ is fully determined as well.

The implication of the latter property is that a validator for *complete* instruction sequences (as they occur in valid modules) can be implemented with a simple left-to-right algorithm that does not require the introduction of type variables.

A typing algorithm capable of handling *partial* instruction sequences (as might be considered for program analysis or program manipulation) needs to introduce type variables and perform substitutions, but it does not need to perform backtracking or record any non-syntactic constraints on these type variables.

Technically, the syntax of heap, value, and result types can be enriched with type variables as follows:

$$\begin{array}{lll}
null & ::= & \mathsf{null}^? \mid \alpha_{null} \\
heaptype & ::= & \dots \mid \alpha_{heaptype} \\
reftype & ::= & \mathsf{ref}\ null\ heaptype \\
valtype & ::= & \dots \mid \alpha_{valtype} \mid \alpha_{numvectype} \\
resulttype & ::= & [\alpha^?_{valtype^*}\ valtype^*]
\end{array}$$

where each α_{xyz} ranges over a set of type variables for syntactic class xyz, respectively. The special class $numvectype$ is defined as $numtype \mid vectype \mid \mathsf{bot}$, and is only needed to handle unannotated select instructions.

A type is *closed* when it does not contain any type variables, and *open* otherwise. A *type substitution* σ is a finite mapping from type variables to closed types of the respective syntactic class. When applied to an open type, it replaces the type variables α from its domain with the respective $\sigma(\alpha)$.

Theorem (Principal Types). If an instruction sequence $instr^*$ is valid with some closed instruction type $instrtype$ (i.e., $C \vdash instr^* : instrtype$), then it is also valid with a possibly open instruction type $instrtype_{\mathrm{min}}$ (i.e., $C \vdash instr^* : instrtype_{\mathrm{min}}$), such that for *every* closed type $instrtype'$ with which $instr^*$ is valid (i.e., for all $C \vdash instr^* : instrtype'$), there exists a substitution σ, such that $\sigma(instrtype_{\mathrm{min}})$ is a subtype of $instrtype'$ (i.e., $C \vdash \sigma(instrtype_{\mathrm{min}}) \leq instrtype'$). Furthermore, $instrtype_{\mathrm{min}}$ is unique up to the choice of type variables.

Theorem (Closed Principal Forward Types). If closed input type $[t_1^*]$ is given and the instruction sequence $instr^*$ is valid with instruction type $[t_1^*] \to_{x^*} [t_2^*]$ (i.e., $C \vdash instr^* : [t_1^*] \to_{x^*} [t_2^*]$), then it is also valid with instruction type $[t_1^*] \to_{x^*} [\alpha_{valtype^*}\ t^*]$ (i.e., $C \vdash instr^* : [t_1^*] \to_{x^*} [\alpha_{valtype^*}\ t^*]$), where all t^* are closed, such that for *every* closed result type $[t_2'^*]$ with which $instr^*$ is valid (i.e., for all $C \vdash instr^* : [t_1^*] \to_{x^*} [t_2'^*]$), there exists a substitution σ, such that $[t_2'^*] = [\sigma(\alpha_{valtype^*})\ t^*]$.

7.4.2 Type Lattice

The Principal Types property depends on the existence of a *greatest lower bound* for any pair of types.

Theorem (Greatest Lower Bounds for Value Types). For any two value types t_1 and t_2 that are valid (i.e., $C \vdash t_1 : \mathsf{ok}$ and $C \vdash t_2 : \mathsf{ok}$), there exists a valid value type t that is a subtype of both t_1 and t_2 (i.e., $C \vdash t : \mathsf{ok}$ and $C \vdash t \leq t_1$ and $C \vdash t \leq t_2$), such that *every* valid value type t' that also is a subtype of both t_1 and t_2 (i.e., for all $C \vdash t' : \mathsf{ok}$ and $C \vdash t' \leq t_1$ and $C \vdash t' \leq t_2$), is a subtype of t (i.e., $C \vdash t' \leq t$).

Note: The greatest lower bound of two types may be bot.

Theorem (Conditional Least Upper Bounds for Value Types). Any two value types t_1 and t_2 that are valid (i.e., $C \vdash t_1 : \mathsf{ok}$ and $C \vdash t_2 : \mathsf{ok}$) either have no common supertype, or there exists a valid value type t that is a supertype of both t_1 and t_2 (i.e., $C \vdash t : \mathsf{ok}$ and $C \vdash t_1 \leq t$ and $C \vdash t_2 \leq t$), such that *every* valid value type t'

that also is a supertype of both t_1 and t_2 (i.e., for all $C \vdash t' : \mathsf{ok}$ and $C \vdash t_1 \leq t'$ and $C \vdash t_2 \leq t'$), is a supertype of t (i.e., $C \vdash t \leq t'$).

Note: If a top type was added to the type system, a least upper bound would exist for any two types.

Corollary (Type Lattice). Assuming the addition of a provisional top type, value types form a lattice with respect to their subtype relation.

Finally, value types can be partitioned into multiple disjoint hierarchies that are not related by subtyping, except through bot.

Theorem (Disjoint Subtype Hierarchies). The greatest lower bound of two value types is bot or $\mathsf{ref\ bot}$ if and only if they do not have a least upper bound.

In other words, types that do not have common supertypes, do not have common subtypes either (other than bot or $\mathsf{ref\ bot}$), and vice versa.

Note: Types from disjoint hierarchies can safely be represented in mutually incompatible ways in an implementation, because their values can never flow to the same place.

7.4.3 Compositionality

Valid instruction sequences can be freely *composed*, as long as their types match up.

Theorem (Composition). If two instruction sequences $instr_1^*$ and $instr_2^*$ are valid with types $[t_1^*] \rightarrow_{x_1^*} [t^*]$ and $[t^*] \rightarrow_{x_2^*} [t_2^*]$, respectively (i.e., $C \vdash instr_1^* : [t_1^*] \rightarrow_{x_1^*} [t^*]$ and $C \vdash instr_1^* : [t^*] \rightarrow_{x_2^*} [t_2^*]$), then the concatenated instruction sequence $(instr_1^*\ instr_2^*)$ is valid with type $[t_1^*] \rightarrow_{x_1^*\,x_2^*} [t_2^*]$ (i.e., $C \vdash instr_1^*\ instr_2^* : [t_1^*] \rightarrow_{x_1^*\,x_2^*} [t_2^*]$).

Note: More generally, instead of a shared type $[t^*]$, it suffices if the output type of $instr_1^*$ is a subtype of the input type of $instr_1^*$, since the subtype can always be weakened to its supertype by subsumption.

Inversely, valid instruction sequences can also freely be *decomposed*, that is, splitting them anywhere produces two instruction sequences that are both valid.

Theorem (Decomposition). If an instruction sequence $instr^*$ that is valid with type $[t_1^*] \rightarrow_{x^*} [t_2^*]$ (i.e., $C \vdash instr^* : [t_1^*] \rightarrow_{x^*} [t_2^*]$) is split into two instruction sequences $instr_1^*$ and $instr_2^*$ at any point (i.e., $instr^* = instr_1^*\ instr_2^*$), then these are separately valid with some types $[t_1^*] \rightarrow_{x_1^*} [t^*]$ and $[t^*] \rightarrow_{x_2^*} [t_2^*]$, respectively (i.e., $C \vdash instr_1^* : [t_1^*] \rightarrow_{x_1^*} [t^*]$ and $C \vdash instr_1^* : [t^*] \rightarrow_{x_2^*} [t_2^*]$), where $x^* = x_1^*\ x_2^*$.

Note: This property holds because validation is required even for unreachable code. Without that, $instr_2^*$ might not be valid in isolation.

7.5 Validation Algorithm

The specification of WebAssembly validation is purely *declarative*. It describes the constraints that must be met by a module or instruction sequence to be valid.

This section sketches the skeleton of a sound and complete *algorithm* for effectively validating code, i.e., sequences of instructions. (Other aspects of validation are straightforward to implement.)

In fact, the algorithm is expressed over the flat sequence of opcodes as occurring in the binary format, and performs only a single pass over it. Consequently, it can be integrated directly into a decoder.

The algorithm is expressed in typed pseudo code whose semantics is intended to be self-explanatory.

7.5.1 Data Structures

Types

Value types are representable as sets of enumerations:

```
type num_type = I32 | I64 | F32 | F64
type vec_type = V128
type heap_type =
  Any | Eq | I31 | Struct | Array | None |
  Func | Nofunc | Extern | Noextern | Bot |
  Def(def : def_type)
type ref_type = Ref(heap : heap_type, null : bool)
type val_type = num_type | vec_type | ref_type | Bot

func is_num(t : val_type) : bool =
  return t = I32 || t = I64 || t = F32 || t = F64 || t = Bot

func is_vec(t : val_type) : bool =
  return t = V128 || t = Bot

func is_ref(t : val_type) : bool =
  return not (is_num t || is_vec t) || t = Bot
```

Similarly, defined types `def_type` can be represented:

```
type pack_type = I8 | I16
type field_type = Field(val : val_type | pack_type, mut : bool)

type struct_type = Struct(fields : list(field_type))
type array_type = Array(fields : field_type)
type func_type = Func(params : list(val_type), results : list(val_type))
type comp_type = struct_type | array_type | func_type

type sub_type = Sub(super : list(def_type), body : comp_type, final : bool)
type rec_type = Rec(types : list(sub_type))

type def_type = Def(rec : rec_type, proj : int32)

func unpack_field(t : field_type) : val_type =
  if (it = I8 || t = I16) return I32
  return t

func expand_def(t : def_type) : comp_type =
  return t.rec.types[t.proj].body
```

These representations assume that all types have been closed by substituting all type indices (in concrete heap types and in sub types) with their respective defined types. This includes *recursive* references to enclosing defined types, such that type representations form graphs and may be *cyclic* for recursive types.

We assume that all types have been *canonicalized*, such that equality on two type representations holds if and only if their closures are syntactically equivalent, making it a constant-time check.

Note: For the purpose of type canonicalization, recursive references from a heap type to an enclosing recursive type (i.e., forward edges in the graph that form a cycle) need to be distinguished from references to previously defined types. However, this distinction does not otherwise affect validation, so is ignored here. In the graph representation, all recursive types are effectively infinitely unrolled.

We further assume that validation and subtyping checks are defined on value types, as well as a few auxiliary functions on composite types:

```
func validate_val_type(t : val_type)
func validate_ref_type(t : ref_type)

func matches_val(t1 : val_type, t2 : val_type) : bool
func matches_ref(t1 : val_type, t2 : val_type) : bool

func is_func(t : comp_type) : bool
func is_struct(t : comp_type) : bool
func is_array(t : comp_type) : bool
```

Finally, the following function computes the least precise supertype of a given heap type (its corresponding top type):

```
func top_heap_type(t : heap_type) : heap_type =
  switch (t)
    case (Any | Eq | I31 | Struct | Array | None)
      return Any
    case (Func | Nofunc)
      return Func
    case (Extern | Noextern)
      return Extern
    case (Def(dt))
      switch (dt.rec.types[dt.proj].body)
        case (Struct(_) | Array(_))
          return Any
        case (Func(_))
          return Func
    case (Bot)
      raise CannotOccurInSource
```

Context

Validation requires a context for checking uses of indices. For the purpose of presenting the algorithm, it is maintained in a set of global variables:

```
var return_type : list(val_type)
var types : array(def_type)
var locals : array(val_type)
var locals_init : array(bool)
var globals : array(global_type)
var funcs : array(func_type)
var tables : array(table_type)
var mems : array(mem_type)
```

This assumes suitable representations for the various types besides `val_type`, which are omitted here.

For locals, there is an additional array recording the initialization status of each local.

Stacks

The algorithm uses three separate stacks: the *value stack*, the *control stack*, and the *initialization stack*. The value stack tracks the types of operand values on the stack. The control stack tracks surrounding structured control instructions and their associated blocks. The initialization stack records all locals that have been initialized since the beginning of the function.

```
type val_stack = stack(val_type)
type init_stack = stack(u32)

type ctrl_stack = stack(ctrl_frame)
type ctrl_frame = {
  opcode : opcode
  start_types : list(val_type)
  end_types : list(val_type)
  val_height : nat
  init_height : nat
  unreachable : bool
}
```

For each entered block, the control stack records a *control frame* with the originating opcode, the types on the top of the operand stack at the start and end of the block (used to check its result as well as branches), the height of the operand stack at the start of the block (used to check that operands do not underflow the current block), the height of the initialization stack at the start of the block (used to reset initialization status at the end of the block), and a flag recording whether the remainder of the block is unreachable (used to handle stack-polymorphic typing after branches).

For the purpose of presenting the algorithm, these stacks are simply maintained as global variables:

```
var vals : val_stack
var inits : init_stack
var ctrls : ctrl_stack
```

However, these variables are not manipulated directly by the main checking function, but through a set of auxiliary functions:

```
func push_val(type : val_type) =
  vals.push(type)

func pop_val() : val_type =
  if (vals.size() = ctrls[0].height && ctrls[0].unreachable) return Bot
  error_if(vals.size() = ctrls[0].height)
  return vals.pop()

func pop_val(expect : val_type) : val_type =
  let actual = pop_val()
  error_if(not matches_val(actual, expect))
  return actual

func pop_num() : num_type | Bot =
  let actual = pop_val()
  error_if(not is_num(actual))
  return actual

func pop_ref() : ref_type =
  let actual = pop_val()
  error_if(not is_ref(actual))
  if (actual = Bot) return Ref(Bot, false)
```

(continues on next page)

(continued from previous page)

```
  return actual

func push_vals(types : list(val_type)) = foreach (t in types) push_val(t)
func pop_vals(types : list(val_type)) : list(val_type) =
  var popped := []
  foreach (t in reverse(types)) popped.prepend(pop_val(t))
  return popped
```

Pushing an operand value simply pushes the respective type to the value stack.

Popping an operand value checks that the value stack does not underflow the current block and then removes one type. But first, a special case is handled where the block contains no known values, but has been marked as unreachable. That can occur after an unconditional branch, when the stack is typed polymorphically. In that case, the `Bot` type is returned, because that is a *principal* choice trivially satisfying all use constraints.

A second function for popping an operand value takes an expected type, which the actual operand type is checked against. The types may differ by subtyping, including the case where the actual type is `Bot`, and thereby matches unconditionally. The function returns the actual type popped from the stack.

Finally, there are accumulative functions for pushing or popping multiple operand types.

Note: The notation `stack[i]` is meant to index the stack from the top, so that, e.g., `ctrls[0]` accesses the element pushed last.

The initialization stack and the initialization status of locals is manipulated through the following functions:

```
func get_local(idx : u32) =
  error_if(not locals_init[idx])

func set_local(idx : u32) =
  if (not locals_init[idx])
    inits.push(idx)
    locals_init[idx] := true

func reset_locals(height : nat) =
  while (inits.size() > height)
    locals_init[inits.pop()] := false
```

Getting a local verifies that it is known to be initialized. When a local is set that was not set already, then its initialization status is updated and the change is recorded in the initialization stack. Thus, the initialization status of all locals can be reset to a previous state by denoting a specific height in the initialization stack.

The size of the initialization stack is bounded by the number of (non-defaultable) locals in a function, so can be preallocated by an algorithm.

The control stack is likewise manipulated through auxiliary functions:

```
func push_ctrl(opcode : opcode, in : list(val_type), out : list(val_type)) =
  let frame = ctrl_frame(opcode, in, out, vals.size(), inits.size(), false)
  ctrls.push(frame)
  push_vals(in)

func pop_ctrl() : ctrl_frame =
  error_if(ctrls.is_empty())
  let frame = ctrls[0]
  pop_vals(frame.end_types)
  error_if(vals.size() =/= frame.val_height)
  reset_locals(frame.init_height)
```

(continues on next page)

(continued from previous page)

```
  ctrls.pop()
  return frame

func label_types(frame : ctrl_frame) : list(val_types) =
  return (if (frame.opcode = loop) frame.start_types else frame.end_types)

func unreachable() =
  vals.resize(ctrls[0].height)
  ctrls[0].unreachable := true
```

Pushing a control frame takes the types of the label and result values. It allocates a new frame record recording them along with the current height of the operand stack and marks the block as reachable.

Popping a frame first checks that the control stack is not empty. It then verifies that the operand stack contains the right types of values expected at the end of the exited block and pops them off the operand stack. Afterwards, it checks that the stack has shrunk back to its initial height. Finally, it undoes all changes to the initialization status of locals that happend inside the block.

The type of the label associated with a control frame is either that of the stack at the start or the end of the frame, determined by the opcode that it originates from.

Finally, the current frame can be marked as unreachable. In that case, all existing operand types are purged from the value stack, in order to allow for the stack-polymorphism logic in `pop_val` to take effect. Because every function has an implicit outermost label that corresponds to an implicit block frame, it is an invariant of the validation algorithm that there always is at least one frame on the control stack when validating an instruction, and hence, *ctrls[0]* is always defined.

Note: Even with the unreachable flag set, consecutive operands are still pushed to and popped from the operand stack. That is necessary to detect invalid examples like (unreachable (i32.const) i64.add). However, a polymorphic stack cannot underflow, but instead generates `Bot` types as needed.

7.5.2 Validation of Opcode Sequences

The following function shows the validation of a number of representative instructions that manipulate the stack. Other instructions are checked in a similar manner.

```
func validate(opcode) =
  switch (opcode)
    case (i32.add)
      pop_val(I32)
      pop_val(I32)
      push_val(I32)

    case (drop)
      pop_val()

    case (select)
      pop_val(I32)
      let t1 = pop_val()
      let t2 = pop_val()
      error_if(not (is_num(t1) && is_num(t2) || is_vec(t1) && is_vec(t2)))
      error_if(t1 =/= t2 && t1 =/= Bot && t2 =/= Bot)
      push_val(if (t1 = Bot) t2 else t1)

    case (select t)
```

(continues on next page)

(continued from previous page)

```
    pop_val(I32)
    pop_val(t)
    pop_val(t)
    push_val(t)

  case (ref.is_null)
    pop_ref()
    push_val(I32)

  case (ref.as_non_null)
    let rt = pop_ref()
    push_val(Ref(rt.heap, false))

  case (ref.test rt)
    validate_ref_type(rt)
    pop_val(Ref(top_heap_type(rt), true))
    push_val(I32)

  case (local.get x)
    get_local(x)
    push_val(locals[x])

  case (local.set x)
    pop_val(locals[x])
    set_local(x)

  case (unreachable)
    unreachable()

  case (block t1*->t2*)
    pop_vals([t1*])
    push_ctrl(block, [t1*], [t2*])

  case (loop t1*->t2*)
    pop_vals([t1*])
    push_ctrl(loop, [t1*], [t2*])

  case (if t1*->t2*)
    pop_val(I32)
    pop_vals([t1*])
    push_ctrl(if, [t1*], [t2*])

  case (end)
    let frame = pop_ctrl()
    push_vals(frame.end_types)

  case (else)
    let frame = pop_ctrl()
    error_if(frame.opcode =/= if)
    push_ctrl(else, frame.start_types, frame.end_types)

  case (br n)
    error_if(ctrls.size() < n)
    pop_vals(label_types(ctrls[n]))
    unreachable()
```

(continues on next page)

(continued from previous page)

```
    case (br_if n)
      error_if(ctrls.size() < n)
      pop_val(I32)
      pop_vals(label_types(ctrls[n]))
      push_vals(label_types(ctrls[n]))

    case (br_table n* m)
      pop_val(I32)
      error_if(ctrls.size() < m)
      let arity = label_types(ctrls[m]).size()
      foreach (n in n*)
        error_if(ctrls.size() < n)
        error_if(label_types(ctrls[n]).size() =/= arity)
        push_vals(pop_vals(label_types(ctrls[n])))
      pop_vals(label_types(ctrls[m]))
      unreachable()

    case (br_on_null n)
      error_if(ctrls.size() < n)
      let rt = pop_ref()
      pop_vals(label_types(ctrls[n]))
      push_vals(label_types(ctrls[n]))
      push_val(Ref(rt.heap, false))

    case (br_on_cast n rt1 rt2)
      validate_ref_type(rt1)
      validate_ref_type(rt2)
      pop_val(rt1)
      push_val(rt2)
      pop_vals(label_types(ctrls[n]))
      push_vals(label_types(ctrls[n]))
      pop_val(rt2)
      push_val(diff_ref_type(rt2, rt1))

    case (return)
      pop_vals(return_types)
      unreachable()

    case (call_ref x)
      let t = expand_def(types[x])
      error_if(not is_func(t))
      pop_vals(t.params)
      pop_val(Ref(Def(types[x])))
      push_vals(t.results)

    case (return_call_ref x)
      let t = expand_def(types[x])
      error_if(not is_func(t))
      pop_vals(t.params)
      pop_val(Ref(Def(types[x])))
      error_if(t.results.len() =/= return_types.len())
      push_vals(t.results)
      pop_vals(return_types)
      unreachable()

    case (struct.new x)
```

(continues on next page)

(continued from previous page)

```
    let t = expand_def(types[x])
    error_if(not is_struct(t))
    for (ti in reverse(t.fields))
      pop_val(unpack_field(ti))
    push_val(Ref(Def(types[x])))

  case (struct.set x n)
    let t = expand_def(types[x])
    error_if(not is_struct(t) || n >= t.fields.len())
    pop_val(Ref(Def(types[x])))
    pop_val(unpack_field(st.fields[n]))
```

Note: It is an invariant under the current WebAssembly instruction set that an operand of `Bot` type is never duplicated on the stack. This would change if the language were extended with stack instructions like `dup`. Under such an extension, the above algorithm would need to be refined by replacing the `Bot` type with proper *type variables* to ensure that all uses are consistent.

7.6 Custom Sections and Annotations

This appendix defines dedicated custom sections for WebAssembly's binary format and annotations for the text format. Such sections or annotations do not contribute to, or otherwise affect, the WebAssembly semantics, and may be ignored by an implementation. However, they provide useful meta data that implementations can make use of to improve user experience or take compilation hints.

7.6.1 Name Section

The *name section* is a custom section whose name string is itself '`name`'. The name section should appear only once in a module, and only after the data section.

The purpose of this section is to attach printable names to definitions in a module, which e.g. can be used by a debugger or when parts of the module are to be rendered in text form.

Note: All names are represented in Unicode[56] encoded in UTF-8. Names need not be unique.

Subsections

The data of a name section consists of a sequence of *subsections*. Each subsection consists of a

- a one-byte subsection *id*,
- the *u32* *size* of the contents, in bytes,
- the actual *contents*, whose structure is dependent on the subsection id.

$$
\begin{array}{llll}
\mathtt{namesec} & ::= & \mathtt{section}_0(\mathtt{namedata}) & \\
\mathtt{namedata} & ::= & n{:}\mathtt{name} & (\text{if } n = \text{`}\mathtt{name}\text{'}) \\
 & & \mathtt{modulenamesubsec}^? & \\
 & & \mathtt{funcnamesubsec}^? & \\
 & & \mathtt{localnamesubsec}^? & \\
\mathtt{namesubsection}_N(\mathtt{B}) & ::= & N{:}\mathtt{byte}\ \ \mathit{size}{:}\mathtt{u32}\ \ \mathtt{B} & (\text{if } \mathit{size} = ||\mathtt{B}||)
\end{array}
$$

The following subsection ids are used:

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

Id	Subsection
0	module name
1	function names
2	local names
4	type names
10	field names

Each subsection may occur at most once, and in order of increasing id.

Name Maps

A *name map* assigns names to indices in a given index space. It consists of a list of index/name pairs in order of increasing index value. Each index must be unique, but the assigned names need not be.

$$\begin{array}{lll} \mathtt{namemap} & ::= & \mathtt{list}(\mathtt{nameassoc}) \\ \mathtt{nameassoc} & ::= & \mathtt{idx}\ \mathtt{name} \end{array}$$

An *indirect name map* assigns names to a two-dimensional index space, where secondary indices are *grouped* by primary indices. It consists of a list of primary index/name map pairs in order of increasing index value, where each name map in turn maps secondary indices to names. Each primary index must be unique, and likewise each secondary index per individual name map.

$$\begin{array}{lll} \mathtt{indirectnamemap} & ::= & \mathtt{list}(\mathtt{indirectnameassoc}) \\ \mathtt{indirectnameassoc} & ::= & \mathtt{idx}\ \mathtt{namemap} \end{array}$$

Module Names

The *module name subsection* has the id 0. It simply consists of a single name that is assigned to the module itself.

$$\mathtt{modulenamesubsec} \quad ::= \quad \mathtt{namesubsection}_0(\mathtt{name})$$

Function Names

The *function name subsection* has the id 1. It consists of a name map assigning function names to function indices.

$$\mathtt{funcnamesubsec} \quad ::= \quad \mathtt{namesubsection}_1(\mathtt{namemap})$$

Local Names

The *local name subsection* has the id 2. It consists of an indirect name map assigning local names to local indices grouped by function indices.

$$\mathtt{localnamesubsec} \quad ::= \quad \mathtt{namesubsection}_2(\mathtt{indirectnamemap})$$

Type Names

The *type name subsection* has the id 4. It consists of a name map assigning type names to type indices.

$$\mathtt{typenamesubsec} ::= \mathtt{namesubsection}_1(\mathtt{namemap})$$

Field Names

The *field name subsection* has the id 10. It consists of an indirect name map assigning field names to field indices grouped by the type indices of their respective structure types.

$$\mathtt{fieldnamesubsec} ::= \mathtt{namesubsection}_2(\mathtt{indirectnamemap})$$

7.6.2 Name Annotations

Name annotations are the textual analogue to the name section and provide a textual representation for it. Consequently, their id is `@name`.

Analogous to the name section, name annotations are allowed on modules, functions, and locals (including parameters). They can be placed where the text format allows binding occurrences of respective identifiers. If both an identifier and a name annotation are given, the annotation is expected *after* the identifier. In that case, the annotation takes precedence over the identifier as a textual representation of the binding's name. At most one name annotation may be given per binding.

All name annotations have the following format:

$$\mathtt{nameannot} ::= \text{‘}(\texttt{@name}\text{’}\ \mathtt{string}\ \text{‘})\text{’}$$

Note: All name annotations can be arbitrary UTF-8 strings. Names need not be unique.

Module Names

A *module name annotation* must be placed on a module definition, directly after the ‘`module`’ keyword, or if present, after the following module identifier.

$$\mathtt{modulenameannot} ::= \mathtt{nameannot}$$

Function Names

A *function name annotation* must be placed on a function definition or function import, directly after the ‘`func`’ keyword, or if present, after the following function identifier or.

$$\mathtt{funcnameannot} ::= \mathtt{nameannot}$$

Parameter Names

A *parameter name annotation* must be placed on a parameter declaration, directly after the ‘param’ keyword, or if present, after the following parameter identifier. It may only be placed on a declaration that declares exactly one parameter.

$$\mathtt{paramnameannot} ::= \mathtt{nameannot}$$

Local Names

A *local name annotation* must be placed on a local declaration, directly after the ‘local’ keyword, or if present, after the following local identifier. It may only be placed on a declaration that declares exactly one local.

$$\mathtt{localnameannot} ::= \mathtt{nameannot}$$

Type Names

A *type name annotation* must be placed on a type declaration, directly after the ‘type’ keyword, or if present, after the following type identifier.

$$\mathtt{typenameannot} ::= \mathtt{nameannot}$$

Field Names

A *field name annotation* must be placed on the field of a structure type, directly after the ‘field’ keyword, or if present, after the following field identifier. It may only be placed on a declaration that declares exactly one field.

$$\mathtt{fieldnameannot} ::= \mathtt{nameannot}$$

7.6.3 Custom Annotations

Custom annotations are a generic textual representation for any custom section. Their id is @custom. By generating custom annotations, tools converting between binary format and text format can maintain and round-trip the content of custom sections even when they do not recognize them.

Custom annotations must be placed inside a module definition. They must occur anywhere after the ‘module’ keyword, or if present, after the following module identifier. They must not be nested into other constructs.

$$
\begin{array}{lcl}
\mathtt{customannot} & ::= & \text{`}(\mathtt{@custom}\text{'}~\mathtt{string}~\mathtt{customplace}^?~\mathtt{datastring}~\text{`})\text{'} \\
\mathtt{customplace} & ::= & \text{`}(\text{'}~\text{`}\mathtt{before}\text{'}~\text{`}\mathtt{first}\text{'}~\text{`})\text{'} \\
 & | & \text{`}(\text{'}~\text{`}\mathtt{before}\text{'}~\mathtt{sec}~\text{`})\text{'} \\
 & | & \text{`}(\text{'}~\text{`}\mathtt{after}\text{'}~\mathtt{sec}~\text{`})\text{'} \\
 & | & \text{`}(\text{'}~\text{`}\mathtt{after}\text{'}~\text{`}\mathtt{last}\text{'}~\text{`})\text{'} \\
\mathtt{sec} & ::= & \text{`}\mathtt{type}\text{'} \\
 & | & \text{`}\mathtt{import}\text{'} \\
 & | & \text{`}\mathtt{func}\text{'} \\
 & | & \text{`}\mathtt{table}\text{'} \\
 & | & \text{`}\mathtt{memory}\text{'} \\
 & | & \text{`}\mathtt{global}\text{'} \\
 & | & \text{`}\mathtt{export}\text{'} \\
 & | & \text{`}\mathtt{start}\text{'} \\
 & | & \text{`}\mathtt{elem}\text{'} \\
 & | & \text{`}\mathtt{code}\text{'} \\
 & | & \text{`}\mathtt{data}\text{'} \\
 & | & \text{`}\mathtt{datacount}\text{'}
\end{array}
$$

The first string in a custom annotation denotes the name of the custom section it represents. The remaining strings collectively represent the section's payload data, written as a data string, which can be split up into a possibly empty sequence of individual string literals (similar to data segments).

An arbitrary number of custom annotations (even of the same name) may occur in a module, each defining a separate custom section when converting to binary format. Placement of the sections in the binary can be customized via explicit *placement* directives, that position them either directly before or directly after a known section. That section must exist and be non-empty in the binary encoding of the annotated module. The placements (`before first`) and (`after last`) denote virtual sections before the first and after the last known section, respectively. When the placement directive is omitted, it defaults to (`after last`).

If multiple placement directives appear for the same position, then the sections are all placed there, in order of their appearance in the text. For this purpose, the position `after` a section is considered different from the position `before` the consecutive section, and the former occurs before the latter.

Note: Future versions of WebAssembly may introduce additional sections between others or at the beginning or end of a module. Using `first` and `last` guarantees that placement will still go before or after any future section, respectively.

If a custom section with a specific section id is given as well as annotations representing the same custom section (e.g., `@name` annotations as well as a `@custom` annotation for a `name` section), then two sections are assumed to be created. Their relative placement will depend on the placement directive given for the `@custom` annotation as well as the implicit placement requirements of the custom section, which are applied to the other annotation.

Note: For example, the following module,

```
(module
  (@custom "A" "aaa")
  (type $t (func))
  (@custom "B" (after func) "bbb")
  (@custom "C" (before func) "ccc")
  (@custom "D" (after last) "ddd")
  (table 10 funcref)
  (func (type $t))
  (@custom "E" (after import) "eee")
  (@custom "F" (before type) "fff")
  (@custom "G" (after data) "ggg")
  (@custom "H" (after code) "hhh")
```

(continues on next page)

(continued from previous page)

```
  (@custom "I" (after func) "iii")
  (@custom "J" (before func) "jjj")
  (@custom "K" (before first) "kkk")
)
```

will result in the following section ordering:

```
custom section "K"
custom section "F"
type section
custom section "E"
custom section "C"
custom section "J"
function section
custom section "B"
custom section "I"
table section
code section
custom section "H"
custom section "G"
custom section "A"
custom section "D"
```

7.7 Change History

Since the original release 1.0 of the WebAssembly specification, a number of proposals for extensions have been integrated. The following sections provide an overview of what has changed.

7.7.1 Release 2.0

Sign extension instructions

Added new numeric instructions for performing sign extension within integer representations.[57]

- New numeric instructions: $\mathsf{i}nn.\mathsf{extend}N\_\mathsf{s}$

Non-trapping float-to-int conversions

Added new conversion instructions that avoid trapping when converting a floating-point number to an integer.[58]

- New numeric instructions: $\mathsf{i}nn.\mathsf{trunc\_sat\_f}mm\_sx$

[57] https://github.com/WebAssembly/spec/tree/main/proposals/sign-extension-ops/

[58] https://github.com/WebAssembly/spec/tree/main/proposals/nontrapping-float-to-int-conversion/

Multiple values

Generalized the result type of blocks and functions to allow for multiple values; in addition, introduced the ability to have block parameters.[59]

- Function types allow more than one result
- Block types can be arbitrary function types

Reference types

Added funcref and externref as new value types and respective instructions.[60]

- New value types: reference types funcref and externref
- New reference instructions: ref.null, ref.func, ref.is_null
- Extended parametric instruction: select with optional type immediate
- New declarative form of element segment

Table instructions

Added instructions to directly access and modify tables.[Page 287, 60]

- Table types allow any reference type as element type
- New table instructions: table.get, table.set, table.size, table.grow

Multiple tables

Added the ability to use multiple tables per module.[60]

- Modules may define, import, and export multiple tables
- Table instructions take a table index immediate: table.get, table.set, table.size, table.grow, call_indirect
- Element segments take a table index

Bulk memory and table instructions

Added instructions that modify ranges of memory or table entries.[60][61]

- New memory instructions: memory.fill, memory.init, memory.copy, data.drop
- New table instructions: table.fill, table.init, table.copy, elem.drop
- New passive form of data segment
- New passive form of element segment
- New data count section in binary format
- Active data and element segments boundaries are no longer checked at compile time but may trap instead

[59] https://github.com/WebAssembly/spec/tree/main/proposals/multi-value/
[60] https://github.com/WebAssembly/spec/tree/main/proposals/reference-types/
[61] https://github.com/WebAssembly/spec/tree/main/proposals/bulk-memory-operations/

Vector instructions

Added vector type and instructions that manipulate multiple numeric values in parallel (also known as *SIMD*, single instruction multiple data)[62]

- New value type: v128
- New memory instructions: v128.load, v128.load$N$xM_sx, v128.load$N$_zero, v128.load$N$_splat, v128.load$N$_lane, v128.store, v128.store$N$_lane
- New constant vector instruction: v128.const
- New unary vector instructions: v128.not, iNxM.abs, iNxM.neg, i8x16.popcnt, fNxM.abs, fNxM.neg, fNxM.sqrt, fNxM.ceil, fNxM.floor, fNxM.trunc, fNxM.nearest
- New binary vector instructions: v128.and, v128.andnot, v128.or, v128.xor, iNxM.add, iNxM.sub, iNxM.mul, iNxM.add_sat_sx, iNxM.sub_sat_sx, iNxM.min_sx, iNxM.max_sx, iNxM.shl, iNxM.shr_sx, fNxM.add, iNxM.extmul_*half*_iN'xM'_sx, i16x8.q15mulr_sat_s, i32x4.dot_i16x8_s, i16x8.extadd_pairwise_i8x16_sx, i32x4.extadd_pairwise_i16x8_sx, i8x16.avgr_u, i16x8.avgr_u, fNxM.sub, fNxM.mul, fNxM.div, fNxM.min, fNxM.max, fNxM.pmin, fNxM.pmax
- New ternary vector instruction: v128.bitselect
- New test vector instructions: v128.any_true, iNxM.all_true
- New relational vector instructions: iNxM.eq, iNxM.ne, iNxM.lt_sx, iNxM.gt_sx, iNxM.le_sx, iNxM.ge_sx, fNxM.eq, fNxM.ne, fNxM.lt, fNxM.gt, fNxM.le, fNxM.ge
- New conversion vector instructions:i32x4.trunc_sat_f32x4_sx, i32x4.trunc_sat_f64x2_sx_zero, f32x4.convert_i32x4_sx, f32x4.demote_f64x2_zero, f64x2.convert_low_i32x4_sx, f64x2.promote_low_f32x4
- New lane access vector instructions: iNxM.extract_lane_$sx^?$, iNxM.replace_lane, fNxM.extract_lane, fNxM.replace_lane
- New lane splitting/combining vector instructions: iNxM.extend_*half*_iN'xM'_sx, i8x16.narrow_i16x8_sx, i16x8.narrow_i32x4_sx
- New byte reordering vector instructions: i8x16.shuffle, i8x16.swizzle
- New injection/projection vector instructions: iNxM.splat, fNxM.splat, iNxM.bitmask

7.7.2 Release 3.0

Extended constant expressions

Allowed basic numeric computations in constant expressions.[63]

- Extended set of constant instructions with inn.add, inn.sub, and inn.mul, and global.get for any previously declared immutable global

Note: The garbage collection added further constant instructions.

[62] https://github.com/WebAssembly/spec/tree/main/proposals/simd/
[63] https://github.com/WebAssembly/extended-const/blob/main/proposals/extended-const/

Tail calls

Added instructions to perform tail calls.[64]

- New control instructions: return_call and return_call_indirect

Multiple memories

Added the ability to use multiple memories per module.[65]

- Modules may define, import, and export multiple memories
- Memory instructions take a memory index immediate: memory.size, memory.grow, memory.fill, memory.copy, memory.init, t.load, t.store, t.load$N$_sx, t.store$N$, v128.load$N$xM_sx, v128.load$N$_zero, v128.load$N$_splat, v128.load$N$_lane, v128.store$N$_lane
- Data segments take a memory index

Typeful references

Added more precise types for references.[66]

- New generalised form of reference types: (ref null$^?$ $heaptype$)
- New class of heap types: func, extern, $typeidx$
- Basic subtyping on reference and value types
- New reference instructions: ref.as_non_null, br_on_null, br_on_non_null
- New control instruction: call_ref
- Refined typing of reference instruction ref.func with more precise result type
- Refined typing of local instructions and instruction sequences to track the initialization status of locals with non-defaultable type
- Extended table definitions with optional initializer expression

Garbage collection

Added managed reference types.[67]

- New forms of heap types: any, eq, i31, struct, array, none, nofunc, noextern
- New reference type short-hands: anyref, eqref, i31ref, structref, arrayref, nullref, nullfuncref, nullexternref
- New forms of type definitions: structure and array types, sub types, and recursive types
- Enriched subtyping based on explicitly declared sub types and the new heap types
- New generic reference instructions: ref.eq, ref.test, ref.cast, br_on_cast, br_on_cast_fail
- New reference instructions for unboxed scalars: ref.i31, i31.get_sx
- New reference instructions for structure types: struct.new, struct.new_default, struct.get_$sx^?$, struct.set
- New reference instructions for array types: array.new, array.new_default, array.new_fixed, array.new_data, array.new_elem, array.get_$sx^?$, array.set, array.len, array.fill, array.copy, array.init_data, array.init_elem
- New reference instructions for converting host types: any.convert_extern, extern.convert_any

[64] https://github.com/WebAssembly/spec/tree/main/proposals/tail-call/
[65] https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/
[66] https://github.com/WebAssembly/spec/tree/main/proposals/function-references/
[67] https://github.com/WebAssembly/spec/tree/main/proposals/gc/

- Extended set of constant instructions with ref.i31, struct.new, struct.new_default, array.new, array.new_default, array.new_fixed, any.convert_extern, extern.convert_any

7.8 Index of Types

Category	Constructor	Binary Opcode
Type index	x	(positive number as s32 or u32)
Number type	i32	0x7F (-1 as s7)
Number type	i64	0x7E (-2 as s7)
Number type	f32	0x7D (-3 as s7)
Number type	f64	0x7C (-4 as s7)
Vector type	v128	0x7B (-5 as s7)
(reserved)		0x7A .. 0x79
Packed type	i8	0x78 (-8 as s7)
Packed type	i16	0x77 (-9 as s7)
(reserved)		0x78 .. 0x74
Heap type	nofunc	0x73 (-13 as s7)
Heap type	noextern	0x72 (-14 as s7)
Heap type	none	0x71 (-15 as s7)
Heap type	func	0x70 (-16 as s7)
Heap type	extern	0x6F (-17 as s7)
Heap type	any	0x6E (-18 as s7)
Heap type	eq	0x6D (-19 as s7)
Heap type	i31	0x6C (-20 as s7)
Heap type	struct	0x6B (-21 as s7)
Heap type	array	0x6A (-22 as s7)
(reserved)		0x69 .. 0x65
Reference type	ref	0x64 (-28 as s7)
Reference type	ref null	0x63 (-29 as s7)
(reserved)		0x62 .. 0x61
Composite type	func $[valtype^*] \rightarrow [valtype^*]$	0x60 (-32 as s7)
Composite type	struct $fieldtype^*$	0x5F (-33 as s7)
Composite type	array $fieldtype$	0x5E (-34 as s7)
(reserved)		0x5D .. 0x51
Sub type	sub $typeidx^*$ $comptype$	0x50 (-48 as s7)
Sub type	sub final $typeidx^*$ $comptype$	0x4F (-49 as s7)
Recursive type	rec $subtype^*$	0x4E (-50 as s7)
(reserved)		0x4D .. 0x41
Result type	$[\epsilon]$	0x40 (-64 as s7)
Table type	$limits$ $reftype$	(none)
Memory type	$limits$	(none)
Global type	mut $valtype$	(none)

7.9 Index of Instructions

Instruction	Binary Opcode	Type	Validation	Execution
unreachable	0x00	$[t_1^*] \to [t_2^*]$	validation	execution
nop	0x01	$[] \to []$	validation	execution
block bt	0x02	$[t_1^*] \to [t_2^*]$	validation	execution
loop bt	0x03	$[t_1^*] \to [t_2^*]$	validation	execution
if bt	0x04	$[t_1^*\ \mathsf{i32}] \to [t_2^*]$	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 $x\ y$	0x11	$[t_1^*\ \mathsf{i32}] \to [t_2^*]$	validation	execution
return_call x	0x12	$[t_1^*] \to [t_2^*]$	validation	execution
return_call_indirect $x\ y$	0x13	$[t_1^*\ \mathsf{i32}] \to [t_2^*]$	validation	execution
call_ref x	0x14	$[t_1^*\ (\mathsf{ref\ null}\ x)] \to [t_2^*]$	validation	execution
return_call_ref x	0x15	$[t_1^*\ (\mathsf{ref\ null}\ x)] \to [t_2^*]$	validation	execution
(reserved)	0x16			
(reserved)	0x17			
(reserved)	0x18			
(reserved)	0x19			
drop	0x1A	$[t] \to []$	validation	execution
select	0x1B	$[t\ t\ \mathsf{i32}] \to [t]$	validation	execution
select t	0x1C	$[t\ t\ \mathsf{i32}] \to [t]$	validation	execution
(reserved)	0x1D			
(reserved)	0x1E			
(reserved)	0x1F			
local.get x	0x20	$[] \to [t]$	validation	execution
local.set x	0x21	$[t] \to []$	validation	execution
local.tee x	0x22	$[t] \to [t]$	validation	execution
global.get x	0x23	$[] \to [t]$	validation	execution
global.set x	0x24	$[t] \to []$	validation	execution
table.get x	0x25	$[\mathsf{i32}] \to [t]$	validation	execution
table.set x	0x26	$[\mathsf{i32}\ t] \to []$	validation	execution
(reserved)	0x27			
i32.load x $memarg$	0x28	$[\mathsf{i32}] \to [\mathsf{i32}]$	validation	execution
i64.load x $memarg$	0x29	$[\mathsf{i32}] \to [\mathsf{i64}]$	validation	execution
f32.load x $memarg$	0x2A	$[\mathsf{i32}] \to [\mathsf{f32}]$	validation	execution
f64.load x $memarg$	0x2B	$[\mathsf{i32}] \to [\mathsf{f64}]$	validation	execution
i32.load8_s x $memarg$	0x2C	$[\mathsf{i32}] \to [\mathsf{i32}]$	validation	execution
i32.load8_u x $memarg$	0x2D	$[\mathsf{i32}] \to [\mathsf{i32}]$	validation	execution
i32.load16_s x $memarg$	0x2E	$[\mathsf{i32}] \to [\mathsf{i32}]$	validation	execution
i32.load16_u x $memarg$	0x2F	$[\mathsf{i32}] \to [\mathsf{i32}]$	validation	execution
i64.load8_s x $memarg$	0x30	$[\mathsf{i32}] \to [\mathsf{i64}]$	validation	execution
i64.load8_u x $memarg$	0x31	$[\mathsf{i32}] \to [\mathsf{i64}]$	validation	execution
i64.load16_s x $memarg$	0x32	$[\mathsf{i32}] \to [\mathsf{i64}]$	validation	execution

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
i64.load16_u x $memarg$	0x33	[i32] → [i64]	validation	execution
i64.load32_s x $memarg$	0x34	[i32] → [i64]	validation	execution
i64.load32_u x $memarg$	0x35	[i32] → [i64]	validation	execution
i32.store x $memarg$	0x36	[i32 i32] → []	validation	execution
i64.store x $memarg$	0x37	[i32 i64] → []	validation	execution
f32.store x $memarg$	0x38	[i32 f32] → []	validation	execution
f64.store x $memarg$	0x39	[i32 f64] → []	validation	execution
i32.store8 x $memarg$	0x3A	[i32 i32] → []	validation	execution
i32.store16 x $memarg$	0x3B	[i32 i32] → []	validation	execution
i64.store8 x $memarg$	0x3C	[i32 i64] → []	validation	execution
i64.store16 x $memarg$	0x3D	[i32 i64] → []	validation	execution
i64.store32 x $memarg$	0x3E	[i32 i64] → []	validation	execution
memory.size x	0x3F	[] → [i32]	validation	execution
memory.grow x	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 (
i32.eq	0x46	[i32 i32] → [i32]	validation	execution (
i32.ne	0x47	[i32 i32] → [i32]	validation	execution (
i32.lt_s	0x48	[i32 i32] → [i32]	validation	execution (
i32.lt_u	0x49	[i32 i32] → [i32]	validation	execution (
i32.gt_s	0x4A	[i32 i32] → [i32]	validation	execution (
i32.gt_u	0x4B	[i32 i32] → [i32]	validation	execution (
i32.le_s	0x4C	[i32 i32] → [i32]	validation	execution (
i32.le_u	0x4D	[i32 i32] → [i32]	validation	execution (
i32.ge_s	0x4E	[i32 i32] → [i32]	validation	execution (
i32.ge_u	0x4F	[i32 i32] → [i32]	validation	execution (
i64.eqz	0x50	[i64] → [i32]	validation	execution (
i64.eq	0x51	[i64 i64] → [i32]	validation	execution (
i64.ne	0x52	[i64 i64] → [i32]	validation	execution (
i64.lt_s	0x53	[i64 i64] → [i32]	validation	execution (
i64.lt_u	0x54	[i64 i64] → [i32]	validation	execution (
i64.gt_s	0x55	[i64 i64] → [i32]	validation	execution (
i64.gt_u	0x56	[i64 i64] → [i32]	validation	execution (
i64.le_s	0x57	[i64 i64] → [i32]	validation	execution (
i64.le_u	0x58	[i64 i64] → [i32]	validation	execution (
i64.ge_s	0x59	[i64 i64] → [i32]	validation	execution (
i64.ge_u	0x5A	[i64 i64] → [i32]	validation	execution (
f32.eq	0x5B	[f32 f32] → [i32]	validation	execution (
f32.ne	0x5C	[f32 f32] → [i32]	validation	execution (
f32.lt	0x5D	[f32 f32] → [i32]	validation	execution (
f32.gt	0x5E	[f32 f32] → [i32]	validation	execution (
f32.le	0x5F	[f32 f32] → [i32]	validation	execution (
f32.ge	0x60	[f32 f32] → [i32]	validation	execution (
f64.eq	0x61	[f64 f64] → [i32]	validation	execution (
f64.ne	0x62	[f64 f64] → [i32]	validation	execution (
f64.lt	0x63	[f64 f64] → [i32]	validation	execution (
f64.gt	0x64	[f64 f64] → [i32]	validation	execution (
f64.le	0x65	[f64 f64] → [i32]	validation	execution (
f64.ge	0x66	[f64 f64] → [i32]	validation	execution (
i32.clz	0x67	[i32] → [i32]	validation	execution (

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
i32.ctz	0x68	[i32] → [i32]	validation	execution (
i32.popcnt	0x69	[i32] → [i32]	validation	execution (
i32.add	0x6A	[i32 i32] → [i32]	validation	execution (
i32.sub	0x6B	[i32 i32] → [i32]	validation	execution (
i32.mul	0x6C	[i32 i32] → [i32]	validation	execution (
i32.div_s	0x6D	[i32 i32] → [i32]	validation	execution (
i32.div_u	0x6E	[i32 i32] → [i32]	validation	execution (
i32.rem_s	0x6F	[i32 i32] → [i32]	validation	execution (
i32.rem_u	0x70	[i32 i32] → [i32]	validation	execution (
i32.and	0x71	[i32 i32] → [i32]	validation	execution (
i32.or	0x72	[i32 i32] → [i32]	validation	execution (
i32.xor	0x73	[i32 i32] → [i32]	validation	execution (
i32.shl	0x74	[i32 i32] → [i32]	validation	execution (
i32.shr_s	0x75	[i32 i32] → [i32]	validation	execution (
i32.shr_u	0x76	[i32 i32] → [i32]	validation	execution (
i32.rotl	0x77	[i32 i32] → [i32]	validation	execution (
i32.rotr	0x78	[i32 i32] → [i32]	validation	execution (
i64.clz	0x79	[i64] → [i64]	validation	execution (
i64.ctz	0x7A	[i64] → [i64]	validation	execution (
i64.popcnt	0x7B	[i64] → [i64]	validation	execution (
i64.add	0x7C	[i64 i64] → [i64]	validation	execution (
i64.sub	0x7D	[i64 i64] → [i64]	validation	execution (
i64.mul	0x7E	[i64 i64] → [i64]	validation	execution (
i64.div_s	0x7F	[i64 i64] → [i64]	validation	execution (
i64.div_u	0x80	[i64 i64] → [i64]	validation	execution (
i64.rem_s	0x81	[i64 i64] → [i64]	validation	execution (
i64.rem_u	0x82	[i64 i64] → [i64]	validation	execution (
i64.and	0x83	[i64 i64] → [i64]	validation	execution (
i64.or	0x84	[i64 i64] → [i64]	validation	execution (
i64.xor	0x85	[i64 i64] → [i64]	validation	execution (
i64.shl	0x86	[i64 i64] → [i64]	validation	execution (
i64.shr_s	0x87	[i64 i64] → [i64]	validation	execution (
i64.shr_u	0x88	[i64 i64] → [i64]	validation	execution (
i64.rotl	0x89	[i64 i64] → [i64]	validation	execution (
i64.rotr	0x8A	[i64 i64] → [i64]	validation	execution (
f32.abs	0x8B	[f32] → [f32]	validation	execution (
f32.neg	0x8C	[f32] → [f32]	validation	execution (
f32.ceil	0x8D	[f32] → [f32]	validation	execution (
f32.floor	0x8E	[f32] → [f32]	validation	execution (
f32.trunc	0x8F	[f32] → [f32]	validation	execution (
f32.nearest	0x90	[f32] → [f32]	validation	execution (
f32.sqrt	0x91	[f32] → [f32]	validation	execution (
f32.add	0x92	[f32 f32] → [f32]	validation	execution (
f32.sub	0x93	[f32 f32] → [f32]	validation	execution (
f32.mul	0x94	[f32 f32] → [f32]	validation	execution (
f32.div	0x95	[f32 f32] → [f32]	validation	execution (
f32.min	0x96	[f32 f32] → [f32]	validation	execution (
f32.max	0x97	[f32 f32] → [f32]	validation	execution (
f32.copysign	0x98	[f32 f32] → [f32]	validation	execution (
f64.abs	0x99	[f64] → [f64]	validation	execution (
f64.neg	0x9A	[f64] → [f64]	validation	execution (
f64.ceil	0x9B	[f64] → [f64]	validation	execution (
f64.floor	0x9C	[f64] → [f64]	validation	execution (

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
f64.trunc	0x9D	[f64] → [f64]	validation	execution (
f64.nearest	0x9E	[f64] → [f64]	validation	execution (
f64.sqrt	0x9F	[f64] → [f64]	validation	execution (
f64.add	0xA0	[f64 f64] → [f64]	validation	execution (
f64.sub	0xA1	[f64 f64] → [f64]	validation	execution (
f64.mul	0xA2	[f64 f64] → [f64]	validation	execution (
f64.div	0xA3	[f64 f64] → [f64]	validation	execution (
f64.min	0xA4	[f64 f64] → [f64]	validation	execution (
f64.max	0xA5	[f64 f64] → [f64]	validation	execution (
f64.copysign	0xA6	[f64 f64] → [f64]	validation	execution (
i32.wrap_i64	0xA7	[i64] → [i32]	validation	execution (
i32.trunc_f32_s	0xA8	[f32] → [i32]	validation	execution (
i32.trunc_f32_u	0xA9	[f32] → [i32]	validation	execution (
i32.trunc_f64_s	0xAA	[f64] → [i32]	validation	execution (
i32.trunc_f64_u	0xAB	[f64] → [i32]	validation	execution (
i64.extend_i32_s	0xAC	[i32] → [i64]	validation	execution (
i64.extend_i32_u	0xAD	[i32] → [i64]	validation	execution (
i64.trunc_f32_s	0xAE	[f32] → [i64]	validation	execution (
i64.trunc_f32_u	0xAF	[f32] → [i64]	validation	execution (
i64.trunc_f64_s	0xB0	[f64] → [i64]	validation	execution (
i64.trunc_f64_u	0xB1	[f64] → [i64]	validation	execution (
f32.convert_i32_s	0xB2	[i32] → [f32]	validation	execution (
f32.convert_i32_u	0xB3	[i32] → [f32]	validation	execution (
f32.convert_i64_s	0xB4	[i64] → [f32]	validation	execution (
f32.convert_i64_u	0xB5	[i64] → [f32]	validation	execution (
f32.demote_f64	0xB6	[f64] → [f32]	validation	execution (
f64.convert_i32_s	0xB7	[i32] → [f64]	validation	execution (
f64.convert_i32_u	0xB8	[i32] → [f64]	validation	execution (
f64.convert_i64_s	0xB9	[i64] → [f64]	validation	execution (
f64.convert_i64_u	0xBA	[i64] → [f64]	validation	execution (
f64.promote_f32	0xBB	[f32] → [f64]	validation	execution (
i32.reinterpret_f32	0xBC	[f32] → [i32]	validation	execution (
i64.reinterpret_f64	0xBD	[f64] → [i64]	validation	execution (
f32.reinterpret_i32	0xBE	[i32] → [f32]	validation	execution (
f64.reinterpret_i64	0xBF	[i64] → [f64]	validation	execution (
i32.extend8_s	0xC0	[i32] → [i32]	validation	execution (
i32.extend16_s	0xC1	[i32] → [i32]	validation	execution (
i64.extend8_s	0xC2	[i64] → [i64]	validation	execution (
i64.extend16_s	0xC3	[i64] → [i64]	validation	execution (
i64.extend32_s	0xC4	[i64] → [i64]	validation	execution (
(reserved)	0xC5			
(reserved)	0xC6			
(reserved)	0xC7			
(reserved)	0xC8			
(reserved)	0xC9			
(reserved)	0xCA			
(reserved)	0xCB			
(reserved)	0xCC			
(reserved)	0xCD			
(reserved)	0xCE			
(reserved)	0xCF			
ref.null ht	0xD0	[] → [(ref null ht)]	validation	execution
ref.is_null	0xD1	[(ref null ht)] → [i32]	validation	execution

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
ref.func x	0xD2	$[] \to [\text{ref } ht]$	validation	execution
ref.eq	0xD3	$[\text{eqref eqref}] \to [\text{i32}]$	validation	execution
ref.as_non_null	0xD4	$[(\text{ref null } ht)] \to [(\text{ref } ht)]$	validation	execution
br_on_null l	0xD5	$[t^* (\text{ref null } ht)] \to [t^* (\text{ref } ht)]$	validation	execution
br_on_non_null l	0xD6	$[t^* (\text{ref null } ht)] \to [t^*]$	validation	execution
(reserved)	0xD7			
(reserved)	0xD8			
(reserved)	0xD9			
(reserved)	0xDA			
(reserved)	0xDB			
(reserved)	0xDC			
(reserved)	0xDD			
(reserved)	0xDE			
(reserved)	0xDF			
(reserved)	0xE0			
(reserved)	0xE1			
(reserved)	0xE2			
(reserved)	0xE3			
(reserved)	0xE4			
(reserved)	0xE5			
(reserved)	0xE6			
(reserved)	0xE7			
(reserved)	0xE8			
(reserved)	0xE9			
(reserved)	0xEA			
(reserved)	0xEB			
(reserved)	0xEC			
(reserved)	0xED			
(reserved)	0xEE			
(reserved)	0xEF			
(reserved)	0xF0			
(reserved)	0xF1			
(reserved)	0xF2			
(reserved)	0xF3			
(reserved)	0xF4			
(reserved)	0xF5			
(reserved)	0xF6			
(reserved)	0xF7			
(reserved)	0xF8			
(reserved)	0xF9			
(reserved)	0xFA			
struct.new x	0xFB 0x00	$[t^*] \to [(\text{ref } x)]$	validation	execution
struct.new_default x	0xFB 0x01	$[] \to [(\text{ref } x)]$	validation	execution
struct.get x y	0xFB 0x02	$[(\text{ref null } x)] \to [t]$	validation	execution
struct.get_s x y	0xFB 0x03	$[(\text{ref null } x)] \to [\text{i32}]$	validation	execution
struct.get_u x y	0xFB 0x04	$[(\text{ref null } x)] \to [\text{i32}]$	validation	execution
struct.set x y	0xFB 0x05	$[(\text{ref null } x)\ t] \to []$	validation	execution
array.new x	0xFB 0x06	$[t] \to [(\text{ref } x)]$	validation	execution
array.new_default x	0xFB 0x07	$[\text{i32}] \to [(\text{ref } x)]$	validation	execution
array.new_fixed x n	0xFB 0x08	$[t^n] \to [(\text{ref } x)]$	validation	execution
array.new_data x y	0xFB 0x09	$[\text{i32 i32}] \to [(\text{ref } x)]$	validation	execution
array.new_elem x y	0xFB 0x0A	$[\text{i32 i32}] \to [(\text{ref } x)]$	validation	execution
array.get x	0xFB 0x0B	$[(\text{ref null } x)\ \text{i32}] \to [t]$	validation	execution

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
array.get_s x	0xFB 0x0C	[(ref null x) i32] → [i32]	validation	execution
array.get_u x	0xFB 0x0D	[(ref null x) i32] → [i32]	validation	execution
array.set x	0xFB 0x0E	[(ref null x) i32 t] → []	validation	execution
array.len	0xFB 0x0F	[(ref null array)] → [i32]	validation	execution
array.fill x	0xFB 0x10	[(ref null x) i32 t i32] → []	validation	execution
array.copy x y	0xFB 0x11	[(ref null x) i32 (ref null y) i32 i32] → []	validation	execution
array.init_data x y	0xFB 0x12	[(ref null x) i32 i32 i32] → []	validation	execution
array.init_elem x y	0xFB 0x13	[(ref null x) i32 i32 i32] → []	validation	execution
ref.test (ref t)	0xFB 0x14	[(ref t')] → [i32]	validation	execution
ref.test (ref null t)	0xFB 0x15	[(REF null t')] → [i32]	validation	execution
ref.cast (ref t)	0xFB 0x16	[(ref t')] → [(ref t)]	validation	execution
ref.cast (ref null t)	0xFB 0x17	[(ref null t')] → [(ref null t)]	validation	execution
br_on_cast t_1 t_2	0xFB 0x18	$[t_1] \to [t_1 \setminus t_2]$	validation	execution
br_on_cast_fail t_1 t_2	0xFB 0x19	$[t_1] \to [t_2]$	validation	execution
any.convert_extern	0xFB 0x1A	[(ref null extern)] → [(ref null any)]	validation	execution
extern.convert_any	0xFB 0x1B	[(ref null any)] → [(ref null extern)]	validation	execution
ref.i31	0xFB 0x1C	[i32] → [(ref i31)]	validation	execution
i31.get_s	0xFB 0x1D	[i31ref] → [i32]	validation	execution
i31.get_u	0xFB 0x1E	[i31ref] → [i32]	validation	execution
(reserved)	0xFB 0x1E . . .			
i32.trunc_sat_f32_s	0xFC 0x00	[f32] → [i32]	validation	execution (
i32.trunc_sat_f32_u	0xFC 0x01	[f32] → [i32]	validation	execution (
i32.trunc_sat_f64_s	0xFC 0x02	[f64] → [i32]	validation	execution (
i32.trunc_sat_f64_u	0xFC 0x03	[f64] → [i32]	validation	execution (
i64.trunc_sat_f32_s	0xFC 0x04	[f32] → [i64]	validation	execution (
i64.trunc_sat_f32_u	0xFC 0x05	[f32] → [i64]	validation	execution (
i64.trunc_sat_f64_s	0xFC 0x06	[f64] → [i64]	validation	execution (
i64.trunc_sat_f64_u	0xFC 0x07	[f64] → [i64]	validation	execution (
memory.init x y	0xFC 0x08	[i32 i32 i32] → []	validation	execution
data.drop x	0xFC 0x09	[] → []	validation	execution
memory.copy x y	0xFC 0x0A	[i32 i32 i32] → []	validation	execution
memory.fill y	0xFC 0x0B	[i32 i32 i32] → []	validation	execution
table.init x y	0xFC 0x0C	[i32 i32 i32] → []	validation	execution
elem.drop x	0xFC 0x0D	[] → []	validation	execution
table.copy x y	0xFC 0x0E	[i32 i32 i32] → []	validation	execution
table.grow x	0xFC 0x0F	[t i32] → [i32]	validation	execution
table.size x	0xFC 0x10	[] → [i32]	validation	execution
table.fill x	0xFC 0x11	[i32 t i32] → []	validation	execution
(reserved)	0xFC 0x1E . . .			
v128.load x $memarg$	0xFD 0x00	[i32] → [v128]	validation	execution
v128.load8x8_s x $memarg$	0xFD 0x01	[i32] → [v128]	validation	execution
v128.load8x8_u x $memarg$	0xFD 0x02	[i32] → [v128]	validation	execution
v128.load16x4_s x $memarg$	0xFD 0x03	[i32] → [v128]	validation	execution
v128.load16x4_u x $memarg$	0xFD 0x04	[i32] → [v128]	validation	execution
v128.load32x2_s x $memarg$	0xFD 0x05	[i32] → [v128]	validation	execution
v128.load32x2_u x $memarg$	0xFD 0x06	[i32] → [v128]	validation	execution
v128.load8_splat x $memarg$	0xFD 0x07	[i32] → [v128]	validation	execution
v128.load16_splat x $memarg$	0xFD 0x08	[i32] → [v128]	validation	execution
v128.load32_splat x $memarg$	0xFD 0x09	[i32] → [v128]	validation	execution
v128.load64_splat x $memarg$	0xFD 0x0A	[i32] → [v128]	validation	execution
v128.store x $memarg$	0xFD 0x0B	[i32 v128] → []	validation	execution
v128.const $i128$	0xFD 0x0C	[] → [v128]	validation	execution
i8x16.shuffle $laneidx$[16]	0xFD 0x0D	[v128 v128] → [v128]	validation	execution

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
i8x16.swizzle	0xFD 0x0E	[v128 v128] → [v128]	validation	execution
i8x16.splat	0xFD 0x0F	[i32] → [v128]	validation	execution
i16x8.splat	0xFD 0x10	[i32] → [v128]	validation	execution
i32x4.splat	0xFD 0x11	[i32] → [v128]	validation	execution
i64x2.splat	0xFD 0x12	[i64] → [v128]	validation	execution
f32x4.splat	0xFD 0x13	[f32] → [v128]	validation	execution
f64x2.splat	0xFD 0x14	[f64] → [v128]	validation	execution
i8x16.extract_lane_s *laneidx*	0xFD 0x15	[v128] → [i32]	validation	execution
i8x16.extract_lane_u *laneidx*	0xFD 0x16	[v128] → [i32]	validation	execution
i8x16.replace_lane *laneidx*	0xFD 0x17	[v128 i32] → [v128]	validation	execution
i16x8.extract_lane_s *laneidx*	0xFD 0x18	[v128] → [i32]	validation	execution
i16x8.extract_lane_u *laneidx*	0xFD 0x19	[v128] → [i32]	validation	execution
i16x8.replace_lane *laneidx*	0xFD 0x1A	[v128 i32] → [v128]	validation	execution
i32x4.extract_lane *laneidx*	0xFD 0x1B	[v128] → [i32]	validation	execution
i32x4.replace_lane *laneidx*	0xFD 0x1C	[v128 i32] → [v128]	validation	execution
i64x2.extract_lane *laneidx*	0xFD 0x1D	[v128] → [i64]	validation	execution
i64x2.replace_lane *laneidx*	0xFD 0x1E	[v128 i64] → [v128]	validation	execution
f32x4.extract_lane *laneidx*	0xFD 0x1F	[v128] → [f32]	validation	execution
f32x4.replace_lane *laneidx*	0xFD 0x20	[v128 f32] → [v128]	validation	execution
f64x2.extract_lane *laneidx*	0xFD 0x21	[v128] → [f64]	validation	execution
f64x2.replace_lane *laneidx*	0xFD 0x22	[v128 f64] → [v128]	validation	execution
i8x16.eq	0xFD 0x23	[v128 v128] → [v128]	validation	execution (
i8x16.ne	0xFD 0x24	[v128 v128] → [v128]	validation	execution (
i8x16.lt_s	0xFD 0x25	[v128 v128] → [v128]	validation	execution (
i8x16.lt_u	0xFD 0x26	[v128 v128] → [v128]	validation	execution (
i8x16.gt_s	0xFD 0x27	[v128 v128] → [v128]	validation	execution (
i8x16.gt_u	0xFD 0x28	[v128 v128] → [v128]	validation	execution (
i8x16.le_s	0xFD 0x29	[v128 v128] → [v128]	validation	execution (
i8x16.le_u	0xFD 0x2A	[v128 v128] → [v128]	validation	execution (
i8x16.ge_s	0xFD 0x2B	[v128 v128] → [v128]	validation	execution (
i8x16.ge_u	0xFD 0x2C	[v128 v128] → [v128]	validation	execution (
i16x8.eq	0xFD 0x2D	[v128 v128] → [v128]	validation	execution (
i16x8.ne	0xFD 0x2E	[v128 v128] → [v128]	validation	execution (
i16x8.lt_s	0xFD 0x2F	[v128 v128] → [v128]	validation	execution (
i16x8.lt_u	0xFD 0x30	[v128 v128] → [v128]	validation	execution (
i16x8.gt_s	0xFD 0x31	[v128 v128] → [v128]	validation	execution (
i16x8.gt_u	0xFD 0x32	[v128 v128] → [v128]	validation	execution (
i16x8.le_s	0xFD 0x33	[v128 v128] → [v128]	validation	execution (
i16x8.le_u	0xFD 0x34	[v128 v128] → [v128]	validation	execution (
i16x8.ge_s	0xFD 0x35	[v128 v128] → [v128]	validation	execution (
i16x8.ge_u	0xFD 0x36	[v128 v128] → [v128]	validation	execution (
i32x4.eq	0xFD 0x37	[v128 v128] → [v128]	validation	execution (
i32x4.ne	0xFD 0x38	[v128 v128] → [v128]	validation	execution (
i32x4.lt_s	0xFD 0x39	[v128 v128] → [v128]	validation	execution (
i32x4.lt_u	0xFD 0x3A	[v128 v128] → [v128]	validation	execution (
i32x4.gt_s	0xFD 0x3B	[v128 v128] → [v128]	validation	execution (
i32x4.gt_u	0xFD 0x3C	[v128 v128] → [v128]	validation	execution (
i32x4.le_s	0xFD 0x3D	[v128 v128] → [v128]	validation	execution (
i32x4.le_u	0xFD 0x3E	[v128 v128] → [v128]	validation	execution (
i32x4.ge_s	0xFD 0x3F	[v128 v128] → [v128]	validation	execution (
i32x4.ge_u	0xFD 0x40	[v128 v128] → [v128]	validation	execution (
f32x4.eq	0xFD 0x41	[v128 v128] → [v128]	validation	execution (
f32x4.ne	0xFD 0x42	[v128 v128] → [v128]	validation	execution (

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
f32x4.lt	0xFD 0x43	[v128 v128] → [v128]	validation	execution (
f32x4.gt	0xFD 0x44	[v128 v128] → [v128]	validation	execution (
f32x4.le	0xFD 0x45	[v128 v128] → [v128]	validation	execution (
f32x4.ge	0xFD 0x46	[v128 v128] → [v128]	validation	execution (
f64x2.eq	0xFD 0x47	[v128 v128] → [v128]	validation	execution (
f64x2.ne	0xFD 0x48	[v128 v128] → [v128]	validation	execution (
f64x2.lt	0xFD 0x49	[v128 v128] → [v128]	validation	execution (
f64x2.gt	0xFD 0x4A	[v128 v128] → [v128]	validation	execution (
f64x2.le	0xFD 0x4B	[v128 v128] → [v128]	validation	execution (
f64x2.ge	0xFD 0x4C	[v128 v128] → [v128]	validation	execution (
v128.not	0xFD 0x4D	[v128] → [v128]	validation	execution (
v128.and	0xFD 0x4E	[v128 v128] → [v128]	validation	execution (
v128.andnot	0xFD 0x4F	[v128 v128] → [v128]	validation	execution (
v128.or	0xFD 0x50	[v128 v128] → [v128]	validation	execution (
v128.xor	0xFD 0x51	[v128 v128] → [v128]	validation	execution (
v128.bitselect	0xFD 0x52	[v128 v128 v128] → [v128]	validation	execution (
v128.any_true	0xFD 0x53	[v128] → [i32]	validation	execution
v128.load8_lane *memarg laneidx*	0xFD 0x54	[i32 v128] → [v128]	validation	execution
v128.load16_lane *memarg laneidx*	0xFD 0x55	[i32 v128] → [v128]	validation	execution
v128.load32_lane *memarg laneidx*	0xFD 0x56	[i32 v128] → [v128]	validation	execution
v128.load64_lane *memarg laneidx*	0xFD 0x57	[i32 v128] → [v128]	validation	execution
v128.store8_lane *memarg laneidx*	0xFD 0x58	[i32 v128] → []	validation	execution
v128.store16_lane *memarg laneidx*	0xFD 0x59	[i32 v128] → []	validation	execution
v128.store32_lane *memarg laneidx*	0xFD 0x5A	[i32 v128] → []	validation	execution
v128.store64_lane *memarg laneidx*	0xFD 0x5B	[i32 v128] → []	validation	execution
v128.load32_zero *memarg*	0xFD 0x5C	[i32] → [v128]	validation	execution
v128.load64_zero *memarg*	0xFD 0x5D	[i32] → [v128]	validation	execution
f32x4.demote_f64x2_zero	0xFD 0x5E	[v128] → [v128]	validation	execution (
f64x2.promote_low_f32x4	0xFD 0x5F	[v128] → [v128]	validation	execution (
i8x16.abs	0xFD 0x60	[v128] → [v128]	validation	execution (
i8x16.neg	0xFD 0x61	[v128] → [v128]	validation	execution (
i8x16.popcnt	0xFD 0x62	[v128] → [v128]	validation	execution (
i8x16.all_true	0xFD 0x63	[v128] → [i32]	validation	execution
i8x16.bitmask	0xFD 0x64	[v128] → [i32]	validation	execution
i8x16.narrow_i16x8_s	0xFD 0x65	[v128 v128] → [v128]	validation	execution
i8x16.narrow_i16x8_u	0xFD 0x66	[v128 v128] → [v128]	validation	execution
f32x4.ceil	0xFD 0x67	[v128] → [v128]	validation	execution (
f32x4.floor	0xFD 0x68	[v128] → [v128]	validation	execution (
f32x4.trunc	0xFD 0x69	[v128] → [v128]	validation	execution (
f32x4.nearest	0xFD 0x6A	[v128] → [v128]	validation	execution (
i8x16.shl	0xFD 0x6B	[v128 i32] → [v128]	validation	execution (
i8x16.shr_s	0xFD 0x6C	[v128 i32] → [v128]	validation	execution (
i8x16.shr_u	0xFD 0x6D	[v128 i32] → [v128]	validation	execution (
i8x16.add	0xFD 0x6E	[v128 v128] → [v128]	validation	execution (
i8x16.add_sat_s	0xFD 0x6F	[v128 v128] → [v128]	validation	execution (
i8x16.add_sat_u	0xFD 0x70	[v128 v128] → [v128]	validation	execution (
i8x16.sub	0xFD 0x71	[v128 v128] → [v128]	validation	execution (
i8x16.sub_sat_s	0xFD 0x72	[v128 v128] → [v128]	validation	execution (
i8x16.sub_sat_u	0xFD 0x73	[v128 v128] → [v128]	validation	execution (
f64x2.ceil	0xFD 0x74	[v128] → [v128]	validation	execution (
f64x2.floor	0xFD 0x75	[v128] → [v128]	validation	execution (
i8x16.min_s	0xFD 0x76	[v128 v128] → [v128]	validation	execution (
i8x16.min_u	0xFD 0x77	[v128 v128] → [v128]	validation	execution (

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
i8x16.max_s	0xFD 0x78	[v128 v128] → [v128]	validation	execution (
i8x16.max_u	0xFD 0x79	[v128 v128] → [v128]	validation	execution (
f64x2.trunc	0xFD 0x7A	[v128] → [v128]	validation	execution (
i8x16.avgr_u	0xFD 0x7B	[v128 v128] → [v128]	validation	execution (
i16x8.extadd_pairwise_i8x16_s	0xFD 0x7C	[v128] → [v128]	validation	execution
i16x8.extadd_pairwise_i8x16_u	0xFD 0x7D	[v128] → [v128]	validation	execution
i32x4.extadd_pairwise_i16x8_s	0xFD 0x7E	[v128] → [v128]	validation	execution
i32x4.extadd_pairwise_i16x8_u	0xFD 0x7F	[v128] → [v128]	validation	execution
i16x8.abs	0xFD 0x80 0x01	[v128] → [v128]	validation	execution (
i16x8.neg	0xFD 0x81 0x01	[v128] → [v128]	validation	execution (
i16x8.q15mulr_sat_s	0xFD 0x82 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.all_true	0xFD 0x83 0x01	[v128] → [i32]	validation	execution
i16x8.bitmask	0xFD 0x84 0x01	[v128] → [i32]	validation	execution
i16x8.narrow_i32x4_s	0xFD 0x85 0x01	[v128 v128] → [v128]	validation	execution
i16x8.narrow_i32x4_u	0xFD 0x86 0x01	[v128 v128] → [v128]	validation	execution
i16x8.extend_low_i8x16_s	0xFD 0x87 0x01	[v128] → [v128]	validation	execution
i16x8.extend_high_i8x16_s	0xFD 0x88 0x01	[v128] → [v128]	validation	execution
i16x8.extend_low_i8x16_u	0xFD 0x89 0x01	[v128] → [v128]	validation	execution
i16x8.extend_high_i8x16_u	0xFD 0x8A 0x01	[v128] → [v128]	validation	execution
i16x8.shl	0xFD 0x8B 0x01	[v128 i32] → [v128]	validation	execution (
i16x8.shr_s	0xFD 0x8C 0x01	[v128 i32] → [v128]	validation	execution (
i16x8.shr_u	0xFD 0x8D 0x01	[v128 i32] → [v128]	validation	execution (
i16x8.add	0xFD 0x8E 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.add_sat_s	0xFD 0x8F 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.add_sat_u	0xFD 0x90 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.sub	0xFD 0x91 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.sub_sat_s	0xFD 0x92 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.sub_sat_u	0xFD 0x93 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.nearest	0xFD 0x94 0x01	[v128] → [v128]	validation	execution (
i16x8.mul	0xFD 0x95 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.min_s	0xFD 0x96 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.min_u	0xFD 0x97 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.max_s	0xFD 0x98 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.max_u	0xFD 0x99 0x01	[v128 v128] → [v128]	validation	execution (
(reserved)	0xFD 0x9A 0x01			
i16x8.avgr_u	0xFD 0x9B 0x01	[v128 v128] → [v128]	validation	execution (
i16x8.extmul_low_i8x16_s	0xFD 0x9C 0x01	[v128 v128] → [v128]	validation	execution
i16x8.extmul_high_i8x16_s	0xFD 0x9D 0x01	[v128 v128] → [v128]	validation	execution
i16x8.extmul_low_i8x16_u	0xFD 0x9E 0x01	[v128 v128] → [v128]	validation	execution
i16x8.extmul_high_i8x16_u	0xFD 0x9F 0x01	[v128 v128] → [v128]	validation	execution
i32x4.abs	0xFD 0xA0 0x01	[v128] → [v128]	validation	execution (
i32x4.neg	0xFD 0xA1 0x01	[v128] → [v128]	validation	execution (
(reserved)	0xFD 0xA2 0x01			
i32x4.all_true	0xFD 0xA3 0x01	[v128] → [i32]	validation	execution
i32x4.bitmask	0xFD 0xA4 0x01	[v128] → [i32]	validation	execution
(reserved)	0xFD 0xA5 0x01			
(reserved)	0xFD 0xA6 0x01			
i32x4.extend_low_i16x8_s	0xFD 0xA7 0x01	[v128] → [v128]	validation	execution
i32x4.extend_high_i16x8_s	0xFD 0xA8 0x01	[v128] → [v128]	validation	execution
i32x4.extend_low_i16x8_u	0xFD 0xA9 0x01	[v128] → [v128]	validation	execution
i32x4.extend_high_i16x8_u	0xFD 0xAA 0x01	[v128] → [v128]	validation	execution
i32x4.shl	0xFD 0xAB 0x01	[v128 i32] → [v128]	validation	execution (
i32x4.shr_s	0xFD 0xAC 0x01	[v128 i32] → [v128]	validation	execution (

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
i32x4.shr_u	0xFD 0xAD 0x01	[v128 i32] → [v128]	validation	execution (
i32x4.add	0xFD 0xAE 0x01	[v128 v128] → [v128]	validation	execution (
(reserved)	0xFD 0xAF 0x01			
(reserved)	0xFD 0xB0 0x01			
i32x4.sub	0xFD 0xB1 0x01	[v128 v128] → [v128]	validation	execution (
(reserved)	0xFD 0xB2 0x01			
(reserved)	0xFD 0xB3 0x01			
(reserved)	0xFD 0xB4 0x01			
i32x4.mul	0xFD 0xB5 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.min_s	0xFD 0xB6 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.min_u	0xFD 0xB7 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.max_s	0xFD 0xB8 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.max_u	0xFD 0xB9 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.dot_i16x8_s	0xFD 0xBA 0x01	[v128 v128] → [v128]	validation	execution
i32x4.extmul_low_i16x8_s	0xFD 0xBC 0x01	[v128 v128] → [v128]	validation	execution
i32x4.extmul_high_i16x8_s	0xFD 0xBD 0x01	[v128 v128] → [v128]	validation	execution
i32x4.extmul_low_i16x8_u	0xFD 0xBE 0x01	[v128 v128] → [v128]	validation	execution
i32x4.extmul_high_i16x8_u	0xFD 0xBF 0x01	[v128 v128] → [v128]	validation	execution
i64x2.abs	0xFD 0xC0 0x01	[v128] → [v128]	validation	execution (
i64x2.neg	0xFD 0xC1 0x01	[v128] → [v128]	validation	execution (
(reserved)	0xFD 0xC2 0x01			
i64x2.all_true	0xFD 0xC3 0x01	[v128] → [i32]	validation	execution
i64x2.bitmask	0xFD 0xC4 0x01	[v128] → [i32]	validation	execution
(reserved)	0xFD 0xC5 0x01			
(reserved)	0xFD 0xC6 0x01			
i64x2.extend_low_i32x4_s	0xFD 0xC7 0x01	[v128] → [v128]	validation	execution
i64x2.extend_high_i32x4_s	0xFD 0xC8 0x01	[v128] → [v128]	validation	execution
i64x2.extend_low_i32x4_u	0xFD 0xC9 0x01	[v128] → [v128]	validation	execution
i64x2.extend_high_i32x4_u	0xFD 0xCA 0x01	[v128] → [v128]	validation	execution
i64x2.shl	0xFD 0xCB 0x01	[v128 i32] → [v128]	validation	execution (
i64x2.shr_s	0xFD 0xCC 0x01	[v128 i32] → [v128]	validation	execution (
i64x2.shr_u	0xFD 0xCD 0x01	[v128 i32] → [v128]	validation	execution (
i64x2.add	0xFD 0xCE 0x01	[v128 v128] → [v128]	validation	execution (
(reserved)	0xFD 0xCF 0x01			
(reserved)	0xFD 0xD0 0x01			
i64x2.sub	0xFD 0xD1 0x01	[v128 v128] → [v128]	validation	execution (
(reserved)	0xFD 0xD2 0x01			
(reserved)	0xFD 0xD3 0x01			
(reserved)	0xFD 0xD4 0x01			
i64x2.mul	0xFD 0xD5 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.eq	0xFD 0xD6 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.ne	0xFD 0xD7 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.lt_s	0xFD 0xD8 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.gt_s	0xFD 0xD9 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.le_s	0xFD 0xDA 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.ge_s	0xFD 0xDB 0x01	[v128 v128] → [v128]	validation	execution (
i64x2.extmul_low_i32x4_s	0xFD 0xDC 0x01	[v128 v128] → [v128]	validation	execution
i64x2.extmul_high_i32x4_s	0xFD 0xDD 0x01	[v128 v128] → [v128]	validation	execution
i64x2.extmul_low_i32x4_u	0xFD 0xDE 0x01	[v128 v128] → [v128]	validation	execution
i64x2.extmul_high_i32x4_u	0xFD 0xDF 0x01	[v128 v128] → [v128]	validation	execution
f32x4.abs	0xFD 0xE0 0x01	[v128] → [v128]	validation	execution (
f32x4.neg	0xFD 0xE1 0x01	[v128] → [v128]	validation	execution (
(reserved)	0xFD 0xE2 0x01			

continues on ne

Table 2 – continued from previous page

Instruction	Binary Opcode	Type	Validation	Execution
f32x4.sqrt	0xFD 0xE3 0x01	[v128] → [v128]	validation	execution (
f32x4.add	0xFD 0xE4 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.sub	0xFD 0xE5 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.mul	0xFD 0xE6 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.div	0xFD 0xE7 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.min	0xFD 0xE8 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.max	0xFD 0xE9 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.pmin	0xFD 0xEA 0x01	[v128 v128] → [v128]	validation	execution (
f32x4.pmax	0xFD 0xEB 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.abs	0xFD 0xEC 0x01	[v128] → [v128]	validation	execution (
f64x2.neg	0xFD 0xED 0x01	[v128] → [v128]	validation	execution (
f64x2.sqrt	0xFD 0xEF 0x01	[v128] → [v128]	validation	execution (
f64x2.add	0xFD 0xF0 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.sub	0xFD 0xF1 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.mul	0xFD 0xF2 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.div	0xFD 0xF3 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.min	0xFD 0xF4 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.max	0xFD 0xF5 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.pmin	0xFD 0xF6 0x01	[v128 v128] → [v128]	validation	execution (
f64x2.pmax	0xFD 0xF7 0x01	[v128 v128] → [v128]	validation	execution (
i32x4.trunc_sat_f32x4_s	0xFD 0xF8 0x01	[v128] → [v128]	validation	execution (
i32x4.trunc_sat_f32x4_u	0xFD 0xF9 0x01	[v128] → [v128]	validation	execution (
f32x4.convert_i32x4_s	0xFD 0xFA 0x01	[v128] → [v128]	validation	execution (
f32x4.convert_i32x4_u	0xFD 0xFB 0x01	[v128] → [v128]	validation	execution (
i32x4.trunc_sat_f64x2_s_zero	0xFD 0xFC 0x01	[v128] → [v128]	validation	execution (
i32x4.trunc_sat_f64x2_u_zero	0xFD 0xFD 0x01	[v128] → [v128]	validation	execution (
f64x2.convert_low_i32x4_s	0xFD 0xFE 0x01	[v128] → [v128]	validation	execution (
f64x2.convert_low_i32x4_u	0xFD 0xFF 0x01	[v128] → [v128]	validation	execution (
(reserved)	0xFD 0x00 0x02 ...			
(reserved)	0xFE			
(reserved)	0xFF			

Note: Multi-byte opcodes are given with the shortest possible encoding in the table. However, what is following the first byte is actually a u32 with variable-length encoding and consequently has multiple possible representations.

7.10 Index of Semantic Rules

7.10.1 Well-formedness of Types

Construct	Judgement
Numeric type	$C \vdash \mathit{numtype} : \mathsf{ok}$
Vector type	$C \vdash \mathit{vectype} : \mathsf{ok}$
Heap type	$C \vdash \mathit{heaptype} : \mathsf{ok}$
Reference type	$C \vdash \mathit{reftype} : \mathsf{ok}$
Value type	$C \vdash \mathit{valtype} : \mathsf{ok}$
Packed type	$C \vdash \mathit{packtype} : \mathsf{ok}$
Storage type	$C \vdash \mathit{storagetype} : \mathsf{ok}$
Field type	$C \vdash \mathit{fieldtype} : \mathsf{ok}$
Result type	$C \vdash \mathit{resulttype} : \mathsf{ok}$
Instruction type	$C \vdash \mathit{instrtype} : \mathsf{ok}$
Function type	$C \vdash \mathit{functype} : \mathsf{ok}$
Structure type	$C \vdash \mathit{structtype} : \mathsf{ok}$
Array type	$C \vdash \mathit{arraytype} : \mathsf{ok}$
Composite type	$C \vdash \mathit{comptype} : \mathsf{ok}$
Sub type	$C \vdash \mathit{subtype} : \mathsf{ok}$
Recursive type	$C \vdash \mathit{rectype} : \mathsf{ok}$
Defined type	$C \vdash \mathit{deftype} : \mathsf{ok}$
Block type	$C \vdash \mathit{blocktype} : \mathit{instrtype}$
Table type	$C \vdash \mathit{tabletype} : \mathsf{ok}$
Memory type	$C \vdash \mathit{memtype} : \mathsf{ok}$
Global type	$C \vdash \mathit{globaltype} : \mathsf{ok}$
External type	$C \vdash \mathit{externtype} : \mathsf{ok}$
Type definitions	$C \vdash \mathit{type}^* : \mathsf{ok}$

7.10.2 Typing of Static Constructs

Construct	Judgement
Instruction	$S; C \vdash \mathit{instr} : \mathit{functype}$
Instruction sequence	$S; C \vdash \mathit{instr}^* : \mathit{functype}$
Expression	$C \vdash \mathit{expr} : \mathit{resulttype}$
Function	$C \vdash \mathit{func} : \mathit{functype}$
Local	$C \vdash \mathit{local} : \mathit{localtype}$
Table	$C \vdash \mathit{table} : \mathit{tabletype}$
Memory	$C \vdash \mathit{mem} : \mathit{memtype}$
Limits	$C \vdash \mathit{limits} : k$
Global	$C \vdash \mathit{global} : \mathit{globaltype}$
Element segment	$C \vdash \mathit{elem} : \mathit{reftype}$
Element mode	$C \vdash \mathit{elemmode} : \mathit{reftype}$
Data segment	$C \vdash \mathit{data} : \mathsf{ok}$
Data mode	$C \vdash \mathit{datamode} : \mathsf{ok}$
Start function	$C \vdash \mathit{start} : \mathsf{ok}$
Export	$C \vdash \mathit{export} : \mathit{externtype}$
Export description	$C \vdash \mathit{exportdesc} : \mathit{externtype}$
Import	$C \vdash \mathit{import} : \mathit{externtype}$
Import description	$C \vdash \mathit{importdesc} : \mathit{externtype}$
Module	$\vdash \mathit{module} : \mathit{externtype}^* \rightarrow \mathit{externtype}^*$

7.10.3 Typing of Runtime Constructs

Construct	Judgement
Value	$S \vdash \mathit{val} : \mathit{valtype}$
Result	$S \vdash \mathit{result} : \mathit{resulttype}$
Packed value	$S \vdash \mathit{packval} : \mathit{packtype}$
Field value	$S \vdash \mathit{fieldval} : \mathit{storagetype}$
External value	$S \vdash \mathit{externval} : \mathit{externtype}$
Function instance	$S \vdash \mathit{funcinst} : \mathit{functype}$
Table instance	$S \vdash \mathit{tableinst} : \mathit{tabletype}$
Memory instance	$S \vdash \mathit{meminst} : \mathit{memtype}$
Global instance	$S \vdash \mathit{globalinst} : \mathit{globaltype}$
Element instance	$S \vdash \mathit{eleminst} : t$
Data instance	$S \vdash \mathit{datainst} : \mathsf{ok}$
Structure instance	$S \vdash \mathit{structinst} : \mathsf{ok}$
Array instance	$S \vdash \mathit{arrayinst} : \mathsf{ok}$
Export instance	$S \vdash \mathit{exportinst} : \mathsf{ok}$
Module instance	$S \vdash \mathit{moduleinst} : C$
Store	$\vdash \mathit{store} : \mathsf{ok}$
Configuration	$\vdash \mathit{config} : [t^*]$
Thread	$S; \mathit{resulttype}^? \vdash \mathit{thread} : \mathit{resulttype}$
Frame	$S \vdash \mathit{frame} : C$

7.10.4 Constantness

Construct	Judgement
Constant expression	$C \vdash \mathit{expr}\,\mathsf{const}$
Constant instruction	$C \vdash \mathit{instr}\,\mathsf{const}$

7.10.5 Matching

Construct	Judgement
Number type	$C \vdash \mathit{numtype}_1 \leq \mathit{numtype}_2$
Vector type	$C \vdash \mathit{vectype}_1 \leq \mathit{vectype}_2$
Heap type	$C \vdash \mathit{heaptype}_1 \leq \mathit{heaptype}_2$
Reference type	$C \vdash \mathit{reftype}_1 \leq \mathit{reftype}_2$
Value type	$C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2$
Packed type	$C \vdash \mathit{packtype}_1 \leq \mathit{packtype}_2$
Storage type	$C \vdash \mathit{storagetype}_1 \leq \mathit{storagetype}_2$
Field type	$C \vdash \mathit{fieldtype}_1 \leq \mathit{fieldtype}_2$
Result type	$C \vdash \mathit{resulttype}_1 \leq \mathit{resulttype}_2$
Instruction type	$C \vdash \mathit{instrtype}_1 \leq \mathit{instrtype}_2$
Function type	$C \vdash \mathit{functype}_1 \leq \mathit{functype}_2$
Structure type	$C \vdash \mathit{structtype}_1 \leq \mathit{structtype}_2$
Array type	$C \vdash \mathit{arraytype}_1 \leq \mathit{arraytype}_2$
Composite type	$C \vdash \mathit{comptype}_1 \leq \mathit{comptype}_2$
Defined type	$C \vdash \mathit{deftype}_1 \leq \mathit{deftype}_2$
Table type	$C \vdash \mathit{tabletype}_1 \leq \mathit{tabletype}_2$
Memory type	$C \vdash \mathit{memtype}_1 \leq \mathit{memtype}_2$
Global type	$C \vdash \mathit{globaltype}_1 \leq \mathit{globaltype}_2$
External type	$C \vdash \mathit{externtype}_1 \leq \mathit{externtype}_2$
Limits	$C \vdash \mathit{limits}_1 \leq \mathit{limits}_2$

7.10.6 Store Extension

Construct	Judgement
Function instance	$\vdash funcinst_1 \preceq funcinst_2$
Table instance	$\vdash tableinst_1 \preceq tableinst_2$
Memory instance	$\vdash meminst_1 \preceq meminst_2$
Global instance	$\vdash globalinst_1 \preceq globalinst_2$
Element instance	$\vdash eleminst_1 \preceq eleminst_2$
Data instance	$\vdash datainst_1 \preceq datainst_2$
Structure instance	$\vdash structinst_1 \preceq structinst_2$
Array instance	$\vdash arrayinst_1 \preceq arrayinst_2$
Store	$\vdash store_1 \preceq store_2$

7.10.7 Execution

Construct	Judgement
Instruction	$S; F; instr^* \hookrightarrow S'; F'; instr'^*$
Expression	$S; F; expr \hookrightarrow S'; F'; expr'$

Index

Symbols

A

B

C

D

E

L

M

N

T

W