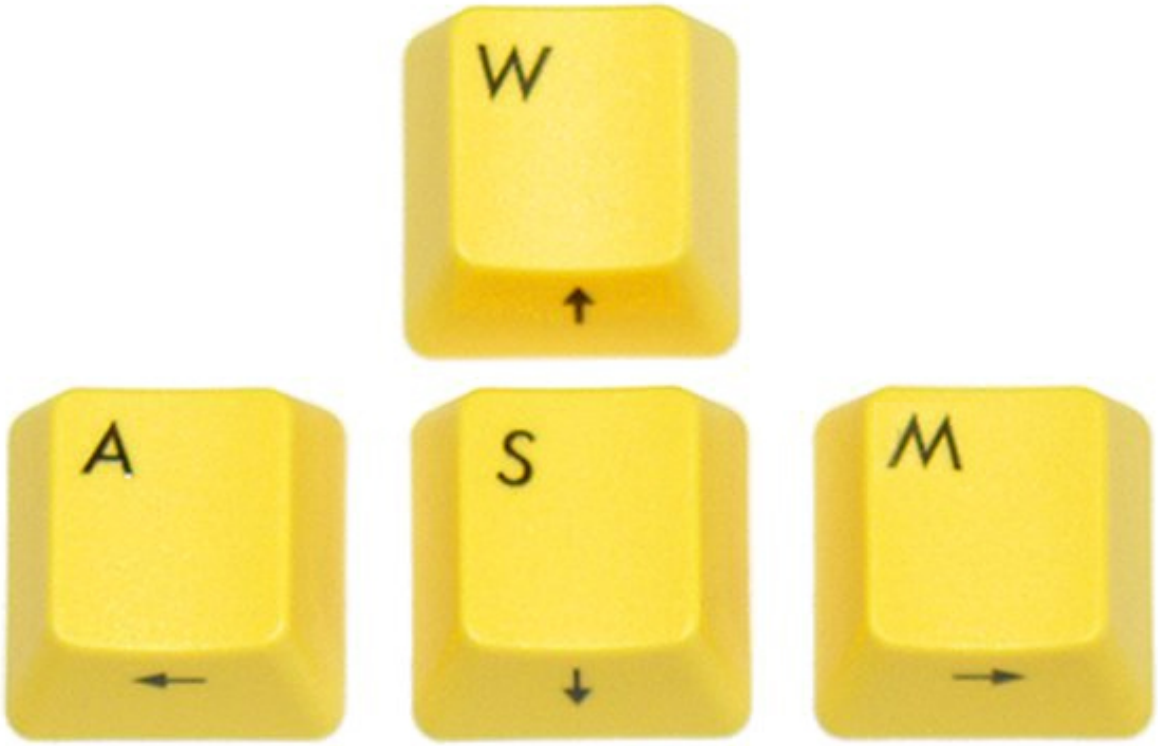

WebAssembly Specification

Release 1.0

WebAssembly Community Group

May 02, 2017

Contents

CHAPTER 1

Introduction

1.1 Introduction

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

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

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 It is intended that it will be superseded by new incremental releases with additional features in the future.

1.1.1 Design Goals

The design goals of WebAssembly are the following:

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

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

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

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

- **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.
- **Efficient**: can be decoded, validated, and compiled in a fast single pass, equally with either just-in-time (JIT) or ahead-of-time (AOT) compilation.
- **Streamable**: allows decoding, validation, and compilation to begin as soon as possible, before all data has been seen.
- **Parallelizable**: allows decoding, validation, and compilation to be split into many independent parallel tasks.
- **Portable**: makes no architectural assumptions that are not broadly supported across modern hardware.

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

1.1.2 Scope

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

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

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

1.2 Overview

1.2.1 Concepts

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

Values WebAssembly provides only four basic *value types*. These are integers and IEEE-754 floating point[4] numbers, each in 32 and 64 bit width. 32 bit integers also serve as Booleans and as memory addresses. The usual operations on these types are available, including the full matrix of conversions between them. There is no distinction between signed and unsigned integer types. Instead, integers are interpreted by respective operations as either unsigned or signed in 2's complement representation.

Instructions The computational model of WebAssembly is based on a *stack machine*. Code consists of sequences of *instructions* that are executed in order. Instructions manipulate values on an implicit *operand stack* [5] and fall into two main categories. Simple instructions perform basic operations on data. They pop arguments from the operand stack and push results back to it. *Control* instructions alter control flow. Control flow is *structured*, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.

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

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

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

Functions Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns a sequence of values as results. [6] Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable *local variables* that are usable as virtual registers.

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

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

Modules A WebAssembly binary takes the form of a *module* that contains definitions for functions, tables, and linear memories, as well as mutable or immutable *global variables*. Definitions can also be *imported*, specifying a module/name pair and a suitable type. Each definition can optionally be *exported* under one or more names. In addition to definitions, a module can define initialization data for its memory or table that takes the form of *segments* copied to given offsets. It 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. An *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 of an instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

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

CHAPTER 2

Structure

2.1 Conventions

WebAssembly is a programming language that does not have a concrete textual syntax (other than the auxiliary text format). For conciseness, however, its structure is described in the form of an *abstract syntax*. All parts of this specification are defined in terms of this abstract syntax, including the decoding of the *binary format*.

2.1.1 Grammar

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

- Terminal symbols (atoms) are written in sans-serif: $\mathsf{i32}, \mathsf{end}$.
- Nonterminal symbols are written in italic: $\mathit{valtype}, \mathit{instr}$.
- A^n is a sequence of $n \geq 0$ iterations of A.
- A^* is a possibly empty sequence of iterations of A. (This is a shorthand for A^n used where n is not relevant.)
- A^+ is a possibly 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 $n \leq 1$.)

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 of length n a sequence s that consists of its i-th to $(i+n-1)$-the element.
- s with $[i] = x$ denotes the same sequence as s, except that the i-the element is replaced with x.
- s with $[i : n] = x^n$ denotes the same sequence as s, except that the sub-sequence $s[i : n]$ is replaced with x^n.

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

$$r \ ::= \ \{\mathsf{field}_1 \ x_1, \mathsf{field}_2 \ x_2, \dots\}$$

The following notation is adopted for manipulating such records:

- $r.\mathsf{field}$ denotes the field component of r.
- r with $\mathsf{field} = x$ denotes the same record as r, except that the field component is replaced with x.

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

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

2.2 Values

2.2.1 Bytes

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

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

Conventions

- The meta variable b range 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 *size* and their *signedness*.

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

The latter class defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. In the abstract syntax, they are represented as unsigned. However, some operations convert them to signed based on a 2's complement interpretation.

Conventions

- The meta variables m, n, i range over integers.
- Numbers may be denoted by simple arithmetics, as in the grammar above.

2.2.3 Floating-Point

Floating-point data consists of values in binary floating-point format according to the IEEE 754[7] standard.

$$fN \quad ::= \quad byte^{N/8}$$

The two possible sizes N are 32 and 64.

2.2.4 Vectors

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

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

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

2.2.5 Names

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

$$\begin{array}{lcl} \mathit{name} & ::= & \mathit{codepoint}^* \\ \mathit{codepoint} & ::= & \text{U+0000} \mid \ldots \mid \text{U+D7FF} \mid \text{U+E000} \mid \ldots \mid \text{U+10FFFF} \end{array}$$

Todo

The definition of a name as an arbitrary sequence of scalar code points is too general. So would be the definition of a vector. Only names whose UTF-8 encoding is within the bounds of the maximum vector lengths must be included. How specify this?

Convention

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

2.3 Types

2.3.1 Value Types

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

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

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

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

Conventions

- The meta variable t ranges over value types where clear from context.
- The notation $|t|$ denotes the *width* of a value type in bytes. (That is, $|\mathsf{i32}| = |\mathsf{f32}| = 4$ and $|\mathsf{i64}| = |\mathsf{f64}| = 8$.)

2.3.2 Result Types

Result types classify the results of functions or blocks, which is a sequence of values.

$$\mathit{resulttype} \quad ::= \quad [\mathit{valtype}^?]$$

Note: In the current version of WebAssembly, at most one value is allowed as a result. However, this may be generalized to sequences of values in future versions.

[8] http://www.unicode.org/versions/latest/
[9] http://ieeexplore.ieee.org/document/4610935/

2.3.3 Function Types

Function types classify the signature of functions, mapping a vector of parameters to a vector of results.

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

Note: In the current version of WebAssembly, the length of the result type vector of a function may be at most 1. This restriction may be removed in future versions.

2.3.4 Limits

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

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

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

2.3.5 Memory Types

Memory types classify linear memories and their size range.

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

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

2.3.6 Table Types

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

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

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

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

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

2.3.7 Global Types

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

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

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 *returning* (pushing) result values.

Note: In the current version of WebAssembly, at most one result value can be pushed by a single instruction. This restriction may be lifted in future versions.

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

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

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

2.4.1 Numeric Instructions

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

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

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

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

Some integer instructions come in two flavours, where a signedness annotation sx distinguishes whether the operands are to be interpreted as *unsigned* or *signed* integers. For the other integer instructions, the use of 2's complement for the signed interpretation means that they behave the same regardless of signedness.

Conventions

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

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

2.4.2 Parametric Instructions

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

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

The drop operator simply throws away a single operand.

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

2.4.3 Variable Instructions

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

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

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

2.4.4 Memory Instructions

Instructions in this group are concerned with linear memory.

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

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

The static address offset is added to the dynamic address operand, yielding a 33 bit *effective address* that is the zero-based index at which the memory is accessed. All values are read and written in little endian[10] byte order. A

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

trap results if any of the accessed memory bytes lies outside the address range implied by the memory's current size.

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

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

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

The precise semantics of memory instructions is *described* in the Instruction section.

2.4.5 Control Instructions

Instructions in this group affect the flow of control.

$$
\begin{array}{rcl}
\mathit{instr} & ::= & \dots \mid \\
& & \mathsf{nop} \mid \\
& & \mathsf{unreachable} \mid \\
& & \mathsf{block}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{loop}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{if}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{br}\ \mathit{labelidx} \mid \\
& & \mathsf{br\_if}\ \mathit{labelidx} \mid \\
& & \mathsf{br\_table}\ \mathit{vec}(\mathit{labelidx})\ \mathit{labelidx} \mid \\
& & \mathsf{return} \mid \\
& & \mathsf{call}\ \mathit{funcidx} \mid \\
& & \mathsf{call\_indirect}\ \mathit{typeidx}
\end{array}
$$

The nop instruction does nothing.

The unreachable instruction causes an unconditional trap.

The block, loop and if instructions are *structured* instructions. They bracket nested sequences of instructions, called *blocks*, terminated with, or separated by, end or else pseudo-instructions. As the grammar prescribes, they must be well-nested. A structured instruction can produce a value as described by the annotated *result type*.

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

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

Branch instructions come in several flavors: br performs an unconditional branch, br_if performs a conditional branch, and br_table performs an indirect branch through an operand indexing into the label vector that is an immediate to the instruction, or to a default target if the operand is out of bounds. The return instruction is a shortcut for an unconditional branch to the outermost block, which implicitly is the body of the current function. Taking a branch *unwinds* the operand stack up to the height where the targeted structured control instruction

was entered. However, forward branches that target a control instruction with a non-empty result type consume a matching operand first and push it back on the operand stack after unwinding, as a result for the terminated instruction.

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

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

2.4.6 Expressions

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

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

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

2.5 Modules

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

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

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

2.5.1 Indices

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

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

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

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

Label indices reference block instructions inside an instruction sequence.

Conventions

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

2.5.2 Types

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

All function types used in a module must be defined in the type section. They are referenced by *type indices*.

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

2.5.3 Functions

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

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

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

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

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

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

2.5.4 Tables

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

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

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

Tables can be initialized through *element segments*.

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

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

2.5.5 Memories

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

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

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

Memories can be initialized through *data segments*.

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

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

2.5.6 Globals

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

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

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

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

2.5.7 Element Segments

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

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

The offset is given by a *constant expression*.

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

2.5.8 Data Segments

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

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

The offset is given by a *constant expression*.

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

2.5.9 Start Function

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

$$start \quad ::= \quad \{\text{func } funcidx\}$$

2.5.10 Exports

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

$$\begin{array}{lll} export & ::= & \{\text{name } name, \text{desc } exportdesc\} \\ exportdesc & ::= & \text{func } funcidx \mid \\ & & \text{table } tableidx \mid \\ & & \text{mem } memidx \mid \\ & & \text{global } globalidx \end{array}$$

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

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

Conventions

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

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

2.5.11 Imports

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

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

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

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

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

CHAPTER 3

Validation

3.1 Conventions

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

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

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

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

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

3.1.1 Contexts

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

- *Types*: the list of types defined in the current module.
- *Functions*: the list of functions declared in the current module, represented by their function type.
- *Tables*: the list of tables declared in the current module, represented by their table type.
- *Memories*: the list of memories declared in the current module, represented by their memory type.
- *Globals*: the list of globals declared in the current module, represented by their global type.
- *Locals*: the list of locals declared in the current function (including parameters), represented by their value type.
- *Labels*: the stack of labels accessible from the current position, represented by their result type.
- *Return*: the return type of the current function, represented as a result type.

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

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

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

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

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

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

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

3.1.2 Textual Notation

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

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

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

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

3.1.3 Formal Notation

Note: This section gives a brief explanation of the notation for specifying typing rules formally. For the interested reader, a more thorough introduction can be found in respective text books. [11]

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

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

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

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

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

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

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

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

An instruction like get_local can be typed as follows:

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

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

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

$$\frac{C, \mathsf{label}\ [t^?] \vdash instr^* : [] \to [t^?]}{C \vdash \mathsf{block}\ [t^?]\ instr^*\ \mathsf{end} : [] \to [t^?]}$$

A block instruction is only valid when the instruction sequence in its body is. Moreover, the result type must match the block's annotation $t^?$. If so, then the block instruction has the same type as the body. Inside the body an additional label of the same type is available, which is expressed by locally extending the context C with the additional label information for the premise.

3.2 Instructions

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

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

Typing extends to *instruction sequences* $instr^*$. Such a sequence has a *function types* $[t_1^*] \to [t_2^*]$ if the accumulative effect of executing the instructions is consuming values of types t_1^* off the operand stack and pushing new values of types t_2^*. For some instructions, the typing rules do not fully constrain the type, and therefor allow for multiple types. Such instructions are called *polymorphic*. Two degrees of polymorphism can be distinguished:

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

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

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

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

and

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

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

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

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

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

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

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

3.2.1 Numeric Instructions

$t.\mathsf{const}\ c$

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

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

$t.unop$

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

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

$t.binop$

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

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

$t.testop$

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

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

$t.relop$

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

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

$t_2.cvtop/t_1$

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

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

3.2.2 Parametric Instructions

drop

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

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

select

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

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

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

3.2.3 Variable Instructions

get_local x

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

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

set_local x

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

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

tee_local x

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

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

get_global x

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

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

set_global x

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

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

3.2.4 Memory Instructions

$t.\mathsf{load}\ memarg$

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

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

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

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

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

t.store $memarg$

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

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

t.store$N$ $memarg$

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

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

current_memory

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

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

grow_memory

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

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

3.2.5 Control Instructions

nop

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

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

unreachable

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

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

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

block $[t^?]$ $instr^*$ end

- Let C' be the same *context* as C, but with the *result type* $[t^?]$ prepended to the labels vector.
- Under context C', the instruction sequence $instr^*$ must be *valid* with type $[] \to [t^?]$.
- Then the compound instruction is valid with type $[] \to [t^?]$.

$$\frac{C, \mathsf{labels}\,[t^?] \vdash instr^* : [] \to [t^?]}{C \vdash \mathsf{block}\ [t^?]\ instr^*\ \mathsf{end} : [] \to [t^?]}$$

Note: The fact that the nested instruction sequence $instr^*$ must have type $[] \to [t^?]$ implies that it cannot access operands that have been pushed on the stack before the block was entered. This may be generalized in future versions of WebAssembly.

---

loop $[t^?]$ $instr^*$ end

- Let C' be the same *context* as C, but with the empty *result type* $[]$ prepended to the labels vector.
- Under context C', the instruction sequence $instr^*$ must be *valid* with type $[] \to [t^?]$.
- Then the compound instruction is valid with type $[] \to [t^?]$.

$$\frac{C, \mathsf{labels}\,[] \vdash instr^* : [] \to [t^?]}{C \vdash \mathsf{loop}\ [t^?]\ instr^*\ \mathsf{end} : [] \to [t^?]}$$

Note: The fact that the nested instruction sequence $instr^*$ must have type $[] \to [t^?]$ implies that it cannot access operands that have been pushed on the stack before the loop was entered. This may be generalized in future versions of WebAssembly.

---

if $[t^?]$ $instr_1^*$ else $instr_2^*$ end

- Let C' be the same *context* as C, but with the empty *result type* $[t^?]$ prepended to the labels vector.
- Under context C', the instruction sequence $instr_1^*$ must be *valid* with type $[] \to [t^?]$.
- Under context C', the instruction sequence $instr_2^*$ must be *valid* with type $[] \to [t^?]$.
- Then the compound instruction is valid with type $[] \to [t^?]$.

$$\frac{C, \mathsf{labels}\,[t^?] \vdash instr_1^* : [] \to [t^?] \qquad C, \mathsf{labels}\,[t^?] \vdash instr_2^* : [] \to [t^?]}{C \vdash \mathsf{if}\ [t^?]\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} : [\mathsf{i32}] \to [t^?]}$$

Note: The fact that the nested instruction sequence $instr^*$ must have type $[] \to [t^?]$ implies that it cannot access operands that have been pushed on the stack before the conditional was entered. This may be generalized in future versions of WebAssembly.

---

br l

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

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

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

br_if l

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

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

br_table $l^*\ l_N$

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

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

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

return

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

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

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

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

call x

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

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

call_indirect x

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

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

3.2.6 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

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

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

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

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

$$\frac{C \vdash \mathit{instr}^* : [t_1^*] \to [t_0^*\ t^*] \qquad C \vdash \mathit{instr}_N : [t^*] \to [t_3^*]}{C \vdash \mathit{instr}^*\ \mathit{instr}_N : [t_1^*] \to [t_0^*\ t_3^*]}$$

3.2.7 Expressions

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

$instr^*$ end

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

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

Constant Expressions

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

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

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

3.3 Modules

Modules are valid when all the definitions they contain are valid. To that end, each definition is classified with a suitable type.

3.3.1 Auxiliary Rules

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

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

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

3.3.2 Functions

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

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

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \to [t_2^*]$ be the *function type* $C.\mathsf{types}[x]$.
- The length of t_2^* must not be larger than 1.
- Let C' be the same *context* as C, but with:

  - locals set to the sequence of *value types* $t_1^*\ t^*$, concatenating parameters and locals,
  - labels set to the singular sequence containing only *result type* $[t_2^*]$.
  - return set to the *result type* $[t_2^*]$.
- Under the context C', the expression *expr* must be valid with type t_2^*.
- Then the function definition is valid with type $[t_1^*] \to [t_2^*]$.

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

Note: The restriction on the length of the result types t_2^* may be lifted in future versions of WebAssembly.

3.3.3 Tables

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

$\{\mathsf{type}\ limits\ elemtype\}$

- The limits *limits* must be *valid*.
- Then the table definition is valid with type *limits elemtype*.

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

3.3.4 Memories

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

$\{\mathsf{type}\ limits\}$

- The limits *limits* must be *valid*.
- Then the memory definition is valid with type *limits*.

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

3.3.5 Globals

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

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

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

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

3.3.6 Element Segments

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

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

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

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

3.3.7 Data Segments

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

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

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

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

3.3.8 Start Function

Start function declarations $start$ are not classified by any type.

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

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

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

3.3.9 Exports

Exports $export$ are classified by their export $name$. Export descriptions $exportdesc$ are not classified by any type.

$\{\text{name } name, \text{desc } exportdesc\}$

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

$$\frac{C \vdash exportdesc \text{ ok}}{C \vdash \{\text{name } name, \text{desc } exportdesc\} : name}$$

func x

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

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

table x

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

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

mem x

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

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

global x

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

$$\frac{C.\text{globals}[x] = \text{var } t}{C \vdash \text{global } x \text{ ok}}$$

3.3.10 Imports

Imports $import$ and import descriptions $importdesc$ are classified by *external types*.

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

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

$$\frac{C \vdash importdesc : externtype}{C \vdash \{\text{module } name_1, \text{name } name_2, \text{desc } importdesc\} : externtype}$$

func x

- The function $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \to [t_2^*]$ be the *function type* $C.\mathsf{types}[x]$.
- The length of t_2^* must not be larger than 1.
- Then the import description is valid with type $\mathsf{func}\ [t_1^*] \to [t_2^*]$.

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

Note: The restriction on the length of the result types t_2^* may be lifted in future versions of WebAssembly.

table *limits elemtype*

- The limits *limits* must be valid.
- Then the import description is valid with type table *limits elemtype*.

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

mem *limits*

- The limits *limits* must be valid.
- Then the import description is valid with type mem *limits*.

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

global *mut t*

- The mutability *mut* must be var.
- Then the import description is valid with type global t.

$$\frac{}{C \vdash \mathsf{global}\ \mathsf{var}\ t : \mathsf{global}\ \mathsf{var}\ t}$$

3.3.11 Modules

A module is entirely *closed*, that is, it only refers 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 types of definitions in the module itself.

- Let *module* be the module to validate.
- Let C be a *context* where:
  - $C.\mathsf{types}$ is $\mathit{module}.\mathsf{types}$,
  - $C.\mathsf{funcs}$ is $\mathsf{funcs}(\mathit{externtype}_i^*)$ concatenated with $\mathit{functype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{functype}_i^*$ as determined below,
  - $C.\mathsf{tables}$ is $\mathsf{tables}(\mathit{externtype}_i^*)$ concatenated with $\mathit{tabletype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{tabletype}_i^*$ as determined below,

  - C.mems is $\mathrm{mems}(\mathit{externtype}_i^*)$ concatenated with $\mathit{memtype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{memtype}_i^*$ as determined below,
  - C.globals is $\mathrm{globals}(\mathit{externtype}_i^*)$ concatenated with $\mathit{globaltype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{globaltype}_i^*$ as determined below.
  - C.locals is empty,
  - C.labels is empty.
  - C.return is empty.
- Under the context C:
  - For each func_i in module.funcs, the definition func_i must be *valid* with a *function type* $\mathit{functype}_i$.
  - For each table_i in module.tables, the definition table_i must be *valid* with a *table type* $\mathit{tabletype}_i$.
  - For each mem_i in module.mems, the definition mem_i must be *valid* with a *memory type* $\mathit{memtype}_i$.
  - For each global_i in module.globals:
    - Let C_i be the *context* where C_i.globals is the sequence $\mathrm{globals}(\mathit{externtype}_i^*)$ concatenated with $\mathit{globaltype}_0 \ \ldots \ \mathit{globaltype}_{i-1}$, and all other fields are empty.
    - Under the context C_i, the definition global_i must be *valid* with a *global type* $\mathit{globaltype}_i$.
  - For each elem_i in module.elem, the segment elem_i must be *valid*.
  - For each data_i in module.data, the segment elem_i must be *valid*.
  - 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* $\mathit{externtype}_i$.
  - For each export_i in module.exports, the segment import_i must be *valid* with a *name* name_i.
- The length of C.tables must not be larger than 1.
- The length of C.mems must not be larger than 1.
- All export names name_i must be different.

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

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

Globals, however, are not recursive. The effect of defining the limited context C' for validating the module's globals is that their initialization expressions can only access imported globals and nothing else.

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

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.

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.

Store, *stack*, and other *runtime structure*, such as *module instances*, are made precise in terms of additional auxiliary *syntax*.

4.1.1 Textual 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, which is referred to as the *current* frame.
- Both the store and the current frame are mutated by *replacing* some of its components. Such replacement is assumed to apply globally.
- The execution of an instruction may *trap*, in which case the entire computation is aborted and no further modifications to the store are performed by it. (Other computations can still be initiated afterwards.)
- The execution of an instruction may also end in a *jump* to a designated target, which defines the next instruction to execute.
- Execution can *enter* and *exit* instruction sequences in a block-like fashion.
- 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, with indications why these are known to hold.

4.1.2 Formal Notation

Note: This section gives a brief explanation of the notation for specifying execution formally. For the interested reader, a more thorough introduction can be found in respective text books. [13]

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

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

A *configuration* is a syntactic description of a program state. Each rule specifies one *step* of execution. As long as there is at most one reduction rule applicable to a given configuration, reduction – and thereby execution – is *deterministic*. WebAssembly has only very few exceptions to this, which are noted explicitly in this specification.

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

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

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

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

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

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

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

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

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

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

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

Note: For example, the following instruction sequence,

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

terminates after three steps:

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

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

---

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

4.2 Runtime Structure

4.2.1 Values

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

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

$$val \quad ::= \quad \text{i32.const } i32 \mid \text{i64.const } i64 \mid \text{f32.const } f32 \mid \text{f64.const } f64$$

4.2.2 Store

The *store* represents all global state that can be manipulated by WebAssembly programs. It consists of the runtime representation of all *instances* of *functions*, *tables*, *memories*, and *globals* that have been *allocated* during the life time of the abstract machine. [15]

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

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

Convention

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

4.2.3 Addresses

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

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

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

A *memory address* $memaddr$ denotes the abstract address *of* a memory *instance* in the store, not an offset *inside* a memory instance.

---

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

4.2.4 Module Instances

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

$$\begin{array}{lcll} moduleinst & ::= & \{ & \mathsf{types} \quad functype^*, \\ & & & \mathsf{funcs} \quad funcaddr^*, \\ & & & \mathsf{tables} \quad tableaddr^*, \\ & & & \mathsf{mems} \quad memaddr^*, \\ & & & \mathsf{globals} \quad globaladdr^* \\ & & & \mathsf{exports} \quad exportinst^* \ \} \end{array}$$

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

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

4.2.5 Function Instances

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

$$funcinst \quad ::= \quad \{\mathsf{module}\ moduleinst, \mathsf{code}\ func\}$$

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

Todo

Host functions?

4.2.6 Table Instances

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

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

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

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

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

4.2.7 Memory Instances

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

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

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

The bytes can be mutated through *memory instructions*, the execution of a *data segment*, or by external means provided by the embedder.

It is an invariant of the semantics that the length of the byte vector, divided by page size, never exceeds the maximum size, if present.

4.2.8 Global Instances

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

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

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

4.2.9 Export Instances

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

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

4.2.10 External Values

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

$$externval \quad ::= \quad \mathsf{func}\ funcaddr \mid \mathsf{table}\ tableaddr \mid \mathsf{mem}\ memaddr \mid \mathsf{global}\ globaladdr$$

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:

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

4.2.11 External Types

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

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

Conventions

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

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

4.2.12 Stack

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

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

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

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

Values

Values are represented by *themselves*.

Labels

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

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

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

Note: For example, a loop label has the form

$$\mathsf{label}_n\{\mathsf{loop}\ [t^?]\ \ldots\ \mathsf{end}\}$$

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

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

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

Frames

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

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

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

Conventions

- The meta variable L ranges over labels where clear from context.
- The meta variable F ranges over frames where clear from context.

Note: In the current version of WebAssembly, the arities of labels and activations cannot be larger than 1. This may be generalized in future versions.

4.2.13 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*:

$$\begin{array}{lll} \mathit{instr} & ::= & \ldots \mid \\ & & \mathsf{trap} \mid \\ & & \mathsf{invoke}\ \mathit{funcaddr} \mid \\ & & \mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{instr}^*\ \mathsf{end} \mid \\ & & \mathsf{frame}_n\{\mathit{frame}\}\ \mathit{instr}^*\ \mathsf{end} \mid \end{array}$$

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

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

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

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

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

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

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

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

Block Contexts

To express *branches*, the following syntax of *block contexts* is defined, indexed by the count k of labels surrounding the hole:

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

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

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

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

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

Evaluation Contexts

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

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

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

4.3 Numerics

Todo

Describe

4.3.1 Auxiliary Operations

$$\begin{array}{lcll} \mathrm{bytes}_N(i) & = & \epsilon & (N = 0 \wedge i = 0) \\ \mathrm{bytes}_N(i) & = & b \; \mathrm{bytes}_{N-8}(j) & (N \geq 8 \wedge i = 8 \cdot j + b) \end{array}$$

$$\begin{array}{lcl} \mathrm{bytes}_{\mathsf{i}N}(i) & = & \mathrm{bytes}_N(i) \\ \mathrm{bytes}_{\mathsf{f}N}(b^N) & = & \mathrm{reverse}(b^N) \end{array}$$

Note that bytes is a bijection, hence the function is invertible.

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

$$\begin{array}{lcl} \mathrm{extend}_{\mathsf{U},N}(i) & = & i \\ \mathrm{extend}_{\mathsf{S},N}(i) & = & \mathrm{signed}_N(i) \\ \mathrm{wrap}_N(i) & = & i \bmod 2^N \end{array}$$

Note: The index N of the extend function is the size extending *from*, where as the index of the wrap function is the size wrapping *to*.

4.3.2 Integer Operations

4.3.3 Floating-Point Operations

4.3.4 Conversions

4.4 Instructions

4.4.1 Numeric Instructions

$t.\mathsf{const}\ c$

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

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

$t.unop$

1. Assert: due to *validation*, a value of *value type* t is on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
3. If $unop_t(c_1)$ is defined, then:
   (a) Let c be the result of computing $unop_t(c_1)$.
   (b) Push the value $t.\mathsf{const}\ c$ to the stack.
4. Else:
   (a) Trap.

$$\begin{array}{lcll} (t.\mathsf{const}\ c_1)\ t.unop & \hookrightarrow & (t.\mathsf{const}\ c) & (\text{if } unop_t(c_1) = c) \\ (t.\mathsf{const}\ c_1)\ t.unop & \hookrightarrow & \mathsf{trap} & (\text{otherwise}) \end{array}$$

$t.binop$

1. Assert: due to *validation*, two values of *value type* t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. If $binop_t(c_1, c_2)$ is defined, then:
   (a) Let c be the result of computing $binop_t(c_1, c_2)$.
   (b) Push the value $t.\mathsf{const}\ c$ to the stack.
5. Else:
   (a) Trap.

$$\begin{array}{lcll} (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & (t.\mathsf{const}\ c) & (\text{if } binop_t(c_1, c_2) = c) \\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & \mathsf{trap} & (\text{otherwise}) \end{array}$$

$t.testop$

1. Assert: due to *validation*, a value of *value type* t is on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
3. Let c be the result of computing $testop_t(c_1)$.
4. Push the value $\mathsf{i32.const}\ c$ to the stack.

$$(t.\mathsf{const}\ c_1)\ t.testop \quad \hookrightarrow \quad (t.\mathsf{const}\ c) \qquad (\text{if } testop_t(c_1) = c)$$

$t.relop$

1. Assert: due to *validation*, two values of *value type* t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. Let c be the result of computing $relop_t(c_1, c_2)$.
5. Push the value $\mathsf{i32.const}\ c$ to the stack.

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

$t_2.cvtop/t_1$

1. Assert: due to *validation*, a value of *value type* t_1 is on the top of the stack.
2. Pop the value $t_1.\mathsf{const}\ c_1$ from the stack.
3. If $cvtop_{t_1,t_2}(c_1)$ is defined:
   (a) Let c_2 be the result of computing $cvtop_{t_1,t_2}(c_1)$.
   (b) Push the value $t_2.\mathsf{const}\ c_2$ to the stack.
4. Else:
   (a) Trap.

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

4.4.2 Parametric Instructions

drop

1. Assert: due to *validation*, a value is on the top of the stack.
2. Pop the value val from the stack.

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

select

1. Assert: due to *validation*, a value *value type* i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. Assert: due to *validation*, two more values (of the same *value type*) are on the top of the stack.
4. Pop the value val_2 from the stack.
5. Pop the value val_1 from the stack.
6. If c is not 0, then:
   (a) Push the value val_1 back to the stack.
7. Else:
   (a) Push the value val_2 back to the stack.

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

4.4.3 Variable Instructions

get_local x

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\mathsf{locals}[x]$ exists.
3. Let val be the value $F.\mathsf{locals}[x]$.
4. Push the value val to the stack.

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

set_local x

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\mathsf{locals}[x]$ exists.
3. Assert: due to *validation*, a value is on the top of the stack.
4. Pop the value val from the stack.
5. Replace $F.\mathsf{locals}[x]$ with the value val.

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

tee_local x

1. Assert: due to *validation*, a value is on the top of the stack.
2. Pop the value val from the stack.
3. Push the value val to the stack.
4. Push the value val to the stack.
5. *Execute* the instruction $(\mathsf{set\_local}\ x)$.

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

get_global x

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

$$S; F; (\text{get\_global } x) \quad \hookrightarrow \quad S; F; val \quad (\text{if } S.\text{globals}[F.\text{module.globals}[x]].\text{value} = val)$$

set_global x

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.globals$[x]$ exists.
3. Let a be the *global address* F.module.globals$[x]$.
4. Assert: due to *validation*, S.globals$[a]$ exists.
5. Let $glob$ be the *global instance* S.globals$[a]$.
6. Assert: due to *validation*, a value is on the top of the stack.
7. Pop the value val from the stack.
8. Replace $glob$.value with the value val.

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

4.4.4 Memory Instructions

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

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.mems$[0]$ exists.
3. Let a be the *memory address* F.module.mems$[0]$.
4. Assert: due to *validation*, S.mems$[a]$ exists.
5. Let mem be the *memory instance* S.mems$[a]$.
6. Assert: due to *validation*, a value *value type* i32 is on the top of the stack.
7. Pop the value i32.const i from the stack.
8. Let ea be $i + memarg$.offset.
9. If N is not part of the instruction, then:
   (a) Let N be the *width* $|t|$ of *value type* t.
10. If $ea + N$ is larger than the length of mem.data, then:
    (a) Trap.
11. Let b^* be the byte sequence mem.data$[ea : N]$.

12. If N and sx are part of the instruction, then:
    (a) Let n be the integer for which $\text{bytes}_N(n) = b^*$.
    (b) Let c be the result of computing $\text{extend}_{sx,N}(n)$.
13. Else:
    (a) Let c be the constant for which $\text{bytes}_t(c) = b^*$.
14. Push the value $t.\text{const}\ c$ to the stack.

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

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

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

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\text{module.mems}[0]$ exists.
3. Let a be the *memory address* $F.\text{module.mems}[0]$.
4. Assert: due to *validation*, $S.\text{mems}[a]$ exists.
5. Let mem be the *memory instance* $S.\text{mems}[a]$.
6. Assert: due to *validation*, a value *value type* i32 is on the top of the stack.
7. Pop the value $\text{i32.const}\ i$ from the stack.
8. Let ea be $i + memarg.\text{offset}$.
9. If N is not part of the instruction, then:
   (a) Let N be the *width* $|t|$ of *value type* t.
10. If $ea + N$ is larger than the length of $mem.\text{data}$, then:
    (a) Trap.
11. Assert: due to *validation*, a value of *value type* t is on the top of the stack.
12. Pop the value $t.\text{const}\ c$ from the stack.
13. If N is part of the instruction, then:
    (a) Let n be the result of computing $\text{wrap}_N(c)$.
    (b) Let b^* be the byte sequence $\text{bytes}_N(n)$.
14. Else:
    (a) Let b^* be the byte sequence $\text{bytes}_t(c)$.
15. Replace the bytes $mem.\text{data}[ea : N]$ with b^*.

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

current_memory

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\text{module.mems}[0]$ exists.
3. Let a be the *memory address* $F.\text{module.mems}[0]$.
4. Assert: due to *validation*, $S.\text{mems}[a]$ exists.
5. Let mem be the *memory instance* $S.\text{mems}[a]$.
6. Let sz be the length of $mem.\text{data}$ divided by the *page size*.
7. Push the value i32.const sz to the stack.

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

grow_memory

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

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

Note: The grow_memory instruction is non-deterministic. It may either succeed, returning the old memory size sz, or fail, returning -1. Failure *must* occur if the referenced memory instance has a maximum size defined that would be exceeded. However, failure *can* occur in other cases as well. In practice, the choice depends on the resources available to the embedder.

4.4.5 Control Instructions

nop

1. Do nothing.

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

unreachable

1. Trap.

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

block $[t^?]$ $instr^*$ end

1. Let n be the arity $|t^?|$ of the *result type* $t^?$.
2. Let L be the label whose arity is n and whose continuation is the end of the block.
3. *Enter* the instruction sequence $instr^*$ with label L.

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

loop $[t^?]$ $instr^*$ end

1. Let L be the label whose arity is 0 and whose continuation is the start of the loop.
2. *Enter* the instruction sequence $instr^*$ with label L.

$$\text{loop } [t^?]\ instr^*\ \text{end} \quad \hookrightarrow \quad \text{label}_0\{\text{loop } [t^?]\ instr^*\ \text{end}\}\ instr^*\ \text{end}$$

if $[t^?]$ $instr_1^*$ else $instr_2^*$ end

1. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. Let n be the arity $|t^?|$ of the *result type* $t^?$.
4. Let L be the label whose arity is n and whose continuation is the end of the if instruction.
5. If c is not 0, then:
   (a) *Enter* the instruction sequence $instr_1^*$ with label L.

6. Else:
   (a) *Enter* the instruction sequence $instr_2^*$ with label L.

$$\begin{array}{lcll} (\mathsf{i32.const}\ c)\ \mathsf{if}\ [t^n]\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_n\{\epsilon\}\ instr_1^*\ \mathsf{end} & (\text{if } c \neq 0) \\ (\mathsf{i32.const}\ c)\ \mathsf{if}\ [t^n]\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_n\{\epsilon\}\ instr_2^*\ \mathsf{end} & (\text{if } c = 0) \end{array}$$

br l

1. Assert: due to *validation*, the stack contains at least $l + 1$ labels.
2. Let L be the l-th label appearing on the stack, starting from the top and counting from zero.
3. Let n be the arity of L.
4. Assert: due to *validation*, there are at least n values on the top of the stack.
5. Pop the values val^n from the stack.
6. Repeat $l + 1$ times:
   (a) While the top of the stack is a value, do:
      i. Pop the value from the stack.
   (b) Assert: due to *validation*, the top of the stack now is a label.
   (c) Pop the label from the stack.
7. Push the values val^n to the stack.
8. Jump to the continuation of L.

$$\mathsf{label}_n\{instr^*\}\ B^l[val^n\ (\mathsf{br}\ l)]\ \mathsf{end} \quad \hookrightarrow \quad val^n\ instr^*$$

br_if l

1. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. If c is 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 $l^*\ l_N$

1. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
2. Pop the value i32.const i from the stack.
3. If i is smaller than the length of l^*, then:
   (a) Let l_i be the label $l^*[i]$.
   (b) *Execute* the instruction $(\mathsf{br}\ l_i)$.
4. Else:
   (a) *Execute* the instruction $(\mathsf{br}\ l_N)$.

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

return

1. Let F be the *current frame*.
2. Let n be the arity of F.
3. Assert: due to *validation*, there are at least n values on the top of the stack.
4. Pop the results val^n from the stack.
5. Assert: due to *validation*, the stack contains at least one *frame*.
6. While the top of the stack is not a frame, do:
   (a) Pop the top element from the stack.
7. Assert: the top of the stack is the frame F.
8. Pop the frame from the stack.
9. Push val^n to the stack.
10. Jump to the instruction after the original call.

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

call x

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\mathsf{module.funcs}[x]$ exists.
3. Let a be the *function address* $F.\mathsf{module.funcs}[x]$.
4. *Invoke* the function instance at address a.

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

call_indirect x

1. Let F be the *current frame*.
2. Assert: due to *validation*, $F.\mathsf{module.tables}[0]$ exists.
3. Let a be the *table address* $F.\mathsf{module.tables}[0]$.
4. Assert: due to *validation*, $S.\mathsf{tables}[a]$ exists.
5. Let tab be the *table instance* $S.\mathsf{tables}[a]$.
6. Assert: due to *validation*, $F.\mathsf{module.types}[x]$ exists.
7. Let ft_{expect} be the *function type* $F.\mathsf{module.types}[x]$.
8. Assert: due to *validation*, a value with *value type* i32 is on the top of the stack.
9. Pop the value $\mathsf{i32.const}\ i$ from the stack.
10. If i is not smaller than the length of $tab.\mathsf{elem}$, then:
    (a) Trap.
11. 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. Assert: due to *validation*, $f.\mathsf{module.types}[f.\mathsf{code.type}]$ exists.
16. Let ft_{actual} be the *function type* $f.\mathsf{module.types}[f.\mathsf{code.type}]$.
15. If ft_{actual} and ft_{expect} differ, then:
    (a) Trap.
17. *Invoke* the function instance at address a.

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

4.4.6 Instruction Sequences

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 labelled instruction sequence is reached without a jump or trap aborting it, then the following steps are performed.

1. Let n be the arity of L.
2. Assert: due to *validation*, there are n values on the top of the stack.
3. Pop the results val^n from the stack.
4. Assert: due to *validation*, the label L is now on the top of the stack.
5. Pop the label from the stack.
6. Push val^n back to the stack.
7. Jump to the position after the end of the *structured control instruction* associated with the label L.

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

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

4.4.7 Function Calls

Invocation of function address a

1. Assert: due to *validation*, $S.\mathsf{funcs}[a]$ exists.
2. Let f be the function instance, $S.\mathsf{funcs}[a]$.
3. Assert: due to *validation*, $f.\mathsf{module.types}[f.\mathsf{code.type}]$ exists.
4. Let $[t_1^n] \to [t_2^m]$ be the *function type* $f.\mathsf{module.types}[f.\mathsf{code.type}]$.
5. Let t^* be the list of *value types* $f.\mathsf{code.locals}$.
6. Let $instr^*$ end be the *expression* $f.\mathsf{code.body}$.
7. Assert: due to *validation*, n values are on the top of the stack.
8. Pop the values val^n from the stack.
9. Let val_0^* be the list of zero values of types t^*.
10. Let F be the *frame* $\{\mathsf{module}\ f.\mathsf{module}, \mathsf{locals}\ val^n\ val_0^*\}$.
11. Push the activation of F with arity m to the stack.
12. *Execute* the instruction $\mathsf{block}\ [t_2^m]\ instr^*\ \mathsf{end}$.

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

Returning from a function

When the end of a funtion is reached without a jump (return) or trap aborting it, then the following steps are performed.

1. Let F be the *current frame*.
2. Let n be the arity of the activation of F.
3. Assert: due to *validation*, there are n values on the top of the stack.
4. Pop the results val^n from the stack.
5. Assert: due to *validation*, the frame F is now on the top of the stack.
6. Pop the frame from the stack.
7. Push val^n back to the stack.
8. Jump to the instruction after the original call.

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

4.4.8 Expressions

$instr^*$ end

Todo

Define

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

4.5 Modules

Todo

Work in progress

CHAPTER 5

Binary Format

5.1 Conventions

The binary format for WebAssembly modules is a dense linear *encoding* of their *abstract syntax*. [16]

The format is defined by an *attribute grammar* whose only terminal symbols are *bytes*. A byte sequence is a well-formed encoding of a module if and only if it is generated by the grammar.

Each production of this grammar has exactly one synthesized attribute: the abstract syntax that the respective byte sequence encodes. Thus, the attribute grammar implicitly defines a *decoding* function.

Except for a few exceptions, the binary grammar closely mirrors the grammar of the abstract syntax.

Note: Some phrases of abstract syntax have multiple possible encodings in the binary format. For example, numbers may be encoded as if they had optional leading zeros. Implementations of decoders must support all possible alternatives; implementations of encoders can pick any one allowed encoding.

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 A^n used where n is not relevant.)
- $B^?$ is an optional occurrence of B. (This is a shorthand for A^n where $n \leq 1$.)
- $x{:}B$ denotes the same language as the nonterminal B, but also binds the variable x to the attribute synthesized for B.
- Productions are written `name` $::= B \Rightarrow A$, where A is the attribute that is synthesized for `name`, usually from attribute variables bound in B.
- 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.

[16] Additional encoding layers – for example, introducing compression – may be defined on top of the basic representation defined here. However, such layers are outside the scope of the current specification.

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

$$\begin{array}{lcll} \texttt{valtype} & ::= & \texttt{0x7F} & \Rightarrow \quad \texttt{i32} \\ & | & \texttt{0x7E} & \Rightarrow \quad \texttt{i64} \\ & | & \texttt{0x7D} & \Rightarrow \quad \texttt{f32} \\ & | & \texttt{0x7C} & \Rightarrow \quad \texttt{f64} \end{array}$$

Consequently, the byte 0x7F encodes the type i32, 0x7E encodes the type i64, and so forth. No other byte value is allowed as the encoding of a value type.

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

$$\begin{array}{lcll} \texttt{limits} & ::= & \texttt{0x00}\ n{:}\texttt{u32} & \Rightarrow \quad \{\min n, \max \epsilon\} \\ & | & \texttt{0x01}\ n{:}\texttt{u32}\ m{:}\texttt{u32} & \Rightarrow \quad \{\min n, \max m\} \end{array}$$

That is, a limits pair is encoded as either the byte 0x00 followed by the encoding of a $u32$ value, or the byte 0x01 followed by two such encodings. The variables n and m name the attributes of the respective u32 nonterminals, which in this case are the actual *unsigned integers* they decode into. The attribute of the complete production then is the abstract syntax for the limit, expressed in terms of the former values.

5.1.2 Auxiliary Notation

When dealing with binary encodings the following notation is also used:

- ϵ denotes the empty byte sequence.
- $||B||$ is the length of the byte sequence generated from the production B in a derivation.

5.2 Values

5.2.1 Bytes

Bytes encode themselves.

$$\begin{array}{lcll} \texttt{byte} & ::= & \texttt{0x00} & \Rightarrow \quad \texttt{0x00} \\ & | & & \dots \\ & | & \texttt{0xFF} & \Rightarrow \quad \texttt{0xFF} \end{array}$$

5.2.2 Integers

All *integers* are encoded using the LEB128[17] variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in unsigned LEB128[18] format. As an additional constraint, the total number of bytes encoding a value of type uN must not exceed $\text{ceil}(N/7)$ bytes.

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

Signed integers are encoded in signed LEB128[19] format, which uses a 2's complement representation. As an additional constraint, the total number of bytes encoding a value of type sN must not exceed $\text{ceil}(N/7)$ bytes.

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

[17] https://en.wikipedia.org/wiki/LEB128
[18] https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128
[19] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

Uninterpreted integers are encoded as signed integers.

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

Note: While the side conditions $N > 7$ in the productions for *non-terminating* bytes restrict the length of the u and s encodings, "trailing zeros" are still allowed within these bounds. For example, `0x03` and `0x83 0x00` are both well-formed encodings for the value 3 as a *u8*. Similarly, either of `0x7e` and `0xFE 0x7F` and `0xFE 0xFF 0x7F` are well-formed encodings of the value -2 as a *s16*.

The side conditions on the value n of *terminating* 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 bit pattern in little endian[20] byte order:

$$\texttt{f}N \quad ::= \quad b^*{:}\texttt{byte}^{N/8} \quad \Rightarrow \quad \text{reverse}(b^*)$$

Here, $\text{reverse}(b^*)$ denotes the byte sequence b^* in reversed order.

5.2.4 Vectors

Vectors are encoded with their length followed by the encoding of their element sequence.

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

5.2.5 Names

Names are encoded like a vector of bytes containing the UTF-8[21] encoding of the name's code point sequence.

$$\begin{array}{lcll}
\texttt{name} & ::= & n{:}\texttt{u32}\ (uc{:}\texttt{codepoint})^* & \Rightarrow \quad uc^* \qquad (|\texttt{codepoint}^*| \\
\texttt{codepoint} & ::= & uv{:}\texttt{codeval}_N & \Rightarrow \quad uv \qquad (uv \geq N \wedge (uv \\
\texttt{codeval}_N & ::= & b_1{:}\texttt{byte} & \Rightarrow \quad b_1 \qquad (b_1 < \texttt{0x80} \wedge N \\
 & | & b_1{:}\texttt{byte}\ b_2{:}\texttt{cont} & \Rightarrow \quad 2^6 \cdot (b_1 - \texttt{0xc0}) + b_2 \qquad (\texttt{0xc0} \leq b_1 < \texttt{0} \\
 & | & b_1{:}\texttt{byte}\ b_2{:}\texttt{cont}\ b_3{:}\texttt{cont} & \Rightarrow \quad 2^{12} \cdot (b_1 - \texttt{0xe0}) + 2^6 \cdot b_2 + b_3 \qquad (\texttt{0xe0} \leq b_1 < \texttt{0} \\
 & | & b_1{:}\texttt{byte}\ b_2{:}\texttt{cont}\ b_3{:}\texttt{cont}\ b_4{:}\texttt{cont} & \Rightarrow \quad 2^{18} \cdot (b_1 - \texttt{0xf0}) + 2^{12} \cdot b_2 + 2^6 \cdot b_3 + b_4 \qquad (\texttt{0xf0} \leq b_1 < \texttt{0} \\
\texttt{cont} & ::= & b{:}\texttt{byte} & \Rightarrow \quad b - \texttt{0x80} \qquad (b \geq \texttt{0x80})
\end{array}$$

Note: The *size*, $||\texttt{codepoint}^*||$ denotes the number of bytes in the encoding of the sequence, not the number of code points.

The index N to `codeval` is the minimum value that a given byte sequence may decode into. The respective side conditions on it exclude encodings using more than the minimal number of bytes to represent a code point.

---

[20] https://en.wikipedia.org/wiki/Endianness#Little-endian
[21] http://www.unicode.org/versions/latest/

5.3 Types

5.3.1 Value Types

Value types are encoded by a single byte.

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

Note: In future versions of WebAssembly, value types may include types denoted by *type indices*. Thus, the binary format for types corresponds to the encodings of small negative s values, so that they can coexist with (positive) type indices.

5.3.2 Result Types

The only *result types* occurring in the binary format are the types of blocks. These are encoded in special compressed form, by either the byte `0x40` indicating the empty type or as a single *value type*.

$$\begin{array}{rcll} \texttt{blocktype} & ::= & \texttt{0x40} & \Rightarrow \ [] \\ & | & t{:}\texttt{valtype} & \Rightarrow \ [t] \end{array}$$

Note: In future versions of WebAssembly, this scheme may be extended to support multiple results or more general block types.

5.3.3 Function Types

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

$$\texttt{functype} \ ::= \ \texttt{0x60} \ t_1^*{:}\texttt{vec(valtype)} \ t_2^*{:}\texttt{vec(valtype)} \ \Rightarrow \ [t_1^*] \to [t_2^*]$$

5.3.4 Limits

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

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

5.3.5 Memory Types

Memory types are encoded with their *limits*.

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

5.3.6 Table Types

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

```
tabletype  ::=  et:elemtype lim:limits  ⇒  lim et
elemtype   ::=  0x70                    ⇒  anyfunc
```

5.3.7 Global Types

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

```
globaltype  ::=  t:valtype m:mut  ⇒  m t
mut         ::=  0x00             ⇒  var
              |  0x01             ⇒  var
```

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 encoding instructions are reserved for future extensions.

5.4.1 Control Instructions

Control instructions have varying encodings. For structured instructions, the nested instruction sequences are terminated with explicit opcodes for end and else.

```
instr  ::=  0x00                                                  ⇒  unreachable
         |  0x01                                                  ⇒  nop
         |  0x02 rt:blocktype (in:instr)* 0x0B                    ⇒  block rt in* end
         |  0x03 rt:blocktype (in:instr)* 0x0B                    ⇒  loop rt in* end
         |  0x04 rt:blocktype (in:instr)* 0x0B                    ⇒  if rt in* else ϵ end
         |  0x04 rt:blocktype (in1:instr)* 0x05 (in2:instr)* 0x0B ⇒  if rt in1* else in2* end
         |  0x0C l:labelidx                                       ⇒  br l
         |  0x0D l:labelidx                                       ⇒  br_if l
         |  0x0E l*:vec(labelidx) lN:labelidx                     ⇒  br_table l* lN
         |  0x0F                                                  ⇒  return
         |  0x10 x:funcidx                                        ⇒  call x
         |  0x11 x:typeidx                                        ⇒  call_indirect x
```

Note: The else opcode 0x05 in the encoding of an if instruction can be omitted if the following instruction sequence is empty.

5.4.2 Parametric Instructions

Parametric instructions are represented by single byte codes.

```
instr ::= ...
        | 0x1A ⇒ drop
        | 0x1B ⇒ select
```

5.4.3 Variable Instructions

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

```
instr ::= ...
        | 0x20 x:localidx ⇒ get_local x
        | 0x21 x:localidx ⇒ set_local x
        | 0x22 x:localidx ⇒ tee_local x
        | 0x23 x:globalidx ⇒ get_global x
        | 0x24 x:globalidx ⇒ set_global x
```

5.4.4 Memory Instructions

Each variant of *memory instruction* is encoded with a different byte code. Loads and stores are followed by the encoding of their *memarg* immediate.

```
memarg ::= a:u32 o:u32 ⇒ {align a, offset o}
instr  ::= ...
         | 0x28 m:memarg ⇒ i32.load m
         | 0x29 m:memarg ⇒ i64.load m
         | 0x2A m:memarg ⇒ f32.load m
         | 0x2B m:memarg ⇒ f64.load m
         | 0x2C m:memarg ⇒ i32.load8_s m
         | 0x2D m:memarg ⇒ i32.load8_u m
         | 0x2E m:memarg ⇒ i32.load16_s m
         | 0x2F m:memarg ⇒ i32.load16_u m
         | 0x30 m:memarg ⇒ i64.load8_s m
         | 0x31 m:memarg ⇒ i64.load8_u m
         | 0x32 m:memarg ⇒ i64.load16_s m
         | 0x33 m:memarg ⇒ i64.load16_u m
         | 0x34 m:memarg ⇒ i64.load32_s m
         | 0x35 m:memarg ⇒ i64.load32_u m
         | 0x36 m:memarg ⇒ i32.store m
         | 0x37 m:memarg ⇒ i64.store m
         | 0x38 m:memarg ⇒ f32.store m
         | 0x39 m:memarg ⇒ f64.store m
         | 0x3A m:memarg ⇒ i32.store8 m
         | 0x3B m:memarg ⇒ i32.store16 m
         | 0x3C m:memarg ⇒ i64.store8 m
         | 0x3D m:memarg ⇒ i64.store16 m
         | 0x3E m:memarg ⇒ i64.store32 m
         | 0x3F 0x00 ⇒ current_memory
         | 0x40 0x00 ⇒ grow_memory
```

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

5.4.5 Numeric Instructions

All variants of *numeric instructions* are represented by separate byte codes.

The const instructions are followed by the respective literal.

```
instr ::= ...
        | 0x41 n:i32 ⇒ i32.const n
        | 0x42 n:i64 ⇒ i64.const n
        | 0x43 z:f32 ⇒ f32.const z
        | 0x44 z:f64 ⇒ f64.const z
```

All other numeric instructions are plain opcodes without any immediates.

```
instr ::= ...
        | 0x45 ⇒ i32.eqz
        | 0x46 ⇒ i32.eq
        | 0x47 ⇒ i32.ne
        | 0x48 ⇒ i32.lt_s
        | 0x49 ⇒ i32.lt_u
        | 0x4A ⇒ i32.gt_s
        | 0x4B ⇒ i32.gt_u
        | 0x4C ⇒ i32.le_s
        | 0x4D ⇒ i32.le_u
        | 0x4E ⇒ i32.ge_s
        | 0x4F ⇒ i32.ge_u

        | 0x50 ⇒ i64.eqz
        | 0x51 ⇒ i64.eq
        | 0x52 ⇒ i64.ne
        | 0x53 ⇒ i64.lt_s
        | 0x54 ⇒ i64.lt_u
        | 0x55 ⇒ i64.gt_s
        | 0x56 ⇒ i64.gt_u
        | 0x57 ⇒ i64.le_s
        | 0x58 ⇒ i64.le_u
        | 0x59 ⇒ i64.ge_s
        | 0x5A ⇒ i64.ge_u

        | 0x5B ⇒ f32.eq
        | 0x5C ⇒ f32.ne
        | 0x5D ⇒ f32.lt
        | 0x5E ⇒ f32.gt
        | 0x5F ⇒ f32.le
        | 0x60 ⇒ f32.ge

        | 0x61 ⇒ f64.eq
        | 0x62 ⇒ f64.ne
        | 0x63 ⇒ f64.lt
        | 0x64 ⇒ f64.gt
        | 0x65 ⇒ f64.le
        | 0x66 ⇒ f64.ge
```

| 0x67 ⇒ i32.clz
| 0x68 ⇒ i32.ctz
| 0x69 ⇒ i32.popcnt
| 0x6A ⇒ i32.add
| 0x6B ⇒ i32.sub
| 0x6C ⇒ i32.mul
| 0x6D ⇒ i32.div_s
| 0x6E ⇒ i32.div_u
| 0x6F ⇒ i32.rem_s
| 0x70 ⇒ i32.rem_u
| 0x71 ⇒ i32.and
| 0x72 ⇒ i32.or
| 0x73 ⇒ i32.xor
| 0x74 ⇒ i32.shl
| 0x75 ⇒ i32.shr_s
| 0x76 ⇒ i32.shr_u
| 0x77 ⇒ i32.rotl
| 0x78 ⇒ i32.rotr

| 0x79 ⇒ i64.clz
| 0x7A ⇒ i64.ctz
| 0x7B ⇒ i64.popcnt
| 0x7C ⇒ i64.add
| 0x7D ⇒ i64.sub
| 0x7E ⇒ i64.mul
| 0x7F ⇒ i64.div_s
| 0x80 ⇒ i64.div_u
| 0x81 ⇒ i64.rem_s
| 0x82 ⇒ i64.rem_u
| 0x83 ⇒ i64.and
| 0x84 ⇒ i64.or
| 0x85 ⇒ i64.xor
| 0x86 ⇒ i64.shl
| 0x87 ⇒ i64.shr_s
| 0x88 ⇒ i64.shr_u
| 0x89 ⇒ i64.rotl
| 0x8A ⇒ i64.rotr

| 0x8B ⇒ f32.abs
| 0x8C ⇒ f32.neg
| 0x8D ⇒ f32.ceil
| 0x8E ⇒ f32.floor
| 0x8F ⇒ f32.trunc
| 0x90 ⇒ f32.nearest
| 0x91 ⇒ f32.sqrt
| 0x92 ⇒ f32.add
| 0x93 ⇒ f32.sub
| 0x94 ⇒ f32.mul
| 0x95 ⇒ f32.div
| 0x96 ⇒ f32.min
| 0x97 ⇒ f32.max
| 0x98 ⇒ f32.copysign

0x99	⇒	f64.abs
0x9A	⇒	f64.neg
0x9B	⇒	f64.ceil
0x9C	⇒	f64.floor
0x9D	⇒	f64.trunc
0x9E	⇒	f64.nearest
0x9F	⇒	f64.sqrt
0xA0	⇒	f64.add
0xA1	⇒	f64.sub
0xA2	⇒	f64.mul
0xA3	⇒	f64.div
0xA4	⇒	f64.min
0xA5	⇒	f64.max
0xA6	⇒	f64.copysign

0xA7	⇒	i32.wrap/i64
0xA8	⇒	i32.trunc_s/f32
0xA9	⇒	i32.trunc_u/f32
0xAA	⇒	i32.trunc_s/f64
0xAB	⇒	i32.trunc_u/f64
0xAC	⇒	i64.extend_s/i32
0xAD	⇒	i64.extend_u/i32
0xAE	⇒	i64.trunc_s/f32
0xAF	⇒	i64.trunc_u/f32
0xB0	⇒	i64.trunc_s/f64
0xB1	⇒	i64.trunc_u/f64
0xB2	⇒	f32.convert_s/i32
0xB3	⇒	f32.convert_u/i32
0xB4	⇒	f32.convert_s/i64
0xB5	⇒	f32.convert_u/i64
0xB6	⇒	f32.demote/f64
0xB7	⇒	f64.convert_s/i32
0xB8	⇒	f64.convert_u/i32
0xB9	⇒	f64.convert_s/i64
0xBA	⇒	f64.convert_u/i64
0xBB	⇒	f64.promote/f32
0xBC	⇒	i32.reinterpret/f32
0xBD	⇒	i64.reinterpret/f64
0xBE	⇒	f32.reinterpret/i32
0xBF	⇒	f64.reinterpret/i64

5.4.6 Expressions

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

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

5.5 Modules

The binary encoding of modules is organized into *sections*. Most sections correspond to one component of a *module* record, except that *function definitions* are split into two sections, separating their type declarations in the *function section* from their bodies in the *code section*.

Note: This separation enables *parallel* and *streaming* compilation of the functions in a module.

5.5.1 Indices

All *indices* are encoded with their respective *u32* value.

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

5.5.2 Sections

Each section consists of

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

Every section is optional; an omitted section is equivalent to the section being present with empty contents.

The following parameterized grammar rule defines the generic structure of a section with id N and contents described by the grammar $\texttt{B}$.

$$\begin{array}{lcll} \texttt{section}_N(\texttt{B}) & ::= & N{:}\texttt{byte}\ \ \mathit{size}{:}\texttt{u32}\ \ \mathit{cont}{:}\texttt{B} & \Rightarrow\ \mathit{cont} \quad (\mathit{size} = ||\texttt{B}||) \\ & | & \epsilon & \Rightarrow\ \epsilon \end{array}$$

For most sections, the contents $\texttt{B}$ encodes a *vector*. In these cases, the empty result ϵ is interpreted as the empty vector.

Note: Other than for unknown *custom sections*, the *size* is not required for decoding, but can be used to skip sections when navigating through a binary. The module is malformed if the size does not match the length of the binary contents $\texttt{B}$.

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.

$$\begin{array}{lcl} \texttt{customsec} & ::= & \texttt{section}_0(\texttt{custom}) \\ \texttt{custom} & ::= & \texttt{name byte}^* \end{array}$$

Note: If an implementation interprets the contents of a custom section, then errors in that contents, or the placement of the section, must not invalidate the module.

5.5.4 Type Section

The *type section* has the id 1. It decodes into a vector of *function types* that represent the types component of a *module*.

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

5.5.5 Import Section

The *import section* has the id 2. It decodes into a vector of *imports* that represent the imports component of a *module*.

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

5.5.6 Function Section

The *function section* has the id 3. It decodes into a vector of *type indices* that represent the type fields of the *functions* in the funcs component of a *module*. The locals and body fields of the respective functions are encoded separately in the *code section*.

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

5.5.7 Table Section

The *table section* has the id 4. It decodes into a vector of *tables* that represent the tables component of a *module*.

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

5.5.8 Memory Section

The *memory section* has the id 5. It decodes into a vector of *memories* that represent the mems component of a *module*.

$$
\begin{array}{lcll}
\texttt{memsec} & ::= & mem^*{:}\texttt{section}_5(\texttt{vec}(\texttt{mem})) & \Rightarrow\ mem^* \\
\texttt{mem} & ::= & mt{:}\texttt{memtype} & \Rightarrow\ \{\text{type } mt\}
\end{array}
$$

5.5.9 Global Section

The *global section* has the id 6. It decodes into a vector of *globals* that represent the globals component of a *module*.

$$
\begin{array}{lcll}
\texttt{globalsec} & ::= & glob^*{:}\texttt{section}_6(\texttt{vec}(\texttt{global})) & \Rightarrow\ glob^* \\
\texttt{global} & ::= & gt{:}\texttt{globaltype}\ e{:}\texttt{expr} & \Rightarrow\ \{\text{type } gt, \text{init } e\}
\end{array}
$$

5.5.10 Export Section

The *export section* has the id 7. It decodes into a vector of *exports* that represent the exports component of a *module*.

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

5.5.11 Start Section

The *start section* has the id 8. It decodes into an optional *start function* that represents the start component of a *module*.

$$
\begin{array}{lcll}
\texttt{startsec} & ::= & st^?{:}\texttt{section}_8(\texttt{start}) & \Rightarrow\ st^? \\
\texttt{start} & ::= & x{:}\texttt{funcidx} & \Rightarrow\ \{\text{func } x\}
\end{array}
$$

5.5.12 Element Section

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

$$
\begin{array}{lcll}
\texttt{elemsec} & ::= & seg^*{:}\texttt{section}_9(\texttt{vec}(\texttt{elem})) & \Rightarrow\ seg \\
\texttt{elem} & ::= & x{:}\texttt{tableidx}\ \ e{:}\texttt{expr}\ \ y^*{:}\texttt{vec}(\texttt{funcidx}) & \Rightarrow\ \{\text{table } x, \text{offset } e, \text{init } y^*\}
\end{array}
$$

5.5.13 Code Section

The *code section* has the id 10. It decodes into a vector of *code* entries that are pairs of *value type* vectors and *expressions*. They represent the locals and body field of the *functions* in the funcs component of a *module*. The type fields of the respective functions are encoded separately in the *function section*.

The encoding of each code entry consists of

- the *u32* *size* of the function code in bytes,
- the actual *function code*, which in turn consists of
  - the declaration of *locals*,
  - the function *body* as an *expression*.

Local declarations are compressed into a vector whose entries consist of

- a *u32* *count*,
- a *value type*,

denoting *count* locals of the same value type.

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

Here, *code* ranges over pairs $(\mathit{valtype}^*, \mathit{expr})$. The meta function $\mathrm{concat}((t^*)^*)$ denotes the sequence of types formed by concatenating all sequences t_i^* in $(t^*)^*$. Any code for which the length of the resulting sequence is out of bounds of the maximum size of a *vector* is malformed.

Note: The *size* is not needed for decoding, but like with *sections*, can be used to skip functions when navigating through a binary. The module is malformed if a size does not match the length of the respective function code.

5.5.14 Data Section

The *data section* has the id 11. It decodes into a vector of *data segments* that represent the `data` component of a *module*.

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

5.5.15 Modules

The encoding of a *module* starts with a preamble containing a 4-byte magic number and a version field. The current version of the WebAssembly binary format is 1.

The preamble is followed by a sequence of *sections*. *Custom sections* may be inserted at any place in this sequence, while other sections must occur at most once and in the prescribed order. All sections can be empty. The lengths

of vectors produced by the (possibly empty) *function* and *code* section must match up.

$$
\begin{array}{lcl}
\texttt{magic} & ::= & \texttt{0x00 0x61 0x73 0x6D} \\
\texttt{version} & ::= & \texttt{0x01 0x00 0x00 0x00} \\
\texttt{module} & ::= & \texttt{magic} \\
 & & \texttt{version} \\
 & & \texttt{customsec}^* \\
 & & \mathit{functype}^* : \texttt{typesec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{import}^* : \texttt{importsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{typeidx}^n : \texttt{funcsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{table}^* : \texttt{tablesec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{mem}^* : \texttt{memsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{global}^* : \texttt{globalsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{export}^* : \texttt{exportsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{start}^? : \texttt{startsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{elem}^* : \texttt{elemsec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{code}^n : \texttt{codesec} \\
 & & \texttt{customsec}^* \\
 & & \mathit{data}^* : \texttt{datasec} \\
 & & \texttt{customsec}^* \quad \Rightarrow \quad \{\, \mathsf{types}\ \mathit{functype}^*, \\
 & & \qquad \mathsf{funcs}\ \mathit{func}^n, \\
 & & \qquad \mathsf{tables}\ \mathit{table}^*, \\
 & & \qquad \mathsf{mems}\ \mathit{mem}^*, \\
 & & \qquad \mathsf{globals}\ \mathit{global}^*, \\
 & & \qquad \mathsf{elem}\ \mathit{elem}^*, \\
 & & \qquad \mathsf{data}\ \mathit{data}^*, \\
 & & \qquad \mathsf{start}\ \mathit{start}^?, \\
 & & \qquad \mathsf{imports}\ \mathit{import}^*, \\
 & & \qquad \mathsf{exports}\ \mathit{export}^*\ \} \\
 & & (\text{where for each } t_i^*, e_i \text{ in } \mathit{code}^n, \\
 & & \ \mathit{func}^n[i] = \{\mathsf{type}\ \mathit{typeidx}^n[i], \mathsf{locals}\ t_i^*, \mathsf{body}\ e_i\})
\end{array}
$$

Note: The version of the WebAssembly binary format may increase in the future if backward-incompatible changes are 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

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

CHAPTER 7

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

CHAPTER 8

Appendix: Text Format

Todo

Describe

CHAPTER 9

Appendix: Name Section

Todo

Describe

Index of Instructions

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

Continued on next page

Table 10.1 – continued from previous page

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

Continued on next page

Table 10.1 – continued from previous page

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

Continued on next page

Table 10.1 – continued from previous page

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

Symbols

A

B

C

D

E

F

G

I

L

M

U

V

W