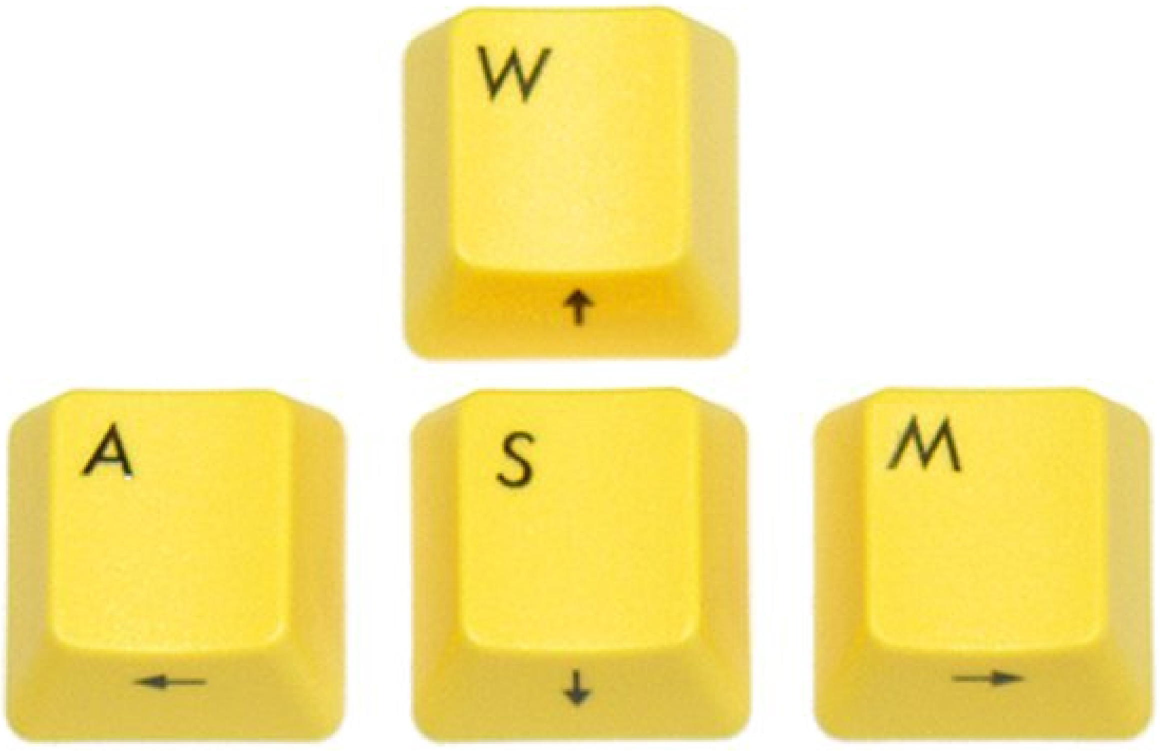

WebAssembly Specification

Release 1.0

WebAssembly Community Group

Mar 17, 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 It is intended that it will be superseded by new incremental releases with additional features in the future.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 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.

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.

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.

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

Conventions

- The meta variable b range over bytes.
- The meta function $\mathrm{byte}(n)$ denotes the byte representing the natural number $n < 256$.

2.2.2 Integers

Different classes of *integers* with different value ranges are distinguished by their *size* and their *signedness*.

$$\begin{array}{lll} \mathit{uint}_N & ::= & 0 \mid 1 \mid \dots \mid 2^N{-}1 \\ \mathit{sint}_N & ::= & -2^{N-1} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid 2^{N-1}{-}1 \\ \mathit{int}_N & ::= & \mathit{uint}_N \mid \mathit{sint}_N \end{array}$$

The latter class defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. A 2's complement conversion is assumed for values that are out-of-range for a chosen interpretation. That is, semantically, when interpreted as unsigned, negative values $-n$ convert to $2^N - n$, and when interpreted as signed, positive values $n \geq 2^{N-1}$ convert to $n - 2^N$.

Conventions

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

2.2.3 Floating-point Numbers

Floating-point numbers are represented as binary values according to the IEEE-754[7] standard.

$$\mathit{float}_N \quad ::= \quad \mathit{byte}^{N/8}$$

The two possible sizes N are 32 and 64.

Conventions

- The meta variable z ranges over floating point values.

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.

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

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

2.2.5 Names

Names are vectors of bytes interpreted as character strings.

$$name \quad ::= \quad vec(byte)$$

Todo

Unicode?

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.

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

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

The types $float_{32}$ and $float_{64}$ classify 32 and 64 bit floating points, respectively. They correspond to single and double precision floating point types as defined by the IEEE-754[8] 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.

$$resulttype \quad ::= \quad [valtype^?]$$

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.

2.3.3 Function Types

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

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

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

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 Memory Types

Memory types classify linear memories and their size range.

$$\begin{array}{lll} \mathit{memtype} & ::= & \mathit{limits} \\ \mathit{limits} & ::= & \{\mathsf{min}\ \mathit{uint}_{32}, \mathsf{max}\ \mathit{uint}_{32}^?\} \end{array}$$

The limits constrain the minimum and optionally the maximum size of a table. If no maximum is given, the table can grow to any size. Both values are given in units of page size.

2.3.5 Table Types

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

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

Like memories, tables are constrained by limits for their minimum and optionally the maximum size. These sizes 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.

2.3.6 Global Types

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

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

2.3.7 External Types

External types classify imports and exports and their respective types.

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

Conventions

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

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

2.4 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*.

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

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

2.4.1 Indices

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

$$
\begin{array}{lll}
\mathit{typeidx} & ::= & \mathit{uint}_{32} \\
\mathit{funcidx} & ::= & \mathit{uint}_{32} \\
\mathit{tableidx} & ::= & \mathit{uint}_{32} \\
\mathit{memidx} & ::= & \mathit{uint}_{32} \\
\mathit{globalidx} & ::= & \mathit{uint}_{32} \\
\mathit{localidx} & ::= & \mathit{uint}_{32} \\
\mathit{labelidx} & ::= & \mathit{uint}_{32} \\
\end{array}
$$

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.

2.4.2 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.

2.4.3 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*.

2.4.4 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.

2.4.5 Memories

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

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

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.

2.4.6 Globals

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

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

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

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

2.4.7 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.

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

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.

2.4.8 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.

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

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.

2.4.9 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.

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

2.4.10 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.

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

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.

2.4.11 Imports

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

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

Each import is 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.

2.5 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.

2.5.1 Numeric Instructions

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

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

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 sign interpretation is irrelevant under a 2's complement interpretation.

2.5.2 Parametric Instructions

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

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

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.

2.5.3 Variable Instructions

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

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

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

2.5.4 Memory Instructions

Instructions in this group are concerned with linear memory.

$$\begin{array}{lll} \mathit{memarg} & ::= & \{\mathsf{offset}\ \mathit{uint}_{32}, \mathsf{align}\ \mathit{uint}_{32}\} \\ \mathit{instr} & ::= & \dots \mid \\ && \mathsf{i}\mathit{nn}.\mathsf{load}\ \mathit{memarg} \mid \mathsf{f}\mathit{nn}.\mathsf{load}\ \mathit{memarg} \mid \\ && \mathsf{i}\mathit{nn}.\mathsf{store}\ \mathit{memarg} \mid \mathsf{f}\mathit{nn}.\mathsf{store}\ \mathit{memarg} \mid \\ && \mathsf{i}\mathit{nn}.\mathsf{load8\_}\mathit{sx}\ \mathit{memarg} \mid \mathsf{i}\mathit{nn}.\mathsf{load16\_}\mathit{sx}\ \mathit{memarg} \mid \mathsf{i64.load32\_}\mathit{sx}\ \mathit{memarg} \mid \\ && \mathsf{i}\mathit{nn}.\mathsf{store8}\ \mathit{memarg} \mid \mathsf{i}\mathit{nn}.\mathsf{store16}\ \mathit{memarg} \mid \mathsf{i64.store32}\ \mathit{memarg} \mid \\ && \mathsf{current\_memory} \mid \\ && \mathsf{grow\_memory} \end{array}$$

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[9] byte order. A trap results if any of the accessed memory bytes lies outside the address range implied by the memory's current size.

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

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

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

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

2.5.5 Control Instructions

Instructions in this group affect the flow of control.

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

The nop instruction does nothing.

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

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.

2.5.6 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.

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.

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

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

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

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. A phrase A is said to be "valid with type T", 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. [10]

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.

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

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

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

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 *instruction* i32.add 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:

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

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 $C.\mathsf{local}[x]$ 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 $[t_1^*] \rightarrow [t_2^*]$ for any possible sequences of value types t_1^* and t_2^*. Consequently,

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

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

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

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

3.2.1 Numeric Instructions

In this section, the following grammar shorthands are adopted:

$$\begin{array}{lll}
\mathit{unop} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \mid \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\
\mathit{binop} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{div\_}\mathit{sx} \mid \mathsf{rem\_}\mathit{sx} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \mid \\
 & & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}\mathit{sx} \mid \mathsf{rotl} \mid \mathsf{rotr} \\
\mathit{testop} & ::= & \mathsf{eqz} \\
\mathit{relop} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} \mid \mathsf{lt\_}\mathit{sx} \mid \mathsf{gt\_}\mathit{sx} \mid \mathsf{le\_}\mathit{sx} \mid \mathsf{ge\_}\mathit{sx} \\
\mathit{cvtop} & ::= & \mathsf{wrap} \mid \mathsf{extend\_}\mathit{sx} \mid \mathsf{trunc\_}\mathit{sx} \mid \mathsf{convert\_}\mathit{sx} \mid \mathsf{demote} \mid \mathsf{promote} \mid \mathsf{reinterpret}
\end{array}$$

$t.\mathsf{const}\ c$

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

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

$t.\mathit{unop}$

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

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

$t.\mathit{binop}$

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

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

$t.\mathit{testop}$

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

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

$t.\mathit{relop}$

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

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

$t_2.\mathit{cvtop}/t_1$

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

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

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 $[t] \rightarrow [t]$.

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

get_global x

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

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

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 mut.
- Then the instruction is valid with type $[t] \rightarrow []$.

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

3.2.4 Memory Instructions

t.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 $[\mathsf{i32}] \rightarrow [t]$.

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

t.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 $[\mathsf{i32}] \rightarrow [t]$.

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

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

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

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

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

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

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

current_memory

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

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

grow_memory

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

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

3.2.5 Control Instructions

nop

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

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

unreachable

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

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

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

$\mathsf{block}\ [t^?]\ \mathit{instr}^*\ \mathsf{end}$

- 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 $[] \to [t^?]$.

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

$\mathsf{loop}\ [t^?]\ \mathit{instr}^*\ \mathsf{end}$

- 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 $[] \to [t^?]$.

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

$\mathsf{if}\ [t^?]\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*\ \mathsf{end}$

- 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 $[] \to [t^?]$.

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

$\mathsf{br}\ l$

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

$\mathsf{br\_if}\ l$

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

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

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 $[t_1^* \ t^?] \to [t_2^*]$, for any sequences of *value types* t_1^* and t_2^*.

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

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

return

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

$$\frac{C.\mathsf{labels}[|C.\mathsf{labels}| - 1] = [t^?]}{C \vdash \mathsf{return} : [t_1^*\ t^?] \to [t_2^*]}$$

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

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

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

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: $instr^*\ instr_N$

- 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 $[t_1^*] \to [t_0^*\ t_3^*]$.

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

3.2.7 Expressions

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

$instr^*$ end

- The instruction sequence 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.const c,
  - or of the form get_global x, in which case C.globals$[x]$ must be a *global type* of the form 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.

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.types is module.types,
  - C.funcs is $\mathrm{funcs}(\mathit{externtype}_i^*)$ concatenated with $\mathit{functype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{functype}_i^*$ as determined below,
  - C.tables is $\mathrm{tables}(\mathit{externtype}_i^*)$ concatenated with $\mathit{tabletype}_i^*$, with the type sequences $\mathit{externtype}_i^*$ and $\mathit{tabletype}_i^*$ as determined below,
  - C.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.
- 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 $\mathit{globaltype}_0 \ \dots \ \mathit{globaltype}_{i-1}$, 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.

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

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 contexts C_i for validating the module's globals is that their initialization expressions can only access imported and previously defined globals and nothing else.

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

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

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

- The type $C.\mathsf{types}[x]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^*]$ be the *function type* $C.\mathsf{types}[x]$.
- The length of t_2^* must not be larger than 1.
- Let C' be the same *context* as C, but with:
  - the locals set to the sequence of *value types* $t_1^*~t^*$, concatenating parameters and locals,
  - the labels set to the singular sequence with *result type* (t_2^*).
- Under the context C', the expression expr must be valid with type t_2^*.
- Then the function definition is valid with type $[t_1^*] \rightarrow [t_2^*]$.

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

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.

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

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}\ \mathit{elemtype}$ be the *table type* $C.\mathsf{tables}[x]$.
- The *element type* $\mathit{elemtype}$ 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.

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

3.3.7 Data Segments

Data segments data are not classified by any type.

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

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

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

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 $\mathit{exportdesc}$ are not classified by any type.

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

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

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

$\mathsf{func}\ x$

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

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

$\mathsf{table}\ x$

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

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

$\mathsf{mem}\ x$

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

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

$\mathsf{global}\ x$

- 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 const.
- Then the export description is valid.

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

3.3.10 Imports

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

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

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

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

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 $\mathsf{table}\ limits\ elemtype$.

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

mem $limits$

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

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

global $mut\ t$

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

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

CHAPTER 4

Instantiation

CHAPTER 5

Execution

CHAPTER 6

Instructions

6.1 Overview

Todo

Describe

6.2 Control Instructions

6.2.1 Block Instructions

6.2.2 Branch Instructions

6.2.3 Call Instructions

6.2.4 Miscellaneous Control Instructions

6.3 Variable Instructions

6.4 Parametric Instructions

6.5 Numeric Instructions

6.5.1 Numeric Instructions

6.5.2 Integer Test Instructions

6.5.3 Integer Comparison Instructions

6.5.4 Floating Point Comparison Instructions

6.5.5 Unary Integer Instructions

6.5.6 Unary Floating Point Instructions

6.5.7 Binary Integer Instructions

6.5.8 Binary Floating Point Instructions

6.5.9 Conversion Instructions

6.5.10 Reinterpretation Instructions

6.6 Memory Instructions

6.6.1 Load Instructions

6.6.2 Memory Instructions

6.6.3 Store Instructions

6.7 Instruction Sequences

Todo

Describe

CHAPTER 7

Binary Format

CHAPTER 8

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

CHAPTER 9

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

CHAPTER 10

Appendix: Text Format

Todo

Describe

CHAPTER 11

Appendix: Name Section

Todo

Describe