CHAPTER 3

Validation

3.1 Conventions

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

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

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

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

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

3.1.1 Contexts

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

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

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

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

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

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

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

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

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

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

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

3.1.2 Prose Notation

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

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

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

- The rules implicitly assume a given context C.

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

3.1.3 Formal Notation

Note: This section gives a brief explanation of the notation for specifying typing rules formally. For the interested reader, a more thorough introduction can be found in respective text books.[16]

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

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

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

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

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

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

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

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

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

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

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

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

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

---

[16] For example: Benjamin Pierce. Types and Programming Languages[Page 27, 17]. The MIT Press 2002

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

3.2 Types

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

3.2.1 Limits

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

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

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

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

3.2.2 Block Types

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

$typeidx$

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

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

$[valtype^?]$

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

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

3.2.3 Function Types

Function types are always valid.

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

- The function type is valid.

$$\frac{}{\vdash [t_1^*] \to [t_2^*]\ \text{ok}}$$

3.2.4 Table Types

limits reftype

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

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

3.2.5 Memory Types

limits

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

$$\frac{\vdash \mathit{limits} : 2^{16}}{\vdash \mathit{limits}\ \text{ok}}$$

3.2.6 Global Types

mut valtype

- The global type is valid.

$$\frac{}{\vdash \mathit{mut}\ \mathit{valtype}\ \text{ok}}$$

3.2.7 External Types

func *functype*

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

$$\frac{\vdash \mathit{functype}\ \text{ok}}{\vdash \text{func}\ \mathit{functype}\ \text{ok}}$$

table $tabletype$

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

$$\frac{\vdash tabletype \text{ ok}}{\vdash \text{table } tabletype \text{ ok}}$$

mem $memtype$

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

$$\frac{\vdash memtype \text{ ok}}{\vdash \text{mem } memtype \text{ ok}}$$

global $globaltype$

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

$$\frac{\vdash globaltype \text{ ok}}{\vdash \text{global } globaltype \text{ ok}}$$

3.2.8 Import Subtyping

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

Limits

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

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

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

Functions

An external type func $functype_1$ matches func $functype_2$ if and only if:

- Both $functype_1$ and $functype_2$ are the same.

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

Tables

An external type table $(limits_1\ reftype_1)$ matches table $(limits_2\ reftype_2)$ if and only if:

- Limits $limits_1$ match $limits_2$.
- Both $reftype_1$ and $reftype_2$ are the same.

$$\frac{\vdash limits_1 \leq limits_2}{\vdash \mathsf{table}\ (limits_1\ reftype) \leq \mathsf{table}\ (limits_2\ reftype)}$$

Memories

An external type mem $limits_1$ matches mem $limits_2$ if and only if:

- Limits $limits_1$ match $limits_2$.

$$\frac{\vdash limits_1 \leq limits_2}{\vdash \mathsf{mem}\ limits_1 \leq \mathsf{mem}\ limits_2}$$

Globals

An external type global $globaltype_1$ matches global $globaltype_2$ if and only if:

- Both $globaltype_1$ and $globaltype_2$ are the same.

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

3.3 Instructions

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

$$\begin{array}{lcl} stacktype & ::= & [opdtype^*] \to [opdtype^*] \\ opdtype & ::= & valtype \mid \bot \end{array}$$

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

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

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

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

Typing extends to instruction sequences $instr^*$. Such a sequence has a stack type $[t_1^*] \to [t_2^*]$ 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 therefore allow for multiple types. Such instructions are called *polymorphic*. Two degrees of polymorphism can be distinguished:

- *value-polymorphic*: the value type t of one or several individual operands is unconstrained. That is the case for all parametric instructions like drop and select.
- *stack-polymorphic*: the entire (or most of the) stack type $[t_1^*] \to [t_2^*]$ 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 $[t\ t\ \text{i32}] \to [t]$, for any possible number type t. Consequently, both instruction sequences

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

and

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

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 operand types t_1^* and t_2^*. Consequently,

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

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

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

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

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

3.3.1 Numeric Instructions

$t.\text{const}\ c$

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

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

$t.unop$

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

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

$t.binop$

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

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

$t.testop$

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

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

$t.relop$

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

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

$t_2.cvtop\_t_1\_sx^?$

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

$$\frac{}{C \vdash t_2.cvtop\_t_1\_sx^? : [t_1] \to [t_2]}$$

3.3.2 Reference Instructions

$\mathsf{ref.null}\ t$

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

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

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

$\mathsf{ref.is\_null}$

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

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

ref.func x

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

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

3.3.3 Vector Instructions

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

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

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

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

v128.const c

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

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

v128.vvunop

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

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

v128.$\mathit{vvbinop}$

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

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

v128.$\mathit{vvternop}$

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

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

v128.$vvtestop$

- The instruction is valid with type [v128] → [i32].

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

i8x16.swizzle

- The instruction is valid with type [v128 v128] → [v128].

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

i8x16.shuffle $laneidx^{16}$

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

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

$shape$.splat

- Let t be $\text{unpacked}(shape)$.
- The instruction is valid with type $[t]$ → [v128].

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

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

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

$$\frac{laneidx < \dim(shape)}{C \vdash shape.\text{extract\_lane\_}sx^?\ laneidx : [\text{v128}] \to [\text{unpacked}(shape)]}$$

$shape$.replace_lane $laneidx$

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

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

shape.*vunop*

- The instruction is valid with type [v128] → [v128].

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

shape.*vbinop*

- The instruction is valid with type [v128 v128] → [v128].

$$\overline{C \vdash shape.vbinop : [\mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

shape.*vrelop*

- The instruction is valid with type [v128 v128] → [v128].

$$\overline{C \vdash shape.vrelop : [\mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

ishape.*vishiftop*

- The instruction is valid with type [v128 i32] → [v128].

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

shape.*vtestop*

- The instruction is valid with type [v128] → [i32].

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

shape.*vcvtop*_*half*?_*shape*_*sx*?_zero?

- The instruction is valid with type [v128] → [v128].

$$\overline{C \vdash shape.vcvtop\_half^?\_shape\_sx^?\_\mathsf{zero}^? : [\mathsf{v128}] \to [\mathsf{v128}]}$$

$ishape_1$.narrow_$ishape_2$_sx

- The instruction is valid with type [v128 v128] → [v128].

$$\overline{C \vdash ishape_1.\mathsf{narrow}\_ishape_2\_sx : [\mathsf{v128}\ \mathsf{v128}] \to [\mathsf{v128}]}$$

ishape.bitmask

- The instruction is valid with type [v128] → [i32].

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

$ishape_1$.dot_$ishape_2$_s

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

$$\frac{}{C \vdash ishape_1\text{.dot\_}ishape_2\text{\_s} : [\text{v128 v128}] \to [\text{v128}]}$$

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

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

$$\frac{}{C \vdash ishape_1\text{.extmul\_}half\text{\_}ishape_2\text{\_}sx : [\text{v128 v128}] \to [\text{v128}]}$$

$ishape_1$.extadd_pairwise_$ishape_2$_sx

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

$$\frac{}{C \vdash ishape_1\text{.extadd\_pairwise\_}ishape_2\text{\_}sx : [\text{v128}] \to [\text{v128}]}$$

3.3.4 Parametric Instructions

drop

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

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

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

select $(t^*)^?$

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

$$\frac{}{C \vdash \text{select } t : [t\ t\ \text{i32}] \to [t]} \qquad \frac{\vdash t \leq numtype}{C \vdash \text{select} : [t\ t\ \text{i32}] \to [t]} \qquad \frac{\vdash t \leq vectype}{C \vdash \text{select} : [t\ t\ \text{i32}] \to [t]}$$

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

3.3.5 Variable Instructions

local.get x

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

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

local.set x

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

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

local.tee x

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

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

global.get x

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

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

global.set x

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

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

3.3.6 Table Instructions

table.get x

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

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

table.set x

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

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

table.size x

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

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

table.grow x

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

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

table.fill x

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

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

table.copy x y

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

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

table.init x y

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

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

elem.drop x

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

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

3.3.7 Memory Instructions

t.load memarg

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

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

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

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

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

$t.\mathsf{store}\ memarg$

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

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

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

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

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

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

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

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

$\mathsf{v128.load}N\mathsf{\_splat}\ memarg$

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

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

v128.load$N$_zero $memarg$

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

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

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

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

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

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

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

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

memory.size

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

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

memory.grow

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

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

memory.fill

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

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

memory.copy

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

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

memory.init x

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

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

data.drop x

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

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

3.3.8 Control Instructions

nop

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

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

unreachable

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

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

Note: The unreachable instruction is stack-polymorphic.

block $blocktype$ $instr^*$ end

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

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

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

loop $blocktype$ $instr^*$ end

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

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

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

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

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

$$\frac{C \vdash blocktype : [t_1^*] \to [t_2^*] \qquad C, \mathsf{labels}\,[t_2^*] \vdash instr_1^* : [t_1^*] \to [t_2^*] \qquad C, \mathsf{labels}\,[t_2^*] \vdash instr_2^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{if}\ blocktype\ instr_1^*\ \mathsf{else}\ instr_2^*\ \mathsf{end} : [t_1^*\ \mathsf{i32}] \to [t_2^*]}$$

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

br l

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

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

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

The br instruction is stack-polymorphic.

br_if l

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

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

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

br_table $l^*\ l_N$

- The label $C.\mathsf{labels}[l_N]$ must be defined in the context.
- For each label l_i in l^*, the label $C.\mathsf{labels}[l_i]$ must be defined in the context.
- There must be a sequence t^* of operand types, such that:
  - The length of the sequence t^* is the same as the length of the sequence $C.\mathsf{labels}[l_N]$.
  - For each operand type t_j in t^* and corresponding type t'_{Nj} in $C.\mathsf{labels}[l_N]$, t_j matches t'_{Nj}.
  - For each label l_i in l^*:
    - The length of the sequence t^* is the same as the length of the sequence $C.\mathsf{labels}[l_i]$.
    - For each operand type t_j in t^* and corresponding type t'_{ij} in $C.\mathsf{labels}[l_i]$, t_j matches t'_{ij}.
- Then the instruction is valid with type $[t_1^*\ t^*\ \mathsf{i32}] \to [t_2^*]$, for any sequences of operand types t_1^* and t_2^*.

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

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

The br_table instruction is stack-polymorphic.

return

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

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

Note: The return instruction is stack-polymorphic.

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

---

call x

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

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

call_indirect x y

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

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

3.3.9 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

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

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

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

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

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

3.3.10 Expressions

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

$instr^*$ end

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

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

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 $\mathsf{ref.null}$,
  - or of the form $\mathsf{ref.func}\ x$,
  - or of the form $\mathsf{global.get}\ x$, in which case $C.\mathsf{globals}[x]$ must be a global type of the form $\mathsf{const}\ t$.

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

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

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

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

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

3.4 Modules

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

3.4.1 Functions

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

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

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

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

3.4.2 Tables

Tables $table$ are classified by table types.

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

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

$$\frac{\vdash tabletype\ \text{ok}}{C \vdash \{\mathsf{type}\ tabletype\} : tabletype}$$

3.4.3 Memories

Memories mem are classified by memory types.

$\{\text{type } \mathit{memtype}\}$

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

$$\frac{\vdash \mathit{memtype}\ \text{ok}}{C \vdash \{\text{type } \mathit{memtype}\} : \mathit{memtype}}$$

3.4.4 Globals

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

$\{\text{type } \mathit{mut}\ t, \text{init } \mathit{expr}\}$

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

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

3.4.5 Element Segments

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

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

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

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

passive

- The element mode is valid with any reference type.

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

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

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

$$\frac{C.\text{tables}[x] = limits\ t \qquad C \vdash expr : [\text{i32}] \qquad C \vdash expr\ \text{const}}{C \vdash \text{active}\ \{\text{table } x, \text{offset } expr\} : t}$$

declarative

- The element mode is valid with any reference type.

$$\frac{}{C \vdash \text{declarative} : reftype}$$

3.4.6 Data Segments

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

$\{\text{init } b^*, \text{mode } datamode\}$

- The data mode $datamode$ must be valid.
- Then the data segment is valid.

$$\frac{C \vdash datamode\ \text{ok}}{C \vdash \{\text{init } b^*, \text{mode } datamode\}\ \text{ok}}$$

passive

- The data mode is valid.

$$\frac{}{C \vdash \text{passive}\ \text{ok}}$$

active $\{\text{memory } x, \text{offset } expr\}$

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

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

3.4.7 Start Function

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

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

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

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

3.4.8 Exports

Exports $export$ and export descriptions $exportdesc$ are classified by their external type.

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

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

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

$\mathsf{func}\ x$

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

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

$\mathsf{table}\ x$

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

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

$\mathsf{mem}\ x$

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

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

global x

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

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

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

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

table $tabletype$

- The table type $tabletype$ must be valid.
- Then the import description is valid with type table $tabletype$.

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

mem $memtype$

- The memory type $memtype$ must be valid.
- Then the import description is valid with type mem $memtype$.

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

global $globaltype$

- The global type $globaltype$ must be valid.
- Then the import description is valid with type global $globaltype$.

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

3.4.10 Modules

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

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

- Let $module$ be the module to validate.
- Let C be a context where:
  - $C.\mathsf{types}$ is $module.\mathsf{types}$,
  - $C.\mathsf{funcs}$ is $\mathrm{funcs}(it^*)$ concatenated with ft^*, with the import's external types it^* and the internal function types ft^* as determined below,
  - $C.\mathsf{tables}$ is $\mathrm{tables}(it^*)$ concatenated with tt^*, with the import's external types it^* and the internal table types tt^* as determined below,
  - $C.\mathsf{mems}$ is $\mathrm{mems}(it^*)$ concatenated with mt^*, with the import's external types it^* and the internal memory types mt^* as determined below,
  - $C.\mathsf{globals}$ is $\mathrm{globals}(it^*)$ concatenated with gt^*, with the import's external types it^* and the internal global types gt^* as determined below,
  - $C.\mathsf{elems}$ is rt^* as determined below,
  - $C.\mathsf{datas}$ is ok^n, where n is the length of the vector $module.\mathsf{datas}$,
  - $C.\mathsf{locals}$ is empty,
  - $C.\mathsf{labels}$ is empty,
  - $C.\mathsf{return}$ is empty.
  - $C.\mathsf{refs}$ is the set $\mathrm{funcidx}(module \text{ with } \mathsf{funcs} = \epsilon \text{ with } \mathsf{start} = \epsilon)$, i.e., the set of function indices occurring in the module, except in its functions or start function.
- Let C' be the same context as C, except that $C'.\mathsf{globals}$ is just the sequence $\mathrm{globals}(it^*)$.
- For each $functype_i$ in $module.\mathsf{types}$, the function type $functype_i$ must be valid.
- Under the context C':
  - For each $table_i$ in $module.\mathsf{tables}$, the definition $table_i$ must be valid with a table type tt_i.
  - For each mem_i in $module.\mathsf{mems}$, the definition mem_i must be valid with a memory type mt_i.
  - For each $global_i$ in $module.\mathsf{globals}$, the definition $global_i$ must be valid with a global type gt_i.
  - For each $elem_i$ in $module.\mathsf{elems}$, the segment $elem_i$ must be valid with reference type rt_i.
  - For each $data_i$ in $module.\mathsf{datas}$, the segment $data_i$ must be valid.
- Under the context C:
  - For each $func_i$ in $module.\mathsf{funcs}$, the definition $func_i$ must be valid with a function type ft_i.
  - If $module.\mathsf{start}$ is non-empty, then $module.\mathsf{start}$ must be valid.

  - For each $import_i$ in $module$.imports, the segment $import_i$ must be valid with an external type it_i.
  - For each $export_i$ in $module$.exports, the segment $export_i$ must be valid with external type et_i.
- The length of C.mems must not be larger than 1.
- All export names $export_i$.name must be different.
- Let ft^* be the concatenation of the internal function types ft_i, in index order.
- Let tt^* be the concatenation of the internal table types tt_i, in index order.
- Let mt^* be the concatenation of the internal memory types mt_i, in index order.
- Let gt^* be the concatenation of the internal global types gt_i, in index order.
- Let rt^* be the concatenation of the reference types rt_i, in index order.
- Let it^* be the concatenation of external types it_i of the imports, in index order.
- Let et^* be the concatenation of external types et_i of the exports, in index order.
- Then the module is valid with external types $it^* \to et^*$.

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

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

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

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