WebAssembly

author

Mar 07, 2024

Contents

1 Syntax **1**
- 1.1 Values . . . 1
- 1.2 Types . . . 1
- 1.3 Instructions . . . 3
- 1.4 Modules . . . 5

2 Validation **7**
- 2.1 Conventions . . . 7
- 2.2 Types . . . 7
- 2.3 Instructions . . . 9
- 2.4 Modules . . . 19

3 Execution **21**
- 3.1 Auxiliary . . . 21
- 3.2 Runtime . . . 22
- 3.3 Instructions . . . 30
- 3.4 Modules . . . 48

CHAPTER 1

Syntax

1.1 Values

1.1.1 Bytes

$$byte \quad ::= \quad nat$$

1.1.2 Integers

$$u32 \quad ::= \quad nat$$

1.2 Types

1.2.1 Number Types

$$\begin{array}{llll} \text{(number type)} & numtype & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\ & \mathsf{i}n & ::= & \mathsf{i32} \mid \mathsf{i64} \\ & \mathsf{f}n & ::= & \mathsf{f32} \mid \mathsf{f64} \end{array}$$

1.2.2 Vector Types

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

1.2.3 Reference Types

$$\mathit{reftype} \quad ::= \quad \mathsf{funcref} \mid \mathsf{externref}$$

1.2.4 Value Types

$$\mathit{valtype} \quad ::= \quad \mathit{numtype} \mid \mathit{vectype} \mid \mathit{reftype} \mid \mathsf{bot}$$

1.2.5 Result Types

$$\mathit{resulttype} \quad ::= \quad \mathit{valtype}^*$$

1.2.6 Limits

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

1.2.7 Memory Types

$$\begin{array}{llll} \text{(memory type)} & \mathit{memtype} & ::= & \mathit{limits}\ \mathsf{i8} \\ \text{(data type)} & \mathit{datatype} & ::= & \mathsf{ok} \end{array}$$

1.2.8 Table Types

$$\begin{array}{llll} \text{(table type)} & \mathit{tabletype} & ::= & \mathit{limits}\ \mathit{reftype} \\ \text{(element type)} & \mathit{elemtype} & ::= & \mathit{reftype} \end{array}$$

1.2.9 Global Types

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

1.2.10 Function Types

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

1.3 Instructions

1.3.1 Numeric Instructions

$$
\begin{array}{llll}
\text{(signedness)} & sx & ::= & \mathsf{u} \mid \mathsf{s} \\
\text{(instruction)} & instr & ::= & \dots \\
& & | & numtype.\mathsf{const}\ c_{numtype} \\
& & | & numtype.unop_{numtype} \\
& & | & numtype.binop_{numtype} \\
& & | & numtype.testop_{numtype} \\
& & | & numtype.relop_{numtype} \\
& & | & numtype.\mathsf{extend}n \\
& & | & numtype.cvtop\_numtype\_sx^{?} \\
& & | & \dots \\
& unop_{\mathsf{ixx}} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\
& unop_{\mathsf{fxx}} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\
& binop_{\mathsf{ixx}} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div\_}sx \mid \mathsf{rem\_}sx \\
& & | & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}sx \mid \mathsf{rotl} \mid \mathsf{rotr} \\
& binop_{\mathsf{fxx}} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \\
& testop_{\mathsf{ixx}} & ::= & \mathsf{eqz} \\
& testop_{\mathsf{fxx}} & ::= & \\
& relop_{\mathsf{ixx}} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt\_}sx \mid \mathsf{gt\_}sx \mid \mathsf{le\_}sx \mid \mathsf{ge\_}sx
\end{array}
$$

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

$$
\begin{array}{lll}
unop_{numtype} & ::= & unop_{\mathsf{ixx}} \mid unop_{\mathsf{fxx}} \\
binop_{numtype} & ::= & binop_{\mathsf{ixx}} \mid binop_{\mathsf{fxx}} \\
relop_{numtype} & ::= & relop_{\mathsf{ixx}} \mid relop_{\mathsf{fxx}} \\
cvtop & ::= & \mathsf{convert} \mid \mathsf{reinterpret}
\end{array}
$$

1.3.2 Reference Instructions

$$
\begin{array}{lll}
instr & ::= & \dots \\
& | & \mathsf{ref.null}\ reftype \\
& | & \mathsf{ref.func}\ funcidx \\
& | & \mathsf{ref.is\_null} \\
& | & \dots
\end{array}
$$

1.3.3 State Instructions

$$
\begin{array}{lll}
\mathit{instr} & ::= & \dots \\
& | & \mathsf{local.get}~\mathit{localidx} \\
& | & \mathsf{local.set}~\mathit{localidx} \\
& | & \mathsf{local.tee}~\mathit{localidx} \\
& | & \mathsf{global.get}~\mathit{globalidx} \\
& | & \mathsf{global.set}~\mathit{globalidx} \\
& | & \mathsf{table.get}~\mathit{tableidx} \\
& | & \mathsf{table.set}~\mathit{tableidx} \\
& | & \mathsf{table.size}~\mathit{tableidx} \\
& | & \mathsf{table.grow}~\mathit{tableidx} \\
& | & \mathsf{table.fill}~\mathit{tableidx} \\
& | & \mathsf{table.copy}~\mathit{tableidx}~\mathit{tableidx} \\
& | & \mathsf{table.init}~\mathit{tableidx}~\mathit{elemidx} \\
& | & \mathsf{elem.drop}~\mathit{elemidx} \\
& | & \mathsf{memory.size} \\
& | & \mathsf{memory.grow} \\
& | & \mathsf{memory.fill} \\
& | & \mathsf{memory.copy} \\
& | & \mathsf{memory.init}~\mathit{dataidx} \\
& | & \mathsf{data.drop}~\mathit{dataidx} \\
& | & \mathit{numtype}.\mathsf{load}(\mathit{n\_sx})^?~\mathit{u32}~\mathit{u32} \\
& | & \mathit{numtype}.\mathsf{store}\mathit{n}^?~\mathit{u32}~\mathit{u32}
\end{array}
$$

1.3.4 Control Instructions

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

1.3.5 Expressions

$$expr \quad ::= \quad instr^*$$

1.4 Modules

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

1.4.1 Indices

$$\begin{array}{llll}
\text{(index)} & idx & ::= & nat \\
\text{(function index)} & funcidx & ::= & idx \\
\text{(global index)} & globalidx & ::= & idx \\
\text{(table index)} & tableidx & ::= & idx \\
\text{(memory index)} & memidx & ::= & idx \\
\text{(elem index)} & elemidx & ::= & idx \\
\text{(data index)} & dataidx & ::= & idx \\
\text{(label index)} & labelidx & ::= & idx \\
\text{(local index)} & localidx & ::= & idx
\end{array}$$

1.4.2 Functions

$$func \quad ::= \quad \mathsf{func}\ functype\ valtype^*\ expr$$

1.4.3 Tables

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

1.4.4 Memories

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

1.4.5 Globals

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

1.4.6 Element Segments

$$elem \quad ::= \quad \mathsf{elem}\ reftype\ expr^*\ elemmode^?$$

1.4.7 Data Segments

$$data \quad ::= \quad \mathsf{data}\ byte^*\ datamode^?$$

1.4.8 Start Function

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

1.4.9 Exports

$$\begin{array}{llcl}
\text{(export)} & export & ::= & \mathsf{export}\ name\ externuse \\
\text{(external use)} & externuse & ::= & \mathsf{func}\ funcidx \mid \mathsf{global}\ globalidx \mid \mathsf{table}\ tableidx \mid \mathsf{mem}\ memidx
\end{array}$$

1.4.10 Imports

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

CHAPTER 2

Validation

2.1 Conventions

2.1.1 Contexts

$$\begin{array}{lll} \mathit{context} & ::= & \{\ \mathsf{func}\ \mathit{functype}^*,\ \mathsf{global}\ \mathit{globaltype}^*,\ \mathsf{table}\ \mathit{tabletype}^*,\ \mathsf{mem}\ \mathit{memtype}^*, \\ & & \quad \mathsf{elem}\ \mathit{elemtype}^*,\ \mathsf{data}\ \mathit{datatype}^*, \\ & & \quad \mathsf{local}\ \mathit{valtype}^*,\ \mathsf{label}\ \mathit{resulttype}^*,\ \mathsf{return}\ \mathit{resulttype}^?\ \} \end{array}$$

2.2 Types

2.2.1 Limits

$$\frac{n_1 \leq n_2 \leq k}{\vdash [n_1..n_2] : k}$$

2.2.2 Function Types

$$\frac{}{\vdash \mathit{ft} : \mathsf{ok}}$$

2.2.3 Table Types

$$\frac{\vdash \mathit{lim} : 2^{32} - 1}{\vdash \mathit{lim}\ \mathit{rt} : \mathsf{ok}}$$

2.2.4 Memory Types

$$\frac{\vdash \mathit{lim} : 2^{16}}{\vdash \mathit{lim}\ \mathsf{i8} : \mathsf{ok}}$$

2.2.5 Global Types

$$\frac{}{\vdash \mathit{gt} : \mathsf{ok}}$$

2.2.6 External Types

$$\frac{\vdash \mathit{functype} : \mathsf{ok}}{\vdash \mathsf{func}\ \mathit{functype} : \mathsf{ok}}\ [\textsc{K-extern-func}]$$

$$\frac{\vdash \mathit{tabletype} : \mathsf{ok}}{\vdash \mathsf{table}\ \mathit{tabletype} : \mathsf{ok}}\ [\textsc{K-extern-table}]$$

$$\frac{\vdash \mathit{memtype} : \mathsf{ok}}{\vdash \mathsf{mem}\ \mathit{memtype} : \mathsf{ok}}\ [\textsc{K-extern-mem}]$$

$$\frac{\vdash \mathit{globaltype} : \mathsf{ok}}{\vdash \mathsf{global}\ \mathit{globaltype} : \mathsf{ok}}\ [\textsc{K-extern-global}]$$

2.2.7 Import Subtyping

$$\frac{n_{11} \geq n_{21} \qquad n_{12} \leq n_{22}}{\vdash [n_{11}..n_{12}] \leq [n_{21}..n_{22}]}\ [\textsc{S-limits}]$$

$$\frac{}{\vdash \mathit{ft} \leq \mathit{ft}}\ [\textsc{S-func}]$$

$$\frac{\vdash \mathit{ft}_1 \leq \mathit{ft}_2}{\vdash \mathsf{func}\ \mathit{ft}_1 \leq \mathsf{func}\ \mathit{ft}_2}\ [\textsc{S-extern-func}]$$

$$\frac{}{\vdash \mathit{gt} \leq \mathit{gt}}\ [\textsc{S-global}]$$

$$\frac{\vdash \mathit{gt}_1 \leq \mathit{gt}_2}{\vdash \mathsf{global}\ \mathit{gt}_1 \leq \mathsf{global}\ \mathit{gt}_2}\ [\textsc{S-extern-global}]$$

$$\frac{\vdash \mathit{lim}_1 \leq \mathit{lim}_2}{\vdash \mathit{lim}_1\ \mathit{rt} \leq \mathit{lim}_2\ \mathit{rt}}\ [\textsc{S-table}]$$

$$\frac{\vdash \mathit{tt}_1 \leq \mathit{tt}_2}{\vdash \mathsf{table}\ \mathit{tt}_1 \leq \mathsf{table}\ \mathit{tt}_2}\ [\textsc{S-extern-table}]$$

$$\frac{\vdash \mathit{lim}_1 \leq \mathit{lim}_2}{\vdash \mathit{lim}_1\ \mathsf{i8} \leq \mathit{lim}_2\ \mathsf{i8}}\ [\textsc{S-mem}]$$

$$\frac{\vdash \mathit{mt}_1 \leq \mathit{mt}_2}{\vdash \mathsf{mem}\ \mathit{mt}_1 \leq \mathsf{mem}\ \mathit{mt}_2}\ [\textsc{S-extern-mem}]$$

2.3 Instructions

2.3.1 Numeric Instructions

unop nt $unop$

- The instruction is valid with type $nt \to nt$.

$$\frac{}{C \vdash nt.unop : nt \to nt} \text{[T-UNOP]}$$

binop nt $binop$

- The instruction is valid with type $nt\ nt \to nt$.

$$\frac{}{C \vdash nt.binop : nt\ nt \to nt} \text{[T-BINOP]}$$

testop nt $testop$

- The instruction is valid with type $nt \to$ i32.

$$\frac{}{C \vdash nt.testop : nt \to \mathsf{i32}} \text{[T-TESTOP]}$$

relop nt $relop$

- The instruction is valid with type $nt\ nt \to$ i32.

$$\frac{}{C \vdash nt.relop : nt\ nt \to \mathsf{i32}} \text{[T-RELOP]}$$

TODO (should change the rule name to cvtop-)

$$\frac{nt_1 \neq nt_2 \qquad |nt_1| = |nt_2|}{C \vdash \mathsf{cvtop}\ nt_1\ \mathsf{reinterpret}\ nt_2 : nt_2 \to nt_1} \text{[T-REINTERPRET]}$$

TODO (should change the rule name to cvtop-)

$$\frac{\mathsf{i}n_1 \neq \mathsf{i}n_2 \qquad sx^? = \epsilon \Leftrightarrow |\mathsf{i}n_1| > |\mathsf{i}n_2|}{C \vdash \mathsf{i}n_1.\mathsf{convert\_i}n_2\_sx^? : \mathsf{i}n_2 \to \mathsf{i}n_1} \text{[T-CONVERT-I]}$$

$$\frac{\mathsf{f}n_1 \neq \mathsf{f}n_2}{C \vdash \mathsf{cvtop}\ \mathsf{f}n_1\ \mathsf{convert}\ \mathsf{f}n_2 : \mathsf{f}n_2 \to \mathsf{f}n_1} \text{[T-CONVERT-F]}$$

2.3.2 Reference Instructions

ref.is_null

- The instruction is valid with type $rt \to \mathsf{i32}$.

$$\frac{}{C \vdash \mathsf{ref.is\_null} : rt \to \mathsf{i32}} \ [\textsc{T-ref.is\_null}]$$

ref.func x

- The length of C.func must be greater than x.
- Let ft be $C.\mathsf{func}[x]$.
- The instruction is valid with type $\epsilon \to \mathsf{funcref}$.

$$\frac{C.\mathsf{func}[x] = ft}{C \vdash \mathsf{ref.func}\ x : \epsilon \to \mathsf{funcref}} \ [\textsc{T-ref.func}]$$

2.3.3 Parametric Instructions

drop

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

$$\frac{}{C \vdash \mathsf{drop} : t \to \epsilon} \ [\textsc{T-drop}]$$

select $t^?$

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

$$\frac{}{C \vdash \mathsf{select}\ t : t\ t\ \mathsf{i32} \to t} \ [\textsc{T-select-expl}]$$

$$\frac{\vdash t \leq t' \qquad t' = \mathit{numtype} \lor t' = \mathit{vectype}}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \to t} \ [\textsc{T-select-impl}]$$

2.3.4 Variable Instructions

local.get x

- The length of C.local must be greater than x.
- Let t be C.local$[x]$.
- The instruction is valid with type $\epsilon \to t$.

$$\frac{C.\mathsf{local}[x] = t}{C \vdash \mathsf{local.get}\ x : \epsilon \to t}\ [\text{T-LOCAL.GET}]$$

local.set x

- The length of C.local must be greater than x.
- Let t be C.local$[x]$.
- The instruction is valid with type $t \to \epsilon$.

$$\frac{C.\mathsf{local}[x] = t}{C \vdash \mathsf{local.set}\ x : t \to \epsilon}\ [\text{T-LOCAL.SET}]$$

local.tee x

- The length of C.local must be greater than x.
- Let t be C.local$[x]$.
- The instruction is valid with type $t \to t$.

$$\frac{C.\mathsf{local}[x] = t}{C \vdash \mathsf{local.tee}\ x : t \to t}\ [\text{T-LOCAL.TEE}]$$

global.get x

- The length of C.global must be greater than x.
- Let $mut^?\ t$ be C.global$[x]$.
- The instruction is valid with type $\epsilon \to t$.

$$\frac{C.\mathsf{global}[x] = \mathsf{mut}^?\ t}{C \vdash \mathsf{global.get}\ x : \epsilon \to t}\ [\text{T-GLOBAL.GET}]$$

global.set x

- The length of C.global must be greater than x.
- Let mut t be C.global$[x]$.
- The instruction is valid with type $t \to \epsilon$.

$$\frac{C.\mathsf{global}[x] = \mathsf{mut}\ t}{C \vdash \mathsf{global.set}\ x : t \to \epsilon} \ [\text{T-GLOBAL.SET}]$$

2.3.5 Table Instructions

table.get x

- The length of C.table must be greater than x.
- Let $lim\ rt$ be C.table$[x]$.
- The instruction is valid with type i32 $\to rt$.

$$\frac{C.\mathsf{table}[x] = lim\ rt}{C \vdash \mathsf{table.get}\ x : \mathsf{i32} \to rt} \ [\text{T-TABLE.GET}]$$

table.set x

- The length of C.table must be greater than x.
- Let $lim\ rt$ be C.table$[x]$.
- The instruction is valid with type i32 $rt \to \epsilon$.

$$\frac{C.\mathsf{table}[x] = lim\ rt}{C \vdash \mathsf{table.set}\ x : \mathsf{i32}\ rt \to \epsilon} \ [\text{T-TABLE.SET}]$$

table.size x

- The length of C.table must be greater than x.
- Let tt be C.table$[x]$.
- The instruction is valid with type $\epsilon \to$ i32.

$$\frac{C.\mathsf{table}[x] = tt}{C \vdash \mathsf{table.size}\ x : \epsilon \to \mathsf{i32}} \ [\text{T-TABLE.SIZE}]$$

table.grow x

- The length of C.table must be greater than x.
- Let $lim\ rt$ be C.table$[x]$.
- The instruction is valid with type rt i32 $\rightarrow$ i32.

$$\frac{C.\mathsf{table}[x] = lim\ rt}{C \vdash \mathsf{table.grow}\ x : rt\ \mathsf{i32} \rightarrow \mathsf{i32}}\ [\textsc{T-table.grow}]$$

table.fill x

- The length of C.table must be greater than x.
- Let $lim\ rt$ be C.table$[x]$.
- The instruction is valid with type i32 rt i32 $\rightarrow \epsilon$.

$$\frac{C.\mathsf{table}[x] = lim\ rt}{C \vdash \mathsf{table.fill}\ x : \mathsf{i32}\ rt\ \mathsf{i32} \rightarrow \epsilon}\ [\textsc{T-table.fill}]$$

table.copy x_1 x_2

- The length of C.table must be greater than x_1.
- The length of C.table must be greater than x_2.
- Let $lim_1\ rt$ be C.table$[x_1]$.
- Let $lim_2\ rt$ be C.table$[x_2]$.
- The instruction is valid with type i32 i32 i32 $\rightarrow \epsilon$.

$$\frac{C.\mathsf{table}[x_1] = lim_1\ rt \qquad C.\mathsf{table}[x_2] = lim_2\ rt}{C \vdash \mathsf{table.copy}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}\ [\textsc{T-table.copy}]$$

table.init x_1 x_2

- The length of C.table must be greater than x_1.
- The length of C.elem must be greater than x_2.
- Let $lim\ rt$ be C.table$[x_1]$.
- C.elem$[x_2]$ must be equal to rt.
- The instruction is valid with type i32 i32 i32 $\rightarrow \epsilon$.

$$\frac{C.\mathsf{table}[x_1] = lim\ rt \qquad C.\mathsf{elem}[x_2] = rt}{C \vdash \mathsf{table.init}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}\ [\textsc{T-table.init}]$$

elem.drop x

- The length of C.elem must be greater than x.
- Let rt be C.elem$[x]$.
- The instruction is valid with type $\epsilon \rightarrow \epsilon$.

$$\frac{C.\mathsf{elem}[x] = rt}{C \vdash \mathsf{elem.drop}\ x : \epsilon \rightarrow \epsilon}\ [\textsc{T-elem.drop}]$$

2.3.6 Memory Instructions

load nt $(n\ sx)^?$ n_A n_O

- The length of C.mem must be greater than 0.
- $n^?$ is ϵ and $sx^?$ is ϵ are equivalent.
- 2^{n_A} must be less than or equal to size$(nt)/8$.
- If n is defined,
  - 2^{n_A} must be less than or equal to $n/8$.
  - $n/8$ must be less than size$(nt)/8$.
- C.mem$[0]$ must be equal to mt.
- If n is defined,
  - nt must be equal to in.
- The instruction is valid with type i32 $\rightarrow nt$.

$$\frac{C.\mathsf{mem}[0] = mt \qquad 2^{n_\mathsf{a}} \leq |nt|/8 \qquad (2^{n_\mathsf{a}} \leq n/8 < |nt|/8)^? \qquad n^? = \epsilon \lor nt = \mathsf{i}n}{C \vdash nt.\mathsf{load}(n\_sx)^?\ n_\mathsf{a}\ n_\mathsf{o} : \mathsf{i32} \rightarrow nt}\ [\textsc{T-load}]$$

store nt $n^?$ n_A n_O

- The length of C.mem must be greater than 0.
- 2^{n_A} must be less than or equal to size$(nt)/8$.
- If n is defined,
  - 2^{n_A} must be less than or equal to $n/8$.
  - $n/8$ must be less than size$(nt)/8$.
- C.mem$[0]$ must be equal to mt.
- If n is defined,
  - nt must be equal to in.
- The instruction is valid with type i32 $nt \rightarrow \epsilon$.

$$\frac{C.\mathsf{mem}[0] = mt \qquad 2^{n_\mathsf{a}} \leq |nt|/8 \qquad (2^{n_\mathsf{a}} \leq n/8 < |nt|/8)^? \qquad n^? = \epsilon \vee nt = \mathsf{i}n}{C \vdash nt.\mathsf{store}n^?\ n_\mathsf{a}\ n_\mathsf{o} : \mathsf{i32}\ nt \to \epsilon}\ [\text{T-STORE}]$$

memory.size

- The length of C.mem must be greater than 0.
- Let mt be C.mem$[0]$.
- The instruction is valid with type $\epsilon \to$ i32.

$$\frac{C.\mathsf{mem}[0] = mt}{C \vdash \mathsf{memory.size} : \epsilon \to \mathsf{i32}}\ [\text{T-MEMORY.SIZE}]$$

memory.grow

- The length of C.mem must be greater than 0.
- Let mt be C.mem$[0]$.
- The instruction is valid with type i32 $\to$ i32.

$$\frac{C.\mathsf{mem}[0] = mt}{C \vdash \mathsf{memory.grow} : \mathsf{i32} \to \mathsf{i32}}\ [\text{T-MEMORY.GROW}]$$

memory.fill

- The length of C.mem must be greater than 0.
- Let mt be C.mem$[0]$.
- The instruction is valid with type i32 i32 i32 $\to \epsilon$.

$$\frac{C.\mathsf{mem}[0] = mt}{C \vdash \mathsf{memory.fill} : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}\ [\text{T-MEMORY.FILL}]$$

memory.copy

- The length of C.mem must be greater than 0.
- Let mt be C.mem$[0]$.
- The instruction is valid with type i32 i32 i32 $\to \epsilon$.

$$\frac{C.\mathsf{mem}[0] = mt}{C \vdash \mathsf{memory.copy} : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \to \epsilon}\ [\text{T-MEMORY.COPY}]$$

memory.init x

- The length of C.mem must be greater than 0.
- The length of C.data must be greater than x.
- C.data$[x]$ must be equal to ok.
- Let mt be C.mem$[0]$.
- The instruction is valid with type i32 i32 i32 $\rightarrow \epsilon$.

$$\frac{C.\mathsf{mem}[0] = mt \qquad C.\mathsf{data}[x] = \mathsf{ok}}{C \vdash \mathsf{memory.init}\ x : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}\ [\textsc{T-memory.init}]$$

data.drop x

- The length of C.data must be greater than x.
- C.data$[x]$ must be equal to ok.
- The instruction is valid with type $\epsilon \rightarrow \epsilon$.

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

2.3.7 Control Instructions

nop

- The instruction is valid with type $\epsilon \rightarrow \epsilon$.

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \rightarrow \epsilon}\ [\textsc{T-nop}]$$

unreachable

- The instruction is valid with type ${t_1}^* \rightarrow {t_2}^*$.

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

block bt $instr^*$

- Under the context C with .label prepended by ${t_2}^*$, $instr^*$ must be valid with type ${t_1}^* \to {t_2}^*$.
- Under the context C, bt must be valid with type ${t_1}^* \to {t_2}^*$.
- The instruction is valid with type ${t_1}^* \to {t_2}^*$.

$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_2^*) \vdash instr^* : t_1^* \to t_2^*}{C \vdash \mathsf{block}\ bt\ instr^* : t_1^* \to t_2^*}\ [\textsc{T-block}]$$

loop bt $instr^*$

- Under the context C with .label prepended by ${t_1}^*$, $instr^*$ must be valid with type ${t_1}^* \to {t_2}^*$.
- Under the context C, bt must be valid with type ${t_1}^* \to {t_2}^*$.
- The instruction is valid with type ${t_1}^* \to {t_2}^*$.

$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_1^*) \vdash instr^* : t_1^* \to t_2^*}{C \vdash \mathsf{loop}\ bt\ instr^* : t_1^* \to t_2^*}\ [\textsc{T-loop}]$$

if bt ${instr_1}^*$ ${instr_2}^*$

- Under the context C with .label prepended by ${t_2}^*$, ${instr_2}^*$ must be valid with type ${t_1}^* \to {t_2}^*$.
- Under the context C, bt must be valid with type ${t_1}^* \to {t_2}^*$.
- Under the context C with .label prepended by ${t_2}^*$, ${instr_1}^*$ must be valid with type ${t_1}^* \to {t_2}^*$.
- The instruction is valid with type ${t_1}^*$ i32 $\to {t_2}^*$.

$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_2^*) \vdash instr_1^* : t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_2^*) \vdash instr_2^* : t_1^* \to t_2^*}{C \vdash \mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^* : t_1^*\ \mathsf{i32} \to t_2^*}\ [\textsc{T-if}]$$

br l

- The length of C.label must be greater than l.
- Let t^* be C.label$[l]$.
- The instruction is valid with type ${t_1}^*\ t^* \to {t_2}^*$.

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

br_if l

- The length of C.label must be greater than l.
- Let t^* be C.label$[l]$.
- The instruction is valid with type t^* i32 $\rightarrow t^*$.

$$\frac{C.\mathsf{label}[l] = t^*}{C \vdash \mathsf{br\_if}\ l : t^*\ \mathsf{i32} \rightarrow t^*}\ [\text{T-BR\_IF}]$$

br_table l^* l'

- For all l in l^*,
  - The length of C.label must be greater than l.
- The length of C.label must be greater than l'.
- For all l in l^*,
  - t^* must match C.label$[l]$.
- t^* must match C.label$[l']$.
- The instruction is valid with type ${t_1}^*\ t^* \rightarrow {t_2}^*$.

$$\frac{(\vdash t^* \leq C.\mathsf{label}[l])^* \qquad \vdash t^* \leq C.\mathsf{label}[l']}{C \vdash \mathsf{br\_table}\ l^*\ l' : t_1^*\ t^* \rightarrow t_2^*}\ [\text{T-BR\_TABLE}]$$

return

- Let ${t^*}^?$ be C.return.
- The instruction is valid with type ${t_1}^*\ t^* \rightarrow {t_2}^*$.

$$\frac{C.\mathsf{return} = (t^*)}{C \vdash \mathsf{return} : t_1^*\ t^* \rightarrow t_2^*}\ [\text{T-RETURN}]$$

call x

- The length of C.func must be greater than x.
- Let ${t_1}^* \rightarrow {t_2}^*$ be C.func$[x]$.
- The instruction is valid with type ${t_1}^* \rightarrow {t_2}^*$.

$$\frac{C.\mathsf{func}[x] = t_1^* \rightarrow t_2^*}{C \vdash \mathsf{call}\ x : t_1^* \rightarrow t_2^*}\ [\text{T-CALL}]$$

call_indirect x ft

- The length of C.table must be greater than x.
- Let lim funcref be C.table$[x]$.
- Let $t_1{}^* \to t_2{}^*$ be ft.
- The instruction is valid with type $t_1{}^*$ i32 $\to t_2{}^*$.

$$\frac{C.\mathsf{table}[x] = lim\ \mathsf{funcref} \qquad ft = t_1^* \to t_2^*}{C \vdash \mathsf{call\_indirect}\ x\ ft : t_1^*\ \mathsf{i32} \to t_2^*}\ [\textsc{T-call\_indirect}]$$

2.4 Modules

2.4.1 Functions

$$\frac{ft = t_1^* \to t_2^* \qquad \vdash ft : \mathsf{ok} \qquad C, \mathsf{local}\ t_1^*\ t^*, \mathsf{label}\ (t_2^*), \mathsf{return}\ (t_2^*) \vdash expr : t_2^*}{C \vdash \mathsf{func}\ ft\ t^*\ expr : ft}$$

2.4.2 Tables

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

2.4.3 Memories

$$\frac{\vdash mt : \mathsf{ok}}{C \vdash \mathsf{memory}\ mt : mt}$$

2.4.4 Globals

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

2.4.5 Element Segments

$$\frac{(C \vdash expr : rt)^* \qquad (C \vdash elemmode : rt)^?}{C \vdash \mathsf{elem}\ rt\ expr^*\ elemmode^? : rt}\ [\textsc{T-elem}]$$

$$\frac{C.\mathsf{table}[x] = lim\ rt \qquad (C \vdash expr : \mathsf{i32}\ \mathsf{const})^*}{C \vdash \mathsf{table}\ x\ expr : rt}\ [\textsc{T-elemmode-active}]$$

$$\frac{}{C \vdash \mathsf{declare} : rt}\ [\textsc{T-elemmode-declare}]$$

2.4.6 Data Segments

$$\frac{(C \vdash \mathit{datamode} : \mathsf{ok})^?}{C \vdash \mathsf{data}\ b^*\ \mathit{datamode}^? : \mathsf{ok}}\ [\text{T-DATA}]$$

$$\frac{C.\mathsf{mem}[0] = \mathit{mt} \qquad (C \vdash \mathit{expr} : \mathsf{i32}\ \mathsf{const})^*}{C \vdash \mathsf{memory}\ 0\ \mathit{expr} : \mathsf{ok}}\ [\text{T-DATAMODE}]$$

2.4.7 Start Function

$$\frac{C.\mathsf{func}[x] = \epsilon \rightarrow \epsilon}{C \vdash \mathsf{start}\ x : \mathsf{ok}}$$

2.4.8 Exports

$$\frac{C \vdash \mathit{externuse} : \mathit{xt}}{C \vdash \mathsf{export}\ \mathit{name}\ \mathit{externuse} : \mathit{xt}}\ [\text{T-EXPORT}]$$

$$\frac{C.\mathsf{func}[x] = \mathit{ft}}{C \vdash \mathsf{func}\ x : \mathsf{func}\ \mathit{ft}}\ [\text{T-EXTERNUSE-FUNC}]$$

$$\frac{C.\mathsf{table}[x] = \mathit{tt}}{C \vdash \mathsf{table}\ x : \mathsf{table}\ \mathit{tt}}\ [\text{T-EXTERNUSE-TABLE}]$$

$$\frac{C.\mathsf{mem}[x] = \mathit{mt}}{C \vdash \mathsf{mem}\ x : \mathsf{mem}\ \mathit{mt}}\ [\text{T-EXTERNUSE-MEM}]$$

$$\frac{C.\mathsf{global}[x] = \mathit{gt}}{C \vdash \mathsf{global}\ x : \mathsf{global}\ \mathit{gt}}\ [\text{T-EXTERNUSE-GLOBAL}]$$

2.4.9 Imports

$$\frac{\vdash \mathit{xt} : \mathsf{ok}}{C \vdash \mathsf{import}\ \mathit{name}_1\ \mathit{name}_2\ \mathit{xt} : \mathit{xt}}$$

2.4.10 Modules

$$\frac{\begin{array}{c} C = \{\mathsf{func}\ \mathit{ft}^*,\ \mathsf{global}\ \mathit{gt}^*,\ \mathsf{table}\ \mathit{tt}^*,\ \mathsf{mem}\ \mathit{mt}^*,\ \mathsf{elem}\ \mathit{rt}^*,\ \mathsf{data}\ \mathsf{ok}^n\} \\ (C \vdash \mathit{func} : \mathit{ft})^* \qquad (C \vdash \mathit{global} : \mathit{gt})^* \qquad (C \vdash \mathit{table} : \mathit{tt})^* \qquad (C \vdash \mathit{mem} : \mathit{mt})^* \\ (C \vdash \mathit{elem} : \mathit{rt})^* \qquad (C \vdash \mathit{data} : \mathsf{ok})^n \qquad (C \vdash \mathit{start} : \mathsf{ok})^? \\ |\mathit{mem}^*| \leq 1 \end{array}}{\vdash \mathsf{module}\ \mathit{import}^*\ \mathit{func}^*\ \mathit{global}^*\ \mathit{table}^*\ \mathit{mem}^*\ \mathit{elem}^*\ \mathit{data}^n\ \mathit{start}^?\ \mathit{export}^* : \mathsf{ok}}$$

CHAPTER 3

Execution

3.1 Auxiliary

3.1.1 General Constants

$\mathrm{Ki}()$

1. Return 1024.

$$\mathrm{Ki} = 1024$$

3.1.2 General Functions

$\min(x_0, x_1)$

1. If x_0 is 0, then:
   a. Return 0.
2. If x_1 is 0, then:
   a. Return 0.
3. Assert: Due to validation, x_0 is greater than or equal to 1.
4. Let i be $x_0 - 1$.
5. Assert: Due to validation, x_1 is greater than or equal to 1.
6. Let j be $x_1 - 1$.
7. Return $\min(i, j)$.

$$\begin{array}{lcl} \min(0, j) & = & 0 \\ \min(i, 0) & = & 0 \\ \min(i+1, j+1) & = & \min(i, j) \end{array}$$

3.1.3 Auxiliary Definitions on Types

$\mathrm{size}(x_0)$

1. If x_0 is i32, then:
   a. Return 32.
2. If x_0 is i64, then:
   a. Return 64.
3. If x_0 is f32, then:
   a. Return 32.
4. If x_0 is f64, then:
   a. Return 64.
5. If x_0 is v128, then:
   a. Return 128.

$$\begin{array}{lcl} |\mathsf{i32}| & = & 32 \\ |\mathsf{i64}| & = & 64 \\ |\mathsf{f32}| & = & 32 \\ |\mathsf{f64}| & = & 64 \\ |\mathsf{v128}| & = & 128 \end{array}$$

3.2 Runtime

3.2.1 Values

$\mathrm{default\_}(x_0)$

1. If x_0 is i32, then:
   a. Return i32.const 0.
2. If x_0 is i64, then:
   a. Return i64.const 0.
3. If x_0 is f32, then:
   a. Return f32.const 0.
4. If x_0 is f64, then:
   a. Return f64.const 0.
5. If x_0 is funcref, then:

a. Return ref.null funcref.

6. If x_0 is externref, then:

a. Return ref.null externref.

$$\begin{array}{lcl}
\mathrm{default}_{\mathsf{i32}} & = & (\mathsf{i32.const}\ 0) \\
\mathrm{default}_{\mathsf{i64}} & = & (\mathsf{i64.const}\ 0) \\
\mathrm{default}_{\mathsf{f32}} & = & (\mathsf{f32.const}\ 0) \\
\mathrm{default}_{\mathsf{f64}} & = & (\mathsf{f64.const}\ 0) \\
\mathrm{default}_{\mathsf{funcref}} & = & (\mathsf{ref.null}\ \mathsf{funcref}) \\
\mathrm{default}_{\mathsf{externref}} & = & (\mathsf{ref.null}\ \mathsf{externref})
\end{array}$$

3.2.2 Results

3.2.3 Store

$$\begin{array}{lcl}
store & ::= & \{\ \mathsf{func}\ funcinst^*, \\
& & \ \ \mathsf{global}\ globalinst^*, \\
& & \ \ \mathsf{table}\ tableinst^*, \\
& & \ \ \mathsf{mem}\ meminst^*, \\
& & \ \ \mathsf{elem}\ eleminst^*, \\
& & \ \ \mathsf{data}\ datainst^*\ \}
\end{array}$$

3.2.4 Addresses

$$\begin{array}{llcl}
\text{(address)} & addr & ::= & nat \\
\text{(function address)} & funcaddr & ::= & addr \\
\text{(global address)} & globaladdr & ::= & addr \\
\text{(table address)} & tableaddr & ::= & addr \\
\text{(memory address)} & memaddr & ::= & addr \\
\text{(elem address)} & elemaddr & ::= & addr \\
\text{(data address)} & dataaddr & ::= & addr \\
\text{(label address)} & labeladdr & ::= & addr \\
\text{(host address)} & hostaddr & ::= & addr
\end{array}$$

3.2.5 Module Instances

$$\begin{array}{lcl}
moduleinst & ::= & \{\ \mathsf{func}\ funcaddr^*, \\
& & \ \ \mathsf{global}\ globaladdr^*, \\
& & \ \ \mathsf{table}\ tableaddr^*, \\
& & \ \ \mathsf{mem}\ memaddr^*, \\
& & \ \ \mathsf{elem}\ elemaddr^*, \\
& & \ \ \mathsf{data}\ dataaddr^*, \\
& & \ \ \mathsf{export}\ exportinst^*\ \}
\end{array}$$

3.2.6 Function Instances

$$\begin{array}{llll} \mathit{funcinst} & ::= & \{ & \mathsf{module}\ \mathit{moduleinst}, \\ & & & \mathsf{code}\ \mathit{func}\ \} \end{array}$$

3.2.7 Table Instances

$$\begin{array}{llll} \mathit{tableinst} & ::= & \{ & \mathsf{type}\ \mathit{tabletype}, \\ & & & \mathsf{elem}\ \mathit{ref}^*\ \} \end{array}$$

3.2.8 Memory Instances

$$\begin{array}{llll} \mathit{meminst} & ::= & \{ & \mathsf{type}\ \mathit{memtype}, \\ & & & \mathsf{data}\ \mathit{byte}^*\ \} \end{array}$$

3.2.9 Global Instances

$$\begin{array}{llll} \mathit{globalinst} & ::= & \{ & \mathsf{type}\ \mathit{globaltype}, \\ & & & \mathsf{value}\ \mathit{val}\ \} \end{array}$$

3.2.10 Element Instances

$$\begin{array}{llll} \mathit{eleminst} & ::= & \{ & \mathsf{type}\ \mathit{elemtype}, \\ & & & \mathsf{elem}\ \mathit{ref}^*\ \} \end{array}$$

3.2.11 Data Instances

$$\begin{array}{lll} \mathit{datainst} & ::= & \{ \mathsf{data}\ \mathit{byte}^*\ \} \end{array}$$

3.2.12 Export Instances

$$\begin{array}{llll} \mathit{exportinst} & ::= & \{ & \mathsf{name}\ \mathit{name}, \\ & & & \mathsf{value}\ \mathit{externval}\ \} \end{array}$$

3.2.13 External Values

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

3.2.14 Stack

Activation Frames

$$\begin{array}{llll} \mathit{frame} & ::= & \{ & \mathsf{local}\ \mathit{val}^*, \\ & & & \mathsf{module}\ \mathit{moduleinst}\ \} \end{array}$$

3.2.15 Administrative Instructions

$$\begin{array}{lcl} \mathit{instr} & ::= & \mathit{instr} \\ & | & \mathsf{ref.func}\ \mathit{funcaddr} \\ & | & \mathsf{ref.extern}\ \mathit{hostaddr} \\ & | & \mathsf{call}\ \mathit{funcaddr} \\ & | & \mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{instr}^* \\ & | & \mathsf{frame}_n\{\mathit{frame}\}\ \mathit{instr}^* \\ & | & \mathsf{trap} \end{array}$$

Configurations

$$\begin{array}{llcl} \text{(state)} & \mathit{state} & ::= & \mathit{store}; \mathit{frame} \\ \text{(configuration)} & \mathit{config} & ::= & \mathit{state}; \mathit{instr}^* \end{array}$$

Evaluation Contexts

$$\begin{array}{lcl} E & ::= & [\_] \\ & | & \mathit{val}^*\ E\ \mathit{instr}^* \\ & | & \mathsf{label}_n\{\mathit{instr}^*\}\ E \end{array}$$

3.2.16 Helper Functions

funcaddr()

1. Let f be the current frame.
2. Return $f.\mathsf{module}.\mathsf{func}$.

$$(s; f).\mathsf{module}.\mathsf{func} \quad = \quad f.\mathsf{module}.\mathsf{func}$$

funcinst()

1. Return $s.\mathsf{func}$.

$$(s; f).\mathsf{func} \quad = \quad s.\mathsf{func}$$

globalinst()

1. Return $s.\mathsf{global}$.

$$(s; f).\mathsf{global} \quad = \quad s.\mathsf{global}$$

$\mathrm{tableinst}()$

1. Return s.table.

$$(s;f).\mathsf{table} \quad = \quad s.\mathsf{table}$$

$\mathrm{meminst}()$

1. Return s.mem.

$$(s;f).\mathsf{mem} \quad = \quad s.\mathsf{mem}$$

$\mathrm{eleminst}()$

1. Return s.elem.

$$(s;f).\mathsf{elem} \quad = \quad s.\mathsf{elem}$$

$\mathrm{datainst}()$

1. Return s.data.

$$(s;f).\mathsf{data} \quad = \quad s.\mathsf{data}$$

$\mathrm{func}(x)$

1. Let f be the current frame.
2. Return $s.\mathsf{func}[f.\mathsf{module}.\mathsf{func}[x]]$.

$$(s;f).\mathsf{func}[x] \quad = \quad s.\mathsf{func}[f.\mathsf{module}.\mathsf{func}[x]]$$

global(x)

1. Let f be the current frame.
2. Return $s.\mathsf{global}[f.\mathsf{module}.\mathsf{global}[x]]$.

$$(s;f).\mathsf{global}[x] \quad = \quad s.\mathsf{global}[f.\mathsf{module}.\mathsf{global}[x]]$$

table(x)

1. Let f be the current frame.
2. Return $s.\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]]$.

$$(s;f).\mathsf{table}[x] \quad = \quad s.\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]]$$

mem(x)

1. Let f be the current frame.
2. Return $s.\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]]$.

$$(s;f).\mathsf{mem}[x] \quad = \quad s.\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]]$$

elem(x)

1. Let f be the current frame.
2. Return $s.\mathsf{elem}[f.\mathsf{module}.\mathsf{elem}[x]]$.

$$(s;f).\mathsf{elem}[x] \quad = \quad s.\mathsf{elem}[f.\mathsf{module}.\mathsf{elem}[x]]$$

data(x)

1. Let f be the current frame.
2. Return $s.\mathsf{data}[f.\mathsf{module}.\mathsf{data}[x]]$.

$$(s;f).\mathsf{data}[x] \quad = \quad s.\mathsf{data}[f.\mathsf{module}.\mathsf{data}[x]]$$

$\mathrm{local}(x)$

1. Let f be the current frame.
2. Return $f.\mathsf{local}[x]$.

$$(s; f).\mathsf{local}[x] \quad = \quad f.\mathsf{local}[x]$$

$\mathrm{with\_local}(x, v)$

1. Let f be the current frame.
2. Replace $f.\mathsf{local}[x]$ with v.

$$(s; f)[\mathsf{local}[x] = v] \quad = \quad s; f[\mathsf{local}[x] = v]$$

$\mathrm{with\_global}(x, v)$

1. Let f be the current frame.
2. Replace $s.\mathsf{global}[f.\mathsf{module}.\mathsf{global}[x]].\mathsf{value}$ with v.

$$(s; f)[\mathsf{global}[x].\mathsf{value} = v] \quad = \quad s[\mathsf{global}[f.\mathsf{module}.\mathsf{global}[x]].\mathsf{value} = v]; f$$

$\mathrm{with\_table}(x, i, r)$

1. Let f be the current frame.
2. Replace $s.\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]].\mathsf{elem}[i]$ with r.

$$(s; f)[\mathsf{table}[x].\mathsf{elem}[i] = r] \quad = \quad s[\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]].\mathsf{elem}[i] = r]; f$$

$\mathrm{with\_tableinst}(x, ti)$

1. Let f be the current frame.
2. Replace $s.\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]]$ with ti.

$$(s; f)[\mathsf{table}[x] = ti] \quad = \quad s[\mathsf{table}[f.\mathsf{module}.\mathsf{table}[x]] = ti]; f$$

$\mathrm{with\_mem}(x, i, j, b^*)$

1. Let f be the current frame.
2. Replace $s.\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]].\mathsf{data}[i : j]$ with b^*.

$$(s; f)[\mathsf{mem}[x].\mathsf{data}[i : j] = b^*] \quad = \quad s[\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]].\mathsf{data}[i : j] = b^*]; f$$

$\mathrm{with\_meminst}(x, mi)$

1. Let f be the current frame.
2. Replace $s.\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]]$ with mi.

$$(s; f)[\mathsf{mem}[x] = mi] \quad = \quad s[\mathsf{mem}[f.\mathsf{module}.\mathsf{mem}[x]] = mi]; f$$

$\mathrm{with\_elem}(x, r^*)$

1. Let f be the current frame.
2. Replace $s.\mathsf{elem}[f.\mathsf{module}.\mathsf{elem}[x]].\mathsf{elem}$ with r^*.

$$(s; f)[\mathsf{elem}[x].\mathsf{elem} = r^*] \quad = \quad s[\mathsf{elem}[f.\mathsf{module}.\mathsf{elem}[x]].\mathsf{elem} = r^*]; f$$

$\mathrm{with\_data}(x, b^*)$

1. Let f be the current frame.
2. Replace $s.\mathsf{data}[f.\mathsf{module}.\mathsf{data}[x]].\mathsf{data}$ with b^*.

$$(s; f)[\mathsf{data}[x].\mathsf{data} = b^*] \quad = \quad s[\mathsf{data}[f.\mathsf{module}.\mathsf{data}[x]].\mathsf{data} = b^*]; f$$

$\mathrm{grow\_table}(ti, n, r)$

1. Let $\{\mathsf{type}\ i\ j\ rt, \mathsf{elem}\ r'^*\}$ be ti.
2. Let i' be $|r'^*| + n$.
3. Let ti' be $\{\mathsf{type}\ i'\ j\ rt, \mathsf{elem}\ r'^*\ r^n\}$.
4. If $ti'.\mathsf{type}$ is valid, then:
   a. Return ti'.

$$\begin{array}{rcll} \mathrm{grow}_{table}(ti,\ n,\ r) & = & ti' & \text{if } ti = \{\mathsf{type}\ [i..j]\ rt,\ \mathsf{elem}\ r'^*\} \\ & & & \wedge\ i' = |r'^*| + n \\ & & & \wedge\ ti' = \{\mathsf{type}\ [i'..j]\ rt,\ \mathsf{elem}\ r'^*\ r^n\} \\ & & & \wedge \vdash ti'.\mathsf{type} : \mathsf{ok} \end{array}$$

$\mathrm{grow\_memory}(mi, n)$

1. Let $\{\mathsf{type}\ \mathsf{i8}\ i\ j, \mathsf{data}\ b^*\}$ be mi.
2. Let i' be $|b^*|/64 \cdot \mathrm{Ki}() + n$.
3. Let mi' be $\{\mathsf{type}\ \mathsf{i8}\ i'\ j, \mathsf{data}\ b^*\ 0^{n \cdot 64} \cdot \mathrm{Ki}()\}$.
4. If $mi'.\mathsf{type}$ is valid, then:
   a. Return mi'.

$$\begin{array}{rcll} \mathrm{grow}_{memory}(mi,\ n) & = & mi' & \text{if } mi = \{\mathsf{type}\ ([i..j]\ \mathsf{i8}),\ \mathsf{data}\ b^*\} \\ & & & \wedge\ i' = |b^*|/(64 \cdot \mathrm{Ki}) + n \\ & & & \wedge\ mi' = \{\mathsf{type}\ ([i'..j]\ \mathsf{i8}),\ \mathsf{data}\ b^*\ 0^{n \cdot 64 \cdot \mathrm{Ki}}\} \\ & & & \wedge \vdash mi'.\mathsf{type} : \mathsf{ok} \end{array}$$

3.3 Instructions

3.3.1 Numeric Instructions

$\mathsf{unop}\ nt\ unop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop $nt.\mathsf{const}\ c_1$ from the stack.
3. If the length of $\mathrm{unop}(unop, nt, c_1)$ is 1, then:
   a. Let c be $\mathrm{unop}(unop, nt, c_1)$.
   b. Push $nt.\mathsf{const}\ c$ to the stack.
4. If $\mathrm{unop}(unop, nt, c_1)$ is ϵ, then:
   a. Trap.

$$\begin{array}{llcll} [\textsc{E-unop-val}] & (nt.\mathsf{const}\ c_1)\ (nt.unop) & \hookrightarrow & (nt.\mathsf{const}\ c) & \text{if } unop_{nt}(c_1) = c \\ [\textsc{E-unop-trap}] & (nt.\mathsf{const}\ c_1)\ (nt.unop) & \hookrightarrow & \mathsf{trap} & \text{if } unop_{nt}(c_1) = \epsilon \end{array}$$

binop nt $binop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c_2 from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop nt.const c_1 from the stack.
5. If the length of $\mathrm{binop}(binop, nt, c_1, c_2)$ is 1, then:
   a. Let c be $\mathrm{binop}(binop, nt, c_1, c_2)$.
   b. Push nt.const c to the stack.
6. If $\mathrm{binop}(binop, nt, c_1, c_2)$ is ϵ, then:
   a. Trap.

$$\begin{array}{llll}[\textsc{E-binop-val}] & (nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.binop) & \hookrightarrow & (nt.\mathsf{const}\ c) & \text{if } binop_{nt}(c_1,\ c_2) = c \\ [\textsc{E-binop-trap}] & (nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.binop) & \hookrightarrow & \mathsf{trap} & \text{if } binop_{nt}(c_1,\ c_2) = \epsilon\end{array}$$

testop nt $testop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c_1 from the stack.
3. Let c be $\mathrm{testop}(testop, nt, c_1)$.
4. Push i32.const c to the stack.

$$[\textsc{E-testop}](nt.\mathsf{const}\ c_1)\ (nt.testop) \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \quad \text{if } c = testop_{nt}(c_1)$$

relop nt $relop$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c_2 from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop nt.const c_1 from the stack.
5. Let c be $\mathrm{relop}(relop, nt, c_1, c_2)$.
6. Push i32.const c to the stack.

$$[\textsc{E-relop}](nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.relop) \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \quad \text{if } c = relop_{nt}(c_1,\ c_2)$$

cvtop nt_2 $cvtop$ nt_1 $sx^?$

1. Assert: Due to validation, a value of value type nt_1 is on the top of the stack.
2. Pop nt_1.const c_1 from the stack.
3. If the length of $\mathrm{cvtop}(nt_1, cvtop, nt_2, sx^?, c_1)$ is 1, then:
   a. Let c be $\mathrm{cvtop}(nt_1, cvtop, nt_2, sx^?, c_1)$.
   b. Push nt_2.const c to the stack.
4. If $\mathrm{cvtop}(nt_1, cvtop, nt_2, sx^?, c_1)$ is ϵ, then:
   a. Trap.

$$\begin{array}{llcll}
[\textsc{E-cvtop-val}] & (nt_1.\mathsf{const}\ c_1)\ (nt_2.cvtop_nt_1_sx^?) & \hookrightarrow & (nt_2.\mathsf{const}\ c) & \text{if } \mathrm{cvtop}(nt_1,\ cvtop,\ nt_2,\ sx^?,\ c_1) = c \\
[\textsc{E-cvtop-trap}] & (nt_1.\mathsf{const}\ c_1)\ (nt_2.cvtop_nt_1_sx^?) & \hookrightarrow & \mathsf{trap} & \text{if } \mathrm{cvtop}(nt_1,\ cvtop,\ nt_2,\ sx^?,\ c_1) = \epsilon
\end{array}$$

3.3.2 Reference Instructions

ref.is_null

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. If val is not of the case ref.null, then:
   a. Push i32.const 0 to the stack.
4. Else:
   a. Push i32.const 1 to the stack.

$$\begin{array}{llcll}
[\textsc{E-ref.is_null-true}] & val\ \mathsf{ref.is_null} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } val = (\mathsf{ref.null}\ rt) \\
[\textsc{E-ref.is_null-false}] & val\ \mathsf{ref.is_null} & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}$$

ref.func x

1. Assert: Due to validation, x is less than the length of $\mathrm{funcaddr}()$.
2. Push ref.func_addr $\mathrm{funcaddr}()[x]$ to the stack.

$$[\textsc{E-ref.func}]\, z; (\mathsf{ref.func}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ z.\mathsf{module.func}[x])$$

3.3.3 Parametric Instructions

drop

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

$$[\textsc{E-drop}]\, val\ \mathsf{drop} \quad \hookrightarrow \quad \epsilon$$

select $t^?$

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const c from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop val_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop val_1 from the stack.
7. If c is not 0, then:
   a. Push val_1 to the stack.
8. Else:
   a. Push val_2 to the stack.

$$\begin{array}{lcll} [\textsc{E-select-true}]\ val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_1 & \text{if } c \neq 0 \\ [\textsc{E-select-false}]\, val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_2 & \text{if } c = 0 \end{array}$$

3.3.4 Variable Instructions

local.get x

1. Push local(x) to the stack.

$$[\textsc{E-local.get}]\, z; (\mathsf{local.get}\ x) \quad \hookrightarrow \quad z.\mathsf{local}[x]$$

local.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. Perform with_local(x, val).

$$[\textsc{E-local.set}] z; val\ (\mathsf{local.set}\ x) \quad \hookrightarrow \quad z[\mathsf{local}[x] = val]; \epsilon$$

local.tee x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. Push val to the stack.
4. Push val to the stack.
5. Execute local.set x.

$$[\textsc{E-local.tee}] val\ (\mathsf{local.tee}\ x) \quad \hookrightarrow \quad val\ val\ (\mathsf{local.set}\ x)$$

global.get x

1. Push global(x).value to the stack.

$$[\textsc{E-global.get}] z; (\mathsf{global.get}\ x) \quad \hookrightarrow \quad z.\mathsf{global}[x].\mathsf{value}$$

global.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. Perform with_global(x, val).

$$[\textsc{E-global.set}] z; val\ (\mathsf{global.set}\ x) \quad \hookrightarrow \quad z[\mathsf{global}[x].\mathsf{value} = val]; \epsilon$$

3.3.5 Table Instructions

table.get x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const i from the stack.
3. If i is greater than or equal to the length of $\mathrm{table}(x)$.elem, then:
   a. Trap.
4. Push $\mathrm{table}(x)$.elem$[i]$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-table.get-trap}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i \geq |z.\mathsf{table}[x].\mathsf{elem}| \\
[\textsc{E-table.get-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ x) & \hookrightarrow & z.\mathsf{table}[x].\mathsf{elem}[i] & \text{if } i < |z.\mathsf{table}[x].\mathsf{elem}|
\end{array}
$$

table.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop i32.const i from the stack.
5. If i is greater than or equal to the length of $\mathrm{table}(x)$.elem, then:
   a. Trap.
6. Perform $\mathrm{with\_table}(x, i, ref)$.

$$
\begin{array}{llcll}
[\textsc{E-table.set-trap}] & z; (\mathsf{i32.const}\ i)\ \mathit{ref}\ (\mathsf{table.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \text{if } i \geq |z.\mathsf{table}[x].\mathsf{elem}| \\
[\textsc{E-table.set-val}] & z; (\mathsf{i32.const}\ i)\ \mathit{ref}\ (\mathsf{table.set}\ x) & \hookrightarrow & z[\mathsf{table}[x].\mathsf{elem}[i] = \mathit{ref}]; \epsilon & \text{if } i < |z.\mathsf{table}[x].\mathsf{elem}|
\end{array}
$$

table.size x

1. Let n be the length of $\mathrm{table}(x)$.elem.
2. Push i32.const n to the stack.

$$
[\textsc{E-table.size}]\ z; (\mathsf{table.size}\ x) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \quad \text{if } |z.\mathsf{table}[x].\mathsf{elem}| = n
$$

table.grow x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop ref from the stack.
5. Either:
   a. Let ti be $\mathrm{grow\_table}(\mathrm{table}(x), n, ref)$.
   b. Push i32.const $|\mathrm{table}(x).\mathsf{elem}|$ to the stack.
   c. Perform $\mathrm{with\_tableinst}(x, ti)$.
6. Or:
   a. Push i32.const -1 to the stack.

$$
\begin{array}{llcll}
[\textsc{E-table.grow-succeed}] & z; ref\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z[\mathsf{table}[x] = ti]; (\mathsf{i32.const}\ |z.\mathsf{table}[x].\mathsf{elem}|) & \text{if } \mathrm{grow}_{table}(z.\mathsf{table}[x] \\
[\textsc{E-table.grow-fail}] & z; ref\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ -1) &
\end{array}
$$

table.fill x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop val from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const i from the stack.
7. If $i + n$ is greater than the length of $\mathrm{table}(x).\mathsf{elem}$, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else:
   a. Push i32.const i to the stack.
   b. Push val to the stack.
   c. Execute table.set x.
   d. Push i32.const $i + 1$ to the stack.
   e. Push val to the stack.
   f. Push i32.const $n - 1$ to the stack.
   g. Execute table.fill x.

$$[\textsc{E-table.fill-trap}]\ z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) \hookrightarrow \mathsf{trap}$$
$$[\textsc{E-table.fill-zero}]\ z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) \hookrightarrow \epsilon$$
$$[\textsc{E-table.fill-succ}]\ z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) \hookrightarrow (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x)\ (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.const}\ n$$

table.copy x y

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop i32.const i from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const j from the stack.
7. If $i + n$ is greater than the length of $\mathrm{table}(y)$.elem or $j + n$ is greater than the length of $\mathrm{table}(x)$.elem, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else:
   a. If j is less than or equal to i, then:
      1) Push i32.const j to the stack.
      2) Push i32.const i to the stack.
      3) Execute table.get y.
      4) Execute table.set x.
      5) Push i32.const $j + 1$ to the stack.
      6) Push i32.const $i + 1$ to the stack.
   b. Else:
      1) Push i32.const $j + n - 1$ to the stack.
      2) Push i32.const $i + n - 1$ to the stack.
      3) Execute table.get y.
      4) Execute table.set x.
      5) Push i32.const j to the stack.
      6) Push i32.const i to the stack.
   c. Push i32.const $n - 1$ to the stack.
   d. Execute table.copy x y.

$$[\textsc{E-table.copy-trap}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) \hookrightarrow \mathsf{trap}$$
$$[\textsc{E-table.copy-zero}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) \hookrightarrow \epsilon$$
$$[\textsc{E-table.copy-le}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) \hookrightarrow (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}$$
$$[\textsc{E-table.copy-gt}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) \hookrightarrow (\mathsf{i32.const}\ j+n-1)\ (\mathsf{i32.const}\ i+n-1)\ (\mathsf{table}$$

table.init x y

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop i32.const i from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const j from the stack.
7. If $i+n$ is greater than the length of $\mathrm{elem}(y)$.elem or $j+n$ is greater than the length of $\mathrm{table}(x)$.elem, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else if i is less than the length of $\mathrm{elem}(y)$.elem, then:
   a. Push i32.const j to the stack.
   b. Push $\mathrm{elem}(y)$.elem$[i]$ to the stack.
   c. Execute table.set x.
   d. Push i32.const $j+1$ to the stack.
   e. Push i32.const $i+1$ to the stack.
   f. Push i32.const $n-1$ to the stack.
   g. Execute table.init x y.

$$
\begin{array}{lll}
[\textsc{E-table.init-trap}]\, z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-table.init-zero}]\, z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \epsilon \\
[\textsc{E-table.init-succ}]\, z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & (\mathsf{i32.const}\ j)\ z.\mathsf{elem}[y].\mathsf{elem}[i]\ (\mathsf{table.set}\ x)\ (\mathsf{i32.cor}
\end{array}
$$

elem.drop x

1. Perform $\mathrm{with\_elem}(x, \epsilon)$.

$$
[\textsc{E-elem.drop}]\, z; (\mathsf{elem.drop}\ x) \quad \hookrightarrow \quad z[\mathsf{elem}[x].\mathsf{elem} = \epsilon]; \epsilon
$$

3.3.6 Memory Instructions

load nt ${x_0}^?$ n_A n_O

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const i from the stack.
3. If ${x_0}^?$ is not defined, then:
   a. If $i + n_O + \mathrm{size}(nt)/8$ is greater than the length of $\mathrm{mem}(0)$.data, then:
      1) Trap.

b. Let c be $inverse\_of\_bytes(\mathrm{size}(nt), \mathrm{mem}(0).\mathsf{data}[i + n_O : \mathrm{size}(nt)/8])$.

c. Push $nt.\mathsf{const}\ c$ to the stack.

4. Else:

a. Let ${y_0}^?$ be ${x_0}^?$.

b. Let $n\ sx$ be y_0.

c. If $i + n_O + n/8$ is greater than the length of $\mathrm{mem}(0).\mathsf{data}$, then:

1) Trap.

d. Let c be $inverse\_of\_bytes(n, \mathrm{mem}(0).\mathsf{data}[i + n_O : n/8])$.

e. Push $nt.\mathsf{const}\ \mathrm{ext}(n, \mathrm{size}(nt), sx, c)$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-load-num-trap}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & \mathsf{trap} & \text{if } i + n_{\mathsf{o}} + |nt|/8 > |z.\mathsf{mem}[0].\mathsf{data}| \\
[\textsc{E-load-num-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & (nt.\mathsf{const}\ c) & \text{if } \mathrm{bytes}_{|nt|}(c) = z.\mathsf{mem}[0].\mathsf{data}[i + n_{\mathsf{o}} : \\
[\textsc{E-load-pack-trap}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}n\_sx\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & \mathsf{trap} & \text{if } i + n_{\mathsf{o}} + n/8 > |z.\mathsf{mem}[0].\mathsf{data}| \\
[\textsc{E-load-pack-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}n\_sx\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & (nt.\mathsf{const}\ \mathrm{ext}_n(|nt|)^{sx}\ (c)) & \text{if } \mathrm{bytes}_n(c) = z.\mathsf{mem}[0].\mathsf{data}[i + n_{\mathsf{o}} : n
\end{array}
$$

$\mathsf{store}\ nt\ {x_0}^?\ n_A\ n_O$

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop $nt.\mathsf{const}\ c$ from the stack.
3. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
4. Pop $\mathsf{i32.const}\ i$ from the stack.
5. If ${x_0}^?$ is not defined, then:

a. If $i + n_O + \mathrm{size}(nt)/8$ is greater than the length of $\mathrm{mem}(0).\mathsf{data}$, then:

1) Trap.

b. Let b^* be $\mathrm{bytes\_}(\mathrm{size}(nt), c)$.

c. Perform $\mathrm{with\_mem}(0, i + n_O, \mathrm{size}(nt)/8, b^*)$.

6. Else:

a. Let $n^?$ be ${x_0}^?$.

b. If $i + n_O + n/8$ is greater than the length of $\mathrm{mem}(0).\mathsf{data}$, then:

1) Trap.

c. Let b^* be $\mathrm{bytes\_}(n, \mathrm{wrap\_}(\mathrm{size}(nt)\ n, c))$.

d. Perform $\mathrm{with\_mem}(0, i + n_O, n/8, b^*)$.

$$
\begin{array}{llcll}
[\textsc{E-store-num-trap}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & z; \mathsf{trap} & \text{if } i + n_{\mathsf{o}} + |nt|/8 \\
[\textsc{E-store-num-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & z[\mathsf{mem}[0].\mathsf{data}[i + n_{\mathsf{o}} : |nt|/8] = b^*]; \epsilon & \text{if } b^* = \mathrm{bytes}_{|nt|}(c \\
[\textsc{E-store-pack-trap}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & z; \mathsf{trap} & \text{if } i + n_{\mathsf{o}} + n/8 > \\
[\textsc{E-store-pack-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ n_{\mathsf{a}}\ n_{\mathsf{o}}) & \hookrightarrow & z[\mathsf{mem}[0].\mathsf{data}[i + n_{\mathsf{o}} : n/8] = b^*]; \epsilon & \text{if } b^* = \mathrm{bytes}_n(\mathrm{wra}
\end{array}
$$

memory.size

1. Let $n \cdot 64 \cdot \mathrm{Ki}()$ be the length of $\mathrm{mem}(0)$.data.
2. Push i32.const n to the stack.

$$[\text{E-MEMORY.SIZE}]\, z; (\mathsf{memory.size}) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \quad \text{if } n \cdot 64 \cdot \mathrm{Ki} = |z.\mathsf{mem}[0].\mathsf{data}|$$

memory.grow

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Either:
   a. Let mi be $\mathrm{grow\_memory}(\mathrm{mem}(0), n)$.
   b. Push i32.const $|\mathrm{mem}(0).\mathsf{data}|/64 \cdot \mathrm{Ki}()$ to the stack.
   c. Perform $\mathrm{with\_meminst}(0, mi)$.
4. Or:
   a. Push i32.const -1 to the stack.

$$[\text{E-MEMORY.GROW-SUCCEED}]\, z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}) \quad \hookrightarrow \quad z[\mathsf{mem}[0] = mi]; (\mathsf{i32.const}\ |z.\mathsf{mem}[0].\mathsf{data}|/(64 \cdot \mathrm{Ki})) \quad \text{if grow}_{memor}$$

$$[\text{E-MEMORY.GROW-FAIL}]\, z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}) \quad \hookrightarrow \quad z; (\mathsf{i32.const}\ -1)$$

memory.fill

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop val from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const i from the stack.
7. If $i + n$ is greater than the length of $\mathrm{mem}(0)$.data, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else:
   a. Push i32.const i to the stack.
   b. Push val to the stack.
   c. Execute store i32 $8^?$ 0 0.
   d. Push i32.const $i + 1$ to the stack.
   e. Push val to the stack.

f. Push i32.const $n-1$ to the stack.

g. Execute memory.fill.

$$
\begin{array}{lcl}
[\textsc{E-memory.fill-trap}]\, z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-memory.fill-zero}]\, z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}) & \hookrightarrow & \epsilon \\
[\textsc{E-memory.fill-succ}]\, z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}) & \hookrightarrow & (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.store8}\ 0\ 0)\ (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.con}
\end{array}
$$

memory.copy

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop i32.const i from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const j from the stack.
7. If $i+n$ is greater than the length of $\mathrm{mem}(0)$.data or $j+n$ is greater than the length of $\mathrm{mem}(0)$.data, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else:
   a. If j is less than or equal to i, then:
      1) Push i32.const j to the stack.
      2) Push i32.const i to the stack.
      3) Execute load i32 8 $\mathsf{u}^?$ 0 0.
      4) Execute store i32 $8^?$ 0 0.
      5) Push i32.const $j+1$ to the stack.
      6) Push i32.const $i+1$ to the stack.
   b. Else:
      1) Push i32.const $j+n-1$ to the stack.
      2) Push i32.const $i+n-1$ to the stack.
      3) Execute load i32 8 $\mathsf{u}^?$ 0 0.
      4) Execute store i32 $8^?$ 0 0.
      5) Push i32.const j to the stack.
      6) Push i32.const i to the stack.
   c. Push i32.const $n-1$ to the stack.
   d. Execute memory.copy.

$$
\begin{array}{llcl}
[\text{E-MEMORY.COPY-TRAP}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}) & \hookrightarrow & \mathsf{trap} \\
[\text{E-MEMORY.COPY-ZERO}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}) & \hookrightarrow & \epsilon \\
[\text{E-MEMORY.COPY-LE}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}) & \hookrightarrow & (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.load8\_u}\ 0\ 0)\ (\mathsf{i32.st} \\
[\text{E-MEMORY.COPY-GT}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}) & \hookrightarrow & (\mathsf{i32.const}\ j+n-1)\ (\mathsf{i32.const}\ i+n-1)\ (\mathsf{i32.lo}
\end{array}
$$

memory.init x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const n from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop i32.const i from the stack.
5. Assert: Due to validation, a value of value type i32 is on the top of the stack.
6. Pop i32.const j from the stack.
7. If $i + n$ is greater than the length of $\mathrm{data}(x)$.data or $j + n$ is greater than the length of $\mathrm{mem}(0)$.data, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else if i is less than the length of $\mathrm{data}(x)$.data, then:
   a. Push i32.const j to the stack.
   b. Push i32.const $\mathrm{data}(x)$.data$[i]$ to the stack.
   c. Execute store i32 $8^?$ 0 0.
   d. Push i32.const $j + 1$ to the stack.
   e. Push i32.const $i + 1$ to the stack.
   f. Push i32.const $n - 1$ to the stack.
   g. Execute memory.init x.

$$
\begin{array}{llcl}
[\text{E-MEMORY.INIT-TRAP}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x) & \hookrightarrow & \mathsf{trap} \\
[\text{E-MEMORY.INIT-ZERO}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x) & \hookrightarrow & \epsilon \\
[\text{E-MEMORY.INIT-SUCC}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x) & \hookrightarrow & (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ z.\mathsf{data}[x].\mathsf{data}[i])\ (\mathsf{i32.sto}
\end{array}
$$

data.drop x

1. Perform $\mathrm{with\_data}(x, \epsilon)$.

$$[\text{E-DATA.DROP}]\, z; (\mathsf{data.drop}\ x) \quad \hookrightarrow \quad z[\mathsf{data}[x].\mathsf{data} = \epsilon]; \epsilon$$

3.3.7 Control Instructions

nop

1. Do nothing.

$$[\textsc{E-nop}]\,\mathsf{nop} \quad \hookrightarrow \quad \epsilon$$

unreachable

1. Trap.

$$[\textsc{E-unreachable}]\,\mathsf{unreachable} \quad \hookrightarrow \quad \mathsf{trap}$$

block $bt\ instr^*$

1. Let ${t_1}^k \to {t_2}^n$ be bt.
2. Assert: Due to validation, there are at least k values on the top of the stack.
3. Pop val^k from the stack.
4. Let L be the label whose arity is n and whose continuation is ϵ.
5. Enter L with label $instr^*\ LABEL$:
   a. Push val^k to the stack.

$$[\textsc{E-block}]\,val^k\ (\mathsf{block}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_n\{\epsilon\}\ val^k\ instr^*) \quad \text{if } bt = t_1^k \to t_2^n$$

loop $bt\ instr^*$

1. Let ${t_1}^k \to {t_2}^n$ be bt.
2. Assert: Due to validation, there are at least k values on the top of the stack.
3. Pop val^k from the stack.
4. Let L be the label whose arity is k and whose continuation is loop $bt\ instr^*$.
5. Enter L with label $instr^*\ LABEL$:
   a. Push val^k to the stack.

$$[\textsc{E-loop}]\,val^k\ (\mathsf{loop}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_k\{\mathsf{loop}\ bt\ instr^*\}\ val^k\ instr^*) \quad \text{if } bt = t_1^k \to t_2^n$$

if $bt\ instr_1{}^*\ instr_2{}^*$

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const c from the stack.
3. If c is not 0, then:
   a. Execute block $bt\ instr_1{}^*$.
4. Else:
   a. Execute block $bt\ instr_2{}^*$.

$$\begin{array}{llll}
[\textsc{E-if-true}]\ (\mathsf{i32.const}\ c)\ (\mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ bt\ instr_1^*) & \mathsf{if}\ c \neq 0 \\
[\textsc{E-if-false}](\mathsf{i32.const}\ c)\ (\mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ bt\ instr_2^*) & \mathsf{if}\ c = 0
\end{array}$$

br x_0

1. Let L be the current label.
2. Let n be the arity of L.
3. Let $instr'^*$ be the continuation of L.
4. Pop all values $x_1{}^*$ from the stack.
5. Exit current context.
6. If x_0 is 0 and the length of $x_1{}^*$ is greater than or equal to n, then:
   a. Let $val'^*\ val^n$ be $x_1{}^*$.
   b. Push val^n to the stack.
   c. Execute the sequence $instr'^*$.
7. If x_0 is greater than or equal to 1, then:
   a. Let l be $x_0 - 1$.
   b. Let val^* be $x_1{}^*$.
   c. Push val^* to the stack.
   d. Execute br l.

$$\begin{array}{lll}
[\textsc{E-br-zero}](\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ 0)\ instr^*) & \hookrightarrow & val^n\ instr'^* \\
[\textsc{E-br-succ}](\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l+1)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l)
\end{array}$$

br_if l

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const c from the stack.
3. If c is not 0, then:
   a. Execute br l.
4. Else:
   a. Do nothing.

$$
\begin{array}{lllll}
[\textsc{E-br\_if-true}] & (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } c \neq 0 \\
[\textsc{E-br\_if-false}] & (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & \text{if } c = 0
\end{array}
$$

br_table l^* l'

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const i from the stack.
3. If i is less than the length of l^*, then:
   a. Execute br $l^*[i]$.
4. Else:
   a. Execute br l'.

$$
\begin{array}{lllll}
[\textsc{E-br\_table-lt}] & (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l^*[i]) & \text{if } i < |l^*| \\
[\textsc{E-br\_table-ge}] & (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l') & \text{if } i \geq |l^*|
\end{array}
$$

return

1. If the current context is frame, then:
   a. Let F be the current frame.
   b. Let n be the arity of F.
   c. Pop val^n from the stack.
   d. Pop all values val'^* from the stack.
   e. Exit current context.
   f. Push val^n to the stack.
2. Else if the current context is label, then:
   a. Pop all values val^* from the stack.
   b. Exit current context.
   c. Push val^* to the stack.
   d. Execute return.

$$[\textsc{E-return-frame}](\mathsf{frame}_n\{f\}\ val'^*\ val^n\ \mathsf{return}\ instr^*) \quad \hookrightarrow \quad val^n$$
$$[\textsc{E-return-label}]\ (\mathsf{label}_k\{instr'^*\}\ val^*\ \mathsf{return}\ instr^*) \quad \hookrightarrow \quad val^*\ \mathsf{return}$$

call x

1. Assert: Due to validation, x is less than the length of funcaddr$()$.
2. Execute call_addr funcaddr$()[x]$.

$$[\textsc{E-call}] z; (\mathsf{call}\ x) \quad \hookrightarrow \quad (\mathsf{call}\ z.\mathsf{module}.\mathsf{func}[x])$$

call_indirect x ft

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const i from the stack.
3. If i is greater than or equal to the length of table(x).elem, then:
   a. Trap.
4. If table(x).elem$[i]$ is not of the case ref.func_addr, then:
   a. Trap.
5. Let ref.func_addr a be table(x).elem$[i]$.
6. If a is greater than or equal to the length of funcinst$()$, then:
   a. Trap.
7. If funcinst$()[a]$.code is not of the case func, then:
   a. Trap.
8. Let func ft' t^* $instr^*$ be funcinst$()[a]$.code.
9. If ft is not ft', then:
   a. Trap.
10. Execute call_addr a.

$$\begin{array}{llll}
[\textsc{E-call_indirect-call}] z; (\mathsf{i32.const}\ i)\ (\mathsf{call_indirect}\ x\ ft) & \hookrightarrow & (\mathsf{call}\ a) & \text{if } z.\mathsf{table}[x].\mathsf{elem}[i] = (\mathsf{ref.func}\ a) \\
& & & \wedge\ z.\mathsf{func}[a].\mathsf{code} = \mathsf{func}\ ft'\ t^*\ instr^* \\
& & & \wedge\ ft = ft' \\
[\textsc{E-call_indirect-trap}] z; (\mathsf{i32.const}\ i)\ (\mathsf{call_indirect}\ x\ ft) & \hookrightarrow & \mathsf{trap} & \text{otherwise}
\end{array}$$

3.3.8 Blocks

label_

1. Pop all values val^* from the stack.
2. Assert: Due to validation, a label is now on the top of the stack.
3. Exit current context.
4. Push val^* to the stack.

$$[\textsc{E-label-vals}](\mathsf{label}_n\{instr^*\}\ val^*) \quad \hookrightarrow \quad val^*$$

3.3.9 Function Calls

call_addr a

1. Assert: Due to validation, a is less than the length of funcinst().
2. Let $\{\mathsf{module}\ m, \mathsf{code}\ func\}$ be funcinst()$[a]$.
3. Assert: Due to validation, $func$ is of the case func.
4. Let func $y_0\ t^*\ instr^*$ be $func$.
5. Let ${t_1}^k \rightarrow {t_2}^n$ be y_0.
6. Assert: Due to validation, there are at least k values on the top of the stack.
7. Pop val^k from the stack.
8. Let f be $\{\mathsf{local}\ val^k\ (\mathrm{default\_}(t))^*, \mathsf{module}\ m\}$.
9. Let F be the activation of f with arity n.
10. Enter F with label $FRAME$:
   a. Let L be the label whose arity is n and whose continuation is ϵ.
   b. Enter L with label $instr^*\ LABEL$:

$$[\textsc{E-call\_addr}]z; val^k\ (\mathsf{call}\ a) \quad \hookrightarrow \quad (\mathsf{frame}_n\{f\}\ (\mathsf{label}_n\{\epsilon\}\ instr^*)) \quad \begin{array}{l} \text{if } z.\mathsf{func}[a] = \{\mathsf{module}\ m,\ \mathsf{code}\ func\} \\ \wedge\ func = \mathsf{func}\ (t_1^k \rightarrow t_2^n)\ t^*\ instr^* \\ \wedge\ f = \{\mathsf{local}\ val^k\ (\mathrm{default}_t)^*,\ \mathsf{module}\ m\} \end{array}$$

frame_

1. Let f be the current frame.
2. Let n be the arity of f.
3. Assert: Due to validation, there are at least n values on the top of the stack.
4. Pop val^n from the stack.
5. Assert: Due to validation, a frame is now on the top of the stack.
6. Exit current context.

7. Push val^n to the stack.

$$[\textsc{E-frame-vals}]\,(\mathsf{frame}_n\{f\}\ val^n) \quad\hookrightarrow\quad val^n$$

3.4 Modules

3.4.1 Allocation

$\mathrm{funcs}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ \mathit{externval}'^*$ be ${x_0}^*$.
3. If y_0 is of the case func, then:
   a. Let $\mathsf{func}\ fa$ be y_0.
   b. Return $fa\ \mathrm{funcs}(\mathit{externval}'^*)$.
4. Let $\mathit{externval}\ \mathit{externval}'^*$ be ${x_0}^*$.
5. Return $\mathrm{funcs}(\mathit{externval}'^*)$.

$$\begin{array}{lcll}
\mathrm{funcs}(\epsilon) & = & \epsilon & \\
\mathrm{funcs}((\mathsf{func}\ fa)\ \mathit{externval}'^*) & = & fa\ \mathrm{funcs}(\mathit{externval}'^*) & \\
\mathrm{funcs}(\mathit{externval}\ \mathit{externval}'^*) & = & \mathrm{funcs}(\mathit{externval}'^*) & \quad\text{otherwise}
\end{array}$$

$\mathrm{globals}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ \mathit{externval}'^*$ be ${x_0}^*$.
3. If y_0 is of the case global, then:
   a. Let $\mathsf{global}\ ga$ be y_0.
   b. Return $ga\ \mathrm{globals}(\mathit{externval}'^*)$.
4. Let $\mathit{externval}\ \mathit{externval}'^*$ be ${x_0}^*$.
5. Return $\mathrm{globals}(\mathit{externval}'^*)$.

$$\begin{array}{lcll}
\mathrm{globals}(\epsilon) & = & \epsilon & \\
\mathrm{globals}((\mathsf{global}\ ga)\ \mathit{externval}'^*) & = & ga\ \mathrm{globals}(\mathit{externval}'^*) & \\
\mathrm{globals}(\mathit{externval}\ \mathit{externval}'^*) & = & \mathrm{globals}(\mathit{externval}'^*) & \quad\text{otherwise}
\end{array}$$

$\mathrm{tables}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ {externval'}^*$ be ${x_0}^*$.
3. If y_0 is of the case table, then:
   a. Let table ta be y_0.
   b. Return $ta\ \mathrm{tables}({externval'}^*)$.
4. Let $externval\ {externval'}^*$ be ${x_0}^*$.
5. Return $\mathrm{tables}({externval'}^*)$.

$$
\begin{array}{lcll}
\mathrm{tables}(\epsilon) & = & \epsilon & \\
\mathrm{tables}((\mathsf{table}\ ta)\ {externval'}^*) & = & ta\ \mathrm{tables}({externval'}^*) & \\
\mathrm{tables}(externval\ {externval'}^*) & = & \mathrm{tables}({externval'}^*) & \text{otherwise}
\end{array}
$$

$\mathrm{mems}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ {externval'}^*$ be ${x_0}^*$.
3. If y_0 is of the case mem, then:
   a. Let mem ma be y_0.
   b. Return $ma\ \mathrm{mems}({externval'}^*)$.
4. Let $externval\ {externval'}^*$ be ${x_0}^*$.
5. Return $\mathrm{mems}({externval'}^*)$.

$$
\begin{array}{lcll}
\mathrm{mems}(\epsilon) & = & \epsilon & \\
\mathrm{mems}((\mathsf{mem}\ ma)\ {externval'}^*) & = & ma\ \mathrm{mems}({externval'}^*) & \\
\mathrm{mems}(externval\ {externval'}^*) & = & \mathrm{mems}({externval'}^*) & \text{otherwise}
\end{array}
$$

$\mathrm{allocfunc}(m, func)$

1. Let fi be $\{\mathsf{module}\ m, \mathsf{code}\ func\}$.
2. Return $s \bigoplus \{.\mathsf{func}\ fi\}\ |s.\mathsf{func}|$.

$$
\mathrm{allocfunc}(s,\ m,\ func) \quad = \quad (s[\mathsf{func} = ..fi],\ |s.\mathsf{func}|) \quad \text{if } fi = \{\mathsf{module}\ m,\ \mathsf{code}\ func\}
$$

$\mathrm{allocfuncs}(m, x_0{}^*)$

1. If $x_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $func\ func'^*$ be $x_0{}^*$.
3. Let fa be $\mathrm{allocfunc}(m, func)$.
4. Let fa'^* be $\mathrm{allocfuncs}(m, func'^*)$.
5. Return $fa\ fa'^*$.

$$\begin{array}{lcll}
\mathrm{allocfuncs}(s,\ m,\ \epsilon) & = & (s,\ \epsilon) & \\
\mathrm{allocfuncs}(s,\ m,\ \mathit{func}\ \mathit{func}'^*) & = & (s_2,\ \mathit{fa}\ \mathit{fa}'^*) & \text{if } (s_1,\ \mathit{fa}) = \mathrm{allocfunc}(s,\ m,\ \mathit{func}) \\
 & & & \wedge\ (s_2,\ \mathit{fa}'^*) = \mathrm{allocfuncs}(s_1,\ m,\ \mathit{func}'^*)
\end{array}$$

$\mathrm{allocglobal}(globaltype, val)$

1. Let gi be $\{\mathsf{type}\ globaltype, \mathsf{value}\ val\}$.
2. Return $s \bigoplus \{.\mathsf{global}\ gi\}\ |s.\mathsf{global}|$.

$$\mathrm{allocglobal}(s,\ \mathit{globaltype},\ \mathit{val}) \quad = \quad (s[\mathsf{global} = ..\mathit{gi}],\ |s.\mathsf{global}|) \quad \text{if } \mathit{gi} = \{\mathsf{type}\ \mathit{globaltype},\ \mathsf{value}\ \mathit{val}\}$$

$\mathrm{allocglobals}(x_0{}^*, x_1{}^*)$

1. If $x_0{}^*$ is ϵ, then:
   a. Assert: Due to validation, $x_1{}^*$ is ϵ.
   b. Return ϵ.
2. Else:
   a. Let $globaltype\ globaltype'^*$ be $x_0{}^*$.
   b. Assert: Due to validation, the length of $x_1{}^*$ is greater than or equal to 1.
   c. Let $val\ val'^*$ be $x_1{}^*$.
   d. Let ga be $\mathrm{allocglobal}(globaltype, val)$.
   e. Let ga'^* be $\mathrm{allocglobals}(globaltype'^*, val'^*)$.
   f. Return $ga\ ga'^*$.

$$\begin{array}{lcll}
\mathrm{allocglobals}(s,\ \epsilon,\ \epsilon) & = & (s,\ \epsilon) & \\
\mathrm{allocglobals}(s,\ \mathit{globaltype}\ \mathit{globaltype}'^*,\ \mathit{val}\ \mathit{val}'^*) & = & (s_2,\ \mathit{ga}\ \mathit{ga}'^*) & \text{if } (s_1,\ \mathit{ga}) = \mathrm{allocglobal}(s,\ \mathit{globaltype},\ \mathit{val}) \\
 & & & \wedge\ (s_2,\ \mathit{ga}'^*) = \mathrm{allocglobals}(s_1,\ \mathit{globaltype}'^*,\ \mathit{val}'^*)
\end{array}$$

$\mathrm{alloctable}(i\ j\ rt)$

1. Let ti be $\{\mathsf{type}\ i\ j\ rt, \mathsf{elem}\ \mathsf{ref.null}\ rt^i\}$.
2. Return $s \bigoplus \{.\mathsf{table}\ ti\}\ |s.\mathsf{table}|$.

$$\mathrm{alloctable}(s,\ [i..j]\ rt) \quad = \quad (s[\mathsf{table} = ..ti],\ |s.\mathsf{table}|) \quad \text{if } ti = \{\mathsf{type}\ ([i..j]\ rt),\ \mathsf{elem}\ (\mathsf{ref.null}\ rt)^i\}$$

$\mathrm{alloctables}(x_0{}^*)$

1. If $x_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $tabletype\ tabletype'^*$ be $x_0{}^*$.
3. Let ta be $\mathrm{alloctable}(tabletype)$.
4. Let ta'^* be $\mathrm{alloctables}(tabletype'^*)$.
5. Return $ta\ ta'^*$.

$$\begin{array}{lcll} \mathrm{alloctables}(s,\ \epsilon) & = & (s,\ \epsilon) & \\ \mathrm{alloctables}(s,\ tabletype\ tabletype'^*) & = & (s_2,\ ta\ ta'^*) & \text{if } (s_1,\ ta) = \mathrm{alloctable}(s,\ tabletype) \\ & & & \wedge\ (s_2,\ ta'^*) = \mathrm{alloctables}(s_1,\ tabletype'^*) \end{array}$$

$\mathrm{allocmem}(\mathsf{i8}\ i\ j)$

1. Let mi be $\{\mathsf{type}\ \mathsf{i8}\ i\ j, \mathsf{data}\ 0^{i \cdot 64} \cdot \mathrm{Ki}()\}$.
2. Return $s \bigoplus \{.\mathsf{mem}\ mi\}\ |s.\mathsf{mem}|$.

$$\mathrm{allocmem}(s,\ [i..j]\ \mathsf{i8}) \quad = \quad (s[\mathsf{mem} = ..mi],\ |s.\mathsf{mem}|) \quad \text{if } mi = \{\mathsf{type}\ ([i..j]\ \mathsf{i8}),\ \mathsf{data}\ 0^{i \cdot 64 \cdot \mathrm{Ki}}\}$$

$\mathrm{allocmems}(x_0{}^*)$

1. If $x_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $memtype\ memtype'^*$ be $x_0{}^*$.
3. Let ma be $\mathrm{allocmem}(memtype)$.
4. Let ma'^* be $\mathrm{allocmems}(memtype'^*)$.
5. Return $ma\ ma'^*$.

$$\begin{array}{lcll} \mathrm{allocmems}(s,\ \epsilon) & = & (s,\ \epsilon) & \\ \mathrm{allocmems}(s,\ memtype\ memtype'^*) & = & (s_2,\ ma\ ma'^*) & \text{if } (s_1,\ ma) = \mathrm{allocmem}(s,\ memtype) \\ & & & \wedge\ (s_2,\ ma'^*) = \mathrm{allocmems}(s_1,\ memtype'^*) \end{array}$$

$\mathrm{allocelem}(rt, ref^*)$

1. Let ei be $\{\mathsf{type}\ rt, \mathsf{elem}\ ref^*\}$.
2. Return $s \bigoplus \{.\mathsf{elem}\ ei\}\ |s.\mathsf{elem}|$.

$$\mathrm{allocelem}(s,\ rt,\ ref^*) \quad = \quad (s[\mathsf{elem} = ..ei],\ |s.\mathsf{elem}|) \quad \text{if } ei = \{\mathsf{type}\ rt,\ \mathsf{elem}\ ref^*\}$$

$\mathrm{allocelems}({x_0}^*, {x_1}^*)$

1. If ${x_0}^*$ is ϵ and ${x_1}^*$ is ϵ, then:
   a. Return ϵ.
2. Assert: Due to validation, the length of ${x_1}^*$ is greater than or equal to 1.
3. Let $ref^*\ (ref'^*)^*$ be ${x_1}^*$.
4. Assert: Due to validation, the length of ${x_0}^*$ is greater than or equal to 1.
5. Let $rt\ rt'^*$ be ${x_0}^*$.
6. Let ea be $\mathrm{allocelem}(rt, ref^*)$.
7. Let ea'^* be $\mathrm{allocelems}(rt'^*, (ref'^*)^*)$.
8. Return $ea\ ea'^*$.

$$\begin{array}{lcll}
\mathrm{allocelems}(s,\ \epsilon,\ \epsilon) & = & (s,\ \epsilon) & \\
\mathrm{allocelems}(s,\ rt\ rt'^*,\ (ref^*)\ (ref'^*)^*) & = & (s_2,\ ea\ ea'^*) & \text{if } (s_1,\ ea) = \mathrm{allocelem}(s,\ rt,\ ref^*) \\
& & & \wedge\ (s_2,\ ea'^*) = \mathrm{allocelems}(s_2,\ rt'^*,\ (ref'^*)^*)
\end{array}$$

$\mathrm{allocdata}(byte^*)$

1. Let di be $\{\mathsf{data}\ byte^*\}$.
2. Return $s \bigoplus \{.\mathsf{data}\ di\}\ |s.\mathsf{data}|$.

$$\mathrm{allocdata}(s,\ byte^*) \quad = \quad (s[\mathsf{data} = ..di],\ |s.\mathsf{data}|) \quad \text{if } di = \{\mathsf{data}\ byte^*\}$$

$\mathrm{allocdatas}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $byte^*\ (byte'^*)^*$ be ${x_0}^*$.
3. Let da be $\mathrm{allocdata}(byte^*)$.
4. Let da'^* be $\mathrm{allocdatas}((byte'^*)^*)$.
5. Return $da\ da'^*$.

$$
\begin{array}{lcll}
\mathrm{allocdatas}(s, \epsilon) & = & (s, \epsilon) & \\
\mathrm{allocdatas}(s, (byte^*)\ (byte'^*)^*) & = & (s_2, da\ da'^*) & \text{if } (s_1, da) = \mathrm{allocdata}(s, byte^*) \\
 & & & \wedge\ (s_2, da'^*) = \mathrm{allocdatas}(s_1, (byte'^*)^*)
\end{array}
$$

$\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ x_0)$

1. If x_0 is of the case func, then:
   a. Let $\mathsf{func}\ x$ be x_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{func}\ fa^*[x]\}$.
2. If x_0 is of the case global, then:
   a. Let $\mathsf{global}\ x$ be x_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{global}\ ga^*[x]\}$.
3. If x_0 is of the case table, then:
   a. Let $\mathsf{table}\ x$ be x_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{table}\ ta^*[x]\}$.
4. Assert: Due to validation, x_0 is of the case mem.
5. Let $\mathsf{mem}\ x$ be x_0.
6. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{mem}\ ma^*[x]\}$.

$$
\begin{array}{lcl}
\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ (\mathsf{func}\ x)) & = & \{\mathsf{name}\ name, \mathsf{value}\ (\mathsf{func}\ fa^*[x])\} \\
\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ (\mathsf{global}\ x)) & = & \{\mathsf{name}\ name, \mathsf{value}\ (\mathsf{global}\ ga^*[x])\} \\
\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ (\mathsf{table}\ x)) & = & \{\mathsf{name}\ name, \mathsf{value}\ (\mathsf{table}\ ta^*[x])\} \\
\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ (\mathsf{mem}\ x)) & = & \{\mathsf{name}\ name, \mathsf{value}\ (\mathsf{mem}\ ma^*[x])\}
\end{array}
$$

$\mathrm{allocmodule}(module, externval^*, val^*, (ref^*)^*)$

1. Let ${fa_{ex}}^*$ be $\mathrm{funcs}(externval^*)$.
2. Let ${ga_{ex}}^*$ be $\mathrm{globals}(externval^*)$.
3. Let ${ma_{ex}}^*$ be $\mathrm{mems}(externval^*)$.
4. Let ${ta_{ex}}^*$ be $\mathrm{tables}(externval^*)$.
5. Assert: Due to validation, $module$ is of the case module.
6. Let $\mathsf{module}\ import^*\ func^{n_{func}}\ y_0\ y_1\ y_2\ y_3\ y_4\ start^?\ export^*$ be $module$.
7. Let $(\mathsf{data}\ byte^*\ datamode^?)^{n_{data}}$ be y_4.
8. Let $(\mathsf{elem}\ rt\ {expr_2}^*\ elemmode^?)^{n_{elem}}$ be y_3.
9. Let $(\mathsf{memory}\ memtype)^{n_{mem}}$ be y_2.
10. Let $(\mathsf{table}\ tabletype)^{n_{table}}$ be y_1.
11. Let $(\mathsf{global}\ globaltype\ expr_1)^{n_{global}}$ be y_0.
12. Let da^* be $(|s.\mathsf{data}| + i_{data})^{(i_{data}<n_{data})}$.

13. Let ea^* be $(|s.\mathsf{elem}| + i_{elem})^{(i_{elem} < n_{elem})}$.
14. Let ma^* be $(|s.\mathsf{mem}| + i_{mem})^{(i_{mem} < n_{mem})}$.
15. Let ta^* be $(|s.\mathsf{table}| + i_{table})^{(i_{table} < n_{table})}$.
16. Let ga^* be $(|s.\mathsf{global}| + i_{global})^{(i_{global} < n_{global})}$.
17. Let fa^* be $(|s.\mathsf{func}| + i_{func})^{(i_{func} < n_{func})}$.
18. Let xi^* be $(\mathrm{instexport}(fa_{ex}{}^*\ fa^*, ga_{ex}{}^*\ ga^*, ta_{ex}{}^*\ ta^*, ma_{ex}{}^*\ ma^*, export))^*$.
19. Let m be $\{\mathsf{func}\ fa_{ex}{}^*\ fa^*, \mathsf{global}\ ga_{ex}{}^*\ ga^*, \mathsf{table}\ ta_{ex}{}^*\ ta^*, \mathsf{mem}\ ma_{ex}{}^*\ ma^*, \mathsf{elem}\ ea^*, \mathsf{data}\ da^*, \mathsf{export}\ xi^*\}$.
20. Let y_0 be $\mathrm{allocfuncs}(m, func^{n_{func}})$.
21. Assert: Due to validation, y_0 is fa^*.
22. Let y_0 be $\mathrm{allocglobals}(globaltype^{n_{global}}, val^*)$.
23. Assert: Due to validation, y_0 is ga^*.
24. Let y_0 be $\mathrm{alloctables}(tabletype^{n_{table}})$.
25. Assert: Due to validation, y_0 is ta^*.
26. Let y_0 be $\mathrm{allocmems}(memtype^{n_{mem}})$.
27. Assert: Due to validation, y_0 is ma^*.
28. Let y_0 be $\mathrm{allocelems}(rt^{n_{elem}}, (ref^*)^*)$.
29. Assert: Due to validation, y_0 is ea^*.
30. Let y_0 be $\mathrm{allocdatas}((byte^*)^{n_{data}})$.
31. Assert: Due to validation, y_0 is da^*.
32. Return m.

$$\begin{array}{lcll}
\mathrm{allocmodule}(s,\ module,\ externval^*,\ val^*,\ (ref^*)^*) & = & (s_6,\ m) & \text{if } module = \mathsf{module}\ import^*\ func^{n_{func}}\ (\mathsf{global}\ globaltype\ e \\
&&& \wedge\ fa_{ex}^* = \mathrm{funcs}(externval^*) \\
&&& \wedge\ ga_{ex}^* = \mathrm{globals}(externval^*) \\
&&& \wedge\ ta_{ex}^* = \mathrm{tables}(externval^*) \\
&&& \wedge\ ma_{ex}^* = \mathrm{mems}(externval^*) \\
&&& \wedge\ fa^* = |s.\mathsf{func}| + i_{func}^(i_func < n_{func}) \\
&&& \wedge\ ga^* = |s.\mathsf{global}| + i_{global}^(i_global < n_{global}) \\
&&& \wedge\ ta^* = |s.\mathsf{table}| + i_{table}^(i_table < n_{table}) \\
&&& \wedge\ ma^* = |s.\mathsf{mem}| + i_{mem}^(i_mem < n_{mem}) \\
&&& \wedge\ ea^* = |s.\mathsf{elem}| + i_{elem}^(i_elem < n_{elem}) \\
&&& \wedge\ da^* = |s.\mathsf{data}| + i_{data}^(i_data < n_{data}) \\
&&& \wedge\ xi^* = \mathrm{instexport}(fa_{ex}^*\ fa^*,\ ga_{ex}^*\ ga^*,\ ta_{ex}^*\ ta^*,\ ma_{ex}^*\ m \\
&&& \wedge\ m = \{\mathsf{func}\ fa_{ex}^*\ fa^*, \\
&&& \qquad \mathsf{global}\ ga_{ex}^*\ ga^*, \\
&&& \qquad \mathsf{table}\ ta_{ex}^*\ ta^*, \\
&&& \qquad \mathsf{mem}\ ma_{ex}^*\ ma^*, \\
&&& \qquad \mathsf{elem}\ ea^*, \\
&&& \qquad \mathsf{data}\ da^*, \\
&&& \qquad \mathsf{export}\ xi^*\} \\
&&& \wedge\ (s_1,\ fa^*) = \mathrm{allocfuncs}(s,\ m,\ func^{n_{func}}) \\
&&& \wedge\ (s_2,\ ga^*) = \mathrm{allocglobals}(s_1,\ globaltype^{n_{global}},\ val^*) \\
&&& \wedge\ (s_3,\ ta^*) = \mathrm{alloctables}(s_2,\ tabletype^{n_{table}}) \\
&&& \wedge\ (s_4,\ ma^*) = \mathrm{allocmems}(s_3,\ memtype^{n_{mem}}) \\
&&& \wedge\ (s_5,\ ea^*) = \mathrm{allocelems}(s_4,\ rt^{n_{elem}},\ (ref^*)^*) \\
&&& \wedge\ (s_6,\ da^*) = \mathrm{allocdatas}(s_5,\ (byte^*)^{n_{data}})
\end{array}$$

3.4.2 Instantiation

$\mathrm{concat\_instr}({x_0}^*)$

1. If ${x_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $instr^*\ (instr'^*)^*$ be ${x_0}^*$.
3. Return $instr^*\ \mathrm{concat\_instr}((instr'^*)^*)$.

$$\begin{array}{lcl}
\mathrm{concat}_{instr}(\epsilon) & = & \epsilon \\
\mathrm{concat}_{instr}((instr^*)\ (instr'^*)^*) & = & instr^*\ \mathrm{concat}_{instr}((instr'^*)^*)
\end{array}$$

$\mathrm{instantiation}(module, externval^*)$

1. Assert: Due to validation, $module$ is of the case module.
2. Let $\mathsf{module}\ import^*\ func^{n_{func}}\ global^*\ table^*\ mem^*\ elem^*\ data^*\ start^?\ export^*$ be $module$.
3. Let m_{init} be $\{\mathsf{func}\ \mathrm{funcs}(externval^*)\ (|s.\mathsf{func}| + i_{func})^{(i_{func}<n_{func})}, \mathsf{global}\ \mathrm{globals}(externval^*), \mathsf{table}\ \epsilon, \mathsf{mem}\ \epsilon, \mathsf{elem}\ \epsilon, \mathsf{da}$
4. Let n_{data} be the length of $data^*$.
5. Let n_{elem} be the length of $elem^*$.
6. Let $(\mathsf{start}\ x)^?$ be $start^?$.

7. Let $(\mathsf{global}\ globaltype\ instr_1{}^*)^*$ be $global^*$.
8. Let $(\mathsf{elem}\ reftype\ (instr_2{}^*)^*\ elemmode^?)^*$ be $elem^*$.
9. Let f_{init} be $\{\mathsf{local}\ \epsilon, \mathsf{module}\ m_{init}\}$.
10. Let $instr_{data}{}^*$ be $\mathrm{concat\_instr}((\mathrm{rundata}(data^*[j], j))^{(j<n_{data})})$.
11. Let $instr_{elem}{}^*$ be $\mathrm{concat\_instr}((\mathrm{runelem}(elem^*[i], i))^{(i<n_{elem})})$.
12. Enter the activation of f_{init} with label $FRAME$:
  a. Let $(ref^*)^*$ be $((exec\_expr\_const(instr_2{}^*))^*)^*$.
13. Enter the activation of f_{init} with label $FRAME$:
  a. Let val^* be $(exec\_expr\_const(instr_1{}^*))^*$.
14. Let m be $\mathrm{allocmodule}(module, externval^*, val^*, (ref^*)^*)$.
15. Let f be $\{\mathsf{local}\ \epsilon, \mathsf{module}\ m\}$.
16. Enter the activation of f with arity 0 with label $FRAME$:
  a. Execute the sequence $instr_{elem}{}^*$.
  b. Execute the sequence $instr_{data}{}^*$.
  c. If x is defined, then:
    1) Let $x_0{}^?$ be x.
    2) Execute $\mathsf{call}\ x_0$.
17. Return m.

$$
\begin{array}{lcll}
\mathrm{instantiation}(s,\ module,\ externval^*) & = & s'; f; instr^*_{elem}\ instr^*_{data}\ (\mathsf{call}\ x)^? & \text{if } module = \mathsf{module}\ import^*\ func^{n_{func}}\ global^* \\
 & & & \wedge\ m_{init} = \{\mathsf{func}\ \mathrm{funcs}(externval^*)\ |s.\mathsf{func}| + \\
 & & & \qquad \mathsf{global}\ \mathrm{globals}(externval^*), \\
 & & & \qquad \mathsf{table}\ \epsilon, \\
 & & & \qquad \mathsf{mem}\ \epsilon, \\
 & & & \qquad \mathsf{elem}\ \epsilon, \\
 & & & \qquad \mathsf{data}\ \epsilon, \\
 & & & \qquad \mathsf{export}\ \epsilon\} \\
 & & & \wedge\ f_{init} = \{\mathsf{local}\ \epsilon,\ \mathsf{module}\ m_{init}\} \\
 & & & \wedge\ global^* = (\mathsf{global}\ globaltype\ instr^*_1)^* \\
 & & & \wedge\ (s; f_{init}; instr^*_1 \hookrightarrow val)^* \\
 & & & \wedge\ elem^* = (\mathsf{elem}\ reftype\ (instr^*_2)^*\ elemmode^? \\
 & & & \wedge\ (s; f_{init}; instr^*_2 \hookrightarrow ref)^{**} \\
 & & & \wedge\ (s',\ m) = \mathrm{allocmodule}(s,\ module,\ externva \\
 & & & \wedge\ f = \{\mathsf{local}\ \epsilon,\ \mathsf{module}\ m\} \\
 & & & \wedge\ n_{elem} = |elem^*| \\
 & & & \wedge\ instr^*_{elem} = \mathrm{concat}_{instr}(\mathrm{runelem}(elem^*[i], \\
 & & & \wedge\ n_{data} = |data^*| \\
 & & & \wedge\ instr^*_{data} = \mathrm{concat}_{instr}(\mathrm{rundata}(data^*[j], j \\
 & & & \wedge\ start^? = (\mathsf{start}\ x)^?
\end{array}
$$

3.4.3 Invocation

$\mathrm{invocation}(fa, val^n)$

1. Let m be $\{\mathsf{func}\ \epsilon, \mathsf{global}\ \epsilon, \mathsf{table}\ \epsilon, \mathsf{mem}\ \epsilon, \mathsf{elem}\ \epsilon, \mathsf{data}\ \epsilon, \mathsf{export}\ \epsilon\}$.
2. Let f be $\{\mathsf{local}\ \epsilon, \mathsf{module}\ m\}$.
3. Assert: Due to validation, $\mathrm{funcinst}()[fa].\mathsf{code}$ is of the case func.
4. Let $\mathsf{func}\ functype\ valtype^*\ expr$ be $\mathrm{funcinst}()[fa].\mathsf{code}$.
5. Let ${valtype_{param}}^n \rightarrow {valtype_{res}}^k$ be $functype$.
6. Enter the activation of f with arity k with label $FRAME$:
   a. Push val^n to the stack.
   b. Execute $\mathsf{call\_addr}\ fa$.
7. Pop val^k from the stack.
8. Return val^k.

$$
\begin{array}{lcll}
\mathrm{invocation}(s,\ fa,\ val^n) & = & s; f; val^n\ (\mathsf{call}\ fa) & \text{if } m = \{\mathsf{func}\ \epsilon, \\
& & & \qquad\qquad \mathsf{global}\ \epsilon, \\
& & & \qquad\qquad \mathsf{table}\ \epsilon, \\
& & & \qquad\qquad \mathsf{mem}\ \epsilon, \\
& & & \qquad\qquad \mathsf{elem}\ \epsilon, \\
& & & \qquad\qquad \mathsf{data}\ \epsilon, \\
& & & \qquad\qquad \mathsf{export}\ \epsilon\} \\
& & & \wedge\ f = \{\mathsf{local}\ \epsilon,\ \mathsf{module}\ m\} \\
& & & \wedge\ (s; f).\mathsf{func}[fa].\mathsf{code} = \mathsf{func}\ functype\ valtype^*\ expr \\
& & & \wedge\ functype = valtype_{param}^n \rightarrow valtype_{res}^k
\end{array}
$$