WebAssembly

author

Mar 07, 2024

Contents

1 Syntax **1**
- 1.1 Values . . . 1
- 1.2 Types . . . 4
- 1.3 Instructions . . . 6
- 1.4 Modules . . . 9

2 Execution **13**
- 2.1 Conventions . . . 13
- 2.2 Numerics . . . 17
- 2.3 Runtime . . . 18
- 2.4 Instructions . . . 24
- 2.5 Modules . . . 59

CHAPTER 1

Syntax

1.1 Values

1.1.1 Bytes

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

1.1.2 Integers

$$
\begin{array}{llll}
\text{(unsigned integer)} & uN & ::= & 0 \mid \dots \mid 2^N - 1 \\
\text{(signed integer)} & sN & ::= & -2^{N-1} \mid \dots \mid -1 \mid 0 \mid +1 \mid \dots \mid 2^{N-1} - 1 \\
\text{(integer)} & iN & ::= & uN \\
\text{(31-bit integer)} & u31 & ::= & 0 \mid \dots \mid 2^{31} - 1 \\
\text{(32-bit integer)} & u32 & ::= & 0 \mid \dots \mid 2^{32} - 1 \\
\text{(64-bit integer)} & u64 & ::= & 0 \mid \dots \mid 2^{64} - 1 \\
\text{(128-bit integer)} & u128 & ::= & 0 \mid \dots \mid 2^{128} - 1 \\
\text{(33-bit signed integer)} & s33 & ::= & -2^{32} \mid \dots \mid 2^{32} - 1
\end{array}
$$

1.1.3 Floating-Point

$$
\begin{array}{lllll}
\text{(floating-point number)} & fN & ::= & +fNmag \mid -fNmag & \\
\text{(floating-point magnitude)} & fNmag & ::= & (1 + m \cdot 2^{-\mathrm{M}_N}) \cdot 2^n & \text{if } 2 - 2^{\mathrm{E}_N - 1} \leq n \leq 2^{\mathrm{E}_N - 1} - 1 \\
 & & \mid & (0 + m \cdot 2^{-\mathrm{M}_N}) \cdot 2^n & \text{if } 2 - 2^{\mathrm{E}_N - 1} = n \\
 & & \mid & \infty & \\
 & & \mid & \mathsf{nan}(n) & \text{if } 1 \leq n < \mathrm{M}_N \\
\text{(32-bit floating-point)} & f32 & ::= & fN & \\
\text{(64-bit floating-point)} & f64 & ::= & fN &
\end{array}
$$

fNzero()

1. Return pos norm 0 0.

$$+0 \quad = \quad +((1 + 0 \cdot 2^{-\mathrm{M}_N}) \cdot 2^0)$$

$\mathrm{signif}(u_0)$

1. If u_0 is 32, then:
   a. Return 23.
2. Assert: Due to validation, u_0 is 64.
3. Return 52.

$$\begin{array}{rcl} \mathrm{signif}(32) & = & 23 \\ \mathrm{signif}(64) & = & 52 \end{array}$$

$\mathrm{m}(N)$

1. Return $\mathrm{signif}(N)$.

$$\mathrm{M}_N \quad = \quad \mathrm{signif}(N)$$

$\mathrm{expon}(u_0)$

1. If u_0 is 32, then:
   a. Return 8.
2. Assert: Due to validation, u_0 is 64.
3. Return 11.

$$\begin{array}{rcl} \mathrm{expon}(32) & = & 8 \\ \mathrm{expon}(64) & = & 11 \end{array}$$

$\mathrm{e}(N)$

1. Return $\mathrm{expon}(N)$.

$$\mathrm{E}_N \quad = \quad \mathrm{expon}(N)$$

1.1.4 Names

$$\begin{array}{llcl}
\text{(name)} & \mathit{name} & ::= & \mathit{char}^* \\
\text{(character)} & \mathit{char} & ::= & \mathrm{U{+}00} \mid \ldots \mid \mathrm{U{+}D7FF} \mid \mathrm{U{+}E000} \mid \ldots \mid \mathrm{U{+}10FFFF}
\end{array}$$

$\mathrm{utf8}(u_0)$

1. If the length of u_0 is 1, then:
   a. Let c be u_0.
   b. If c is less than 128, then:
      1) Let b be c.
      2) Return b.
   c. If 128 is less than or equal to c and c is less than 2048 and c is greater than or equal to $b_2 - 128$, then:
      1) Let $2^6 \cdot b_1 - 192$ be $c - b_2 - 128$.
      2) Return $b_1\ b_2$.
   d. If 2048 is less than or equal to c and c is less than 55296 or 57344 is less than or equal to c and c is less than 65536 and c is greater than or equal to $b_3 - 128$, then:
      1) Let $2^{12} \cdot b_1 - 224 + 2^6 \cdot b_2 - 128$ be $c - b_3 - 128$.
      2) Return $b_1\ b_2\ b_3$.
   e. If 65536 is less than or equal to c and c is less than 69632 and c is greater than or equal to $b_4 - 128$, then:
      1) Let $2^{18} \cdot b_1 - 240 + 2^{12} \cdot b_2 - 128 + 2^6 \cdot b_3 - 128$ be $c - b_4 - 128$.
      2) Return $b_1\ b_2\ b_3\ b_4$.
2. Let c^* be u_0.
3. Return $\mathrm{concat\_bytes}((\mathrm{utf8}(c))^*)$.

$$\begin{array}{lcll}
\mathrm{utf8}(c) & = & b & \text{if } c < \mathrm{U{+}80} \wedge c = b \\
\mathrm{utf8}(c) & = & b_1\ b_2 & \text{if } \mathrm{U{+}80} \leq c < \mathrm{U{+}0800} \wedge c = 2^6 \cdot (b_1 - \mathtt{0xC0}) + (b_2 - \mathtt{0x80}) \\
\mathrm{utf8}(c) & = & b_1\ b_2\ b_3 & \text{if } (\mathrm{U{+}0800} \leq c < \mathrm{U{+}D800} \vee \mathrm{U{+}E000} \leq c < \mathrm{U{+}10000}) \wedge c = 2^{12} \cdot (b_1 - \mathtt{0xE0}) + 2^6 \cdot (b \\
\mathrm{utf8}(c) & = & b_1\ b_2\ b_3\ b_4 & \text{if } (\mathrm{U{+}10000} \leq c < \mathrm{U{+}11000}) \wedge c = 2^{18} \cdot (b_1 - \mathtt{0xF0}) + 2^{12} \cdot (b_2 - \mathtt{0x80}) + 2^6 \cdot (b_3 - \mathtt{0x8} \\
\mathrm{utf8}(c^*) & = & \mathrm{concat}(\mathrm{utf8}(c)^*) &
\end{array}$$

1.2 Types

1.2.1 Number Types

$$\begin{array}{llll} \text{(number type)} & \mathit{numtype} & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\ & c_{\mathit{numtype}} & ::= & \mathit{nat} \end{array}$$

1.2.2 Vector Types

$$\begin{array}{llll} \text{(vector type)} & \mathit{vectype} & ::= & \mathsf{v128} \\ & c_{\mathit{vectype}} & ::= & \mathit{nat} \end{array}$$

1.2.3 Heap Types

$$\begin{array}{llll} \text{(abstract heap type)} & \mathit{absheaptype} & ::= & \mathsf{any} \mid \mathsf{eq} \mid \mathsf{i31} \mid \mathsf{struct} \mid \mathsf{array} \mid \mathsf{none} \\ & & \mid & \mathsf{func} \mid \mathsf{nofunc} \\ & & \mid & \mathsf{extern} \mid \mathsf{noextern} \\ & & \mid & \dots \\ \text{(abstract heap type)} & \mathit{absheaptype} & ::= & \dots \mid \mathsf{bot} \\ \text{(heap type)} & \mathit{heaptype} & ::= & \mathit{absheaptype} \\ & & \mid & \mathit{typeidx} \\ & & \mid & \dots \\ \text{(heap type)} & \mathit{heaptype} & ::= & \dots \mid \mathit{deftype} \\ & & \mid & \\ & & \mid & \mathsf{rec}~\mathit{nat} \end{array}$$

1.2.4 Reference Types

$$\begin{array}{lll} \mathit{nul} & ::= & \mathsf{null}^? \\ \mathit{reftype} & ::= & \mathsf{ref}~\mathit{nul}~\mathit{heaptype} \end{array}$$

1.2.5 Value Types

$$\begin{array}{lll} \mathit{valtype} & ::= & \mathit{numtype} \mid \mathit{vectype} \mid \mathit{reftype} \mid \dots \\ \mathit{valtype} & ::= & \dots \mid \mathsf{bot} \end{array}$$

1.2.6 Result Types

$$\begin{array}{lll} \mathit{resulttype} & ::= & \mathit{valtype}^* \end{array}$$

1.2.7 Function Types

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

1.2.8 Aggregate Types

$$\begin{array}{llll}
\text{(packed type)} & \mathit{packedtype} & ::= & \mathsf{i8} \mid \mathsf{i16} \\
 & c_{\mathit{packedtype}} & ::= & \mathit{nat} \\
\text{(storage type)} & \mathit{storagetype} & ::= & \mathit{valtype} \mid \mathit{packedtype} \\
\text{(field type)} & \mathit{fieldtype} & ::= & \mathit{mut}~\mathit{storagetype}
\end{array}$$

1.2.9 Composite Types

$$\begin{array}{lll}
\mathit{comptype} & ::= & \mathsf{struct}~\mathit{fieldtype}^* \\
 & | & \mathsf{array}~\mathit{fieldtype} \\
 & | & \mathsf{func}~\mathit{functype}
\end{array}$$

1.2.10 Recursive Types

$$\begin{array}{llll}
\text{(recursive type)} & \mathit{rectype} & ::= & \mathsf{rec}~\mathit{subtype}^* \\
\text{(sub type)} & \mathit{subtype} & ::= & \mathsf{sub}~\mathit{fin}~\mathit{typeidx}^*~\mathit{comptype} \mid \ldots \\
\text{(sub type)} & \mathit{subtype} & ::= & \ldots \\
 & & | & \mathsf{sub}~\mathit{fin}~\mathit{heaptype}^*~\mathit{comptype} \\
 & \mathit{fin} & ::= & \mathsf{final}^?
\end{array}$$

1.2.11 Limits

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

1.2.12 Memory Types

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

1.2.13 Table Types

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

1.2.14 Global Types

$$\begin{array}{llll} \text{(global type)} & \mathit{globaltype} & ::= & \mathit{mut}~\mathit{valtype} \\ & \mathit{mut} & ::= & \mathsf{mut}^? \end{array}$$

1.2.15 External Types

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

1.3 Instructions

1.3.1 Numeric Instructions

$$\begin{array}{llll} & \mathsf{i}\mathit{n} & ::= & \mathsf{i32} \mid \mathsf{i64} \\ & \mathsf{f}\mathit{n} & ::= & \mathsf{f32} \mid \mathsf{f64} \\ \text{(signedness)} & \mathit{sx} & ::= & \mathsf{u} \mid \mathsf{s} \\ \text{(instruction)} & \mathit{instr} & ::= & \dots \\ & & | & \mathit{numtype}.\mathsf{const}~c_{\mathit{numtype}} \\ & & | & \mathit{numtype}.\mathit{unop}_{\mathit{numtype}} \\ & & | & \mathit{numtype}.\mathit{binop}_{\mathit{numtype}} \\ & & | & \mathit{numtype}.\mathit{testop}_{\mathit{numtype}} \\ & & | & \mathit{numtype}.\mathit{relop}_{\mathit{numtype}} \\ & & | & \mathit{numtype}.\mathsf{extend}\mathit{n} \\ & & | & \mathit{numtype}.\mathit{cvtop}\_\mathit{numtype}\_\mathit{sx}^? \\ & & | & \dots \\ & \mathit{unop}_{\mathsf{ixx}} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\ & \mathit{unop}_{\mathsf{fxx}} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\ & \mathit{binop}_{\mathsf{ixx}} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}\_\mathit{sx} \mid \mathsf{rem}\_\mathit{sx} \\ & & | & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr}\_\mathit{sx} \mid \mathsf{rotl} \mid \mathsf{rotr} \\ & \mathit{binop}_{\mathsf{fxx}} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \\ & \mathit{testop}_{\mathsf{ixx}} & ::= & \mathsf{eqz} \\ & \mathit{testop}_{\mathsf{fxx}} & ::= & \\ & \mathit{relop}_{\mathsf{ixx}} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt}\_\mathit{sx} \mid \mathsf{gt}\_\mathit{sx} \mid \mathsf{le}\_\mathit{sx} \mid \mathsf{ge}\_\mathit{sx} \\ & \mathit{refop}_{\mathsf{fxx}} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} \end{array}$$

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

$$\begin{array}{lll} \mathit{unop}_{\mathit{numtype}} & ::= & \mathit{unop}_{\mathsf{ixx}} \mid \mathit{unop}_{\mathsf{fxx}} \\ \mathit{binop}_{\mathit{numtype}} & ::= & \mathit{binop}_{\mathsf{ixx}} \mid \mathit{binop}_{\mathsf{fxx}} \\ \mathit{testop}_{\mathit{numtype}} & ::= & \mathit{testop}_{\mathsf{ixx}} \mid \mathit{testop}_{\mathsf{fxx}} \\ \mathit{relop}_{\mathit{numtype}} & ::= & \mathit{relop}_{\mathsf{ixx}} \mid \mathit{refop}_{\mathsf{fxx}} \\ \mathit{cvtop} & ::= & \mathsf{convert} \mid \mathsf{reinterpret} \mid \mathsf{convert\_sat} \end{array}$$

1.3.2 Reference Instructions

$$
\begin{array}{lcl}
\mathit{instr} & ::= & \dots \\
 & | & \mathsf{ref.null}~\mathit{heaptype} \\
 & | & \mathsf{ref.i31} \\
 & | & \mathsf{ref.func}~\mathit{funcidx} \\
 & | & \mathsf{ref.is\_null} \\
 & | & \mathsf{ref.as\_non\_null} \\
 & | & \mathsf{ref.eq} \\
 & | & \mathsf{ref.test}~\mathit{reftype} \\
 & | & \mathsf{ref.cast}~\mathit{reftype} \\
 & | & \dots
\end{array}
$$

1.3.3 Aggregate Instructions

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

1.3.4 Variable Instructions

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

1.3.5 Table Instructions

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

1.3.6 Memory Instructions

$$
\begin{array}{llrl}
\text{(instruction)} & \mathit{instr} & ::= & \ldots \\
& & | & \mathsf{memory.size}\ \mathit{memidx} \\
& & | & \mathsf{memory.grow}\ \mathit{memidx} \\
& & | & \mathsf{memory.fill}\ \mathit{memidx} \\
& & | & \mathsf{memory.copy}\ \mathit{memidx}\ \mathit{memidx} \\
& & | & \mathsf{memory.init}\ \mathit{memidx}\ \mathit{dataidx} \\
& & | & \mathsf{data.drop}\ \mathit{dataidx} \\
& & | & \mathit{numtype}.\mathsf{load}(\mathit{n\_sx})^?\ \mathit{memidx}\ \mathit{memop} \\
& & | & \mathit{numtype}.\mathsf{store}\mathit{n}^?\ \mathit{memidx}\ \mathit{memop} \\
\text{(memory operator)} & \mathit{memop} & ::= & \{\ \mathsf{align}\ \mathit{u32},\ \mathsf{offset}\ \mathit{u32}\ \}
\end{array}
$$

memop0()

1. Return $\{\mathsf{align}\ 0, \mathsf{offset}\ 0\}$.

$$
= \quad \{\mathsf{align}\ 0,\ \mathsf{offset}\ 0\}
$$

1.3.7 Control Instructions

$$
\begin{array}{llll}
\text{(block type)} & \mathit{blocktype} & ::= & \mathit{valtype}^? \\
& & | & \mathit{funcidx} \\
\text{(instruction)} & \mathit{instr} & ::= & \mathsf{unreachable} \\
& & | & \mathsf{nop} \\
& & | & \mathsf{drop} \\
& & | & \mathsf{select}\ (\mathit{valtype}^*)^? \\
& & | & \mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^* \\
& & | & \mathsf{loop}\ \mathit{blocktype}\ \mathit{instr}^* \\
& & | & \mathsf{if}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^* \\
& & | & \mathsf{br}\ \mathit{labelidx} \\
& & | & \mathsf{br\_if}\ \mathit{labelidx} \\
& & | & \mathsf{br\_table}\ \mathit{labelidx}^*\ \mathit{labelidx} \\
& & | & \mathsf{br\_on\_null}\ \mathit{labelidx} \\
& & | & \mathsf{br\_on\_non\_null}\ \mathit{labelidx} \\
& & | & \mathsf{br\_on\_cast}\ \mathit{labelidx}\ \mathit{reftype}\ \mathit{reftype} \\
& & | & \mathsf{br\_on\_cast\_fail}\ \mathit{labelidx}\ \mathit{reftype}\ \mathit{reftype} \\
& & | & \mathsf{call}\ \mathit{funcidx} \\
& & | & \mathsf{call\_ref}\ \mathit{typeidx}^? \\
& & | & \mathsf{call\_indirect}\ \mathit{tableidx}\ \mathit{typeidx} \\
& & | & \mathsf{return} \\
& & | & \mathsf{return\_call}\ \mathit{funcidx} \\
& & | & \mathsf{return\_call\_ref}\ \mathit{typeidx}^? \\
& & | & \mathsf{return\_call\_indirect}\ \mathit{tableidx}\ \mathit{typeidx} \\
& & | & \dots
\end{array}
$$

1.3.8 Expressions

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

1.4 Modules

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

1.4.1 Indices

$$
\begin{array}{llll}
\text{(index)} & \mathit{idx} & ::= & \mathit{u32} \\
\text{(type index)} & \mathit{typeidx} & ::= & \mathit{idx} \\
\text{(function index)} & \mathit{funcidx} & ::= & \mathit{idx} \\
\text{(table index)} & \mathit{tableidx} & ::= & \mathit{idx} \\
\text{(memory index)} & \mathit{memidx} & ::= & \mathit{idx} \\
\text{(global index)} & \mathit{globalidx} & ::= & \mathit{idx} \\
\text{(elem index)} & \mathit{elemidx} & ::= & \mathit{idx} \\
\text{(data index)} & \mathit{dataidx} & ::= & \mathit{idx} \\
\text{(local index)} & \mathit{localidx} & ::= & \mathit{idx} \\
\text{(label index)} & \mathit{labelidx} & ::= & \mathit{idx}
\end{array}
$$

1.4.2 Types

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

1.4.3 Functions

$$\begin{array}{llll}
\text{(function)} & \mathit{func} & ::= & \mathsf{func}\ \mathit{typeidx}\ \mathit{local}^*\ \mathit{expr} \\
\text{(local)} & \mathit{local} & ::= & \mathsf{local}\ \mathit{valtype}
\end{array}$$

1.4.4 Tables

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

1.4.5 Memories

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

1.4.6 Globals

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

1.4.7 Element Segments

$$\begin{array}{llll}
\text{(table segment)} & \mathit{elem} & ::= & \mathsf{elem}\ \mathit{reftype}\ \mathit{expr}^*\ \mathit{elemmode} \\
 & \mathit{elemmode} & ::= & \mathsf{active}\ \mathit{tableidx}\ \mathit{expr} \mid \mathsf{passive} \mid \mathsf{declare}
\end{array}$$

1.4.8 Data Segments

$$\begin{array}{llll}
\text{(memory segment)} & \mathit{data} & ::= & \mathsf{data}\ \mathit{byte}^*\ \mathit{datamode} \\
 & \mathit{datamode} & ::= & \mathsf{active}\ \mathit{memidx}\ \mathit{expr} \mid \mathsf{passive}
\end{array}$$

1.4.9 Start Function

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

1.4.10 Exports

$$\begin{array}{llll}
\text{(export)} & \mathit{export} & ::= & \mathsf{export}\ \mathit{name}\ \mathit{externidx} \\
\text{(external index)} & \mathit{externidx} & ::= & \mathsf{func}\ \mathit{funcidx} \mid \mathsf{global}\ \mathit{globalidx} \mid \mathsf{table}\ \mathit{tableidx} \mid \mathsf{mem}\ \mathit{memidx}
\end{array}$$

1.4.11 Imports

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

CHAPTER 2

Execution

2.1 Conventions

2.1.1 General Constants

$\mathrm{Ki}()$

1. Return 1024.

$$\mathrm{Ki} \quad = \quad 1024$$

2.1.2 Formal Notation

$$\begin{array}{llll}[\textsc{E-pure}]\, z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \\ [\textsc{E-read}]\, z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } z; \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \end{array}$$

2.1.3 Size

$\mathrm{size}(u_0)$

1. If u_0 is $I32$, then:
   a. Return 32.
2. If u_0 is $I64$, then:
   a. Return 64.
3. If u_0 is $F32$, then:
   a. Return 32.
4. If u_0 is $F64$, then:

a. Return 64.

5. If u_0 is $V128$, then:

a. Return 128.

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

$\mathrm{packedsize}(u_0)$

1. If u_0 is $\mathsf{i8}$, then:

a. Return 8.

2. Assert: Due to validation, u_0 is $\mathsf{i16}$.

3. Return 16.

$$
\begin{array}{lcl}
|\mathsf{i8}| & = & 8 \\
|\mathsf{i16}| & = & 16
\end{array}
$$

$\mathrm{storagesize}(u_0)$

1. If the type of u_0 is valtype, then:

a. Let $valtype$ be u_0.

b. Return $\mathrm{size}(valtype)$.

2. Assert: Due to validation, the type of u_0 is packedtype.

3. Let $packedtype$ be u_0.

4. Return $\mathrm{packedsize}(packedtype)$.

$$
\begin{array}{lcl}
|valtype| & = & |valtype| \\
|packedtype| & = & |packedtype|
\end{array}
$$

2.1.4 Projections

$\mathrm{funcsxt}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ {et}^*$ be ${u_0}^*$.
3. If y_0 is of the case func, then:
   a. Let $\mathsf{func}\ dt$ be y_0.
   b. Return $dt\ \mathrm{funcsxt}({et}^*)$.
4. Let $externtype\ {et}^*$ be ${u_0}^*$.
5. Return $\mathrm{funcsxt}({et}^*)$.

$$\begin{array}{lcll}
\mathrm{funcs}(\epsilon) & = & \epsilon & \\
\mathrm{funcs}((\mathsf{func}\ dt)\ {et}^*) & = & dt\ \mathrm{funcs}({et}^*) & \\
\mathrm{funcs}(externtype\ {et}^*) & = & \mathrm{funcs}({et}^*) & \text{otherwise}
\end{array}$$

$\mathrm{globalsxt}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ {et}^*$ be ${u_0}^*$.
3. If y_0 is of the case global, then:
   a. Let $\mathsf{global}\ gt$ be y_0.
   b. Return $gt\ \mathrm{globalsxt}({et}^*)$.
4. Let $externtype\ {et}^*$ be ${u_0}^*$.
5. Return $\mathrm{globalsxt}({et}^*)$.

$$\begin{array}{lcll}
\mathrm{globals}(\epsilon) & = & \epsilon & \\
\mathrm{globals}((\mathsf{global}\ gt)\ {et}^*) & = & gt\ \mathrm{globals}({et}^*) & \\
\mathrm{globals}(externtype\ {et}^*) & = & \mathrm{globals}({et}^*) & \text{otherwise}
\end{array}$$

$\mathrm{tablesxt}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ {et}^*$ be ${u_0}^*$.
3. If y_0 is of the case table, then:
   a. Let $\mathsf{table}\ tt$ be y_0.
   b. Return $tt\ \mathrm{tablesxt}({et}^*)$.

4. Let $externtype\ et^*$ be $u_0{}^*$.
5. Return $\mathrm{tablesxt}(et^*)$.

$$\begin{array}{lcll} \mathrm{tables}(\epsilon) & = & \epsilon & \\ \mathrm{tables}((\mathsf{table}\ tt)\ et^*) & = & tt\ \mathrm{tables}(et^*) & \\ \mathrm{tables}(externtype\ et^*) & = & \mathrm{tables}(et^*) & \text{otherwise} \end{array}$$

$\mathrm{memsxt}(u_0{}^*)$

1. If $u_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ et^*$ be $u_0{}^*$.
3. If y_0 is of the case mem, then:
   a. Let $\mathsf{mem}\ mt$ be y_0.
   b. Return $mt\ \mathrm{memsxt}(et^*)$.
4. Let $externtype\ et^*$ be $u_0{}^*$.
5. Return $\mathrm{memsxt}(et^*)$.

$$\begin{array}{lcll} \mathrm{mems}(\epsilon) & = & \epsilon & \\ \mathrm{mems}((\mathsf{mem}\ mt)\ et^*) & = & mt\ \mathrm{mems}(et^*) & \\ \mathrm{mems}(externtype\ et^*) & = & \mathrm{mems}(et^*) & \text{otherwise} \end{array}$$

2.1.5 Packed Fields

$\mathrm{packval}(u_0, u_1)$

1. If the type of u_0 is valtype, then:
   a. Let val be u_1.
   b. Return val.
2. Assert: Due to validation, u_1 is of the case const.
3. Let $y_0.\mathsf{const}\ i$ be u_1.
4. Assert: Due to validation, y_0 is $\mathsf{i32}$.
5. Assert: Due to validation, the type of u_0 is packedtype.
6. Let pt be u_0.
7. Return $\mathsf{pack}\ pt\ \mathrm{wrap}(32, \mathrm{packedsize}(pt), i)$.

$$\begin{array}{lcl} \mathrm{pack}_t(val) & = & val \\ \mathrm{pack}_{pt}(\mathsf{i32.const}\ i) & = & pt.\mathsf{pack}\ \mathrm{wrap}_{32,|pt|}(i) \end{array}$$

$\mathrm{unpackval}(u_0, {u_1}^?, u_2)$

1. If ${u_1}^?$ is not defined, then:
   a. Assert: Due to validation, the type of u_0 is valtype.
   b. Assert: Due to validation, the type of u_2 is val.
   c. Let val be u_2.
   d. Return val.
2. Else:
   a. Let $sx^?$ be ${u_1}^?$.
   b. Assert: Due to validation, u_2 is of the case pack.
   c. Let pack pt i be u_2.
   d. Assert: Due to validation, u_0 is pt.
   e. Return i32.const $\mathrm{ext}(\mathrm{packedsize}(pt), 32, sx, i)$.

$$\begin{array}{lcl} \mathrm{unpack}_t^{\epsilon}(val) & = & val \\ \mathrm{unpack}_{pt}^{sx}(pt.\mathsf{pack}\ i) & = & \mathsf{i32.const}\ \mathrm{ext}_{|pt|,32}^{sx}(i) \end{array}$$

$\mathrm{sxfield}(u_0)$

1. If the type of u_0 is valtype, then:
   a. Return ϵ.
2. Assert: Due to validation, the type of u_0 is packedtype.
3. Return $\mathsf{s}^?$.

$$\begin{array}{lcl} \mathrm{sx}(valtype) & = & \epsilon \\ \mathrm{sx}(packedtype) & = & \mathsf{s} \end{array}$$

2.2 Numerics

2.2.1 Sign Interpretation

$\mathrm{signed}(N, i)$

1. If 0 is less than or equal to 2^{N-1}, then:
   a. Return i.
2. Assert: Due to validation, 2^{N-1} is less than or equal to i.
3. Assert: Due to validation, i is less than 2^N.
4. Return $i - 2^N$.

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

$\mathrm{invsigned}(N, i)$

1. Let j be $inverse_{of_signed}(N, i)$.
2. Return j.

$$\mathrm{signed}^{-1}{}_N\, i \quad = \quad j \quad \text{if } \mathrm{signed}_N(j) = i$$

2.3 Runtime

2.3.1 Values

$$\begin{array}{llcl} \text{(number)} & num & ::= & numtype.\mathsf{const}\ c_{numtype} \\ \text{(address reference)} & addrref & ::= & \mathsf{ref.i31}\ u31 \\ & & | & \mathsf{ref.struct}\ structaddr \\ & & | & \mathsf{ref.array}\ arrayaddr \\ & & | & \mathsf{ref.func}\ funcaddr \\ & & | & \mathsf{ref.host}\ hostaddr \\ & & | & \mathsf{ref.extern}\ addrref \\ \text{(reference)} & ref & ::= & addrref \mid \mathsf{ref.null}\ heaptype \\ & & | & \\ \text{(value)} & val & ::= & num \mid ref \end{array}$$

$\mathrm{default}(u_0)$

1. If u_0 is $I32$, then:
   a. Return i32.const $0^?$.
2. If u_0 is $I64$, then:
   a. Return i64.const $0^?$.
3. If u_0 is $F32$, then:
   a. Return f32.const $0^?$.
4. If u_0 is $F64$, then:
   a. Return f64.const $0^?$.
5. Assert: Due to validation, u_0 is of the case REF.
6. Let $REF\ y_0\ ht$ be u_0.
7. If y_0 is null $\epsilon^?$, then:
   a. Return ref.null $ht^?$.
8. Assert: Due to validation, y_0 is null ϵ.
9. Return ϵ.

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

2.3.2 Results

$$
\mathit{result} \ ::= \ \mathit{val}^* ~|~ \mathsf{trap}
$$

2.3.3 Store

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

2.3.4 Addresses

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

2.3.5 Module Instances

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

2.3.6 Function Instances

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

2.3.7 Table Instances

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

2.3.8 Memory Instances

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

2.3.9 Global Instances

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

2.3.10 Element Instances

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

2.3.11 Data Instances

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

2.3.12 Export Instances

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

2.3.13 External Values

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

$\mathrm{funcsxv}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be ${u_0}^*$.
3. If y_0 is of the case func, then:
   a. Let $\mathsf{func}\ fa$ be y_0.
   b. Return $fa\ \mathrm{funcsxv}(xv^*)$.

4. Let $external val\ xv^*$ be $u_0{}^*$.
5. Return $\mathrm{funcsxv}(xv^*)$.

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

$\mathrm{tablesxv}(u_0{}^*)$

1. If $u_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be $u_0{}^*$.
3. If y_0 is of the case table, then:
   a. Let table ta be y_0.
   b. Return $ta\ \mathrm{tablesxv}(xv^*)$.
4. Let $externval\ xv^*$ be $u_0{}^*$.
5. Return $\mathrm{tablesxv}(xv^*)$.

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

$\mathrm{memsxv}(u_0{}^*)$

1. If $u_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be $u_0{}^*$.
3. If y_0 is of the case mem, then:
   a. Let mem ma be y_0.
   b. Return $ma\ \mathrm{memsxv}(xv^*)$.
4. Let $externval\ xv^*$ be $u_0{}^*$.
5. Return $\mathrm{memsxv}(xv^*)$.

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

$\mathrm{globalsxv}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be ${u_0}^*$.
3. If y_0 is of the case global, then:
   a. Let global ga be y_0.
   b. Return $ga\ \mathrm{globalsxv}(xv^*)$.
4. Let $externval\ xv^*$ be ${u_0}^*$.
5. Return $\mathrm{globalsxv}(xv^*)$.

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

2.3.14 Aggregate Instances

$$\begin{array}{llll} \text{(structure instance)} & structinst & ::= & \{\ \mathsf{type}\ deftype, \\ & & & \ \ \mathsf{field}\ fieldval^*\ \} \\ \text{(array instance)} & arrayinst & ::= & \{\ \mathsf{type}\ deftype, \\ & & & \ \ \mathsf{field}\ fieldval^*\ \} \\ \text{(field value)} & fieldval & ::= & val \mid packedval \\ \text{(packed value)} & packedval & ::= & packedtype.\mathsf{pack}\ c_{packedtype} \end{array}$$

2.3.15 Stack

Activation Frames

$$\begin{array}{lll} frame & ::= & \{\ \mathsf{local}\ (val^?)^*, \\ & & \ \ \mathsf{module}\ moduleinst\ \} \end{array}$$

2.3.16 Administrative Instructions

$$\begin{array}{lll} instr & ::= & instr \\ & | & addrref \\ & | & \mathsf{label}_n\{instr^*\}\ instr^* \\ & | & \mathsf{frame}_n\{frame\}\ instr^* \\ & | & \mathsf{trap} \end{array}$$

2.3.17 Configurations

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

2.3.18 Evaluation Contexts

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

2.3.19 Typing

store()

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

frame()

1. Let f be the current frame.
2. Return f.

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

$$\frac{}{s \vdash \mathsf{ref.null}~\mathit{ht} : (\mathsf{ref~null}~\mathit{ht})}~[\textsc{Ref\_ok-null}]$$

$$\frac{}{s \vdash \mathsf{ref.i31}~i : (\mathsf{ref}~\epsilon~\mathsf{i31})}~[\textsc{Ref\_ok-i31}]$$

$$\frac{s.\mathsf{struct}[a].\mathsf{type} = \mathit{dt}}{s \vdash \mathsf{ref.struct}~a : (\mathsf{ref}~\epsilon~\mathit{dt})}~[\textsc{Ref\_ok-struct}]$$

$$\frac{s.\mathsf{array}[a].\mathsf{type} = \mathit{dt}}{s \vdash \mathsf{ref.array}~a : (\mathsf{ref}~\epsilon~\mathit{dt})}~[\textsc{Ref\_ok-array}]$$

$$\frac{s.\mathsf{func}[a].\mathsf{type} = \mathit{dt}}{s \vdash \mathsf{ref.func}~a : (\mathsf{ref}~\epsilon~\mathit{dt})}~[\textsc{Ref\_ok-func}]$$

$$\frac{}{s \vdash \mathsf{ref.host}~a : (\mathsf{ref}~\epsilon~\mathsf{any})}~[\textsc{Ref\_ok-host}]$$

$$\frac{}{s \vdash \mathsf{ref.extern}~\mathit{addrref} : (\mathsf{ref}~\epsilon~\mathsf{extern})}~[\textsc{Ref\_ok-extern}]$$

2.4 Instructions

2.4.1 Numeric Instructions

unop nt $unop$

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. 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.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.

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

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 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 cvtop$(cvtop, nt_1, nt_2, sx^?, c_1)$ is 1, then:
   a. Let c be cvtop$(cvtop, nt_1, nt_2, sx^?, c_1)$.
   b. Push nt_2.const c to the stack.
4. If cvtop$(cvtop, nt_1, nt_2, sx^?, c_1)$ is ϵ, then:
   a. Trap.

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

2.4.2 Reference Instructions

ref.func x

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

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

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 of the case ref.null, then:
   a. Push i32.const 1 to the stack.
4. Else:
   a. Push i32.const 0 to the stack.

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

ref.as_non_null

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. If ref is of the case ref.null, then:
   a. Trap.
4. Push ref to the stack.

$$
\begin{array}{llcll}
[\textsc{E-ref.as\_non\_null-null}] & ref\ \mathsf{ref.as\_non\_null} & \hookrightarrow & \mathsf{trap} & \text{if } ref = (\mathsf{ref.null}\ ht) \\
[\textsc{E-ref.as\_non\_null-addr}] & ref\ \mathsf{ref.as\_non\_null} & \hookrightarrow & ref & \text{otherwise}
\end{array}
$$

ref.eq

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref_2 from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop ref_1 from the stack.
5. If ref_1 is of the case ref.null and ref_2 is of the case ref.null, then:
   a. Push i32.const 1 to the stack.
6. Else if ref_1 is ref_2, then:
   a. Push i32.const 1 to the stack.
7. Else:
   a. Push i32.const 0 to the stack.

$$
\begin{array}{llcll}
[\textsc{E-ref.eq-null}] & ref_1\ ref_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } ref_1 = \mathsf{ref.null}\ ht_1 \wedge ref_2 = \mathsf{ref.null}\ ht_2 \\
[\textsc{E-ref.eq-true}] & ref_1\ ref_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{otherwise, if } ref_1 = ref_2 \\
[\textsc{E-ref.eq-false}] & ref_1\ ref_2\ \mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}
$$

ref.test rt

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. Let rt' be $ref_{type_of}(ref)$.
4. If rt' matches inst_reftype(moduleinst(), rt), then:
   a. Push i32.const 1 to the stack.
5. Else:
   a. Push i32.const 0 to the stack.

$$\begin{array}{llcll}
[\textsc{E-ref.test-true}] & z; \mathit{ref}\ (\mathsf{ref.test}\ rt) & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } z.\mathsf{store} \vdash \mathit{ref} : rt' \\
 & & & & \wedge \{\} \vdash rt' \leq \mathrm{inst}_{z.\mathsf{module}}(rt) \\
[\textsc{E-ref.test-false}] & z; \mathit{ref}\ (\mathsf{ref.test}\ rt) & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}$$

ref.cast rt

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. Let rt' be $ref_{type_of}(ref)$.
4. If not rt' matches inst_reftype(moduleinst(), rt), then:
   a. Trap.
5. Push ref to the stack.

$$\begin{array}{lcll}
z; \mathit{ref}\ (\mathsf{ref.cast}\ rt) & \hookrightarrow & \mathit{ref} & \text{if } z.\mathsf{store} \vdash \mathit{ref} : rt' \\
 & & & \wedge \{\} \vdash rt' \leq \mathrm{inst}_{z.\mathsf{module}}(rt) \\
z; \mathit{ref}\ (\mathsf{ref.cast}\ rt) & \hookrightarrow & \mathsf{trap} & \text{otherwise}
\end{array}$$

ref.i31

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const i from the stack.
3. Push ref.i31_num $\mathrm{wrap}(32, 31, i)$ to the stack.

$$[\textsc{E-ref.i31}] (\mathsf{i32.const}\ i)\ \mathsf{ref.i31} \quad \hookrightarrow \quad (\mathsf{ref.i31}\ \mathrm{wrap}_{32,31}(i))$$

i31.get sx

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop u_0 from the stack.
3. If u_0 is of the case ref.null, then:
   a. Trap.
4. If u_0 is of the case ref.i31_num, then:
   a. Let ref.i31_num i be u_0.
   b. Push i32.const $\mathrm{ext}(31, 32, sx, i)$ to the stack.

$$
\begin{array}{lcl}
[\textsc{E-i31.get-null}] (\mathsf{ref.null}\ ht)\ (\mathsf{i31.get\_}sx) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-i31.get-num}]\ (\mathsf{ref.i31}\ i)\ (\mathsf{i31.get\_}sx) & \hookrightarrow & (\mathsf{i32.const}\ \mathrm{ext}^{sx}_{31,32}(i))
\end{array}
$$

$\mathrm{ext\_structinst}(si^*)$

1. Let f be the current frame.
2. Return $s \bigoplus \{.\mathsf{struct}\ si^*\}\ f$.

$$
(s; f)[\mathsf{struct} = ..si^*] \quad = \quad s[\mathsf{struct} = ..si^*]; f
$$

struct.new x

1. Assert: Due to validation, $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case struct.
2. Let struct y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.
3. Let $(mut\ zt)^n$ be y_0.
4. Assert: Due to validation, there are at least n values on the top of the stack.
5. Pop val^n from the stack.
6. Let si be $\{\mathsf{type}\ \mathrm{type}(x), \mathsf{field}\ (\mathrm{packval}(zt, val))^n\}$.
7. Push ref.struct_addr $|\mathrm{structinst}()|$ to the stack.
8. Perform $\mathrm{ext\_structinst}(si)$.

$$
\begin{array}{lcll}
[\textsc{E-struct.new}] z; val^n\ (\mathsf{struct.new}\ x) & \hookrightarrow & z[\mathsf{struct} = ..si]; (\mathsf{ref.struct}\ |z.\mathsf{struct}|) & \text{if } z.\mathsf{type}\,[x] \approx \mathsf{struct}\ (mut\ zt)^n \\
 & & & \wedge\ si = \{\mathsf{type}\ z.\mathsf{type}\,[x],\ \mathsf{field}\ (\mathrm{pack}_{zt}(val))^n\}
\end{array}
$$

struct.new_default x

1. Assert: Due to validation, $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case struct.
2. Let struct y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.
3. Let $(mut\ zt)^*$ be y_0.
4. Assert: Due to validation, the length of mut^* is the length of zt^*.
5. Assert: Due to validation, $\mathrm{default}(\mathrm{unpacktype}(zt))$ is defined.
6. Let $(val^?)^*$ be $(\mathrm{default}(\mathrm{unpacktype}(zt)))^*$.
7. Assert: Due to validation, the length of val^* is the length of zt^*.
8. Push val^* to the stack.
9. Execute struct.new x.

$$[\text{E-STRUCT.NEW\_DEFAULT}]\, z; (\mathsf{struct.new\_default}\ x) \quad \hookrightarrow \quad val^*\ (\mathsf{struct.new}\ x) \quad \text{if } z.\mathsf{type}\,[x] \approx \mathsf{struct}\ (mut\ zt)^* \\ \wedge\ ((\mathrm{default}\ \mathrm{unpack}(zt) = val))^*$$

struct.get $sx^?\ x\ i$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop u_0 from the stack.
3. If u_0 is of the case ref.null, then:
   a. Trap.
4. If u_0 is of the case ref.struct_addr, then:
   a. Let ref.struct_addr a be u_0.
   b. If a is less than the length of $\mathrm{structinst}()$, then:
      1) Let si be $\mathrm{structinst}()[a]$.
      2) If i is less than the length of si.field and $\mathrm{expanddt}(si.\mathsf{type})$ is of the case struct, then:
         a) Let struct y_0 be $\mathrm{expanddt}(si.\mathsf{type})$.
         b) Let $(mut\ zt)^*$ be y_0.
         c) If the length of mut^* is the length of zt^* and i is less than the length of zt^*, then:
            1. Push $\mathrm{unpackval}(zt^*[i], sx^?, si.\mathsf{field}[i])$ to the stack.

$$[\text{E-STRUCT.GET-NULL}]\quad z; (\mathsf{ref.null}\ ht)\ (\mathsf{struct.get\_}sx^?\ x\ i) \quad \hookrightarrow \quad \mathsf{trap}$$

$$[\text{E-STRUCT.GET-STRUCT}]\, z; (\mathsf{ref.struct}\ a)\ (\mathsf{struct.get\_}sx^?\ x\ i) \quad \hookrightarrow \quad \mathrm{unpack}^{sx^?}_{zt^*[i]}(si.\mathsf{field}[i]) \quad \text{if } z.\mathsf{struct}[a] = si \\ \wedge\ si.\mathsf{type} \approx \mathsf{struct}\ (mut\ zt)^*$$

struct.set x i

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop u_0 from the stack.
5. If u_0 is of the case ref.null, then:
   a. Trap.
6. If u_0 is of the case ref.struct_addr, then:
   a. Let ref.struct_addr a be u_0.
   b. If a is less than the length of $\mathrm{structinst}()$ and $\mathrm{expanddt}(\mathrm{structinst}()[a].\mathsf{type})$ is of the case struct, then:
      1) Let struct y_0 be $\mathrm{expanddt}(\mathrm{structinst}()[a].\mathsf{type})$.
      2) Let $(mut\ zt)^*$ be y_0.
      3) If the length of mut^* is the length of zt^* and i is less than the length of zt^*, then:
         a) Let fv be $\mathrm{packval}(zt^*[i], val)$.
         b) Perform $\mathrm{with\_struct}(a, i, fv)$.

$$
\begin{array}{llcll}
[\textsc{E-struct.set-null}] & z; (\mathsf{ref.null}\ ht)\ val\ (\mathsf{struct.set}\ x\ i) & \hookrightarrow & z; \mathsf{trap} & \\
[\textsc{E-struct.set-struct}] & z; (\mathsf{ref.struct}\ a)\ val\ (\mathsf{struct.set}\ x\ i) & \hookrightarrow & \mathrm{with}_{struct}(z,\ a,\ i,\ fv); \epsilon & \text{if } z.\mathsf{struct}[a].\mathsf{type} \approx \mathsf{struct}\ (mut\ zt)^* \\
 & & & & \wedge\ fv = \mathrm{pack}_{zt^*[i]}(val)
\end{array}
$$

array.new x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop 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. Push val^n to the stack.
6. Execute array.new_fixed x n.

$$
[\textsc{E-array.new}]\ z; val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new}\ x) \quad \hookrightarrow \quad val^n\ (\mathsf{array.new\_fixed}\ x\ n)
$$

array.new_default x

1. 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, $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case array.
4. Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.
5. Let $mut\ zt$ be y_0.
6. Assert: Due to validation, $\mathrm{default}(\mathrm{unpacktype}(zt))$ is defined.
7. Let $val^?$ be $\mathrm{default}(\mathrm{unpacktype}(zt))$.
8. Push val^n to the stack.
9. Execute array.new_fixed x n.

$$[\text{E-ARRAY.NEW\_DEFAULT}]z; (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_default}\ x) \quad \hookrightarrow \quad val^n\ (\mathsf{array.new\_fixed}\ x\ n) \quad \text{if } z.\mathsf{type}\,[x] \approx \mathsf{array}\ (mut\ zt) \wedge \mathrm{default\ unpack}(zt) = val$$

$\mathrm{ext\_arrayinst}(ai^*)$

1. Let f be the current frame.
2. Return $s \bigoplus \{.\mathsf{array}\ ai^*\}\ f$.

$$(s; f)[\mathsf{array} = ..ai^*] \quad = \quad s[\mathsf{array} = ..ai^*]; f$$

array.new_fixed x n

1. Assert: Due to validation, there are at least n values on the top of the stack.
2. Pop val^n from the stack.
3. Assert: Due to validation, $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case array.
4. Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.
5. Let $mut\ zt$ be y_0.
6. Let ai be $\{\mathsf{type}\ \mathrm{type}(x), \mathsf{field}\ (\mathrm{packval}(zt, val))^n\}$.
7. Push ref.array_addr $|\mathrm{arrayinst}()|$ to the stack.
8. Perform $\mathrm{ext\_arrayinst}(ai)$.

$$[\text{E-ARRAY.NEW\_FIXED}]z; val^n\ (\mathsf{array.new\_fixed}\ x\ n) \quad \hookrightarrow \quad z[\mathsf{array} = ..ai]; (\mathsf{ref.array}\ |z.\mathsf{array}|) \quad \text{if } z.\mathsf{type}\,[x] \approx \mathsf{array}\ (mut\ zt) \wedge ai = \{\mathsf{type}\ z.\mathsf{type}\,[x],\ \mathsf{field}\ (\mathrm{pack}_{zt}$$

array.new_elem 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. If $i + n$ is greater than the length of $\mathrm{elem}(y)$.elem, then:
   a. Trap.
6. Let ref^n be $\mathrm{elem}(y)$.elem$[i : n]$.
7. Push ref^n to the stack.
8. Execute array.new_fixed x n.

$$\begin{array}{llcll}
[\textsc{E-array.new_elem-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} & \text{if } i + n > |z.\mathsf{elem}[y] \\
[\textsc{E-array.new_elem-alloc}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new_elem}\ x\ y) & \hookrightarrow & ref^n\ (\mathsf{array.new_fixed}\ x\ n) & \text{if } ref^n = z.\mathsf{elem}[y].\mathsf{e}
\end{array}$$

$\mathrm{concat\_bytes}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $b^*\ ({b'}^*)^*$ be ${u_0}^*$.
3. Return $b^*\ \mathrm{concat\_bytes}(({b'}^*)^*)$.

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

array.new_data 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. If $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case array, then:
   a. Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.
   b. Let $mut\ zt$ be y_0.
   c. If $i + n \cdot \mathrm{storagesize}(zt)/8$ is greater than the length of $\mathrm{data}(y)$.data, then:
      1) Trap.
   d. Let nt be $\mathrm{unpacknumtype}(zt)$.
   e. Let b^* be $\mathrm{data}(y)$.data$[i : n \cdot \mathrm{storagesize}(zt)/8]$.
   f. Let gb^* be $group_{bytes_{by}}(\mathrm{storagesize}(zt)/8, b^*)$.

g. Let c^n be $(inverse_{of_ibytes}(\mathrm{storagesize}(zt), gb))^*$.

h. Push $(nt.\mathsf{const}\ c)^n$ to the stack.

i. Execute $\mathsf{array.new\_fixed}\ x\ n$.

$$[\text{E-ARRAY.NEW\_DATA-OOB}]\quad z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) \hookrightarrow \mathsf{trap} \quad \text{if } z.\mathsf{type}\,[x] \ \ \wedge i + n \cdot |zt$$

$$[\text{E-ARRAY.NEW\_DATA-ALLOC}]\quad z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) \hookrightarrow (nt.\mathsf{const}\ c)^n\ (\mathsf{array.new\_fixed}\ x\ n) \quad \text{if } z.\mathsf{type}\,[x] \ \ \wedge nt = \text{unpa} \ \ \wedge \text{concat}(\text{by}$$

$\mathsf{array.get}\ sx^?\ x$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop $\mathsf{i32.const}\ i$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop u_0 from the stack.
5. If u_0 is of the case $\mathsf{ref.null}$, then:
   a. Trap.
6. If u_0 is of the case $\mathsf{ref.array\_addr}$, then:
   a. Let $\mathsf{ref.array\_addr}\ a$ be u_0.
   b. If a is less than the length of $\mathrm{arrayinst}()$ and i is greater than or equal to the length of $\mathrm{arrayinst}()[a].\mathsf{field}$, then:
      1) Trap.
   c. If i is less than the length of $\mathrm{arrayinst}()[a].\mathsf{field}$ and a is less than the length of $\mathrm{arrayinst}()$, then:
      1) Let fv be $\mathrm{arrayinst}()[a].\mathsf{field}[i]$.
      2) If $\mathrm{expanddt}(\mathrm{arrayinst}()[a].\mathsf{type})$ is of the case array, then:
         a) Let $\mathsf{array}\ y_0$ be $\mathrm{expanddt}(\mathrm{arrayinst}()[a].\mathsf{type})$.
         b) Let $mut\ zt$ be y_0.
         c) Push $\mathrm{unpackval}(zt, sx^?, fv)$ to the stack.

$$[\text{E-ARRAY.GET-NULL}]\quad z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) \hookrightarrow \mathsf{trap}$$

$$[\text{E-ARRAY.GET-OOB}]\quad z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) \hookrightarrow \mathsf{trap} \quad \text{if } i \geq |z.\mathsf{array}[a].\mathsf{field}|$$

$$[\text{E-ARRAY.GET-ARRAY}]\quad z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) \hookrightarrow \mathrm{unpack}_{zt}^{sx^?}(fv) \quad \text{if } fv = z.\mathsf{array}[a].\mathsf{field}[i] \ \ \wedge z.\mathsf{array}[a].\mathsf{type} \approx \mathsf{array}\ (mut\ zt)$$

array.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val 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 is on the top of the stack.
6. Pop u_0 from the stack.
7. If u_0 is of the case ref.null, then:
   a. Trap.
8. If u_0 is of the case ref.array_addr, then:
   a. Let ref.array_addr a be u_0.
   b. If a is less than the length of $\mathrm{arrayinst}()$, then:
      1) If i is greater than or equal to the length of $\mathrm{arrayinst}()[a]$.field, then:
         a) Trap.
      2) If $\mathrm{expanddt}(\mathrm{arrayinst}()[a].\mathsf{type})$ is of the case array, then:
         a) Let array y_0 be $\mathrm{expanddt}(\mathrm{arrayinst}()[a].\mathsf{type})$.
         b) Let $mut\ zt$ be y_0.
         c) Let fv be $\mathrm{packval}(zt, val)$.
         d) Perform $\mathrm{with\_array}(a, i, fv)$.

$$
\begin{array}{llcll}
[\textsc{E-array.set-null}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \\
[\textsc{E-array.set-oob}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} & \text{if } i \geq |z.\mathsf{array}[a].\mathsf{field}| \\
[\textsc{E-array.set-array}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & \mathrm{with}_{array}(z,\ a,\ i,\ fv); \epsilon & \text{if } z.\mathsf{array}[a].\mathsf{type} \approx \mathsf{array}\ (mut\ zt) \\
 & & & & \wedge\ fv = \mathrm{pack}_{zt}(val)
\end{array}
$$

array.len

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop u_0 from the stack.
3. If u_0 is of the case ref.null, then:
   a. Trap.
4. If u_0 is of the case ref.array_addr, then:
   a. Let ref.array_addr a be u_0.
   b. If a is less than the length of $\mathrm{arrayinst}()$, then:
      1) Let n be the length of $\mathrm{arrayinst}()[a]$.field.
      2) Push i32.const n to the stack.

$$
\begin{array}{llcll}
[\textsc{E-array.len-null}] & z; (\mathsf{ref.null}\ ht)\ \mathsf{array.len} & \hookrightarrow & \mathsf{trap} & \\
[\textsc{E-array.len-array}] & z; (\mathsf{ref.array}\ a)\ \mathsf{array.len} & \hookrightarrow & (\mathsf{i32.const}\ n) & \text{if } n = |z.\mathsf{array}[a].\mathsf{field}|
\end{array}
$$

array.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. Assert: Due to validation, a value is on the top of the stack.
8. Pop u_0 from the stack.
9. If u_0 is of the case ref.null, then:
   a. Trap.
10. If u_0 is of the case ref.array_addr, then:

a. Let ref.array_addr a be u_0.

b. If a is less than the length of $\mathrm{arrayinst}()$ and $i + n$ is greater than the length of $\mathrm{arrayinst}()[a]$.field, then:

   1) Trap.

c. If n is 0, then:

   1) Do nothing.

d. Else:

   1) Let ref.array_addr a be u_0.
   2) Push ref.array_addr a to the stack.
   3) Push i32.const i to the stack.
   4) Push val to the stack.
   5) Execute array.set x.
   6) Push ref.array_addr a to the stack.
   7) Push i32.const $i + 1$ to the stack.
   8) Push val to the stack.
   9) Push i32.const $n - 1$ to the stack.
   10) Execute array.fill x.

$$
\begin{array}{llcl}
[\text{E-ARRAY.FILL-NULL}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.FILL-OOB}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.FILL-ZERO}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \epsilon \\
[\text{E-ARRAY.FILL-SUCC}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x)\ (\mathsf{ref.a}
\end{array}
$$

array.copy x_1 x_2

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_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop u_1 from the stack.
7. Assert: Due to validation, a value of value type i32 is on the top of the stack.
8. Pop i32.const i_1 from the stack.
9. Assert: Due to validation, a value is on the top of the stack.
10. Pop u_0 from the stack.
11. If u_0 is of the case ref.null and the type of u_1 is ref, then:

 a. Trap.

12. If u_1 is of the case ref.null and the type of u_0 is ref, then:

 a. Trap.

13. If u_0 is of the case ref.array_addr, then:

 a. Let ref.array_addr a_1 be u_0.

 b. If u_1 is of the case ref.array_addr, then:

   1) If a_1 is less than the length of $\mathrm{arrayinst}()$ and $i_1 + n$ is greater than the length of $\mathrm{arrayinst}()[a_1]$.field, then:

     a) Trap.

   2) Let ref.array_addr a_2 be u_1.

   3) If a_2 is less than the length of $\mathrm{arrayinst}()$ and $i_2 + n$ is greater than the length of $\mathrm{arrayinst}()[a_2]$.field, then:

     a) Trap.

 c. If n is 0, then:

   1) If u_1 is of the case ref.array_addr, then:

     a) Do nothing.

 d. Else if i_1 is greater than i_2, then:

   1) If $\mathrm{expanddt}(\mathrm{type}(x_2))$ is of the case array, then:

     a) Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x_2))$.

     b) Let $mut\ zt_2$ be y_0.

     c) Let ref.array_addr a_1 be u_0.

     d) If u_1 is of the case ref.array_addr, then:

       1. Let ref.array_addr a_2 be u_1.
       2. Let $sx^?$ be $\mathrm{sxfield}(zt_2)$.
       3. Push ref.array_addr a_1 to the stack.
       4. Push i32.const $i_1 + n - 1$ to the stack.
       5. Push ref.array_addr a_2 to the stack.

6. Push i32.const $i_2 + n - 1$ to the stack.
7. Execute array.get $sx^?$ x_2.
8. Execute array.set x_1.
9. Push ref.array_addr a_1 to the stack.
10. Push i32.const i_1 to the stack.
11. Push ref.array_addr a_2 to the stack.
12. Push i32.const i_2 to the stack.
13. Push i32.const $n - 1$ to the stack.
14. Execute array.copy x_1 x_2.

e. Else:

1) If $\mathrm{expanddt}(\mathrm{type}(x_2))$ is of the case array, then:

a) Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x_2))$.

b) Let $mut\ zt_2$ be y_0.

c) Let ref.array_addr a_1 be u_0.

d) If u_1 is of the case ref.array_addr, then:

1. Let ref.array_addr a_2 be u_1.
2. Let $sx^?$ be $\mathrm{sxfield}(zt_2)$.
3. Push ref.array_addr a_1 to the stack.
4. Push i32.const i_1 to the stack.
5. Push ref.array_addr a_2 to the stack.
6. Push i32.const i_2 to the stack.
7. Execute array.get $sx^?$ x_2.
8. Execute array.set x_1.
9. Push ref.array_addr a_1 to the stack.
10. Push i32.const $i_1 + 1$ to the stack.
11. Push ref.array_addr a_2 to the stack.
12. Push i32.const $i_2 + 1$ to the stack.
13. Push i32.const $n - 1$ to the stack.
14. Execute array.copy x_1 x_2.

[E-ARRAY.COPY-NULL1] $z; (\mathsf{ref.null}\ ht_1)\ (\mathsf{i32.const}\ i_1)\ ref\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow \mathsf{trap}$

[E-ARRAY.COPY-NULL2] $z; ref\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.null}\ ht_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow \mathsf{trap}$

[E-ARRAY.COPY-OOB1] $z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow \mathsf{trap}$

[E-ARRAY.COPY-OOB2] $z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow \mathsf{trap}$

[E-ARRAY.COPY-ZERO] $z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow \epsilon$

[E-ARRAY.COPY-LE] $z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow (\mathsf{ref.array}\ a_1)\ (\mathsf{i}$

[E-ARRAY.COPY-GT] $z; (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.copy}\ x_1\ x_2) \hookrightarrow (\mathsf{ref.array}\ a_1)\ (\mathsf{i}$

array.init_elem 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 j 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. Assert: Due to validation, a value is on the top of the stack.
8. Pop u_0 from the stack.
9. If u_0 is of the case ref.null, then:
   a. Trap.
10. If u_0 is of the case ref.array_addr, then:
   a. Let ref.array_addr a be u_0.
   b. If a is less than the length of $\mathrm{arrayinst}()$ and $i + n$ is greater than the length of $\mathrm{arrayinst}()[a]$.field, then:
      1) Trap.
11. If $j + n$ is greater than the length of $\mathrm{elem}(y)$.elem, then:
   a. If u_0 is of the case ref.array_addr, then:
      1) Trap.
   b. If n is 0 and j is less than the length of $\mathrm{elem}(y)$.elem, then:
      1) Let ref be $\mathrm{elem}(y)$.elem$[j]$.
      2) If u_0 is of the case ref.array_addr, then:
         a) Let ref.array_addr a be u_0.
         b) Push ref.array_addr a to the stack.
         c) Push i32.const i to the stack.
         d) Push ref to the stack.
         e) Execute array.set x.
         f) Push ref.array_addr a to the stack.
         g) Push i32.const $i + 1$ to the stack.
         h) Push i32.const $j + 1$ to the stack.
         i) Push i32.const $n - 1$ to the stack.
         j) Execute array.init_elem x y.
12. Else if n is 0, then:
   a. If u_0 is of the case ref.array_addr, then:
      1) Do nothing.
13. Else:
   a. If j is less than the length of $\mathrm{elem}(y)$.elem, then:
      1) Let ref be $\mathrm{elem}(y)$.elem$[j]$.
      2) If u_0 is of the case ref.array_addr, then:
         a) Let ref.array_addr a be u_0.

b) Push ref.array_addr a to the stack.

c) Push i32.const i to the stack.

d) Push ref to the stack.

e) Execute array.set x.

f) Push ref.array_addr a to the stack.

g) Push i32.const $i+1$ to the stack.

h) Push i32.const $j+1$ to the stack.

i) Push i32.const $n-1$ to the stack.

j) Execute array.init_elem x y.

$$
\begin{array}{lll}
[\text{E-ARRAY.INIT\_ELEM-NULL}]\, z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.INIT\_ELEM-OOB1}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.INIT\_ELEM-OOB2}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.INIT\_ELEM-ZERO}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & \epsilon \\
[\text{E-ARRAY.INIT\_ELEM-SUCC}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_elem}\ x\ y) & \hookrightarrow & (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ r
\end{array}
$$

array.init_data 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 j 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. Assert: Due to validation, a value is on the top of the stack.
8. Pop u_0 from the stack.
9. If u_0 is of the case ref.null, then:
   a. Trap.
10. If u_0 is of the case ref.array_addr, then:

a. Let ref.array_addr a be u_0.

b. If a is less than the length of $\mathrm{arrayinst}()$ and $i+n$ is greater than the length of $\mathrm{arrayinst}()[a]$.field, then:

   1) Trap.

11. If $\mathrm{expanddt}(\mathrm{type}(x))$ is not of the case array, then:

a. If n is 0 and u_0 is of the case ref.array_addr, then:

   1) Do nothing.

12. Else:

a. Let array y_0 be $\mathrm{expanddt}(\mathrm{type}(x))$.

b. Let $mut\ zt$ be y_0.

c. If u_0 is of the case ref.array_addr, then:

1) If $j + n \cdot \mathrm{storagesize}(zt)/8$ is greater than the length of $\mathrm{data}(y).\mathsf{data}$, then:

   a) Trap.

2) If n is 0, then:

   a) Do nothing.

3) Else:

   a) Let $\mathsf{array}\ y_0$ be $\mathrm{expanddt}(\mathrm{type}(x))$.

   b) Let $mut\ zt$ be y_0.

   c) Let $\mathsf{ref.array\_addr}\ a$ be u_0.

   d) Let c be $inverse_{of_z tbytes}(zt, \mathrm{data}(y).\mathsf{data}[j : \mathrm{storagesize}(zt)/8])$.

   e) Let nt be $\mathrm{unpacknumtype}(zt)$.

   f) Push $\mathsf{ref.array\_addr}\ a$ to the stack.

   g) Push $\mathsf{i32.const}\ i$ to the stack.

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

   i) Execute $\mathsf{array.set}\ x$.

   j) Push $\mathsf{ref.array\_addr}\ a$ to the stack.

   k) Push $\mathsf{i32.const}\ i + 1$ to the stack.

   l) Push $\mathsf{i32.const}\ j + \mathrm{storagesize}(zt)/8$ to the stack.

   m) Push $\mathsf{i32.const}\ n - 1$ to the stack.

   n) Execute $\mathsf{array.init\_data}\ x\ y$.

$$\begin{array}{lll}
[\textsc{E-array.init_data-null}]\, z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-array.init_data-oob1}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-array.init_data-oob2}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
\\
[\textsc{E-array.init_data-zero}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_data}\ x\ y) & \hookrightarrow & \epsilon \\
[\textsc{E-array.init_data-succ}]\, z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_data}\ x\ y) & \hookrightarrow & (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (
\end{array}$$

extern.convert_any

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop u_0 from the stack.
3. If u_0 is of the case $\mathsf{ref.null}$, then:
   a. Push $\mathsf{ref.null}\ EXTERN$ to the stack.
4. If the type of u_0 is addrref, then:
   a. Let $addrref$ be u_0.
   b. Push $\mathsf{ref.extern}\ addrref$ to the stack.

$$
\begin{array}{llcl}
[\textsc{E-extern.convert\_any-null}] & (\mathsf{ref.null}\ ht)\ \mathsf{extern.convert\_any} & \hookrightarrow & (\mathsf{ref.null}\ \mathsf{extern}) \\
[\textsc{E-extern.convert\_any-addr}] & addrref\ \mathsf{extern.convert\_any} & \hookrightarrow & (\mathsf{ref.extern}\ addrref)
\end{array}
$$

any.convert_extern

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop u_0 from the stack.
3. If u_0 is of the case ref.null, then:
   a. Push ref.null ANY to the stack.
4. If u_0 is of the case ref.extern, then:
   a. Let ref.extern $addrref$ be u_0.
   b. Push $addrref$ to the stack.

$$
\begin{array}{llcl}
[\textsc{E-any.convert\_extern-null}] & (\mathsf{ref.null}\ ht)\ \mathsf{any.convert\_extern} & \hookrightarrow & (\mathsf{ref.null}\ \mathsf{any}) \\
[\textsc{E-any.convert\_extern-addr}] & (\mathsf{ref.extern}\ addrref)\ \mathsf{any.convert\_extern} & \hookrightarrow & addrref
\end{array}
$$

2.4.3 Parametric Instructions

drop

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. Do nothing.

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

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.

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

2.4.4 Variable Instructions

local.get x

1. Assert: Due to validation, $\mathrm{local}(x)$ is defined.
2. Let $val^?$ be $\mathrm{local}(x)$.
3. Push val to the stack.

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

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 $\mathrm{with\_local}(x, val)$.

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

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).

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

2.4.5 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 table(x).elem, then:
   a. Trap.
4. Push table(x).elem$[i]$ to the stack.

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

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 table(x).elem, then:
   a. Trap.
6. Perform with_table(x, i, ref).

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

table.size x

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

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

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 $\mathrm{growtable}(\mathrm{table}(x), n, ref)$.
   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 $\mathrm{invsigned}(32, -1)$ to the stack.

$$\begin{array}{llll}
[\text{E-TABLE.GROW-SUCCEED}]\, z; ref\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z[\mathsf{table}[x] = ti]; (\mathsf{i32.const}\ |z.\mathsf{table}[x].\mathsf{elem}|) & \text{if } ti = \mathrm{growtable}(z.\mathsf{ta} \\
[\text{E-TABLE.GROW-FAIL}] \quad z; ref\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \mathrm{signed}^{-1}{}_{32}{-1}) &
\end{array}$$

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 $\mathrm{table}(x)$.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.

$$
\begin{array}{llcl}
[\text{E-TABLE.FILL-OOB}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \mathsf{trap} \\
[\text{E-TABLE.FILL-ZERO}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \epsilon \\
[\text{E-TABLE.FILL-SUCC}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x)\ (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.const}\ n
\end{array}
$$

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.

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

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.

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

elem.drop x

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

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

2.4.6 Memory Instructions

load nt ${u_0}^?$ x mo

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 $i + mo.\mathsf{offset} + \mathrm{size}(nt)/8$ is greater than the length of $\mathrm{mem}(x).\mathsf{data}$ and ${u_0}^?$ is not defined, then:
   a. Trap.
4. If ${u_0}^?$ is not defined, then:
   a. Let c be $inverse_{of_n tbytes}(nt, \mathrm{mem}(x).\mathsf{data}[i + mo.\mathsf{offset} : \mathrm{size}(nt)/8])$.
   b. Push nt.const c to the stack.
5. Else:
   a. Let ${y_0}^?$ be ${u_0}^?$.
   b. Let n sx be y_0.
   c. If $i + mo.\mathsf{offset} + n/8$ is greater than the length of $\mathrm{mem}(x).\mathsf{data}$, then:
      1) Trap.
   d. Let c be $inverse_{of_i bytes}(n, \mathrm{mem}(x).\mathsf{data}[i + mo.\mathsf{offset} : n/8])$.
   e. Push nt.const $\mathrm{ext}(n, \mathrm{size}(nt), sx, c)$ to the stack.

[E-LOAD-NUM-OOB]	$z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ mo)$	$\hookrightarrow$	trap	if $i + mo.\mathsf{offset} + \lvert nt\rvert/8 > \lvert z.\mathsf{mem}[x].\mathsf{data}\rvert$
[E-LOAD-NUM-VAL]	$z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ mo)$	$\hookrightarrow$	$(nt.\mathsf{const}\ c)$	if $\mathrm{bytes}_{nt}(c) = z.\mathsf{mem}[x].\mathsf{data}[i + mo.\mathsf{offset}$:
[E-LOAD-PACK-OOB]	$z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}n\_sx\ x\ mo)$	$\hookrightarrow$	trap	if $i + mo.\mathsf{offset} + n/8 > \lvert z.\mathsf{mem}[x].\mathsf{data}\rvert$
[E-LOAD-PACK-VAL]	$z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}n\_sx\ x\ mo)$	$\hookrightarrow$	$(nt.\mathsf{const}\ \mathrm{ext}^{sx}_{n,\lvert nt\rvert}(c))$	if $\mathrm{bytes}_{in}(c) = z.\mathsf{mem}[x].\mathsf{data}[i + mo.\mathsf{offset}$:

store nt ${u_0}^?$ x mo

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c 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 $i + mo.\mathsf{offset} + \mathrm{size}(nt)/8$ is greater than the length of $\mathrm{mem}(x).\mathsf{data}$ and ${u_0}^?$ is not defined, then:
   a. Trap.
6. If ${u_0}^?$ is not defined, then:
   a. Let b^* be $\mathrm{ntbytes}(nt, c)$.
   b. Perform $\mathrm{with\_mem}(x, i + mo.\mathsf{offset}, \mathrm{size}(nt)/8, b^*)$.
7. Else:
   a. Let $n^?$ be ${u_0}^?$.
   b. If $i + mo.\mathsf{offset} + n/8$ is greater than the length of $\mathrm{mem}(x).\mathsf{data}$, then:
      1) Trap.
   c. Let b^* be $\mathrm{ibytes}(n, \mathrm{wrap}(\mathrm{size}(nt), n, c))$.

d. Perform $\mathrm{with\_mem}(x, i + mo.\mathsf{offset}, n/8, b^*)$.

$$
\begin{array}{llll}
[\textsc{E-store-num-oob}]\ z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ mo) & \hookrightarrow & z; \mathsf{trap} & \text{if } i + mo.\mathsf{of} \\
[\textsc{E-store-num-val}]\ z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ mo) & \hookrightarrow & z[\mathsf{mem}[x].\mathsf{data}[i + mo.\mathsf{offset} : |nt|/8] = b^*]; \epsilon & \text{if } b^* = \text{byte} \\
[\textsc{E-store-pack-oob}]\ z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ mo) & \hookrightarrow & z; \mathsf{trap} & \text{if } i + mo.\mathsf{of} \\
[\textsc{E-store-pack-val}]\ z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ mo) & \hookrightarrow & z[\mathsf{mem}[x].\mathsf{data}[i + mo.\mathsf{offset} : n/8] = b^*]; \epsilon & \text{if } b^* = \text{byte}
\end{array}
$$

memory.size x

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

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

memory.grow x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop $\mathsf{i32.const}\ n$ from the stack.
3. Either:
   a. Let mi be $\mathrm{growmemory}(\mathrm{mem}(x), n)$.
   b. Push $\mathsf{i32.const}\ |\mathrm{mem}(0).\mathsf{data}|/64 \cdot \mathrm{Ki}()$ to the stack.
   c. Perform $\mathrm{with\_meminst}(x, mi)$.
4. Or:
   a. Push $\mathsf{i32.const}\ \mathrm{invsigned}(32, -1)$ to the stack.

$$
\begin{array}{llll}
[\textsc{E-memory.grow-succeed}]\ z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z[\mathsf{mem}[x] = mi]; (\mathsf{i32.const}\ |z.\mathsf{mem}[0].\mathsf{data}|/(64 \cdot \mathrm{Ki})) & \text{if } mi = \text{gro} \\
[\textsc{E-memory.grow-fail}]\quad z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \text{signed}^{-1}{}_{32}{-1}) &
\end{array}
$$

memory.fill x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop $\mathsf{i32.const}\ n$ 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 $\mathsf{i32.const}\ i$ from the stack.
7. If $i + n$ is greater than the length of $\mathrm{mem}(x).\mathsf{data}$, 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 $8^?$ x $\mathrm{memop0}()$.
   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 x.

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

memory.copy x_1 x_2

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_2 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_1 from the stack.
7. If i_1+n is greater than the length of $\mathrm{mem}(x_1)$.data or i_2+n is greater than the length of $\mathrm{mem}(x_2)$.data, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else:
   a. If i_1 is less than or equal to i_2, then:
      1) Push i32.const i_1 to the stack.
      2) Push i32.const i_2 to the stack.
      3) Execute load i32 8 $\mathsf{u}^?$ x_2 $\mathrm{memop0}()$.
      4) Execute store i32 $8^?$ x_1 $\mathrm{memop0}()$.
      5) Push i32.const i_1+1 to the stack.
      6) Push i32.const i_2+1 to the stack.
   b. Else:
      1) Push i32.const i_1+n-1 to the stack.
      2) Push i32.const i_2+n-1 to the stack.

3) Execute load i32 8 $\mathsf{u}^?$ x_2 $\mathrm{memop0}()$.
4) Execute store i32 $8^?$ x_1 $\mathrm{memop0}()$.
5) Push i32.const i_1 to the stack.
6) Push i32.const i_2 to the stack.

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

d. Execute memory.copy x_1 x_2.

$$[\textsc{E-memory.copy-oob}]\ z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) \hookrightarrow \mathsf{trap}$$
$$[\textsc{E-memory.copy-zero}]\ z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) \hookrightarrow \epsilon$$
$$[\textsc{E-memory.copy-le}]\ z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) \hookrightarrow (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.load8\_u}\ x$$
$$[\textsc{E-memory.copy-gt}]\ z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) \hookrightarrow (\mathsf{i32.const}\ i_1+n-1)\ (\mathsf{i32.const}\ i_2+n-$$

memory.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 $\mathrm{data}(y)$.data or $j+n$ is greater than the length of $\mathrm{mem}(x)$.data, then:
   a. Trap.
8. If n is 0, then:
   a. Do nothing.
9. Else if i is less than the length of $\mathrm{data}(y)$.data, then:
   a. Push i32.const j to the stack.
   b. Push i32.const $\mathrm{data}(y)$.data$[i]$ to the stack.
   c. Execute store i32 $8^?$ x $\mathrm{memop0}()$.
   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 y.

$$[\textsc{E-memory.init-oob}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) \hookrightarrow \mathsf{trap}$$
$$[\textsc{E-memory.init-zero}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) \hookrightarrow \epsilon$$
$$[\textsc{E-memory.init-succ}]\ z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) \hookrightarrow (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ z.\mathsf{data}[y].\mathsf{data}[i])\ (\mathsf{i32.s}$$

data.drop x

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

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

2.4.7 Control Instructions

nop

1. Do nothing.

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

unreachable

1. Trap.

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

$\mathrm{blocktype}(u_1)$

1. If u_1 is _result ϵ, then:
   a. Return $\epsilon \to \epsilon$.
2. If u_1 is of the case _result, then:
   a. Let _result y_0 be u_1.
   b. If y_0 is defined, then:
      1) Let $t^?$ be y_0.
      2) Return $\epsilon \to t$.
3. Assert: Due to validation, u_1 is of the case _idx.
4. Let _idx x be u_1.
5. Assert: Due to validation, $\mathrm{expanddt}(\mathrm{type}(x))$ is of the case func.
6. Let func ft be $\mathrm{expanddt}(\mathrm{type}(x))$.
7. Return ft.

$$\begin{array}{lcll} \mathrm{blocktype}_z(\epsilon) & = & \epsilon \to \epsilon & \\ \mathrm{blocktype}_z(t) & = & \epsilon \to t & \\ \mathrm{blocktype}_z(x) & = & ft & \text{if } z.\mathsf{type}\,[x] \approx \mathsf{func}\ ft \end{array}$$

block $bt\ instr^*$

1. Let $t_1{}^k \to t_2{}^n$ be blocktype(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.

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

loop $bt\ instr^*$

1. Let $t_1{}^k \to t_2{}^n$ be blocktype(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.

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

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 $bt\ instr_2{}^*$.

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

br u_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 ${u_1}^*$ from the stack.
5. Exit current context.
6. If u_0 is 0 and the length of ${u_1}^*$ is greater than or equal to n, then:
   a. Let $val'^*\ val^n$ be ${u_1}^*$.
   b. Push val^n to the stack.
   c. Execute the sequence $instr'^*$.
7. If u_0 is greater than or equal to 1, then:
   a. Let l be $u_0 - 1$.
   b. Let val^* be ${u_1}^*$.
   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.

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

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

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

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

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

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. If val is of the case $\mathsf{ref.null}$, then:
   a. Execute $\mathsf{br}\ l$.
4. Else:
   a. Push val to the stack.

$$\begin{array}{lllll}
[\textsc{E-br_on_null-null}] & val\ (\mathsf{br_on_null}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } val = \mathsf{ref.null}\ ht \\
[\textsc{E-br_on_null-addr}] & val\ (\mathsf{br_on_null}\ l) & \hookrightarrow & val & \text{otherwise}
\end{array}$$

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

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop val from the stack.
3. If val is of the case $\mathsf{ref.null}$, then:
   a. Do nothing.
4. Else:
   a. Push val to the stack.
   b. Execute $\mathsf{br}\ l$.

$$\begin{array}{lllll}
[\textsc{E-br_on_non_null-null}] & val\ (\mathsf{br_on_non_null}\ l) & \hookrightarrow & \epsilon & \text{if } val = \mathsf{ref.null}\ ht \\
[\textsc{E-br_on_non_null-addr}] & val\ (\mathsf{br_on_non_null}\ l) & \hookrightarrow & val\ (\mathsf{br}\ l) & \text{otherwise}
\end{array}$$

br_on_cast l rt_1 rt_2

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. Let rt be $ref_{type_of}(ref)$.
4. If not rt matches $\mathrm{inst\_reftype}(\mathrm{moduleinst}(), rt_2)$, then:
   a. Push ref to the stack.
5. Else:
   a. Push ref to the stack.
   b. Execute br l.

$$
\begin{array}{llcll}
[\textsc{E-br\_on\_cast-succeed}] & z; \mathit{ref}\ (\mathsf{br\_on\_cast}\ l\ \mathit{rt}_1\ \mathit{rt}_2) & \hookrightarrow & \mathit{ref}\ (\mathsf{br}\ l) & \text{if } z.\mathsf{store} \vdash \mathit{ref} : \mathit{rt} \\
 & & & & \wedge\ \{\} \vdash \mathit{rt} \leq \mathrm{inst}_{z.\mathsf{module}}(\mathit{rt}_2) \\
[\textsc{E-br\_on\_cast-fail}] & z; \mathit{ref}\ (\mathsf{br\_on\_cast}\ l\ \mathit{rt}_1\ \mathit{rt}_2) & \hookrightarrow & \mathit{ref} & \text{otherwise}
\end{array}
$$

br_on_cast_fail l rt_1 rt_2

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. Let rt be $ref_{type_of}(ref)$.
4. If rt matches $\mathrm{inst\_reftype}(\mathrm{moduleinst}(), rt_2)$, then:
   a. Push ref to the stack.
5. Else:
   a. Push ref to the stack.
   b. Execute br l.

$$
\begin{array}{llcll}
[\textsc{E-br\_on\_cast\_fail-succeed}] & z; \mathit{ref}\ (\mathsf{br\_on\_cast\_fail}\ l\ \mathit{rt}_1\ \mathit{rt}_2) & \hookrightarrow & \mathit{ref} & \text{if } z.\mathsf{store} \vdash \mathit{ref} : \mathit{rt} \\
 & & & & \wedge\ \{\} \vdash \mathit{rt} \leq \mathrm{inst}_{z.\mathsf{module}}(\mathit{rt}_2) \\
[\textsc{E-br\_on\_cast\_fail-fail}] & z; \mathit{ref}\ (\mathsf{br\_on\_cast\_fail}\ l\ \mathit{rt}_1\ \mathit{rt}_2) & \hookrightarrow & \mathit{ref}\ (\mathsf{br}\ l) & \text{otherwise}
\end{array}
$$

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.

$$\begin{array}{lll}[\text{E-RETURN-FRAME}](\mathsf{frame}_n\{f\}\ val'^*\ val^n\ \mathsf{return}\ instr^*) & \hookrightarrow & val^n \\ [\text{E-RETURN-LABEL}]\ (\mathsf{label}_k\{instr'^*\}\ val^*\ \mathsf{return}\ instr^*) & \hookrightarrow & val^*\ \mathsf{return}\end{array}$$

call x

1. Assert: Due to validation, x is less than the length of funcaddr().
2. Push ref.func_addr funcaddr()$[x]$ to the stack.
3. Execute call_ref ϵ.

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

call_ref x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop ref from the stack.
3. If ref is of the case ref.null, then:
   a. Trap.
4. Assert: Due to validation, ref is of the case ref.func_addr.
5. Let ref.func_addr a be ref.
6. If a is less than the length of funcinst(), then:
   a. Let fi be funcinst()$[a]$.
   b. If $fi.CODE$ is of the case func, then:
      1) Let func y_0 y_1 $instr^*$ be $fi.CODE$.
      2) Let (local t)* be y_1.
      3) If expanddt(fi.type) is of the case func, then:
         a) Let func y_0 be expanddt(fi.type).
         b) Let $t_1{}^n \rightarrow t_2{}^m$ be y_0.
         c) Assert: Due to validation, there are at least n values on the top of the stack.
         d) Pop val^n from the stack.
         e) Let f be $\{$local $(val^?)^n$ (default$(t))^*$, module fi.module$\}$.
         f) Let F be the activation of f with arity m.
         g) Enter F with label $FRAME$:
            1. Let L be the label whose arity is m and whose continuation is ϵ.

2. Enter L with label $instr^*\ LABEL$:

$$[\text{E-CALL\_REF-NULL}]\, z; (\mathsf{ref.null}\ ht)\ (\mathsf{call\_ref}\ x^?) \quad \hookrightarrow \quad \mathsf{trap}$$

$$[\text{E-CALL\_REF-FUNC}]\, z; val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{call\_ref}\ x^?) \quad \hookrightarrow \quad (\mathsf{frame}_m\{f\}\ (\mathsf{label}_m\{\epsilon\}\ instr^*)) \quad \begin{array}{l} \text{if } z.\mathsf{func}[a] = fi \\ \wedge\ fi.\mathsf{type} \approx \mathsf{func}\ (t_1^n \to t_2^m) \\ \wedge\ fi.\mathsf{code} = \mathsf{func}\ x'\ (\mathsf{local}\ t)^*\ (instr^*) \\ \wedge\ f = \{\mathsf{local}\ val^n\ (\mathrm{default}\ t)^*,\ \mathsf{modul} \end{array}$$

call_indirect x y

1. Execute table.get x.
2. Execute ref.cast ref null $\epsilon^?$ $\mathrm{idx}(y)$.
3. Execute call_ref $y^?$.

$$[\text{E-CALL\_INDIRECT-CALL}](\mathsf{call\_indirect}\ x\ y) \quad \hookrightarrow \quad (\mathsf{table.get}\ x)\ (\mathsf{ref.cast}\ (\mathsf{ref\ null}\ y))\ (\mathsf{call\_ref}\ y)$$

return_call 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.
3. Execute return_call_ref ϵ.

$$[\text{E-RETURN\_CALL}]\, z; (\mathsf{return\_call}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ z.\mathsf{module.func}[x])\ (\mathsf{return\_call\_ref})$$

return_call_ref $x^?$

1. If not the current context is frame, then:
   a. If the current context is label, then:
      1) Pop all values val^* from the stack.
      2) Exit current context.
      3) Push val^* to the stack.
      4) Execute return_call_ref $x^?$.
2. Else:
   a. Pop u_0 from the stack.
   b. Pop all values $u_1{}^*$ from the stack.
   c. Exit current context.
   d. If u_0 is of the case ref.func_addr, then:
      1) Let ref.func_addr a be u_0.

2) If a is less than the length of funcinst() and expanddt(funcinst()$[a]$.type) is of the case func, then:

   a) Let func y_0 be expanddt(funcinst()$[a]$.type).

   b) Let ${t_1}^n \rightarrow {t_2}^m$ be y_0.

   c) If the length of ${u_1}^*$ is greater than or equal to n, then:

      1. Let $val'^* \; val^n$ be ${u_1}^*$.
      2. Push val^n to the stack.
      3. Push ref.func_addr a to the stack.
      4. Execute call_ref $x^?$.

e. If u_0 is of the case ref.null, then:

   1) Trap.

$$\begin{array}{llcl}
[\textsc{E-return_call_ref-frame-null}] & z; (\mathsf{frame}_k\{f\} \; val^* \; (\mathsf{ref.null} \; ht) \; (\mathsf{return_call_ref} \; x^?) \; instr^*) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-return_call_ref-frame-addr}] & z; (\mathsf{frame}_k\{f\} \; val'^* \; val^n \; (\mathsf{ref.func} \; a) \; (\mathsf{return_call_ref} \; x^?) \; instr^*) & \hookrightarrow & val^n \; (\mathsf{ref.func} \; a) \; (\mathsf{call_ref} \; x^? \\
[\textsc{E-return_call_ref-label}] & z; (\mathsf{label}_k\{instr'^*\} \; val^* \; (\mathsf{return_call_ref} \; x^?) \; instr^*) & \hookrightarrow & val^* \; (\mathsf{return_call_ref} \; x^?)
\end{array}$$

return_call_indirect x y

1. Execute table.get x.
2. Execute ref.cast ref null $\epsilon^?$ idx(y).
3. Execute return_call_ref $y^?$.

$$[\textsc{E-return\_call\_indirect}] (\mathsf{return\_call\_indirect} \; x \; y) \quad \hookrightarrow \quad (\mathsf{table.get} \; x) \; (\mathsf{ref.cast} \; (\mathsf{ref} \; \mathsf{null} \; y)) \; (\mathsf{return\_call\_ref} \; y)$$

2.4.8 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.

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

2.4.9 Function Calls

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.

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

2.4.10 Expressions

$$\begin{array}{llcll}
[\textsc{E-expr-done}] & z; val^* & \hookrightarrow^* & z; val^* & \\
[\textsc{E-expr-step}] & z; instr^* & \hookrightarrow^* & z''; val^* & \text{if } z; instr^* \hookrightarrow z'; instr'^* \\
& & & & \wedge\ z'; instr' \hookrightarrow^* z''; val^* \\
[\textsc{E-expr}] & z; instr^* & \hookrightarrow^* & z'; val^* & \text{if } z; instr^* \hookrightarrow^* z; val^*
\end{array}$$

2.5 Modules

2.5.1 Allocation

$\mathrm{alloctypes}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $rectype'^*\ rectype$ be ${u_0}^*$.
3. Let $deftype'^*$ be $\mathrm{alloctypes}(rectype'^*)$.
4. Let x be the length of $deftype'^*$.
5. Let $deftype^*$ be $\mathrm{subst\_all\_deftypes}(\mathrm{rolldt}(x, rectype), deftype'^*)$.
6. Return $deftype'^*\ deftype^*$.

$$\begin{array}{lcll}
\mathrm{alloctypes}(\epsilon) & = & \epsilon & \\
\mathrm{alloctypes}(rectype'^*\ rectype) & = & deftype'^*\ deftype^* & \text{if } deftype'^* = \mathrm{alloctypes}(rectype'^*) \\
& & & \wedge\ deftype^* = \mathrm{roll}_x(rectype)[:= deftype'^*] \\
& & & \wedge\ x = |deftype'^*|
\end{array}$$

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

1. Assert: Due to validation, $func$ is of the case func.
2. Let $\mathsf{func}\ x\ local^*\ expr$ be $func$.
3. Let fi be $\{\mathsf{type}\ mm.\mathsf{type}[x], \mathsf{module}\ mm, \mathsf{code}\ func\}$.
4. Let a be the length of $s.\mathsf{func}$.
5. Append fi to the $s.\mathsf{func}$.
6. Return a.

$$\begin{array}{lcll}
\mathrm{allocfunc}(s,\ mm,\ func) & = & (s[\mathsf{func} = ..fi],\ |s.\mathsf{func}|) & \text{if } func = \mathsf{func}\ x\ local^*\ expr \\
& & & \wedge\ fi = \{\mathsf{type}\ mm.\mathsf{type}[x],\ \mathsf{module}\ mm,\ \mathsf{code}\ func\}
\end{array}$$

$\mathrm{allocfuncs}(mm, {u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $func\ func'^*$ be ${u_0}^*$.
3. Let fa be $\mathrm{allocfunc}(mm, func)$.
4. Let fa'^* be $\mathrm{allocfuncs}(mm, func'^*)$.
5. Return $fa\ fa'^*$.

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

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

1. Let gi be $\{\mathsf{type}\ globaltype, \mathsf{value}\ val\}$.
2. Let a be the length of $s.\mathsf{global}$.
3. Append gi to the $s.\mathsf{global}$.
4. Return a.

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

$\text{allocglobals}(u_0{}^*, u_1{}^*)$

1. If $u_0{}^*$ is ϵ, then:
   a. Assert: Due to validation, $u_1{}^*$ is ϵ.
   b. Return ϵ.
2. Else:
   a. Let $globaltype\ globaltype'^*$ be $u_0{}^*$.
   b. Assert: Due to validation, the length of $u_1{}^*$ is greater than or equal to 1.
   c. Let $val\ val'^*$ be $u_1{}^*$.
   d. Let ga be $\text{allocglobal}(globaltype, val)$.
   e. Let ga'^* be $\text{allocglobals}(globaltype'^*, val'^*)$.
   f. Return $ga\ ga'^*$.

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

$\text{alloctable}(i\ j\ rt, ref)$

1. Let ti be $\{\mathsf{type}\ i\ j\ rt, \mathsf{elem}\ ref^i\}$.
2. Let a be the length of $s.\mathsf{table}$.
3. Append ti to the $s.\mathsf{table}$.
4. Return a.

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

$\text{alloctables}(u_0{}^*, u_1{}^*)$

1. If $u_0{}^*$ is ϵ and $u_1{}^*$ is ϵ, then:
   a. Return ϵ.
2. Assert: Due to validation, the length of $u_1{}^*$ is greater than or equal to 1.
3. Let $ref\ ref'^*$ be $u_1{}^*$.
4. Assert: Due to validation, the length of $u_0{}^*$ is greater than or equal to 1.
5. Let $tabletype\ tabletype'^*$ be $u_0{}^*$.
6. Let ta be $\text{alloctable}(tabletype, ref)$.
7. Let ta'^* be $\text{alloctables}(tabletype'^*, ref'^*)$.
8. Return $ta\ ta'^*$.

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

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

1. Let mi be $\{\mathsf{type}\ \mathsf{i8}\ i\ j, \mathsf{data}\ 0^{i \cdot 64} \cdot \mathrm{Ki}()\}$.
2. Let a be the length of $s.\mathsf{mem}$.
3. Append mi to the $s.\mathsf{mem}$.
4. Return a.

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

$\mathrm{allocmems}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $\mathit{memtype}\ \mathit{memtype'}^*$ be ${u_0}^*$.
3. Let ma be $\mathrm{allocmem}(\mathit{memtype})$.
4. Let ma'^* be $\mathrm{allocmems}(\mathit{memtype'}^*)$.
5. Return $ma\ ma'^*$.

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

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

1. Let ei be $\{\mathsf{type}\ rt, \mathsf{elem}\ ref^*\}$.
2. Let a be the length of $s.\mathsf{elem}$.
3. Append ei to the $s.\mathsf{elem}$.
4. Return a.

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

$\text{allocelems}(u_0{}^*, u_1{}^*)$

1. If $u_0{}^*$ is ϵ and $u_1{}^*$ is ϵ, then:
   a. Return ϵ.
2. Assert: Due to validation, the length of $u_1{}^*$ is greater than or equal to 1.
3. Let ref^* $(ref'^*)^*$ be $u_1{}^*$.
4. Assert: Due to validation, the length of $u_0{}^*$ is greater than or equal to 1.
5. Let rt rt'^* be $u_0{}^*$.
6. Let ea be $\text{allocelem}(rt, ref^*)$.
7. Let ea'^* be $\text{allocelems}(rt'^*, (ref'^*)^*)$.
8. Return ea ea'^*.

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

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

1. Let di be $\{\mathsf{data}\ byte^*\}$.
2. Let a be the length of $s.\mathsf{data}$.
3. Append di to the $s.\mathsf{data}$.
4. Return a.

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

$\text{allocdatas}(u_0{}^*)$

1. If $u_0{}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $byte^*$ $(byte'^*)^*$ be $u_0{}^*$.
3. Let da be $\text{allocdata}(byte^*)$.
4. Let da'^* be $\text{allocdatas}((byte'^*)^*)$.
5. Return da da'^*.

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

$\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, \mathsf{export}\ name\ u_0)$

1. If u_0 is of the case func, then:
   a. Let $\mathsf{func}\ x$ be u_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{func}\ fa^*[x]\}$.
2. If u_0 is of the case global, then:
   a. Let $\mathsf{global}\ x$ be u_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{global}\ ga^*[x]\}$.
3. If u_0 is of the case table, then:
   a. Let $\mathsf{table}\ x$ be u_0.
   b. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{table}\ ta^*[x]\}$.
4. Assert: Due to validation, u_0 is of the case mem.
5. Let $\mathsf{mem}\ x$ be u_0.
6. Return $\{\mathsf{name}\ name, \mathsf{value}\ \mathsf{mem}\ ma^*[x]\}$.

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

$\mathrm{allocmodule}(module, externval^*, {val_g}^*, {ref_t}^*, ({ref_e}^*)^*)$

1. Let ${fa_{ex}}^*$ be $\mathrm{funcsxv}(externval^*)$.
2. Let ${ga_{ex}}^*$ be $\mathrm{globalsxv}(externval^*)$.
3. Let ${ma_{ex}}^*$ be $\mathrm{memsxv}(externval^*)$.
4. Let ${ta_{ex}}^*$ be $\mathrm{tablesxv}(externval^*)$.
5. Assert: Due to validation, $module$ is of the case module.
6. Let $\mathsf{module}\ y_0\ import^*\ func^{n_f}\ y_1\ y_2\ y_3\ y_4\ y_5\ start^?\ export^*$ be $module$.
7. Let $(\mathsf{data}\ byte^*\ datamode)^{n_d}$ be y_5.
8. Let $(\mathsf{elem}\ reftype\ {expr_e}^*\ elemmode)^{n_e}$ be y_4.
9. Let $(\mathsf{memory}\ memtype)^{n_m}$ be y_3.
10. Let $(\mathsf{table}\ tabletype\ expr_t)^{n_t}$ be y_2.
11. Let $(\mathsf{global}\ globaltype\ expr_g)^{n_g}$ be y_1.
12. Let $(\mathsf{type}\ rectype)^*$ be y_0.
13. Let dt^* be $\mathrm{alloctypes}(rectype^*)$.
14. Let fa^* be $(|s.\mathsf{func}| + i_f)^{(i_f<n_f)}$.
15. Let ga^* be $(|s.\mathsf{global}| + i_g)^{(i_g<n_g)}$.
16. Let ta^* be $(|s.\mathsf{table}| + i_t)^{(i_t<n_t)}$.
17. Let ma^* be $(|s.\mathsf{mem}| + i_m)^{(i_m<n_m)}$.
18. Let ea^* be $(|s.\mathsf{elem}| + i_e)^{(i_e<n_e)}$.

19. Let da^* be $(|s.\mathsf{data}| + i_d)^{(i_d<n_d)}$.
20. Let xi^* be $(\mathrm{instexport}({fa_{ex}}^*\ fa^*, {ga_{ex}}^*\ ga^*, {ta_{ex}}^*\ ta^*, {ma_{ex}}^*\ ma^*, export))^*$.
21. Let mm be $\{\mathsf{type}\ dt^*, \mathsf{func}\ {fa_{ex}}^*\ fa^*, \mathsf{global}\ {ga_{ex}}^*\ ga^*, \mathsf{table}\ {ta_{ex}}^*\ ta^*, \mathsf{mem}\ {ma_{ex}}^*\ ma^*, \mathsf{elem}\ ea^*, \mathsf{data}\ da^*, \mathsf{export}\ xi^*\}$.
22. Let y_0 be $\mathrm{allocfuncs}(mm, func^{n_f})$.
23. Assert: Due to validation, y_0 is fa^*.
24. Let y_0 be $\mathrm{allocglobals}(globaltype^{n_g}, {val_g}^*)$.
25. Assert: Due to validation, y_0 is ga^*.
26. Let y_0 be $\mathrm{alloctables}(tabletype^{n_t}, {ref_t}^*)$.
27. Assert: Due to validation, y_0 is ta^*.
28. Let y_0 be $\mathrm{allocmems}(memtype^{n_m})$.
29. Assert: Due to validation, y_0 is ma^*.
30. Let y_0 be $\mathrm{allocelems}(reftype^{n_e}, ({ref_e}^*)^*)$.
31. Assert: Due to validation, y_0 is ea^*.
32. Let y_0 be $\mathrm{allocdatas}((byte^*)^{n_d})$.
33. Assert: Due to validation, y_0 is da^*.
34. Return mm.

$$
\begin{array}{lcll}
\mathrm{allocmodule}(s, module, externval^*, val_g^*, ref_t^*, (ref_e^*)^*) & = & (s_6, mm) & \text{if } module = \mathsf{module}\ (\mathsf{type}\ rectype)^*\ import^*\ func^* \\
&&& \wedge\ fa_{ex}^* = \mathrm{funcs}(externval^*) \\
&&& \wedge\ ga_{ex}^* = \mathrm{globals}(externval^*) \\
&&& \wedge\ ta_{ex}^* = \mathrm{tables}(externval^*) \\
&&& \wedge\ ma_{ex}^* = \mathrm{mems}(externval^*) \\
&&& \wedge\ fa^* = |s.\mathsf{func}| + i_f^{i_f<n_f} \\
&&& \wedge\ ga^* = |s.\mathsf{global}| + i_g^{i_g<n_g} \\
&&& \wedge\ ta^* = |s.\mathsf{table}| + i_t^{i_t<n_t} \\
&&& \wedge\ ma^* = |s.\mathsf{mem}| + i_m^{i_m<n_m} \\
&&& \wedge\ ea^* = |s.\mathsf{elem}| + i_e^{i_e<n_e} \\
&&& \wedge\ da^* = |s.\mathsf{data}| + i_d^{i_d<n_d} \\
&&& \wedge\ xi^* = \mathrm{instexport}(fa_{ex}^*\ fa^*, ga_{ex}^*\ ga^*, ta_{ex}^*\ ta^*, \\
&&& \wedge\ mm = \{\mathsf{type}\ dt^*, \\
&&& \qquad\qquad \mathsf{func}\ fa_{ex}^*\ fa^*, \\
&&& \qquad\qquad \mathsf{global}\ ga_{ex}^*\ ga^*, \\
&&& \qquad\qquad \mathsf{table}\ ta_{ex}^*\ ta^*, \\
&&& \qquad\qquad \mathsf{mem}\ ma_{ex}^*\ ma^*, \\
&&& \qquad\qquad \mathsf{elem}\ ea^*, \\
&&& \qquad\qquad \mathsf{data}\ da^*, \\
&&& \qquad\qquad \mathsf{export}\ xi^*\} \\
&&& \wedge\ dt^* = \mathrm{alloctypes}(rectype^*) \\
&&& \wedge\ (s_1, fa^*) = \mathrm{allocfuncs}(s, mm, func^{n_f}) \\
&&& \wedge\ (s_2, ga^*) = \mathrm{allocglobals}(s_1, globaltype^{n_g}, val_g^*) \\
&&& \wedge\ (s_3, ta^*) = \mathrm{alloctables}(s_2, tabletype^{n_t}, ref_t^*) \\
&&& \wedge\ (s_4, ma^*) = \mathrm{allocmems}(s_3, memtype^{n_m}) \\
&&& \wedge\ (s_5, ea^*) = \mathrm{allocelems}(s_4, reftype^{n_e}, (ref_e^*)^*) \\
&&& \wedge\ (s_6, da^*) = \mathrm{allocdatas}(s_5, (byte^*)^{n_d})
\end{array}
$$

2.5.2 Instantiation

$\mathrm{inst\_reftype}(mm, rt)$

1. Let dt^* be mm.type.
2. Return $\mathrm{subst\_all\_reftype}(rt, dt^*)$.

$$\mathrm{inst}_{mm}(rt) \quad = \quad rt[:= dt^*] \quad \text{if } dt^* = mm.\mathsf{type}$$

$\mathrm{concat\_instr}({u_0}^*)$

1. If ${u_0}^*$ is ϵ, then:
   a. Return ϵ.
2. Let $instr^*$ $({instr'}^*)^*$ be ${u_0}^*$.
3. Return $instr^*$ $\mathrm{concat\_instr}(({instr'}^*)^*)$.

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

$\mathrm{rundata}(\mathsf{data}\ byte^*\ u_0, y)$

1. If u_0 is passive, then:
   a. Return ϵ.
2. Assert: Due to validation, u_0 is of the case active.
3. Let active x $instr^*$ be u_0.
4. Return $instr^*$ i32.const 0 i32.const $|byte^*|$ memory.init x y data.drop y.

$$\begin{array}{lcl} \mathrm{rundata}(\mathsf{data}\ byte^*\ (\mathsf{passive}),\ y) & = & \epsilon \\ \mathrm{rundata}(\mathsf{data}\ byte^*\ (\mathsf{active}\ x\ instr^*),\ y) & = & instr^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ |byte^*|)\ (\mathsf{memory.init}\ x\ y)\ (\mathsf{data.drop}\ y) \end{array}$$

$\mathrm{runelem}(\mathsf{elem}\ reftype\ expr^*\ u_0, y)$

1. If u_0 is passive, then:
   a. Return ϵ.
2. If u_0 is declare, then:
   a. Return elem.drop y.
3. Assert: Due to validation, u_0 is of the case active.
4. Let active x $instr^*$ be u_0.
5. Return $instr^*$ i32.const 0 i32.const $|expr^*|$ table.init x y elem.drop y.

$$\mathrm{runelem}(\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}^*\ (\mathsf{passive}),\ y) = \epsilon$$
$$\mathrm{runelem}(\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}^*\ (\mathsf{declare}),\ y) = (\mathsf{elem.drop}\ y)$$
$$\mathrm{runelem}(\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}^*\ (\mathsf{active}\ x\ \mathit{instr}^*),\ y) = \mathit{instr}^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ |\mathit{expr}^*|)\ (\mathsf{table.init}\ x\ y)\ (\mathsf{elem.drop}\ y)$$

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

1. Assert: Due to validation, module is of the case module.
2. Let $\mathsf{module}\ y_0\ \mathit{import}^*\ \mathit{func}^{n_{func}}\ \mathit{global}^*\ \mathit{table}^*\ \mathit{mem}^*\ \mathit{elem}^*\ \mathit{data}^*\ \mathit{start}^?\ \mathit{export}^*$ be module.
3. Let $(\mathsf{type}\ \mathit{rectype})^*$ be y_0.
4. Let n_d be the length of data^*.
5. Let n_e be the length of elem^*.
6. Let $(\mathsf{start}\ x)^?$ be $\mathit{start}^?$.
7. Let mm_{init} be $\{\mathsf{type}\ \mathrm{alloctypes}(\mathit{rectype}^*), \mathsf{func}\ \mathrm{funcsxv}(\mathit{externval}^*)\ (|s.\mathsf{func}| + i_{func})^{(i_{func}<n_{func})}, \mathsf{global}\ \mathrm{globalsxv}(\mathit{ext}$
8. Let $(\mathsf{global}\ \mathit{globaltype}\ \mathit{expr}_g)^*$ be global^*.
9. Let $(\mathsf{table}\ \mathit{tabletype}\ \mathit{expr}_t)^*$ be table^*.
10. Let $(\mathsf{elem}\ \mathit{reftype}\ {\mathit{expr}_e}^*\ \mathit{elemmode})^*$ be elem^*.
11. Let ${\mathit{instr}_d}^*$ be $\mathrm{concat\_instr}((\mathrm{rundata}(\mathit{data}^*[j], j))^{(j<n_d)})$.
12. Let ${\mathit{instr}_e}^*$ be $\mathrm{concat\_instr}((\mathrm{runelem}(\mathit{elem}^*[i], i))^{(i<n_e)})$.
13. Let z be $s\ \{\mathsf{local}\ \epsilon, \mathsf{module}\ \mathit{mm}_{init}\}$.
14. Let f be z.
15. Push the activation of f to the stack.
16. Let $(\mathit{val}_g)^*$ be $(\mathit{eval}_{expr}(\mathit{expr}_g))^*$.
17. Pop the activation of f from the stack.
18. Let f be z.
19. Push the activation of f to the stack.
20. Let $(\mathit{ref}_t)^*$ be $(\mathit{eval}_{expr}(\mathit{expr}_t))^*$.
21. Pop the activation of f from the stack.
22. Let f be z.
23. Push the activation of f to the stack.
24. Let $((\mathit{ref}_e)^*)^*$ be $((\mathit{eval}_{expr}(\mathit{expr}_e))^*)^*$.
25. Pop the activation of f from the stack.
26. Let mm be $\mathrm{allocmodule}(\mathit{module}, \mathit{externval}^*, {\mathit{val}_g}^*, {\mathit{ref}_t}^*, ({\mathit{ref}_e}^*)^*)$.
27. Let f be $\{\mathsf{local}\ \epsilon, \mathsf{module}\ \mathit{mm}\}$.
28. Enter the activation of f with arity 0 with label FRAME:
   a. Execute the sequence ${\mathit{instr}_e}^*$.
   b. Execute the sequence ${\mathit{instr}_d}^*$.
   c. If x is defined, then:
      1) Let ${x_0}^?$ be x.

2) Execute call x_0.

29. Return mm.

$$
\begin{array}{lcll}
\mathrm{instantiate}(s,\ module,\ externval^*) & = & s'; f; instr_e^*\ instr_d^*\ (\mathsf{call}\ x)^? & \text{if } module = \mathsf{module}\ (\mathsf{type}\ rectype)^*\ import^*\ func^{n_{func}} \\
& & & \wedge\ global^* = (\mathsf{global}\ globaltype\ expr_g)^* \\
& & & \wedge\ table^* = (\mathsf{table}\ tabletype\ expr_t)^* \\
& & & \wedge\ elem^* = (\mathsf{elem}\ reftype\ expr_e^*\ elemmode)^* \\
& & & \wedge\ start^? = (\mathsf{start}\ x)^? \\
& & & \wedge\ n_e = |elem^*| \\
& & & \wedge\ n_d = |data^*| \\
& & & \wedge\ mm_{init} = \{\mathsf{type}\ \mathrm{alloctypes}(rectype^*), \\
& & & \qquad \mathsf{func}\ \mathrm{funcs}(externval^*)\ |s.\mathsf{func}| + i_{func}^{i_{func}<} \\
& & & \qquad \mathsf{global}\ \mathrm{globals}(externval^*), \\
& & & \qquad \mathsf{table}\ \epsilon, \\
& & & \qquad \mathsf{mem}\ \epsilon, \\
& & & \qquad \mathsf{elem}\ \epsilon, \\
& & & \qquad \mathsf{data}\ \epsilon, \\
& & & \qquad \mathsf{export}\ \epsilon\} \\
& & & \wedge\ z = s; \{\mathsf{module}\ mm_{init}\} \\
& & & \wedge\ (z; expr_g \hookrightarrow^* z; val_g)^* \\
& & & \wedge\ (z; expr_t \hookrightarrow^* z; ref_t)^* \\
& & & \wedge\ (z; expr_e \hookrightarrow^* z; ref_e)^{**} \\
& & & \wedge\ (s',\ mm) = \mathrm{allocmodule}(s,\ module,\ externval^*,\ va \\
& & & \wedge\ f = \{\mathsf{module}\ mm\} \\
& & & \wedge\ instr_e^* = \mathrm{concat}_{instr}(\mathrm{runelem}(elem^*[i],\ i)^{i<n_e}) \\
& & & \wedge\ instr_d^* = \mathrm{concat}_{instr}(\mathrm{rundata}(data^*[j],\ j)^{j<n_d})
\end{array}
$$

2.5.3 Invocation

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

1. Let mm be $\{$type s.func$[fa]$.type, func ϵ, global ϵ, table ϵ, mem ϵ, elem ϵ, data ϵ, export $\epsilon\}$.
2. Assert: Due to validation, $\mathrm{expanddt}(s.\mathsf{func}[fa].\mathsf{type})$ is of the case func.
3. Let func y_0 be $\mathrm{expanddt}(s.\mathsf{func}[fa].\mathsf{type})$.
4. Let ${t_1}^n \rightarrow {t_2}^*$ be y_0.
5. Let f be $\{$local ϵ, module $mm\}$.
6. Assert: Due to validation, $\mathrm{funcinst}()[fa]$.code is of the case func.
7. Let k be the length of ${t_2}^*$.
8. Enter the activation of f with arity k with label $FRAME$:
   a. Push val^n to the stack.
   b. Push ref.func_addr fa to the stack.
   c. Execute call_ref $0^?$.
9. Pop val^k from the stack.
10. Return val^k.

$$
\begin{array}{rcll}
\mathrm{invoke}(s,\,fa,\,val^n) & = & s;f;val^n\ (\mathsf{ref.func}\ fa)\ (\mathsf{call\_ref}\ 0) & \mathrm{if}\ mm = \{\mathsf{type}\ s.\mathsf{func}[fa].\mathsf{type}, \\
& & & \qquad\qquad \mathsf{func}\ \epsilon, \\
& & & \qquad\qquad \mathsf{global}\ \epsilon, \\
& & & \qquad\qquad \mathsf{table}\ \epsilon, \\
& & & \qquad\qquad \mathsf{mem}\ \epsilon, \\
& & & \qquad\qquad \mathsf{elem}\ \epsilon, \\
& & & \qquad\qquad \mathsf{data}\ \epsilon, \\
& & & \qquad\qquad \mathsf{export}\ \epsilon\} \\
& & & \wedge\ f = \{\mathsf{module}\ mm\} \\
& & & \wedge\ (s;f).\mathsf{func}[fa].\mathsf{code} = \mathsf{func}\ x\ local^*\ expr \\
& & & \wedge\ s.\mathsf{func}[fa].\mathsf{type} \approx \mathsf{func}\ (t_1^n \to t_2^*)
\end{array}
$$

2.5.4 Address Getters

funcaddr()

1. Let f be the current frame.
2. Return f.module.func.

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

2.5.5 Instance Getters

funcinst()

1. Return s.func.

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

globalinst()

1. Return s.global.

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

tableinst()

1. Return s.table.

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

meminst()

1. Return s.mem.

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

eleminst()

1. Return s.elem.

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

datainst()

1. Return s.data.

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

structinst()

1. Return s.struct.

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

arrayinst()

1. Return s.array.

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

moduleinst()

1. Let f be the current frame.
2. Return f.module.

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

2.5.6 Getters

type(x)

1. Let f be the current frame.
2. Return f.module.type[x].

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

func(x)

1. Let f be the current frame.
2. Return s.func[f.module.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.global[f.module.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.table[f.module.table[x]].

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

$\mathrm{mem}(x)$

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

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

$\mathrm{elem}(x)$

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

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

$\mathrm{data}(x)$

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 $f.\mathsf{local}[x]$.

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

2.5.7 Setters

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

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

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

$\mathrm{with\_locals}(C, u_0{}^*, u_1{}^*)$

1. If $u_0{}^*$ is ϵ and $u_1{}^*$ is ϵ, then:
   a. Return C.
2. Assert: Due to validation, the length of $u_1{}^*$ is greater than or equal to 1.
3. Let $lt_1\ lt^*$ be $u_1{}^*$.
4. Assert: Due to validation, the length of $u_0{}^*$ is greater than or equal to 1.
5. Let $x_1\ x^*$ be $u_0{}^*$.
6. Return $\mathrm{with\_locals}(C with.\mathsf{local}[x_1] replacedby lt_1, x^*, lt^*)$.

$$\begin{array}{lcl} C[\mathsf{local}[\epsilon] = \epsilon] & = & C \\ C[\mathsf{local}[x_1\ x^*] = lt_1\ lt^*] & = & C[\mathsf{local}[x_1] = lt_1][\mathsf{local}[x^*] = lt^*] \end{array}$$

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

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^*.

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

$\mathrm{with\_array}(a, i, fv)$

1. Replace $s.\mathsf{array}[a].\mathsf{field}[i]$ with fv.

$$\mathrm{with}_{array}((s; f),\ a,\ i,\ fv) \quad = \quad s[\mathsf{array}[a].\mathsf{field}[i] = fv]; f$$

$\text{with\_struct}(a, i, fv)$

1. Replace $s.\mathsf{struct}[a].\mathsf{field}[i]$ with fv.

$$\text{with}_{struct}((s; f),\ a,\ i,\ fv) \quad = \quad s[\mathsf{struct}[a].\mathsf{field}[i] = fv]; f$$

$\text{growtable}(ti, n, r)$

1. Let $\{\mathsf{type}\ i\ j\ rt, \mathsf{elem}\ r'^*\}$ be ti.
2. Let i' be $|r'^*| + n$.
3. If i' is less than or equal to j, then:
   a. Let ti' be $\{\mathsf{type}\ i'\ j\ rt, \mathsf{elem}\ r'^*\ r^n\}$.
   b. Return ti'.

$$\begin{array}{lcll} \text{growtable}(ti,\ n,\ r) & = & ti' & \text{if } ti = \{\mathsf{type}\ ([i..j]\ rt),\ \mathsf{elem}\ r'^*\} \\ & & & \wedge\ i' = |r'^*| + n \\ & & & \wedge\ ti' = \{\mathsf{type}\ ([i'..j]\ rt),\ \mathsf{elem}\ r'^*\ r^n\} \\ & & & \wedge\ i' \leq j \end{array}$$

$\text{growmemory}(mi, n)$

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

$$\begin{array}{lcll} \text{growmemory}(mi,\ n) & = & mi' & \text{if } mi = \{\mathsf{type}\ ([i..j]\ \mathsf{i8}),\ \mathsf{data}\ b^*\} \\ & & & \wedge\ i' = |b^*|/(64 \cdot \mathrm{Ki}) + n \\ & & & \wedge\ mi' = \{\mathsf{type}\ ([i'..j]\ \mathsf{i8}),\ \mathsf{data}\ b^*\ 0^{n \cdot 64 \cdot \mathrm{Ki}}\} \\ & & & \wedge\ i' \leq j \end{array}$$