WebAssembly Specification

Release 2.0 (Auto-generated Draft 2024-04-03)

Anonymous Authors

Apr 03, 2024

Contents

CHAPTER 1

Introduction

This automatically generated document describes version 2.0 of the core WebAssembly standard. It defines the abstract syntax, validation in formal and prose notation, and execution semantics in formal and prose notation. It omits all editorial text from the official document that explains the meaning of definitions and notation.

CHAPTER 2

Structure

2.1 Values

2.1.1 Bytes

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

2.1.2 Integers

$$\begin{array}{llcl}
\text{(unsigned integer)} & uN & ::= & 0 \mid \ldots \mid 2^N - 1 \\
\text{(signed integer)} & sN & ::= & -2^{N-1} \mid \ldots \mid -1 \mid 0 \mid +1 \mid \ldots \mid 2^{N-1} - 1 \\
\text{(integer)} & iN & ::= & uN \\
 & u31 & ::= & u31 \\
 & u32 & ::= & u32 \\
 & u64 & ::= & u64 \\
 & u128 & ::= & u128 \\
 & s33 & ::= & s33
\end{array}$$

2.1.3 Floating-Point

$$\begin{array}{llcll}
\text{(floating-point number)} & fN & ::= & +fN\,mag \mid -fN\,mag & \\
\text{(floating-point magnitude)} & fN\,mag & ::= & (1 + m \cdot 2^{-M}) \cdot 2^n & \text{if } m < 2^M \wedge 2 - 2^{E-1} \leq n \leq 2^{E-1} - 1 \\
 & & \mid & (0 + m \cdot 2^{-M}) \cdot 2^n & \text{if } m < 2^M \wedge 2 - 2^{E-1} = n \\
 & & \mid & \infty & \\
 & & \mid & \mathsf{nan}\,m & \text{if } 1 \leq m < 2^M \\
 & f32 & ::= & f32 & \\
 & f64 & ::= & f64 &
\end{array}$$

$+0$

1. Return $(+((0 + 0 \cdot 2^{-M}) \cdot 2^n))$.

$$+0 \quad = \quad +((0 + 0 \cdot 2^{-M}) \cdot 2^n)$$

$\mathrm{signif}(N_{u0})$

1. If N_{u0} is 32, then:
   a. Return 23.
2. Assert: Due to validation, N_{u0} is 64.
3. Return 52.

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

M

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

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

$\mathrm{expon}(N_{u0})$

1. If N_{u0} is 32, then:
   a. Return 8.
2. Assert: Due to validation, N_{u0} is 64.
3. Return 11.

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

E

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

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

2.1.4 Names

$$\begin{array}{llcll}
\text{(name)} & \mathit{name} & ::= & \mathit{char}^* & \text{if } |\mathrm{utf8}(\mathit{char}^*)| < 2^{32} \\
\text{(character)} & \mathit{char} & ::= & \text{U+00} \mid \dots \mid \text{U+D7FF} \mid \text{U+E000} \mid \dots \mid \text{U+10FFFF} &
\end{array}$$

$\mathrm{utf8}(\mathit{char}^*_{u0})$

1. If $|\mathit{char}^*_{u0}|$ is 1, then:
   a. Let ch be char^*_{u0}.
   b. If ch is less than 128, then:
      1) Let b be ch.
      2) Return b.
   c. If 128 is less than or equal to ch and ch is less than 2048 and ch is greater than or equal to $b_2 - 128$, then:
      1) Let $2^6 \cdot b_1 - 192$ be $\mathit{ch} - b_2 - 128$.
      2) Return $b_1\ b_2$.
   d. If 2048 is less than or equal to ch and ch is less than 55296 or 57344 is less than or equal to ch and ch is less than 65536 and ch 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 $\mathit{ch} - b_3 - 128$.
      2) Return $b_1\ b_2\ b_3$.
   e. If 65536 is less than or equal to ch and ch is less than 69632 and ch 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 $\mathit{ch} - b_4 - 128$.
      2) Return $b_1\ b_2\ b_3\ b_4$.
2. Let ch^* be char^*_{u0}.
3. Return $\mathrm{concat}_{\mathrm{utf8}(\mathit{ch})^*}$.

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

2.2 Types

2.2.1 Number Types

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

2.2.2 Vector Types

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

2.2.3 Heap Types

$$\begin{array}{llll} & \mathit{absheaptype} & ::= & \mathsf{any} \mid \mathsf{eq} \mid \mathsf{i31} \mid \mathsf{struct} \mid \mathsf{array} \mid \mathsf{none} \\ & & | & \mathsf{func} \mid \mathsf{nofunc} \\ & & | & \mathsf{extern} \mid \mathsf{noextern} \\ & & | & \mathsf{bot} \\ \text{(heap type)} & \mathit{heaptype} & ::= & \mathit{absheaptype} \mid \mathit{typeuse} \end{array}$$

2.2.4 Reference Types

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

2.2.5 Value Types

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

2.2.6 Result Types

$$\begin{array}{lll} \mathit{resulttype} & ::= & \mathit{list}(\mathit{valtype}) \end{array}$$

2.2.7 Function Types

$$\begin{array}{lll} \mathit{functype} & ::= & \mathit{resulttype} \rightarrow \mathit{resulttype} \end{array}$$

2.2.8 Aggregate Types

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

2.2.9 Composite Types

$$\begin{array}{llll} & \mathit{comptype} & ::= & \mathsf{struct}~\mathit{structtype} \\ & & | & \mathsf{array}~\mathit{arraytype} \\ & & | & \mathsf{func}~\mathit{functype} \end{array}$$

2.2.10 Recursive Types

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

2.2.11 Limits

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

2.2.12 Memory Types

$$\mathit{memtype} ::= \mathit{limits}~\mathsf{i8}$$

2.2.13 Table Types

$$\mathit{tabletype} ::= \mathit{limits}~\mathit{reftype}$$

2.2.14 Global Types

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

2.2.15 Element Types

$$\mathit{elemtype} ::= \mathit{reftype}$$

2.2.16 Data Types

$$\mathit{datatype} ::= \mathsf{ok}$$

2.2.17 External Types

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

2.3 Instructions

2.3.1 Numeric Instructions

$$
\begin{array}{llcll}
 & \mathsf{i}n & ::= & \mathsf{i32} \mid \mathsf{i64} & \\
 & \mathsf{f}n & ::= & \mathsf{f32} \mid \mathsf{f64} & \\
\text{(signedness)} & \mathit{sx} & ::= & \mathsf{u} \mid \mathsf{s} & \\
\text{(instruction)} & \mathit{instr} & ::= & \dots & \\
 & & \mid & \mathit{numtype}.\mathsf{const}~\mathit{num}_{\mathit{numtype}} & \\
 & & \mid & \mathit{numtype}.\mathit{unop}_{\mathit{numtype}} & \\
 & & \mid & \mathit{numtype}.\mathit{binop}_{\mathit{numtype}} & \\
 & & \mid & \mathit{numtype}.\mathit{testop}_{\mathit{numtype}} & \\
 & & \mid & \mathit{numtype}.\mathit{relop}_{\mathit{numtype}} & \\
 & & \mid & \mathit{numtype}_1.\mathit{cvtop}\_\mathit{numtype}_2\_\mathit{sx}^? & \text{if } \mathit{numtype}_1 \neq \mathit{numtype}_2 \\
 & & \mid & \mathit{numtype}.\mathsf{extend}n\_\mathsf{s} & \\
 & & \mid & \dots & \\
 & \mathit{unop}_{\mathsf{i}n} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \mid \mathsf{extend}~n & \\
 & \mathit{unop}_{\mathsf{f}n} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} & \\
 & \mathit{binop}_{\mathsf{i}n} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}\_\mathit{sx} \mid \mathsf{rem}\_\mathit{sx} & \\
 & & \mid & \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{f}n} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} & \\
 & \mathit{testop}_{\mathsf{i}n} & ::= & \mathsf{eqz} & \\
 & \mathit{relop}_{\mathsf{i}n} & ::= & \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{relop}_{\mathsf{f}n} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} & \\
 & \mathit{cvtop} & ::= & \mathsf{convert} \mid \mathsf{convert\_sat} \mid \mathsf{reinterpret} &
\end{array}
$$

2.3.2 Reference Instructions

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

2.3.3 Aggregate Instructions

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

2.3.4 Variable Instructions

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

2.3.5 Table Instructions

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

2.3.6 Memory Instructions

$$
\begin{array}{llcll}
\text{(instruction)} & \mathit{instr} & ::= & \dots \\
& & | & \mathit{numtype}.\mathsf{load}(\mathit{w\_sx})^?~\mathit{memidx}~\mathit{memop} & \text{if } (\mathit{numtype} = \mathsf{i}\mathit{n} \wedge w < |\mathsf{i}\mathit{n}|)^? \\
& & | & \mathit{numtype}.\mathsf{store}w^?~\mathit{memidx}~\mathit{memop} & \text{if } (\mathit{numtype} = \mathsf{i}\mathit{n} \wedge w < |\mathsf{i}\mathit{n}|)^? \\
& & | & \mathsf{v128.load}\mathit{vloadop}^?~\mathit{memidx}~\mathit{memop} \\
& & | & \mathsf{v128.load}w\mathsf{\_lane}~\mathit{memidx}~\mathit{memop}~\mathit{laneidx} \\
& & | & \mathsf{v128.store}~\mathit{memidx}~\mathit{memop} \\
& & | & \mathsf{v128.store}w\mathsf{\_lane}~\mathit{memidx}~\mathit{memop}~\mathit{laneidx} \\
& & | & \mathsf{memory.size}~\mathit{memidx} \\
& & | & \mathsf{memory.grow}~\mathit{memidx} \\
& & | & \mathsf{memory.fill}~\mathit{memidx} \\
& & | & \mathsf{memory.copy}~\mathit{memidx}~\mathit{memidx} \\
& & | & \mathsf{memory.init}~\mathit{memidx}~\mathit{dataidx} \\
& & | & \dots \\
\text{(instruction)} & \mathit{instr} & ::= & \dots \\
& & | & \mathsf{data.drop}~\mathit{dataidx} \\
\text{(memory operator)} & \mathit{memop} & ::= & \{\mathsf{align}~\mathit{u32},~\mathsf{offset}~\mathit{u32}\}
\end{array}
$$

%{definition-prose: memop0}

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

2.3.7 Control Instructions

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

2.3.8 Expressions

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

2.4 Modules

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

2.4.1 Indices

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

2.4.2 Types

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

2.4.3 Functions

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

2.4.4 Tables

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

2.4.5 Memories

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

2.4.6 Globals

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

2.4.7 Element Segments

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

2.4.8 Data Segments

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

2.4.9 Start Function

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

2.4.10 Exports

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

2.4.11 Imports

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

CHAPTER 3

Validation

3.1 Conventions

3.1.1 Types

$(\mathsf{ref}\ nul_1\ ht_1) \setminus (\mathsf{ref}\ (\mathsf{null}\ uo^?)\ ht_2)$

1. If $uo^?$ is $()$, then:
   a. Return $(\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ ht_1)$.
2. Assert: Due to validation, $uo^?$ is not defined.
3. Return $(\mathsf{ref}\ nul_1\ ht_1)$.

$$
\begin{array}{lcl}
(\mathsf{ref}\ nul_1\ ht_1) \setminus (\mathsf{ref}\ \mathsf{null}\ ht_2) & = & (\mathsf{ref}\ \epsilon\ ht_1) \\
(\mathsf{ref}\ nul_1\ ht_1) \setminus (\mathsf{ref}\ \epsilon\ ht_2) & = & (\mathsf{ref}\ nul_1\ ht_1)
\end{array}
$$

3.1.2 Injection

$$
\begin{array}{lcl}
typevar & ::= & typeidx \mid \mathsf{rec}\ nat
\end{array}
$$

x

1. Return x.

$$
x \quad = \quad x
$$

3.1.3 Defined Types

$$deftype \quad ::= \quad rectype.nat$$

3.1.4 Unpacking

$\mathrm{unpack}(stora_{u0})$

1. If the type of $stora_{u0}$ is valtype, then:
   a. Let $valtype$ be $stora_{u0}$.
   b. Return $valtype$.
2. Assert: Due to validation, the type of $stora_{u0}$ is packtype.
3. Return i32.

$$\begin{array}{lcl} \mathrm{unpack}(valtype) & = & valtype \\ \mathrm{unpack}(packtype) & = & \mathsf{i32} \end{array}$$

$\mathrm{unpack}(stora_{u0})$

1. If the type of $stora_{u0}$ is numtype, then:
   a. Let $numtype$ be $stora_{u0}$.
   b. Return $numtype$.
2. If the type of $stora_{u0}$ is packtype, then:
   a. Return i32.

$$\begin{array}{lcl} \mathrm{unpack}(numtype) & = & numtype \\ \mathrm{unpack}(packtype) & = & \mathsf{i32} \end{array}$$

3.1.5 Substitutions

$xx[typev^*_{u0} := typeu^*_{u1}]$

1. If $typev^*_{u0}$ is ϵ and $typeu^*_{u1}$ is ϵ, then:
   a. Return xx.
2. Assert: Due to validation, $|typeu^*_{u1}|$ is greater than or equal to 1.
3. Let $tu_1\ tu'^*$ be $typeu^*_{u1}$.
4. If $|typev^*_{u0}|$ is greater than or equal to 1, then:
   a. Let $xx_1\ xx'^*$ be $typev^*_{u0}$.
   b. If xx is xx_1, then:
      1) Return tu_1.
5. Let $tu_1\ tu'^*$ be $typeu^*_{u1}$.

6. Assert: Due to validation, $|typev^*_{u0}|$ is greater than or equal to 1.
7. Let $xx_1\ xx'^*$ be $typev^*_{u0}$.
8. Return $xx[xx'^* := tu'^*]$.

$$\begin{array}{lcll}
xx[\epsilon := \epsilon] & = & xx & \\
xx[xx_1\ xx'^* := tu_1\ tu'^*] & = & tu_1 & \text{if } xx = xx_1 \\
xx[xx_1\ xx'^* := tu_1\ tu'^*] & = & xx[xx'^* := tu'^*] & \text{otherwise}
\end{array}$$

$nt[xx^* := tu^*]$

1. Return nt.

$$nt[xx^* := tu^*] \quad = \quad nt$$

$vt[xx^* := tu^*]$

1. Return vt.

$$vt[xx^* := tu^*] \quad = \quad vt$$

$heapt_{u0}[xx^* := tu^*]$

1. If the type of $heapt_{u0}$ is typevar, then:
   a. Let xx' be $heapt_{u0}$.
   b. Return $xx'[xx^* := tu^*]$.
2. If the type of $heapt_{u0}$ is deftype, then:
   a. Let dt be $heapt_{u0}$.
   b. Return $dt[xx^* := tu^*]$.
3. Let ht be $heapt_{u0}$.
4. Return ht.

$$\begin{array}{lcll}
xx'[xx^* := tu^*] & = & xx'[xx^* := tu^*] & \\
dt[xx^* := tu^*] & = & dt[xx^* := tu^*] & \\
ht[xx^* := tu^*] & = & ht & \text{otherwise}
\end{array}$$

$(\mathsf{ref}\ nul\ ht)[xx^* := tu^*]$

1. Return $(\mathsf{ref}\ nul\ ht[xx^* := tu^*])$.

$$(\mathsf{ref}\ nul\ ht)[xx^* := tu^*] \quad = \quad \mathsf{ref}\ nul\ ht[xx^* := tu^*]$$

$valty_{u0}[xx^* := tu^*]$

1. If the type of $valty_{u0}$ is numtype, then:
   a. Let nt be $valty_{u0}$.
   b. Return $nt[xx^* := tu^*]$.
2. If the type of $valty_{u0}$ is vectype, then:
   a. Let vt be $valty_{u0}$.
   b. Return $vt[xx^* := tu^*]$.
3. If the type of $valty_{u0}$ is reftype, then:
   a. Let rt be $valty_{u0}$.
   b. Return $rt[xx^* := tu^*]$.
4. Assert: Due to validation, $valty_{u0}$ is bot.
5. Return bot.

$$\begin{array}{lcl} nt[xx^* := tu^*] & = & nt[xx^* := tu^*] \\ vt[xx^* := tu^*] & = & vt[xx^* := tu^*] \\ rt[xx^* := tu^*] & = & rt[xx^* := tu^*] \\ \mathsf{bot}[xx^* := tu^*] & = & \mathsf{bot} \end{array}$$

$pt[xx^* := tu^*]$

1. Return pt.

$$pt[xx^* := tu^*] \quad = \quad pt$$

$stora_{u0}[xx^* := tu^*]$

1. If the type of $stora_{u0}$ is valtype, then:
   a. Let t be $stora_{u0}$.
   b. Return $t[xx^* := tu^*]$.
2. Assert: Due to validation, the type of $stora_{u0}$ is packtype.
3. Let pt be $stora_{u0}$.
4. Return $pt[xx^* := tu^*]$.

$$\begin{array}{lcl} t[xx^* := tu^*] &=& t[xx^* := tu^*] \\ pt[xx^* := tu^*] &=& pt[xx^* := tu^*] \end{array}$$

$(mut,\ zt)[xx^* := tu^*]$

1. Return $(mut,\ zt[xx^* := tu^*])$.

$$(mut\ zt)[xx^* := tu^*] \quad = \quad mut\ zt[xx^* := tu^*]$$

$compt_{u0}[xx^* := tu^*]$

1. If $compt_{u0}$ is of the case struct, then:
   a. Let $(\mathsf{struct}\ yt^*)$ be $compt_{u0}$.
   b. Return $(\mathsf{struct}\ yt[xx^* := tu^*]^*)$.
2. If $compt_{u0}$ is of the case array, then:
   a. Let $(\mathsf{array}\ yt)$ be $compt_{u0}$.
   b. Return $(\mathsf{array}\ yt[xx^* := tu^*])$.
3. Assert: Due to validation, $compt_{u0}$ is of the case func.
4. Let $(\mathsf{func}\ ft)$ be $compt_{u0}$.
5. Return $(\mathsf{func}\ ft[xx^* := tu^*])$.

$$\begin{array}{lcl} (\mathsf{struct}\ yt^*)[xx^* := tu^*] &=& \mathsf{struct}\ yt[xx^* := tu^*]^* \\ (\mathsf{array}\ yt)[xx^* := tu^*] &=& \mathsf{array}\ yt[xx^* := tu^*] \\ (\mathsf{func}\ ft)[xx^* := tu^*] &=& \mathsf{func}\ ft[xx^* := tu^*] \end{array}$$

$(\mathsf{sub}\ fin\ tu'^*\ ct)[xx^* := tu^*]$

1. Return $(\mathsf{sub}\ fin\ tu'[xx^* := tu^*]^*\ ct[xx^* := tu^*])$.

$$(\mathsf{sub}\ fin\ tu'^*\ ct)[xx^* := tu^*] \quad = \quad \mathsf{sub}\ fin\ tu'[xx^* := tu^*]^*\ ct[xx^* := tu^*]$$

$(\mathsf{rec}\ st^*)[xx^* := tu^*]$

1. Return $(\mathsf{rec}\ st[xx^* := tu^*]^*)$.

$$(\mathsf{rec}\ st^*)[xx^* := tu^*] \quad = \quad \mathsf{rec}\ st[xx^* := tu^*]^*$$

$(qt.i)[xx^* := tu^*]$

1. Return $(qt[xx^* := tu^*].i)$.

$$(qt.i)[xx^* := tu^*] \quad = \quad qt[xx^* := tu^*].i$$

$(mut,\ t)[xx^* := tu^*]$

1. Return $(mut,\ t[xx^* := tu^*])$.

$$(mut\ t)[xx^* := tu^*] \quad = \quad mut\ t[xx^* := tu^*]$$

$t_1^* \rightarrow t_2^*[xx^* := tu^*]$

1. Return $t_1[xx^* := tu^*]^* \rightarrow t_2[xx^* := tu^*]^*$.

$$(t_1^* \rightarrow t_2^*)[xx^* := tu^*] \quad = \quad t_1[xx^* := tu^*]^* \rightarrow t_2[xx^* := tu^*]^*$$

$(lim,\ rt)[xx^* := tu^*]$

1. Return $(lim,\ rt[xx^* := tu^*])$.

$$(lim\ rt)[xx^* := tu^*] \quad = \quad lim\ rt[xx^* := tu^*]$$

$(\mathsf{i8}\ lim)[xx^* := tu^*]$

1. Return $(\mathsf{i8}\ lim)$.

$$(lim\ \mathsf{i8})[xx^* := tu^*] \quad = \quad lim\ \mathsf{i8}$$

$exter_{u0}[xx^* := tu^*]$

1. If $exter_{u0}$ is of the case func, then:
   a. Let $(\mathsf{func}\ dt)$ be $exter_{u0}$.
   b. Return $(\mathsf{func}\ dt[xx^* := tu^*])$.
2. If $exter_{u0}$ is of the case global, then:
   a. Let $(\mathsf{global}\ gt)$ be $exter_{u0}$.
   b. Return $(\mathsf{global}\ gt[xx^* := tu^*])$.
3. If $exter_{u0}$ is of the case table, then:
   a. Let $(\mathsf{table}\ tt)$ be $exter_{u0}$.
   b. Return $(\mathsf{table}\ tt[xx^* := tu^*])$.
4. Assert: Due to validation, $exter_{u0}$ is of the case mem.
5. Let $(\mathsf{mem}\ mt)$ be $exter_{u0}$.
6. Return $(\mathsf{mem}\ mt[xx^* := tu^*])$.

$$\begin{array}{lcl}
(\mathsf{func}\ dt)[xx^* := tu^*] &=& \mathsf{func}\ dt[xx^* := tu^*] \\
(\mathsf{global}\ gt)[xx^* := tu^*] &=& \mathsf{global}\ gt[xx^* := tu^*] \\
(\mathsf{table}\ tt)[xx^* := tu^*] &=& \mathsf{table}\ tt[xx^* := tu^*] \\
(\mathsf{mem}\ mt)[xx^* := tu^*] &=& \mathsf{mem}\ mt[xx^* := tu^*]
\end{array}$$

$rt[:= tu^n]$

1. Return $rt[i^{i<n} := tu^n]$.

$$rt[:= tu^n] \quad = \quad rt[i^{i<n} := tu^n]$$

$dt[:= tu^n]$

1. Return $dt[i^{i<n} := tu^n]$.

$$dt[:= tu^n] \quad = \quad dt[i^{i<n} := tu^n]$$

$defty^*_{u0}[:= tu^*]$

1. If $defty^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $dt_1\ dt^*$ be $defty^*_{u0}$.
3. Return $dt_1[:= tu^*]\ dt^*[:= tu^*]$.

$$\begin{array}{lcl} \epsilon[:= tu^*] & = & \epsilon \\ dt_1\ dt^*[:= tu^*] & = & dt_1[:= tu^*]\ dt^*[:= tu^*] \end{array}$$

3.1.6 Rolling and Unrolling

$\mathrm{roll}_x((\mathsf{rec}\ st^n))$

1. Return $(\mathsf{rec}\ st[x + i^{i<n} := (\mathsf{rec}\ i)^{i<n}]^n)$.

$$\mathrm{roll}_x(\mathsf{rec}\ st^n) \quad = \quad \mathsf{rec}\ (st[(x + i)^{i<n} := (\mathsf{rec}\ i)^{i<n}])^n$$

$\mathrm{unroll}((\mathsf{rec}\ st^n))$

1. Let qt be $(\mathsf{rec}\ st^n)$.
2. Return $(\mathsf{rec}\ st[(\mathsf{rec}\ i)^{i<n} := (qt.i)^{i<n}]^n)$.

$$\mathrm{unroll}(\mathsf{rec}\ st^n) \quad = \quad \mathsf{rec}\ (st[(\mathsf{rec}\ i)^{i<n} := (qt.i)^{i<n}])^n \qquad \text{if } qt = \mathsf{rec}\ st^n$$

$\mathrm{roll}_x(qt)$

1. Assert: Due to validation, $\mathrm{roll}_x(qt)$ is of the case rec.
2. Let $(\mathsf{rec}\ st^n)$ be $\mathrm{roll}_x(qt)$.
3. Return $((\mathsf{rec}\ st^n).i)^{i<n}$.

$$\mathrm{roll}_x(qt) \quad = \quad ((\mathsf{rec}\ st^n).i)^{i<n} \qquad \text{if } \mathrm{roll}_x(qt) = \mathsf{rec}\ st^n$$

$\mathrm{unroll}((qt.i))$

1. Assert: Due to validation, $\mathrm{unroll}(qt)$ is of the case rec.
2. Let $(\mathsf{rec}\ st^*)$ be $\mathrm{unroll}(qt)$.
3. Return $st^*[i]$.

$$\mathrm{unroll}(qt.i) \quad = \quad st^*[i] \qquad \text{if } \mathrm{unroll}(qt) = \mathsf{rec}\ st^*$$

$\mathrm{unroll}_C(heapt_{u0})$

1. If the type of $heapt_{u0}$ is deftype, then:
   a. Let $deftype$ be $heapt_{u0}$.
   b. Return $\mathrm{unroll}(deftype)$.
2. If $heapt_{u0}$ is of the case , then:
   a. Let $typeidx$ be $heapt_{u0}$.
   b. Return $\mathrm{unroll}(C.\mathsf{types}[typeidx])$.
3. Assert: Due to validation, $heapt_{u0}$ is of the case rec.
4. Let $(\mathsf{rec}\ i)$ be $heapt_{u0}$.
5. Return $C.\mathsf{rec}[i]$.

$$\begin{array}{lcl}
\mathrm{unroll}_C(deftype) & = & \mathrm{unroll}(deftype) \\
\mathrm{unroll}_C(typeidx) & = & \mathrm{unroll}(C.\mathsf{types}[typeidx]) \\
\mathrm{unroll}_C(\mathsf{rec}\ i) & = & C.\mathsf{rec}[i]
\end{array}$$

$\mathrm{expand}(dt)$

1. Assert: Due to validation, $\mathrm{unroll}(dt)$ is of the case sub.
2. Let $(\mathsf{sub}\ fin\ tu^*\ ct)$ be $\mathrm{unroll}(dt)$.
3. Return ct.

$$\mathrm{expand}(dt) \quad = \quad ct \qquad \text{if } \mathrm{unroll}(dt) = \mathsf{sub}\ fin\ tu^*\ ct$$

$$dt \quad \approx \quad ct \qquad \text{if } \mathrm{expand}(dt) = ct$$

3.1.7 Instruction Types

$$instrtype \quad ::= \quad resulttype \rightarrow_{localidx^*} resulttype$$

3.1.8 Local Types

$$\begin{array}{llcl}
\text{(local type)} & localtype & ::= & init\ valtype \\
\text{(initialization status)} & init & ::= & \mathsf{set} \mid \mathsf{unset}
\end{array}$$

3.1.9 Contexts

$$\begin{array}{lll} \mathit{context} & ::= & \{\mathsf{types}~\mathit{deftype}^\ast,~\mathsf{rec}~\mathit{subtype}^\ast, \\ & & \quad \mathsf{funcs}~\mathit{deftype}^\ast,~\mathsf{globals}~\mathit{globaltype}^\ast,~\mathsf{tables}~\mathit{tabletype}^\ast,~\mathsf{mems}~\mathit{memtype}^\ast, \\ & & \quad \mathsf{elems}~\mathit{elemtype}^\ast,~\mathsf{datas}~\mathit{datatype}^\ast, \\ & & \quad \mathsf{locals}~\mathit{localtype}^\ast,~\mathsf{labels}~\mathit{resulttype}^\ast,~\mathsf{return}~\mathit{resulttype}^?\} \end{array}$$

$\mathrm{clos}~C~(\mathit{dt})$

1. Let ${\mathit{dt}'}^\ast$ be $\mathrm{clos}^\ast~(C.\mathsf{types})$.
2. Return $\mathit{dt}[:= {\mathit{dt}'}^\ast]$.

$$\mathrm{clos}~C~(\mathit{dt}) \quad = \quad \mathit{dt}[:= {\mathit{dt}'}^\ast] \qquad \text{if } {\mathit{dt}'}^\ast = \mathrm{clos}^\ast~(C.\mathsf{types})$$

$\mathrm{clos}^\ast~(\mathit{defty}^\ast_{u0})$

1. If $\mathit{defty}^\ast_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $\mathit{dt}^\ast~\mathit{dt}_N$ be $\mathit{defty}^\ast_{u0}$.
3. Let ${\mathit{dt}'}^\ast$ be $\mathrm{clos}^\ast~(\mathit{dt}^\ast)$.
4. Return ${\mathit{dt}'}^\ast~\mathit{dt}_N[:= {\mathit{dt}'}^\ast]$.

$$\begin{array}{lcll} \mathrm{clos}^\ast~(\epsilon) & = & \epsilon & \\ \mathrm{clos}^\ast~(\mathit{dt}^\ast~\mathit{dt}_N) & = & {\mathit{dt}'}^\ast~\mathit{dt}_N[:= {\mathit{dt}'}^\ast] & \text{if } {\mathit{dt}'}^\ast = \mathrm{clos}^\ast~(\mathit{dt}^\ast) \end{array}$$

3.2 Types

3.2.1 Number Types

$$\frac{}{C \vdash \mathit{numtype} : \mathsf{ok}}$$

3.2.2 Vector Types

$$\frac{}{C \vdash \mathit{vectype} : \mathsf{ok}}$$

3.2.3 Heap Types

$$\frac{}{C \vdash \mathit{absheaptype} : \mathsf{ok}}\ [\textsc{K-heap-abs}] \qquad \frac{C.\mathsf{types}[\mathit{typeidx}] = \mathit{dt}}{C \vdash \mathit{typeidx} : \mathsf{ok}}\ [\textsc{K-heap-typeidx}] \qquad \frac{C.\mathsf{rec}[i] = \mathit{st}}{C \vdash \mathsf{rec}\ i : \mathsf{ok}}\ [\textsc{K-heap-rec}]$$

3.2.4 Reference Types

$$\frac{C \vdash \mathit{heaptype} : \mathsf{ok}}{C \vdash \mathsf{ref}\ \mathsf{null}^?\ \mathit{heaptype} : \mathsf{ok}}$$

3.2.5 Value Types

$$\frac{C \vdash \mathit{numtype} : \mathsf{ok}}{C \vdash \mathit{numtype} : \mathsf{ok}}\ [\textsc{K-val-num}] \qquad \frac{C \vdash \mathit{vectype} : \mathsf{ok}}{C \vdash \mathit{vectype} : \mathsf{ok}}\ [\textsc{K-val-vec}] \qquad \frac{C \vdash \mathit{reftype} : \mathsf{ok}}{C \vdash \mathit{reftype} : \mathsf{ok}}\ [\textsc{K-val-ref}] \qquad \frac{}{C \vdash \mathsf{bot} : \mathsf{ok}}\ [\textsc{K-val-bot}]$$

3.2.6 Block Types

$$\frac{(C \vdash \mathit{valtype} : \mathsf{ok})^?}{C \vdash \mathit{valtype}^? : \epsilon \to \mathit{valtype}^?}\ [\textsc{K-block-valtype}] \qquad \frac{C.\mathsf{types}[\mathit{typeidx}] \approx \mathsf{func}\ (t_1^* \to t_2^*)}{C \vdash \mathit{typeidx} : t_1^* \to t_2^*}\ [\textsc{K-block-typeidx}]$$

3.2.7 Result Types

$$\frac{(C \vdash t : \mathsf{ok})^*}{C \vdash t^* : \mathsf{ok}}$$

3.2.8 Instruction Types

$$\frac{C \vdash t_1^* : \mathsf{ok} \qquad C \vdash t_2^* : \mathsf{ok} \qquad (C.\mathsf{locals}[x] = \mathit{lt})^*}{C \vdash t_1^* \to_{x^*} t_2^* : \mathsf{ok}}$$

3.2.9 Function Types

$$\frac{C \vdash t_1^* : \mathsf{ok} \qquad C \vdash t_2^* : \mathsf{ok}}{C \vdash t_1^* \to t_2^* : \mathsf{ok}}$$

3.2.10 Composite Types

$$\frac{(C \vdash \mathit{fieldtype} : \mathsf{ok})^*}{C \vdash \mathsf{struct}\ \mathit{fieldtype}^* : \mathsf{ok}}\ [\textsc{K-comp-struct}] \qquad \frac{C \vdash \mathit{fieldtype} : \mathsf{ok}}{C \vdash \mathsf{array}\ \mathit{fieldtype} : \mathsf{ok}}\ [\textsc{K-comp-array}] \qquad \frac{C \vdash \mathit{functype} : \mathsf{ok}}{C \vdash \mathsf{func}\ \mathit{functype} : \mathsf{ok}}\ [\textsc{K-comp-func}]$$

3.2.11 Field Types

$$\frac{}{C \vdash \mathit{packtype} : \mathsf{ok}}\ [\textsc{K-pack}]$$

$$\frac{C \vdash \mathit{valtype} : \mathsf{ok}}{C \vdash \mathit{valtype} : \mathsf{ok}}\ [\textsc{K-storage-val}] \qquad \frac{C \vdash \mathit{packtype} : \mathsf{ok}}{C \vdash \mathit{packtype} : \mathsf{ok}}\ [\textsc{K-storage-pack}]$$

$$\frac{C \vdash \mathit{storagetype} : \mathsf{ok}}{C \vdash \mathsf{mut}^?\ \mathit{storagetype} : \mathsf{ok}}\ [\textsc{K-field}]$$

3.2.12 Recursive Types

$typeu_{u0} \prec x, i$

1. If the type of $typeu_{u0}$ is deftype, then:
   a. Return true.
2. If $typeu_{u0}$ is of the case , then:
   a. Let $typeidx$ be $typeu_{u0}$.
   b. Return $typeidx$ is less than x.
3. Assert: Due to validation, $typeu_{u0}$ is of the case rec.
4. Let $(\mathsf{rec}\ j)$ be $typeu_{u0}$.
5. Return j is less than i.

$$\begin{array}{lcl} deftype \prec x, i &=& \mathsf{true} \\ typeidx \prec x, i &=& typeidx < x \\ \mathsf{rec}\ j \prec x, i &=& j < i \end{array}$$

$$\begin{array}{rcl} \mathsf{ok}(typeidx) &::=& \mathsf{ok}typeidx \\ \mathsf{ok}(typeidx,\ n) &::=& \mathsf{ok}(typeidx,\ nat) \end{array}$$

$$\frac{}{C \vdash \mathsf{rec}\ \epsilon : \mathsf{ok}(x)}\ [\textsc{K-rect-empty}] \qquad \frac{C \vdash subtype_1 : \mathsf{ok}(x) \qquad C \vdash \mathsf{rec}\ subtype^* : \mathsf{ok}(x+1)}{C \vdash \mathsf{rec}\ (subtype_1\ subtype^*) : \mathsf{ok}(x)}\ [\textsc{K-rect-cons}] \qquad \frac{C, \mathsf{rec}\ subtype^* \vdash \mathsf{re}}{C \vdash \mathsf{rec}\ subt}$$

$$\frac{}{C \vdash \mathsf{rec}\ \epsilon : \mathsf{ok}(x,\ i)}\ [\textsc{K-rec2-empty}] \qquad \frac{C \vdash subtype_1 : \mathsf{ok}(x,\ i) \qquad C \vdash \mathsf{rec}\ subtype^* : \mathsf{ok}(x+1,\ i+1)}{C \vdash \mathsf{rec}\ (subtype_1\ subtype^*) : \mathsf{ok}(x,\ i)}\ [\textsc{K}$$

$$\frac{\begin{array}{c} |x^*| \leq 1 \qquad (x < x_0)^* \qquad (\mathrm{unroll}(C.\mathsf{types}[x]) = \mathsf{sub}\ x'^*\ comptype')^* \\ C \vdash comptype : \mathsf{ok} \qquad (C \vdash comptype \leq comptype')^* \end{array}}{C \vdash \mathsf{sub}\ \mathsf{final}^?\ typeidx^*\ comptype : \mathsf{ok}(x_0)}\ [\textsc{K-sub}]$$

$$\frac{\begin{array}{c} |typeuse^*| \leq 1 \qquad (typeuse \prec x, i)^* \qquad (\mathrm{unroll}_C(typeuse) = \mathsf{sub}\ typeuse'^*\ comptype')^* \\ C \vdash comptype : \mathsf{ok} \qquad (C \vdash comptype \leq comptype')^* \end{array}}{C \vdash \mathsf{sub}\ \mathsf{final}^?\ typeuse^*\ compttype : \mathsf{ok}(x,\ i)}\ [\textsc{K-sub}$$

3.2.13 Defined Types

$$\frac{C \vdash rectype : \mathsf{ok}(x) \qquad rectype = \mathsf{rec}\ subtype^n \qquad i < n}{C \vdash rectype.i : \mathsf{ok}}$$

3.2.14 Limits

$$\frac{n \leq m \leq k}{C \vdash [n..m] : k}$$

3.2.15 Table Types

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

3.2.16 Memory Types

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

3.2.17 Global Types

$$\frac{C \vdash t : \mathsf{ok}}{C \vdash \mathsf{mut}^?\ t : \mathsf{ok}}$$

3.2.18 External Types

$$\frac{C \vdash \mathit{deftype} : \mathsf{ok} \qquad \mathit{deftype} \approx \mathsf{func}\ \mathit{functype}}{C \vdash \mathsf{func}\ \mathit{deftype} : \mathsf{ok}}\ [\text{K-EXTERN-FUNC}] \qquad \frac{C \vdash \mathit{globaltype} : \mathsf{ok}}{C \vdash \mathsf{global}\ \mathit{globaltype} : \mathsf{ok}}\ [\text{K-EXTERN-GLOBAL}] \qquad \frac{C \vdash \mathit{tabletype}}{C \vdash \mathsf{table}\ \mathit{tablet}}$$

3.3 Matching

3.3.1 Number Types

$$\frac{}{C \vdash \mathit{numtype} \leq \mathit{numtype}}$$

3.3.2 Vector Types

$$\frac{}{C \vdash \mathit{vectype} \leq \mathit{vectype}}$$

3.3.3 Heap Types

$$\frac{}{C \vdash \mathit{heaptype} \leq \mathit{heaptype}}\ [\text{S-HEAP-REFL}] \qquad \frac{C \vdash \mathit{heaptype}' : \mathsf{ok} \qquad C \vdash \mathit{heaptype}_1 \leq \mathit{heaptype}' \qquad C \vdash \mathit{heaptype}' \leq \mathit{heaptype}_2}{C \vdash \mathit{heaptype}_1 \leq \mathit{heaptype}_2}$$

3.3.4 Reference Types

$$\frac{C \vdash \mathit{ht}_1 \leq \mathit{ht}_2}{C \vdash \mathsf{ref}\ \mathit{ht}_1 \leq \mathsf{ref}\ \mathit{ht}_2}\ [\text{S-REF-NONNULL}]$$

$$\frac{C \vdash \mathit{ht}_1 \leq \mathit{ht}_2}{C \vdash \mathsf{ref}\ \mathsf{null}^?\ \mathit{ht}_1 \leq \mathsf{ref}\ \mathsf{null}\ \mathit{ht}_2}\ [\text{S-REF-NULL}]$$

3.3.5 Value Types

$$\frac{C \vdash \mathit{numtype}_1 \leq \mathit{numtype}_2}{C \vdash \mathit{numtype}_1 \leq \mathit{numtype}_2}\,[\textsc{S-val-num}] \qquad \frac{C \vdash \mathit{vectype}_1 \leq \mathit{vectype}_2}{C \vdash \mathit{vectype}_1 \leq \mathit{vectype}_2}\,[\textsc{S-val-vec}] \qquad \frac{C \vdash \mathit{reftype}_1 \leq \mathit{reftype}_2}{C \vdash \mathit{reftype}_1 \leq \mathit{reftype}_2}\,[\textsc{S-val-ref}] \qquad \frac{}{C \vdash}$$

3.3.6 Result Types

$$\frac{(C \vdash t_1 \leq t_2)^*}{C \vdash t_1^* \leq t_2^*}$$

3.3.7 Instruction Types

$$\frac{C \vdash t_{21}^* \leq t_{11}^* \qquad C \vdash t_{12}^* \leq t_{22}^* \qquad x^* = x_2^* \setminus x_1^* \qquad ((C.\mathsf{locals}[x] = \mathsf{set}\ t))^*}{C \vdash t_{11}^* \rightarrow_{x_1^*} t_{12}^* \leq t_{21}^* \rightarrow_{x_2^*} t_{22}^*}$$

3.3.8 Function Types

$$\frac{}{C \vdash \mathit{ft} \leq \mathit{ft}}$$

3.3.9 Composite Types

$$\frac{(C \vdash \mathit{yt}_1 \leq \mathit{yt}_2)^*}{C \vdash \mathsf{struct}\ (\mathit{yt}_1^*\ \mathit{yt'}_1) \leq \mathsf{struct}\ \mathit{yt}_2^*}\,[\textsc{S-comp-struct}] \qquad \frac{C \vdash \mathit{yt}_1 \leq \mathit{yt}_2}{C \vdash \mathsf{array}\ \mathit{yt}_1 \leq \mathsf{array}\ \mathit{yt}_2}\,[\textsc{S-comp-array}] \qquad \frac{C \vdash \mathit{ft}_1 \leq \mathit{ft}_2}{C \vdash \mathsf{func}\ \mathit{ft}_1 \leq \mathsf{func}\ \mathit{ft}_2}\,[\textsc{S-comp-f}$$

3.3.10 Field Types

$$\frac{C \vdash \mathit{zt}_1 \leq \mathit{zt}_2}{C \vdash \mathit{zt}_1 \leq \mathit{zt}_2}\,[\textsc{S-field-const}] \qquad \frac{C \vdash \mathit{zt}_1 \leq \mathit{zt}_2 \qquad C \vdash \mathit{zt}_2 \leq \mathit{zt}_1}{C \vdash \mathsf{mut}\ \mathit{zt}_1 \leq \mathsf{mut}\ \mathit{zt}_2}\,[\textsc{S-field-var}]$$

$$\frac{C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2}{C \vdash \mathit{valtype}_1 \leq \mathit{valtype}_2}\,[\textsc{S-storage-val}] \qquad \frac{C \vdash \mathit{packtype}_1 \leq \mathit{packtype}_2}{C \vdash \mathit{packtype}_1 \leq \mathit{packtype}_2}\,[\textsc{S-storage-pack}]$$

$$\frac{}{C \vdash \mathit{packtype} \leq \mathit{packtype}}\,[\textsc{S-pack}]$$

3.3.11 Defined Types

$$\frac{\mathrm{clos}_C\ (\mathit{deftype}_1) = \mathrm{clos}_C\ (\mathit{deftype}_2)}{C \vdash \mathit{deftype}_1 \leq \mathit{deftype}_2}\,[\textsc{S-def-refl}] \qquad \frac{\mathrm{unroll}(\mathit{deftype}_1) = \mathsf{sub}\ \mathit{fin}\ (y_1^*\ y\ y_2^*)\ \mathit{ct} \qquad C \vdash y \leq \mathit{deftype}_2}{C \vdash \mathit{deftype}_1 \leq \mathit{deftype}_2}\,[\textsc{S-def-super}]$$

3.3.12 Limits

$$\frac{n_{11} \geq n_{21} \qquad n_{12} \leq n_{22}}{C \vdash [n_{11} .. n_{12}] \leq [n_{21} .. n_{22}]}$$

3.3.13 Table Types

$$\frac{C \vdash lim_1 \leq lim_2 \qquad C \vdash rt_1 \leq rt_2 \qquad C \vdash rt_2 \leq rt_1}{C \vdash lim_1\ rt_1 \leq lim_2\ rt_2}$$

3.3.14 Memory Types

$$\frac{C \vdash lim_1 \leq lim_2}{C \vdash lim_1\ \mathsf{i8} \leq lim_2\ \mathsf{i8}}$$

3.3.15 Global Types

$$\frac{C \vdash t_1 \leq t_2}{C \vdash t_1 \leq t_2}\ [\textsc{S-global-const}] \qquad \frac{C \vdash t_1 \leq t_2 \qquad C \vdash t_2 \leq t_1}{C \vdash \mathsf{mut}\ t_1 \leq \mathsf{mut}\ t_2}\ [\textsc{S-global-var}]$$

3.3.16 External Types

$$\frac{C \vdash dt_1 \leq dt_2}{C \vdash \mathsf{func}\ dt_1 \leq \mathsf{func}\ dt_2}\ [\textsc{S-extern-func}] \qquad \frac{C \vdash gt_1 \leq gt_2}{C \vdash \mathsf{global}\ gt_1 \leq \mathsf{global}\ gt_2}\ [\textsc{S-extern-global}] \qquad \frac{C \vdash tt_1 \leq tt_2}{C \vdash \mathsf{table}\ tt_1 \leq \mathsf{table}\ tt_2}\ [\textsc{S-extern-tabl}$$

3.4 Instructions

$$\frac{C \vdash numtype_1 \leq numtype_2}{C \vdash numtype_1 \leq numtype_2}\ [\textsc{S-val-num}] \qquad \frac{C \vdash vectype_1 \leq vectype_2}{C \vdash vectype_1 \leq vectype_2}\ [\textsc{S-val-vec}] \qquad \frac{C \vdash reftype_1 \leq reftype_2}{C \vdash reftype_1 \leq reftype_2}\ [\textsc{S-val-ref}] \qquad \frac{}{C \vdash}$$

$$\frac{(C \vdash t_1 \leq t_2)^*}{C \vdash t_1^* \leq t_2^*}\ [\textsc{S-result}]$$

3.4.1 Numeric Instructions

$nt.\mathsf{const}\ c_{nt}$

- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} nt$.

$$\frac{}{C \vdash nt.\mathsf{const}\ c_{nt} : \epsilon \rightarrow nt}$$

$nt.unop_{nt}$

- The instruction is valid with type $nt \rightarrow_{\epsilon} nt$.

$$\frac{}{C \vdash nt.unop_{nt} : nt \rightarrow nt}$$

$nt.binop_{nt}$

- The instruction is valid with type $nt \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash nt.binop_{nt} : nt\ nt \rightarrow nt}$$

$nt.testop_{nt}$

- The instruction is valid with type $nt \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{}{C \vdash nt.testop_{nt} : nt \rightarrow \mathsf{i32}}$$

$nt.relop_{nt}$

- The instruction is valid with type $nt \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash nt.relop_{nt} : nt\ nt \rightarrow \mathsf{i32}}$$

$nt_1.\mathsf{reinterpret\_}nt_2\mathsf{\_}\epsilon$

- $|nt_1|$ must be equal to $|nt_2|$.
- The instruction is valid with type $nt_2 \rightarrow_{\epsilon} nt_1$.

$$\frac{|nt_1| = |nt_2|}{C \vdash nt_1.\mathsf{reinterpret\_}nt_2 : nt_2 \rightarrow nt_1}\ [\textsc{T-cvtop-reinterpret}] \qquad \frac{sx^? = \epsilon \Leftrightarrow |\mathsf{i}n_1| > |\mathsf{i}n_2|}{C \vdash \mathsf{i}n_1.\mathsf{convert\_i}n_2\mathsf{\_}sx^? : \mathsf{i}n_2 \rightarrow \mathsf{i}n_1}\ [\textsc{T-cvtop-convert-i}] \qquad \frac{}{C \vdash \mathsf{f}n_1.}$$

3.4.2 Reference Instructions

$\mathsf{ref.null}\ ht$

- Under the context C, ht must be valid.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ ())\ ht)$.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.null}\ ht : \epsilon \rightarrow (\mathsf{ref}\ \mathsf{null}\ ht)}$$

ref.func x

- $|C.\mathsf{funcs}|$ must be greater than x.
- Let dt be $C.\mathsf{funcs}[x]$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} (\mathsf{ref}~(\mathsf{null}~\epsilon)~dt)$.

$$\frac{C.\mathsf{funcs}[x] = dt}{C \vdash \mathsf{ref.func}~x : \epsilon \rightarrow (\mathsf{ref}~dt)}$$

ref.is_null

- Under the context C, ht must be valid.
- The instruction is valid with type $(\mathsf{ref}~(\mathsf{null}~())~ht) \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.is\_null} : (\mathsf{ref~null}~ht) \rightarrow \mathsf{i32}}$$

ref.as_non_null

- Under the context C, ht must be valid.
- The instruction is valid with type $(\mathsf{ref}~(\mathsf{null}~())~ht) \rightarrow_{\epsilon} (\mathsf{ref}~(\mathsf{null}~\epsilon)~ht)$.

$$\frac{C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{ref.as\_non\_null} : (\mathsf{ref~null}~ht) \rightarrow (\mathsf{ref}~ht)}$$

ref.eq

- The instruction is valid with type $(\mathsf{ref}~(\mathsf{null}~())~\mathsf{eq}) \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash \mathsf{ref.eq} : (\mathsf{ref~null~eq})~(\mathsf{ref~null~eq}) \rightarrow \mathsf{i32}}$$

ref.test rt

- Under the context C, rt must be valid.
- YetI: TODO: prem_to_instrs rule_sub.
- Under the context C, rt' must be valid.
- The instruction is valid with type $rt' \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{C \vdash rt : \mathsf{ok} \qquad C \vdash rt' : \mathsf{ok} \qquad C \vdash rt \leq rt'}{C \vdash \mathsf{ref.test}~rt : rt' \rightarrow \mathsf{i32}}$$

ref.cast rt

- Under the context C, rt must be valid.
- YetI: TODO: prem_to_instrs rule_sub.
- Under the context C, rt' must be valid.
- The instruction is valid with type $rt' \rightarrow_{\epsilon} rt$.

$$\frac{C \vdash rt : \mathsf{ok} \qquad C \vdash rt' : \mathsf{ok} \qquad C \vdash rt \leq rt'}{C \vdash \mathsf{ref.cast}\ rt : rt' \rightarrow rt}$$

3.4.3 Aggregate Reference Instructions

struct.new x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{struct}\ (mut,\ zt)^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- $|zt^*|$ must be equal to $|mut^*|$.
- The instruction is valid with type $\mathrm{unpack}(zt)^* \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ x)$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ (mut\ zt)^*}{C \vdash \mathsf{struct.new}\ x : \mathrm{unpack}(zt)^* \rightarrow (\mathsf{ref}\ x)}$$

struct.new_default x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{struct}\ (mut,\ zt)^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- $|zt^*|$ must be equal to $|mut^*|$.
- YetI: TODO: prem_to_intrs iter.
- $|zt^*|$ must be equal to $|val^*|$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ x)$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ (mut\ zt)^* \qquad (\mathrm{default}_{\mathrm{unpack}(zt)} = val)^*}{C \vdash \mathsf{struct.new\_default}\ x : \epsilon \rightarrow (\mathsf{ref}\ x)}$$

$\mathsf{struct.get\_}sx^?\ x\ i$

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{struct}\ yt^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- $|yt^*|$ must be greater than i.
- Let $(mut,\ zt)$ be $yt^*[i]$.
- $sx^?$ is ϵ if and only if zt is $\mathrm{unpack}(zt)$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\epsilon} \mathrm{unpack}(zt)$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ yt^* \qquad yt^*[i] = mut\ zt \qquad sx^? = \epsilon \Leftrightarrow zt = \mathrm{unpack}(zt)}{C \vdash \mathsf{struct.get\_}sx^?\ x\ i : (\mathsf{ref}\ \mathsf{null}\ x) \rightarrow \mathrm{unpack}(zt)}$$

$\mathsf{struct.set}\ x\ i$

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{struct}\ yt^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- $|yt^*|$ must be greater than i.
- Let $((\mathsf{mut}\ ()),\ zt)$ be $yt^*[i]$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{struct}\ yt^* \qquad yt^*[i] = \mathsf{mut}\ zt}{C \vdash \mathsf{struct.set}\ x\ i : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathrm{unpack}(zt) \rightarrow \epsilon}$$

$\mathsf{array.new}\ x$

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ (mut,\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- The instruction is valid with type $\mathrm{unpack}(zt) \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt)}{C \vdash \mathsf{array.new}\ x : \mathrm{unpack}(zt)\ \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

$\mathsf{array.new\_default}\ x$

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ (mut,\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- Let val be $\mathrm{default}_{\mathrm{unpack}(zt)}$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ x)$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \qquad \mathrm{default}_{\mathrm{unpack}(zt)} = val}{C \vdash \mathsf{array.new\_default}\ x : \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

array.new_fixed x n

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ (mut,\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- The instruction is valid with type $\mathrm{unpack}(zt)^n \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ x)$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt)}{C \vdash \mathsf{array.new\_fixed}\ x\ n : \mathrm{unpack}(zt)^n \rightarrow (\mathsf{ref}\ x)}$$

array.new_elem x y

- $|C.\mathsf{types}|$ must be greater than x.
- $|C.\mathsf{elems}|$ must be greater than y.
- Let $(\mathsf{array}\ (mut,\ rt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- YetI: TODO: prem_to_instrs rule_sub.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ rt) \qquad C \vdash C.\mathsf{elems}[y] \leq rt}{C \vdash \mathsf{array.new\_elem}\ x\ y : \mathsf{i32}\ \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

array.new_data x y

- $|C.\mathsf{types}|$ must be greater than x.
- $|C.\mathsf{datas}|$ must be greater than y.
- $C.\mathsf{datas}[y]$ must be equal to ok.
- Let $(\mathsf{array}\ (mut,\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- Let $numtype$ be $\mathrm{unpack}(zt)$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \qquad \mathrm{unpack}(zt) = numtype \vee \mathrm{unpack}(zt) = vectype \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{array.new\_data}\ x\ y : \mathsf{i32}\ \mathsf{i32} \rightarrow (\mathsf{ref}\ x)}$$

array.get_$sx^?$ x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ (mut,\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- $sx^?$ is ϵ if and only if zt is $\mathrm{unpack}(zt)$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \qquad sx^? = \epsilon \Leftrightarrow zt = \mathrm{unpack}(zt)}{C \vdash \mathsf{array.get\_}sx^?\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32} \rightarrow \mathrm{unpack}(zt)}$$

array.set x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\mathrm{unpack}(zt)}\ \rightarrow$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt)}{C \vdash \mathsf{array.set}\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathrm{unpack}(zt) \rightarrow \epsilon}$$

array.len

- Let $\mathrm{expand}(C.\mathsf{types}[x])$ be $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt))$.
- $|C.\mathsf{types}|$ must be greater than x.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ \mathsf{array}) \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt)}{C \vdash \mathsf{array.len} : (\mathsf{ref}\ \mathsf{null}\ \mathsf{array}) \rightarrow \mathsf{i32}}$$

array.fill x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\mathrm{unpack}(zt)} \mathsf{i32}$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt)}{C \vdash \mathsf{array.fill}\ x : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathrm{unpack}(zt)\ \mathsf{i32} \rightarrow \epsilon}$$

array.copy x_1 x_2

- $|C.\mathsf{types}|$ must be greater than x_1.
- $|C.\mathsf{types}|$ must be greater than x_2.
- Let $(\mathsf{array}\ (mut,\ zt_2))$ be $\mathrm{expand}(C.\mathsf{types}[x_2])$.
- YetI: TODO: prem_to_instrs rule_sub.
- $\mathrm{expand}(C.\mathsf{types}[x_1])$ must be equal to $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt_1))$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x_1) \rightarrow_{\mathsf{ref}\ (\mathsf{null}\ ())\ x_2} \mathsf{i32}$.

$$\frac{C.\mathsf{types}[x_1] \approx \mathsf{array}\ (\mathsf{mut}\ zt_1) \qquad C.\mathsf{types}[x_2] \approx \mathsf{array}\ (mut\ zt_2) \qquad C \vdash zt_2 \leq zt_1}{C \vdash \mathsf{array.copy}\ x_1\ x_2 : (\mathsf{ref}\ \mathsf{null}\ x_1)\ \mathsf{i32}\ (\mathsf{ref}\ \mathsf{null}\ x_2)\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

array.init_data x y

- $|C.\mathsf{types}|$ must be greater than x.
- $|C.\mathsf{datas}|$ must be greater than y.
- $C.\mathsf{datas}[y]$ must be equal to ok.
- Let $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt))$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- Let $numtype$ be $\mathrm{unpack}(zt)$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\mathsf{i32}} \mathsf{i32}$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt) \qquad \mathrm{unpack}(zt) = numtype \vee \mathrm{unpack}(zt) = vectype \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{array.init\_data}\ x\ y : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

array.init_elem x y

- $|C.\mathsf{types}|$ must be greater than x.
- $|C.\mathsf{elems}|$ must be greater than y.
- YetI: TODO: prem_to_instrs rule_sub.
- $\mathrm{expand}(C.\mathsf{types}[x])$ must be equal to $(\mathsf{array}\ ((\mathsf{mut}\ ()),\ zt))$.
- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ x) \rightarrow_{\mathsf{i32}} \mathsf{i32}$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}\ zt) \qquad C \vdash C.\mathsf{elems}[y] \leq zt}{C \vdash \mathsf{array.init\_elem}\ x\ y : (\mathsf{ref}\ \mathsf{null}\ x)\ \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

3.4.4 Scalar Reference Instructions

ref.i31

- The instruction is valid with type $\mathsf{i32} \rightarrow_{\epsilon} (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ \mathsf{i31})$.

$$\overline{C \vdash \mathsf{ref.i31} : \mathsf{i32} \rightarrow (\mathsf{ref}\ \mathsf{i31})}$$

i31.get_sx

- The instruction is valid with type $(\mathsf{ref}\ (\mathsf{null}\ ())\ \mathsf{i31}) \rightarrow_{\epsilon} \mathsf{i32}$.

$$\overline{C \vdash \mathsf{i31.get\_}sx : (\mathsf{ref\ null\ i31}) \rightarrow \mathsf{i32}}$$

3.4.5 Vector Instructions

v128.const c

- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} \mathsf{v128}$.

$$\overline{C \vdash \mathsf{v128.const}\ c : \epsilon \rightarrow \mathsf{v128}}$$

v128.$vvunop$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{v128}$.

$$\overline{C \vdash \mathsf{v128}.vvunop : \mathsf{v128} \rightarrow \mathsf{v128}}$$

v128.$vvbinop$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\overline{C \vdash \mathsf{v128}.vvbinop : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$\mathsf{v128}.\mathit{vvternop}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\mathsf{v128}} \rightarrow$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvternop} : \mathsf{v128}\ \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$\mathsf{v128}.\mathit{vvtestop}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{}{C \vdash \mathsf{v128}.\mathit{vvtestop} : \mathsf{v128} \rightarrow \mathsf{i32}}$$

$\mathit{sh}.\mathsf{shuffle}\ i^*$

- For all i in i^*,
  - i must be less than $2 \cdot \mathrm{dim}(\mathit{sh})$.
- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{(i < 2 \cdot \mathrm{dim}(\mathit{sh}))^*}{C \vdash \mathit{sh}.\mathsf{shuffle}\ i^* : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$\mathit{sh}.\mathsf{splat}$

- The instruction is valid with type $\mathrm{unpack}(\mathit{sh}) \rightarrow_{\epsilon} \mathsf{v128}$.

$$\frac{}{C \vdash \mathit{sh}.\mathsf{splat} : \mathrm{unpack}(\mathit{sh}) \rightarrow \mathsf{v128}}$$

$\mathit{sh}.\mathsf{extract\_lane\_}\mathit{sx}^?\ i$

- i must be less than $\mathrm{dim}(\mathit{sh})$.
- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathrm{unpack}(\mathit{sh})$.

$$\frac{i < \mathrm{dim}(\mathit{sh})}{C \vdash \mathit{sh}.\mathsf{extract\_lane\_}\mathit{sx}^?\ i : \mathsf{v128} \rightarrow \mathrm{unpack}(\mathit{sh})}$$

$sh.\mathsf{replace\_lane}\ i$

- i must be less than $\dim(sh)$.
- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{i < \dim(sh)}{C \vdash sh.\mathsf{replace\_lane}\ i : \mathsf{v128}\ \mathrm{unpack}(sh) \rightarrow \mathsf{v128}}$$

$sh.vunop_{sh}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{v128}$.

$$\frac{}{C \vdash sh.vunop_{sh} : \mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh.vbinop_{sh}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash sh.vbinop_{sh} : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh.vrelop_{sh}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash sh.vrelop_{sh} : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh.vshiftop_{sh}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash sh.vshiftop_{sh} : \mathsf{v128}\ \mathsf{i32} \rightarrow \mathsf{v128}}$$

$sh.vtestop_{sh}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{}{C \vdash sh.vtestop_{sh} : \mathsf{v128} \rightarrow \mathsf{i32}}$$

$sh_1.vcvtop\_hf^?\_sh_2\_sx^?\_zero^?$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{v128}$.

$$\frac{}{C \vdash sh_1.vcvtop\_hf^?\_sh_2\_sx^?\_zero^? : \mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh_1.\mathsf{narrow}\_sh_2\_sx$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash sh_1.\mathsf{narrow}\_sh_2\_sx : \mathsf{v128}~\mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh.\mathsf{bitmask}$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{}{C \vdash sh.\mathsf{bitmask} : \mathsf{v128} \rightarrow \mathsf{i32}}$$

$sh_1.vextunop\_sh_2\_sx$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\epsilon} \mathsf{v128}$.

$$\frac{}{C \vdash sh_1.vextunop\_sh_2\_sx : \mathsf{v128} \rightarrow \mathsf{v128}}$$

$sh_1.vextbinop\_sh_2\_sx$

- The instruction is valid with type $\mathsf{v128} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{}{C \vdash sh_1.vextbinop\_sh_2\_sx : \mathsf{v128}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

3.4.6 External Reference Instructions

extern.convert_any

- The instruction is valid with type $(\mathsf{ref}\ nul\ \mathsf{any}) \rightarrow_{\epsilon} (\mathsf{ref}\ nul\ \mathsf{extern})$.

$$\frac{}{C \vdash \mathsf{extern.convert\_any} : (\mathsf{ref}\ nul\ \mathsf{any}) \rightarrow (\mathsf{ref}\ nul\ \mathsf{extern})}$$

any.convert_extern

- The instruction is valid with type $(\mathsf{ref}\ nul\ \mathsf{extern}) \rightarrow_{\epsilon} (\mathsf{ref}\ nul\ \mathsf{any})$.

$$\frac{}{C \vdash \mathsf{any.convert\_extern} : (\mathsf{ref}\ nul\ \mathsf{extern}) \rightarrow (\mathsf{ref}\ nul\ \mathsf{any})}$$

3.4.7 Parametric Instructions

drop

- Under the context C, t must be valid.
- The instruction is valid with type $t \rightarrow_{\epsilon} \epsilon$.

$$\frac{C \vdash t : \mathsf{ok}}{C \vdash \mathsf{drop} : t \rightarrow \epsilon}$$

select t

- Under the context C, t must be valid.
- The instruction is valid with type $t \rightarrow_{\mathsf{i32}}\ \rightarrow$.

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

3.4.8 Variable Instructions

$\mathsf{local.get}\ x$

- $|C.\mathsf{locals}|$ must be greater than x.
- Let $(\mathsf{set},\ t)$ be $C.\mathsf{locals}[x]$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} t$.

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

$\mathsf{local.set}\ x$

- $|C.\mathsf{locals}|$ must be greater than x.
- Let $(\mathit{init},\ t)$ be $C.\mathsf{locals}[x]$.
- The instruction is valid with type $t \rightarrow_{x} \epsilon$.

$$\frac{C.\mathsf{locals}[x] = \mathit{init}\ t}{C \vdash \mathsf{local.set}\ x : t \rightarrow_{x} \epsilon}\ [\textsc{T-local.set}]$$

$\mathsf{local.tee}\ x$

- $|C.\mathsf{locals}|$ must be greater than x.
- Let $(\mathit{init},\ t)$ be $C.\mathsf{locals}[x]$.
- The instruction is valid with type $t \rightarrow_{x} t$.

$$\frac{C.\mathsf{locals}[x] = \mathit{init}\ t}{C \vdash \mathsf{local.tee}\ x : t \rightarrow_{x} t}\ [\textsc{T-local.tee}]$$

$\mathsf{global.get}\ x$

- $|C.\mathsf{globals}|$ must be greater than x.
- Let $(\mathit{mut},\ t)$ be $C.\mathsf{globals}[x]$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} t$.

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

global.set x

- $|C.\mathsf{globals}|$ must be greater than x.
- Let $((\mathsf{mut}\ ()),\ t)$ be $C.\mathsf{globals}[x]$.
- The instruction is valid with type $t \to_{\epsilon} \epsilon$.

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

3.4.9 Table Instructions

table.get x

- $|C.\mathsf{tables}|$ must be greater than x.
- Let $(lim,\ rt)$ be $C.\mathsf{tables}[x]$.
- The instruction is valid with type $\mathsf{i32} \to_{\epsilon} rt$.

$$\frac{C.\mathsf{tables}[x] = lim\ rt}{C \vdash \mathsf{table.get}\ x : \mathsf{i32} \to rt}$$

table.set x

- $|C.\mathsf{tables}|$ must be greater than x.
- Let $(lim,\ rt)$ be $C.\mathsf{tables}[x]$.
- The instruction is valid with type $\mathsf{i32} \to_{\to} \epsilon$.

$$\frac{C.\mathsf{tables}[x] = lim\ rt}{C \vdash \mathsf{table.set}\ x : \mathsf{i32}\ rt \to \epsilon}$$

table.size x

- $|C.\mathsf{tables}|$ must be greater than x.
- Let $(lim,\ rt)$ be $C.\mathsf{tables}[x]$.
- The instruction is valid with type $\epsilon \to_{\epsilon} \mathsf{i32}$.

$$\frac{C.\mathsf{tables}[x] = lim\ rt}{C \vdash \mathsf{table.size}\ x : \epsilon \to \mathsf{i32}}$$

table.grow x

- $|C.\mathsf{tables}|$ must be greater than x.
- Let $(\mathit{lim},\ \mathit{rt})$ be $C.\mathsf{tables}[x]$.
- The instruction is valid with type $\mathit{rt} \rightarrow_{\rightarrow} \epsilon$.

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

table.fill x

- $|C.\mathsf{tables}|$ must be greater than x.
- Let $(\mathit{lim},\ \mathit{rt})$ be $C.\mathsf{tables}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}}\ \rightarrow$.

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

table.copy x_1 x_2

- $|C.\mathsf{tables}|$ must be greater than x_1.
- $|C.\mathsf{tables}|$ must be greater than x_2.
- Let $(\mathit{lim}_1,\ \mathit{rt}_1)$ be $C.\mathsf{tables}[x_1]$.
- Let $(\mathit{lim}_2,\ \mathit{rt}_2)$ be $C.\mathsf{tables}[x_2]$.
- YetI: TODO: prem_to_instrs rule_sub.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}}\ \rightarrow$.

$$\frac{C.\mathsf{tables}[x_1] = \mathit{lim}_1\ \mathit{rt}_1 \qquad C.\mathsf{tables}[x_2] = \mathit{lim}_2\ \mathit{rt}_2 \qquad C \vdash \mathit{rt}_2 \leq \mathit{rt}_1}{C \vdash \mathsf{table.copy}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

table.init x y

- $|C.\mathsf{tables}|$ must be greater than x.
- $|C.\mathsf{elems}|$ must be greater than y.
- Let rt_2 be $C.\mathsf{elems}[y]$.
- Let $(\mathit{lim},\ \mathit{rt}_1)$ be $C.\mathsf{tables}[x]$.
- YetI: TODO: prem_to_instrs rule_sub.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}}\ \rightarrow$.

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt}_1 \qquad C.\mathsf{elems}[y] = \mathit{rt}_2 \qquad C \vdash \mathit{rt}_2 \leq \mathit{rt}_1}{C \vdash \mathsf{table.init}\ x\ y : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

$\mathsf{elem.drop}\ x$

- $|C.\mathsf{elems}|$ must be greater than x.
- Let rt be $C.\mathsf{elems}[x]$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} \epsilon$.

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

3.4.10 Memory Instructions

$nt.\mathsf{load}(n,\ sx)^?\ x\ memop$

- $|C.\mathsf{mems}|$ must be greater than x.
- $n^?$ is ϵ if and only if $sx^?$ is ϵ.
- $2^{memop.\mathsf{align}}$ must be less than or equal to $|nt|/8$.
- If n is defined,
  - $2^{memop.\mathsf{align}}$ must be less than or equal to $n/8$.
  - $n/8$ must be less than $|nt|/8$.
- $n^?$ must be equal to ϵ.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\epsilon} nt$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} \leq |nt|/8 \qquad (2^{memop.\mathsf{align}} \leq n/8 < |nt|/8)^? \qquad n^? = \epsilon \lor nt = \mathsf{i}n}{C \vdash nt.\mathsf{load}(n\_sx)^?\ x\ memop : \mathsf{i32} \rightarrow nt}$$

$nt.\mathsf{store}n^?\ x\ memop$

- $|C.\mathsf{mems}|$ must be greater than x.
- $2^{memop.\mathsf{align}}$ must be less than or equal to $|nt|/8$.
- If n is defined,
  - $2^{memop.\mathsf{align}}$ must be less than or equal to $n/8$.
  - $n/8$ must be less than $|nt|/8$.
- $n^?$ must be equal to ϵ.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} \leq |nt|/8 \qquad (2^{memop.\mathsf{align}} \leq n/8 < |nt|/8)^? \qquad n^? = \epsilon \lor nt = \mathsf{i}n}{C \vdash nt.\mathsf{store}n^?\ x\ memop : \mathsf{i32}\ nt \rightarrow \epsilon}$$

$\mathsf{v128.loadshape}M N sx\ x\ memop$

- $|C.\mathsf{mems}|$ must be greater than x.
- $2^{memop.\mathsf{align}}$ must be less than or equal to $M/8 \cdot N$.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\epsilon} \mathsf{v128}$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} \leq M/8 \cdot N}{C \vdash \mathsf{v128.loadshape}M\mathsf{x}N sx\ x\ memop : \mathsf{i32} \rightarrow \mathsf{v128}}$$

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} \leq n/8}{C \vdash \mathsf{v128.loadsplat}n\ x\ memop : \mathsf{i32} \rightarrow \mathsf{v128}}$$

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} < n/8}{C \vdash \mathsf{v128.loadzero}n\ x\ memop : \mathsf{i32} \rightarrow \mathsf{v128}}$$

$\mathsf{v128.load}n\mathsf{\_lane}\ x\ memop\ laneidx$

- $|C.\mathsf{mems}|$ must be greater than x.
- $2^{memop.\mathsf{align}}$ must be less than $n/8$.
- $laneidx$ must be less than $128/n$.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} < n/8 \qquad laneidx < 128/n}{C \vdash \mathsf{v128.load}n\mathsf{\_lane}\ x\ memop\ laneidx : \mathsf{i32}\ \mathsf{v128} \rightarrow \mathsf{v128}}$$

$\mathsf{v128.store}\ x\ memop$

- $|C.\mathsf{mems}|$ must be greater than x.
- $2^{memop.\mathsf{align}}$ must be less than or equal to $|\mathsf{v128}|/8$.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} \leq |\mathsf{v128}|/8}{C \vdash \mathsf{v128.store}\ x\ memop : \mathsf{i32}\ \mathsf{v128} \rightarrow \epsilon}$$

$\mathsf{v128.store}n\_\mathsf{lane}\ x\ memop\ laneidx$

- $|C.\mathsf{mems}|$ must be greater than x.
- $2^{memop.\mathsf{align}}$ must be less than $n/8$.
- $laneidx$ must be less than $128/n$.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad 2^{memop.\mathsf{align}} < n/8 \qquad laneidx < 128/n}{C \vdash \mathsf{v128.store}n\_\mathsf{lane}\ x\ memop\ laneidx : \mathsf{i32}\ \mathsf{v128} \rightarrow \epsilon}$$

$\mathsf{memory.size}\ x$

- $|C.\mathsf{mems}|$ must be greater than x.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{C.\mathsf{mems}[x] = mt}{C \vdash \mathsf{memory.size}\ x : \epsilon \rightarrow \mathsf{i32}}$$

$\mathsf{memory.grow}\ x$

- $|C.\mathsf{mems}|$ must be greater than x.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\epsilon} \mathsf{i32}$.

$$\frac{C.\mathsf{mems}[x] = mt}{C \vdash \mathsf{memory.grow}\ x : \mathsf{i32} \rightarrow \mathsf{i32}}$$

$\mathsf{memory.fill}\ x$

- $|C.\mathsf{mems}|$ must be greater than x.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}}\ \rightarrow$.

$$\frac{C.\mathsf{mems}[x] = mt}{C \vdash \mathsf{memory.fill}\ x : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

memory.copy x_1 x_2

- $|C.\mathsf{mems}|$ must be greater than x_1.
- $|C.\mathsf{mems}|$ must be greater than x_2.
- Let mt_1 be $C.\mathsf{mems}[x_1]$.
- Let mt_2 be $C.\mathsf{mems}[x_2]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}} \rightarrow$.

$$\frac{C.\mathsf{mems}[x_1] = mt_1 \qquad C.\mathsf{mems}[x_2] = mt_2}{C \vdash \mathsf{memory.copy}\ x_1\ x_2 : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

memory.init x y

- $|C.\mathsf{mems}|$ must be greater than x.
- $|C.\mathsf{datas}|$ must be greater than y.
- $C.\mathsf{datas}[y]$ must be equal to ok.
- Let mt be $C.\mathsf{mems}[x]$.
- The instruction is valid with type $\mathsf{i32} \rightarrow_{\mathsf{i32}} \rightarrow$.

$$\frac{C.\mathsf{mems}[x] = mt \qquad C.\mathsf{datas}[y] = \mathsf{ok}}{C \vdash \mathsf{memory.init}\ x\ y : \mathsf{i32}\ \mathsf{i32}\ \mathsf{i32} \rightarrow \epsilon}$$

data.drop x

- $|C.\mathsf{datas}|$ must be greater than x.
- $C.\mathsf{datas}[x]$ must be equal to ok.
- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} \epsilon$.

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

3.4.11 Control Instructions

nop

- The instruction is valid with type $\epsilon \rightarrow_{\epsilon} \epsilon$.

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \rightarrow \epsilon}$$

unreachable

- Under the context C, $t_1^* \rightarrow_\epsilon t_2^*$ must be valid.
- The instruction is valid with type $t_1^* \rightarrow_\epsilon t_2^*$.

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

block bt $instr^*$

- Under the context $C[\mathsf{labels} = ..t_2^*]$, $instr^*$ must be valid with type $t_1^* \rightarrow_{x^*} t_2^*$.
- Under the context C, bt must be valid with type $t_1^* \rightarrow_\epsilon t_2^*$.
- The instruction is valid with type $t_1^* \rightarrow_\epsilon t_2^*$.

$$\frac{C \vdash bt : t_1^* \rightarrow t_2^* \qquad C, \mathsf{labels}\ (t_2^*) \vdash instr^* : t_1^* \rightarrow_{x^*} t_2^*}{C \vdash \mathsf{block}\ bt\ instr^* : t_1^* \rightarrow t_2^*}$$

loop bt $instr^*$

- Under the context $C[\mathsf{labels} = ..t_1^*]$, $instr^*$ must be valid with type $t_1^* \rightarrow_{x^*} t_2^*$.
- Under the context C, bt must be valid with type $t_1^* \rightarrow_\epsilon t_2^*$.
- The instruction is valid with type $t_1^* \rightarrow_\epsilon t_2^*$.

$$\frac{C \vdash bt : t_1^* \rightarrow t_2^* \qquad C, \mathsf{labels}\ (t_1^*) \vdash instr^* : t_1^* \rightarrow_{x^*} t_2^*}{C \vdash \mathsf{loop}\ bt\ instr^* : t_1^* \rightarrow t_2^*}$$

if bt $instr_1^*$ $instr_2^*$

- Under the context $C[\mathsf{labels} = ..t_2^*]$, $instr_1^*$ must be valid with type $t_1^* \rightarrow_{x_1^*} t_2^*$.
- Under the context C, bt must be valid with type $t_1^* \rightarrow_\epsilon t_2^*$.
- Under the context $C[\mathsf{labels} = ..t_2^*]$, $instr_2^*$ must be valid with type $t_1^* \rightarrow_{x_2^*} t_2^*$.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

$$\frac{C \vdash bt : t_1^* \rightarrow t_2^* \qquad C, \mathsf{labels}\ (t_2^*) \vdash instr_1^* : t_1^* \rightarrow_{x_1^*} t_2^* \qquad C, \mathsf{labels}\ (t_2^*) \vdash instr_2^* : t_1^* \rightarrow_{x_2^*} t_2^*}{C \vdash \mathsf{if}\ bt\ instr_1^*\ \mathsf{else}\ instr_2^* : t_1^*\ \mathsf{i32} \rightarrow t_2^*}$$

br l

- $|C.\mathsf{labels}|$ must be greater than l.
- Let t^* be $C.\mathsf{labels}[l]$.
- Under the context C, $t_1^* \rightarrow_\epsilon t_2^*$ must be valid.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

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

br_if l

- $|C.\mathsf{labels}|$ must be greater than l.
- Let t^* be $C.\mathsf{labels}[l]$.
- The instruction is valid with type $t^* \rightarrow_\rightarrow \epsilon$.

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

br_table l^* l'

- For all l in l^*,
  - $|C.\mathsf{labels}|$ must be greater than l.
- $|C.\mathsf{labels}|$ must be greater than l'.
- For all l in l^*,
  - YetI: TODO: prem_to_instrs rule_sub.
- YetI: TODO: prem_to_instrs rule_sub.
- Under the context C, $t_1^* \rightarrow_\epsilon t_2^*$ must be valid.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

$$\frac{(C \vdash t^* \leq C.\mathsf{labels}[l])^* \qquad C \vdash t^* \leq C.\mathsf{labels}[l'] \qquad C \vdash t_1^* \rightarrow t_2^* : \mathsf{ok}}{C \vdash \mathsf{br\_table}\ l^*\ l' : t_1^*\ t^* \rightarrow t_2^*}$$

br_on_null l

- $|C.\mathsf{labels}|$ must be greater than l.
- Under the context C, ht must be valid.
- Let t^* be $C.\mathsf{labels}[l]$.
- The instruction is valid with type $t^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{labels}[l] = t^* \qquad C \vdash ht : \mathsf{ok}}{C \vdash \mathsf{br\_on\_null}\ l : t^*\ (\mathsf{ref\ null}\ ht) \rightarrow t^*\ (\mathsf{ref}\ ht)}$$

br_on_non_null l

- $|C.\mathsf{labels}|$ must be greater than l.
- Let $t^*\ (\mathsf{ref}\ (\mathsf{null}\ \epsilon)\ ht)$ be $C.\mathsf{labels}[l]$.
- The instruction is valid with type $t^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{labels}[l] = t^*\ (\mathsf{ref}\ ht)}{C \vdash \mathsf{br\_on\_non\_null}\ l : t^*\ (\mathsf{ref\ null}\ ht) \rightarrow t^*}$$

br_on_cast l rt_1 rt_2

- $|C.\mathsf{labels}|$ must be greater than l.
- Under the context C, rt_1 must be valid.
- Under the context C, rt_2 must be valid.
- YetI: TODO: prem_to_instrs rule_sub.
- Let $t^*\ rt$ be $C.\mathsf{labels}[l]$.
- YetI: TODO: prem_to_instrs rule_sub.
- The instruction is valid with type $t^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{labels}[l] = t^*\ rt \qquad C \vdash rt_1 : \mathsf{ok} \qquad C \vdash rt_2 : \mathsf{ok} \qquad C \vdash rt_2 \leq rt_1 \qquad C \vdash rt_2 \leq rt}{C \vdash \mathsf{br\_on\_cast}\ l\ rt_1\ rt_2 : t^*\ rt_1 \rightarrow t^*\ (rt_1 \setminus rt_2)}$$

TODO (typo in DSL typing rule)

$$\frac{C.\mathsf{labels}[l] = t^*\ rt \qquad C \vdash rt_1 : \mathsf{ok} \qquad C \vdash rt_2 : \mathsf{ok} \qquad C \vdash rt_2 \leq rt_1 \qquad C \vdash rt_1 \setminus rt_2 \leq rt}{C \vdash \mathsf{br\_on\_cast\_fail}\ l\ rt_1\ rt_2 : t^*\ rt_1 \rightarrow t^*\ rt_2}$$

return

- Let t^* be C.return.
- Under the context C, $t_1^* \rightarrow_\epsilon t_2^*$ must be valid.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

$$\frac{C.\mathsf{return} = (t^*) \qquad C \vdash t_1^* \rightarrow t_2^* : \mathsf{ok}}{C \vdash \mathsf{return} : t_1^*\ t^* \rightarrow t_2^*}$$

call x

- $|C.\mathsf{funcs}|$ must be greater than x.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{funcs}[x])$.
- The instruction is valid with type $t_1^* \rightarrow_\epsilon t_2^*$.

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*)}{C \vdash \mathsf{call}\ x : t_1^* \rightarrow t_2^*}$$

call_ref x

- $|C.\mathsf{types}|$ must be greater than x.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*)}{C \vdash \mathsf{call\_ref}\ x : t_1^*\ (\mathsf{ref\ null}\ x) \rightarrow t_2^*}\ [\textsc{T-call\_ref}]$$

call_indirect x y

- $|C.\mathsf{tables}|$ must be greater than x.
- $|C.\mathsf{types}|$ must be greater than y.
- Let $(\mathit{lim}, \mathit{rt})$ be $C.\mathsf{tables}[x]$.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{types}[y])$.
- YetI: TODO: prem_to_instrs rule_sub.
- The instruction is valid with type $t_1^* \rightarrow_\rightarrow \epsilon$.

$$\frac{C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt} \qquad C \vdash \mathit{rt} \leq (\mathsf{ref\ null\ func}) \qquad C.\mathsf{types}[y] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*)}{C \vdash \mathsf{call\_indirect}\ x\ y : t_1^*\ \mathsf{i32} \rightarrow t_2^*}\ [\textsc{T-call\_indirect}]$$

return_call x

- $|C.\mathsf{funcs}|$ must be greater than x.
- Under the context C, $t_3^* \rightarrow_\epsilon t_4^*$ must be valid.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{funcs}[x])$.
- YetI: TODO: prem_to_instrs rule_sub.
- $C.\mathsf{return}$ must be equal to ${t'}_2^*$.
- The instruction is valid with type $t_3^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = ({t'}_2^*) \qquad C \vdash t_2^* \leq {t'}_2^* \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok}}{C \vdash \mathsf{return\_call}\ x : t_3^*\ t_1^* \rightarrow t_4^*}\ [\textsc{T-return\_call}]$$

return_call_ref x

- $|C.\mathsf{types}|$ must be greater than x.
- Under the context C, $t_3^* \rightarrow_\epsilon t_4^*$ must be valid.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{types}[x])$.
- YetI: TODO: prem_to_instrs rule_sub.
- $C.\mathsf{return}$ must be equal to ${t'}_2^*$.
- The instruction is valid with type $t_3^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = ({t'}_2^*) \qquad C \vdash t_2^* \leq {t'}_2^* \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok}}{C \vdash \mathsf{return\_call\_ref}\ x : t_3^*\ t_1^*\ (\mathsf{ref\ null}\ x) \rightarrow t_4^*}\ [\textsc{T-return\_call\_ref}]$$

return_call_indirect x y

- $|C.\mathsf{tables}|$ must be greater than x.
- $|C.\mathsf{types}|$ must be greater than y.
- Under the context C, $t_3^* \rightarrow_\epsilon t_4^*$ must be valid.
- Let $(\mathit{lim}, \mathit{rt})$ be $C.\mathsf{tables}[x]$.
- Let $(\mathsf{func}\ t_1^* \rightarrow t_2^*)$ be $\mathrm{expand}(C.\mathsf{types}[y])$.
- YetI: TODO: prem_to_instrs rule_sub.
- YetI: TODO: prem_to_instrs rule_sub.
- $C.\mathsf{return}$ must be equal to ${t'}_2^*$.
- The instruction is valid with type $t_3^* \rightarrow_{\rightarrow} \epsilon$.

$$\frac{\begin{array}{c} C.\mathsf{tables}[x] = \mathit{lim}\ \mathit{rt} \qquad C \vdash \mathit{rt} \leq (\mathsf{ref\ null\ func}) \\ C.\mathsf{types}[y] \approx \mathsf{func}\ (t_1^* \rightarrow t_2^*) \qquad C.\mathsf{return} = ({t'}_2^*) \qquad C \vdash t_2^* \leq {t'}_2^* \qquad C \vdash t_3^* \rightarrow t_4^* : \mathsf{ok} \end{array}}{C \vdash \mathsf{return\_call\_indirect}\ x\ y : t_3^*\ t_1^*\ \mathsf{i32} \rightarrow t_4^*}\ [\textsc{T-return\_call\_indirect}]$$

3.4.12 Instruction Sequences

$$\frac{}{C \vdash \epsilon : \epsilon \rightarrow \epsilon}\,[\textsc{T-instr*-empty}]$$

$$\frac{C \vdash \mathit{instr}_1 : t_1^* \rightarrow_{x_1^*} t_2^* \qquad (C.\mathsf{locals}[x_1] = \mathit{init}\ t)^* \qquad C[\mathsf{local}[x_1^*] = (\mathsf{set}\ t)^*] \vdash \mathit{instr}_2^* : t_2^* \rightarrow}{C \vdash \mathit{instr}_1\ \mathit{instr}_2^* : t_1^* \rightarrow_{x_1^*\ x_2^*} t_3^*}$$

3.4.13 Expressions

$$\frac{}{C \vdash (\mathit{nt}.\mathsf{const}\ c_{\mathit{nt}})\ \mathsf{const}}\,[\textsc{C-instr-const}] \qquad \frac{}{C \vdash (\mathit{vt}.\mathsf{const}\ c_{\mathit{vt}})\ \mathsf{const}}\,[\textsc{C-instr-vconst}] \qquad \frac{}{C \vdash (\mathsf{ref.null}\ \mathit{ht})\ \mathsf{const}}\,[\textsc{C-instr-ref.null}]$$

$\mathit{binop} \in \mathit{binop}^*_{u0}$

1. Return false.
2. Assert: Due to validation, $|\mathit{binop}^*_{u0}|$ is greater than or equal to 1.
3. Let $\mathit{ibinop}_1\ \mathit{ibinop}'^*$ be binop^*_{u0}.
4. Return binop is ibinop_1 or $\mathit{binop} \in \mathit{ibinop}'^*$.

$$\begin{array}{lcl}
\mathit{binop} \in \mathit{epsilon} & = & \mathsf{false} \\
\mathit{binop} \in (\mathit{ibinop}_1)\ \mathit{ibinop}'^* & = & \mathit{binop} = \mathit{ibinop}_1 \vee \mathit{binop} \in \mathit{ibinop}'^*
\end{array}$$

$\mathit{nt} \in \mathit{numty}^*_{u0}$

1. Return false.
2. Assert: Due to validation, $|\mathit{numty}^*_{u0}|$ is greater than or equal to 1.
3. Let $\mathit{nt}_1\ \mathit{nt}'^*$ be numty^*_{u0}.
4. Return nt is nt_1 or $\mathit{nt} \in \mathit{nt}'^*$.

$$\begin{array}{lcl}
\mathit{nt} \in \mathit{epsilon} & = & \mathsf{false} \\
\mathit{nt} \in \mathit{nt}_1\ \mathit{nt}'^* & = & \mathit{nt} = \mathit{nt}_1 \vee \mathit{nt} \in \mathit{nt}'^*
\end{array}$$

3.5 Modules

3.5.1 Types

$$\frac{x = |C.\mathsf{types}| \qquad dt^* = \mathrm{roll}_x(\mathit{rectype}) \qquad C[\mathsf{types} = ..dt^*] \vdash \mathit{rectype} : \mathsf{ok}(x)}{C \vdash \mathsf{type}\ \mathit{rectype} : dt^*} \ [\textsc{T-type}]$$

$$\frac{}{C \vdash \epsilon : \epsilon} \ [\textsc{T-types-empty}] \qquad \frac{C \vdash \mathit{type}_1 : dt_1 \qquad C[\mathsf{types} = ..dt_1^*] \vdash \mathit{type}^* : dt^*}{C \vdash \mathit{type}_1\ \mathit{type}^* : dt_1^*\ dt^*} \ [\textsc{T-types-cons}]$$

3.5.2 Functions

$$\frac{C.\mathsf{types}[x] \approx \mathsf{func}\ (t_1^* \to t_2^*) \qquad (C \vdash \mathit{local} : \mathit{lt})^* \qquad C, \mathsf{locals}\ (\mathsf{set}\ t_1)^*\ \mathit{lt}^*, \mathsf{labels}\ (t_2^*), \mathsf{return}\ (t_2^*) \vdash \mathit{expr} : t_2^*}{C \vdash \mathsf{func}\ x\ \mathit{local}^*\ \mathit{expr} : C.\mathsf{types}[x]}$$

3.5.3 Locals

$$\frac{\mathrm{default}_t \neq \epsilon}{C \vdash \mathsf{local}\ t : \mathsf{set}\ t} \ [\textsc{T-local-set}] \qquad \frac{\mathrm{default}_t = \epsilon}{C \vdash \mathsf{local}\ t : \mathsf{unset}\ t} \ [\textsc{T-local-unset}]$$

3.5.4 Tables

$$\frac{C \vdash \mathit{tt} : \mathsf{ok} \qquad \mathit{tt} = \mathit{limits}\ \mathit{rt} \qquad C \vdash \mathit{expr} : \mathit{rt}\ \mathsf{const}}{C \vdash \mathsf{table}\ \mathit{tt}\ \mathit{expr} : \mathit{tt}}$$

3.5.5 Memories

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

3.5.6 Globals

$$\frac{C \vdash \mathit{gt} : \mathsf{ok} \qquad \mathit{gt} = \mathit{mut}\ t \qquad C \vdash \mathit{expr} : t\ \mathsf{const}}{C \vdash \mathsf{global}\ \mathit{gt}\ \mathit{expr} : \mathit{gt}} \ [\textsc{T-global}]$$

$$\frac{}{C \vdash \epsilon : \epsilon} \ [\textsc{T-globals-empty}] \qquad \frac{C \vdash \mathit{global} : \mathit{gt}_1 \qquad C[\mathsf{globals} = ..\mathit{gt}_1] \vdash \mathit{global}^* : \mathit{gt}^*}{C \vdash \mathit{global}_1\ \mathit{global}^* : \mathit{gt}_1\ \mathit{gt}^*} \ [\textsc{T-globals-cons}]$$

3.5.7 Element Segments

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

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

3.5.8 Data Segments

$$\frac{C \vdash \mathit{datamode} : \mathsf{ok}}{C \vdash \mathsf{data}\ b^*\ \mathit{datamode} : \mathsf{ok}}\ [\textsc{T-data}]$$

$$\frac{C.\mathsf{mems}[x] = \mathit{mt} \qquad (C \vdash \mathit{expr} : \mathsf{i32}\ \mathsf{const})^*}{C \vdash \mathsf{active}\ x\ \mathit{expr} : \mathsf{ok}}\ [\textsc{T-datamode-active}] \qquad \frac{}{C \vdash \mathsf{passive} : \mathsf{ok}}\ [\textsc{T-datamode-passive}]$$

3.5.9 Start Function

$$\frac{C.\mathsf{funcs}[x] \approx \mathsf{func}\ (\epsilon \to \epsilon)}{C \vdash \mathsf{start}\ x : \mathsf{ok}}$$

3.5.10 Exports

$$\frac{C \vdash \mathit{externidx} : \mathit{xt}}{C \vdash \mathsf{export}\ \mathit{name}\ \mathit{externidx} : \mathit{xt}}\ [\textsc{T-export}]$$

$$\frac{C.\mathsf{funcs}[x] = \mathit{dt}}{C \vdash \mathsf{func}\ x : \mathsf{func}\ \mathit{dt}}\ [\textsc{T-externidx-func}] \qquad \frac{C.\mathsf{globals}[x] = \mathit{gt}}{C \vdash \mathsf{global}\ x : \mathsf{global}\ \mathit{gt}}\ [\textsc{T-externidx-global}] \qquad \frac{C.\mathsf{tables}[x] = \mathit{tt}}{C \vdash \mathsf{table}\ x : \mathsf{table}\ \mathit{tt}}\ [\textsc{T-externidx-table}]$$

3.5.11 Imports

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

3.5.12 Modules

$$\frac{\begin{array}{c} \{\} \vdash \mathit{type}^* : \mathit{dt'}^* \qquad (\{\mathsf{types}\ \mathit{dt'}^*\} \vdash \mathit{import} : \mathit{ixt})^* \\ C' \vdash \mathit{global}^* : \mathit{gt}^* \qquad (C' \vdash \mathit{table} : \mathit{tt})^* \qquad (C' \vdash \mathit{mem} : \mathit{mt})^* \qquad (C \vdash \mathit{func} : \mathit{dt})^* \\ (C \vdash \mathit{elem} : \mathit{rt})^* \qquad (C \vdash \mathit{data} : \mathsf{ok})^n \qquad (C \vdash \mathit{start} : \mathsf{ok})^? \qquad (C \vdash \mathit{export} : \mathit{xt})^* \\ C = \{\mathsf{types}\ \mathit{dt'}^*,\ \mathsf{funcs}\ \mathit{idt}^*\ \mathit{dt}^*,\ \mathsf{globals}\ \mathit{igt}^*\ \mathit{gt}^*,\ \mathsf{tables}\ \mathit{itt}^*\ \mathit{tt}^*,\ \mathsf{mems}\ \mathit{imt}^*\ \mathit{mt}^*,\ \mathsf{elems}\ \mathit{rt}^*,\ \mathsf{datas}\ \mathsf{ok}^n\} \\ C' = \{\mathsf{types}\ \mathit{dt'}^*,\ \mathsf{funcs}\ \mathit{idt}^*\ \mathit{dt}^*,\ \mathsf{globals}\ \mathit{igt}^*\} \\ \mathit{idt}^* = \mathrm{funcs}(\mathit{ixt}^*) \qquad \mathit{igt}^* = \mathrm{globals}(\mathit{ixt}^*) \qquad \mathit{itt}^* = \mathrm{tables}(\mathit{ixt}^*) \qquad \mathit{imt}^* = \mathrm{mems}(\mathit{ixt}^*) \end{array}}{\vdash \mathsf{module}\ \mathit{type}^*\ \mathit{import}^*\ \mathit{func}^*\ \mathit{global}^*\ \mathit{table}^*\ \mathit{mem}^*\ \mathit{elem}^*\ \mathit{data}^n\ \mathit{start}^?\ \mathit{export}^* : \mathsf{ok}}$$

CHAPTER 4

Execution

4.1 Conventions

4.1.1 General Constants

Ki

1. Return 1024.

$$\mathrm{Ki} = 1024$$

$\mathrm{concat}_{X^*_{u0}}$

1. If X^*_{u0} is ϵ, then:

    a. Return ϵ.

2. Let $w^*\ w'^{**}$ be X^*_{u0}.
3. Return $w^*\ \mathrm{concat}_{w'^{**}}$.

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

4.1.2 Formal Notation

$$
\begin{array}{llcll}
[\textsc{E-pure}] & z; instr^* & \hookrightarrow & z; {instr'}^* & \text{if } instr^* \hookrightarrow {instr'}^* \\
[\textsc{E-read}] & z; instr^* & \hookrightarrow & z; {instr'}^* & \text{if } z; instr^* \hookrightarrow {instr'}^* \\
[\textsc{E-refl}] & z; instr^* & \hookrightarrow^* & z; instr^* & \\
[\textsc{E-trans}] & z; instr^* & \hookrightarrow^* & z''; {instr''}^* & \text{if } z; instr^* \hookrightarrow z'; {instr'}^* \\
 & & & & \wedge\ z'; instr' \hookrightarrow^* z''; {instr''}^*
\end{array}
$$

4.1.3 Size

$|numty_{u0}|$

1. If $numty_{u0}$ is $\mathsf{i32}$, then:
   a. Return 32.
2. If $numty_{u0}$ is $\mathsf{i64}$, then:
   a. Return 64.
3. If $numty_{u0}$ is $\mathsf{f32}$, then:
   a. Return 32.
4. Assert: Due to validation, $numty_{u0}$ is $\mathsf{f64}$.
5. Return 64.

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

$|packt_{u0}|$

1. If $packt_{u0}$ is $\mathsf{i8}$, then:
   a. Return 8.
2. Assert: Due to validation, $packt_{u0}$ is $\mathsf{i16}$.
3. Return 16.

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

$|stora_{u0}|$

1. If the type of $stora_{u0}$ is numtype, then:
   a. Let $numtype$ be $stora_{u0}$.
   b. Return $|numtype|$.
2. If the type of $stora_{u0}$ is vectype, then:
   a. Let $vectype$ be $stora_{u0}$.
   b. Return $|vectype|$.
3. Assert: Due to validation, the type of $stora_{u0}$ is packtype.
4. Let $packtype$ be $stora_{u0}$.
5. Return $|packtype|$.

$$\begin{array}{lcl} |numtype| & = & |numtype| \\ |vectype| & = & |vectype| \\ |packtype| & = & |packtype| \end{array}$$

4.1.4 Projections

$\mathrm{funcs}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xt^*$ be $exter^*_{u0}$.
3. If y_0 is of the case func, then:
   a. Let $(\mathsf{func}\ dt)$ be y_0.
   b. Return $dt\ \mathrm{funcs}(xt^*)$.
4. Let $externtype\ xt^*$ be $exter^*_{u0}$.
5. Return $\mathrm{funcs}(xt^*)$.

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

$\mathrm{globals}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xt^*$ be $exter^*_{u0}$.
3. If y_0 is of the case global, then:
   a. Let $(\mathsf{global}\ gt)$ be y_0.
   b. Return $gt\ \mathrm{globals}(xt^*)$.

4. Let $externtype\ xt^*$ be $exter^*_{u0}$.
5. Return $\mathrm{globals}(xt^*)$.

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

$\mathrm{tables}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xt^*$ be $exter^*_{u0}$.
3. If y_0 is of the case table, then:
   a. Let $(\mathsf{table}\ tt)$ be y_0.
   b. Return $tt\ \mathrm{tables}(xt^*)$.
4. Let $externtype\ xt^*$ be $exter^*_{u0}$.
5. Return $\mathrm{tables}(xt^*)$.

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

$\mathrm{mems}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xt^*$ be $exter^*_{u0}$.
3. If y_0 is of the case mem, then:
   a. Let $(\mathsf{mem}\ mt)$ be y_0.
   b. Return $mt\ \mathrm{mems}(xt^*)$.
4. Let $externtype\ xt^*$ be $exter^*_{u0}$.
5. Return $\mathrm{mems}(xt^*)$.

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

4.1.5 Packed Fields

$\mathrm{pack}_{\mathit{stora}_{u\,0}}(\mathit{val}_{u\,1})$

1. If the type of $\mathit{stora}_{u\,0}$ is valtype, then:
   a. Let val be $\mathit{val}_{u\,1}$.
   b. Return val.
2. Assert: Due to validation, $\mathit{val}_{u\,1}$ is of the case const.
3. Let $(y_0.\mathsf{const}\ i)$ be $\mathit{val}_{u\,1}$.
4. Assert: Due to validation, y_0 is $\mathsf{i32}$.
5. Assert: Due to validation, the type of $\mathit{stora}_{u\,0}$ is packtype.
6. Let pt be $\mathit{stora}_{u\,0}$.
7. Return $(\mathit{pt}.\mathsf{pack}\ \mathrm{wrap}_{32,|\mathit{pt}|}(i))$.

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

$\mathrm{unpack}^{\mathit{sx}^?_{u\,1}}_{\mathit{stora}_{u\,0}}(\mathit{field}_{u\,2})$

1. If $\mathit{sx}^?_{u\,1}$ is not defined, then:
   a. Assert: Due to validation, the type of $\mathit{stora}_{u\,0}$ is valtype.
   b. Assert: Due to validation, the type of $\mathit{field}_{u\,2}$ is val.
   c. Let val be $\mathit{field}_{u\,2}$.
   d. Return val.
2. Else:
   a. Let sx be $\mathit{sx}^?_{u\,1}$.
   b. Assert: Due to validation, $\mathit{field}_{u\,2}$ is of the case pack.
   c. Let $(\mathit{pt}.\mathsf{pack}\ i)$ be $\mathit{field}_{u\,2}$.
   d. Assert: Due to validation, $\mathit{stora}_{u\,0}$ is pt.
   e. Return $(\mathsf{i32}.\mathsf{const}\ \mathrm{ext}^{\mathit{sx}}_{|\mathit{pt}|,32}(i))$.

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

$\mathrm{sx}(\mathit{stora}_{u0})$

1. If the type of stora_{u0} is consttype, then:
   a. Return ϵ.
2. Assert: Due to validation, the type of stora_{u0} is packtype.
3. Return s.

$$\begin{array}{lcl} \mathrm{sx}(\mathit{consttype}) & = & \epsilon \\ \mathrm{sx}(\mathit{packtype}) & = & \mathsf{s} \end{array}$$

4.2 Numerics

4.2.1 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 $i - 2^N$.

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

$\mathrm{signed}_N^{-1}(i)$

1. Let j be $\mathrm{inverse}_{of_{signed}}(N, i)$.
2. Return j.

$$\begin{array}{lcll} \mathrm{signed}_N^{-1}(i) & = & j & \text{if } \mathrm{signed}_N(j) = i \end{array}$$

4.3 Runtime Structure

4.3.1 Values

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

$\mathrm{default}_{\mathit{valty}_{u0}}$

1. If valty_{u0} is $\mathsf{i32}$, then:
   a. Return $(\mathsf{i32.const}\ 0)$.
2. If valty_{u0} is $\mathsf{i64}$, then:
   a. Return $(\mathsf{i64.const}\ 0)$.
3. If valty_{u0} is $\mathsf{f32}$, then:
   a. Return $(\mathsf{f32.const}\ {+0})$.
4. If valty_{u0} is $\mathsf{f64}$, then:
   a. Return $(\mathsf{f64.const}\ {+0})$.
5. If valty_{u0} is $\mathsf{v128}$, then:
   a. Return $(\mathsf{v128.const}\ 0)$.
6. Assert: Due to validation, valty_{u0} is of the case ref.
7. Let $(\mathsf{ref}\ y_0\ \mathit{ht})$ be valty_{u0}.
8. If y_0 is $(\mathsf{null}\ ())$, then:
   a. Return $(\mathsf{ref.null}\ \mathit{ht})$.
9. Assert: Due to validation, y_0 is $(\mathsf{null}\ \epsilon)$.
10. Return ϵ.

$$
\begin{array}{lcl}
\mathrm{default}_{\mathsf{i32}} & = & (\mathsf{i32.const}\ 0) \\
\mathrm{default}_{\mathsf{i64}} & = & (\mathsf{i64.const}\ 0) \\
\mathrm{default}_{\mathsf{f32}} & = & (\mathsf{f32.const}\ {+0}) \\
\mathrm{default}_{\mathsf{f64}} & = & (\mathsf{f64.const}\ {+0}) \\
\mathrm{default}_{\mathsf{v128}} & = & (\mathsf{v128.const}\ 0) \\
\mathrm{default}_{\mathsf{ref}\ \mathsf{null}\ \mathit{ht}} & = & (\mathsf{ref.null}\ \mathit{ht}) \\
\mathrm{default}_{\mathsf{ref}\ \epsilon\ \mathit{ht}} & = & \epsilon
\end{array}
$$

4.3.2 Results

$$\mathit{result} \quad ::= \quad \mathit{val}^* \mid \mathsf{trap}$$

4.3.3 Store

$$\begin{array}{lll} \mathit{store} & ::= & \{\mathsf{funcs}~\mathit{funcinst}^*, \\ && \phantom{\{}\mathsf{globals}~\mathit{globalinst}^*, \\ && \phantom{\{}\mathsf{tables}~\mathit{tableinst}^*, \\ && \phantom{\{}\mathsf{mems}~\mathit{meminst}^*, \\ && \phantom{\{}\mathsf{elems}~\mathit{eleminst}^*, \\ && \phantom{\{}\mathsf{datas}~\mathit{datainst}^*, \\ && \phantom{\{}\mathsf{structs}~\mathit{structinst}^*, \\ && \phantom{\{}\mathsf{arrays}~\mathit{arrayinst}^*\} \end{array}$$

4.3.4 Addresses

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

4.3.5 Module Instances

$$\begin{array}{lll} \mathit{moduleinst} & ::= & \{\mathsf{types}~\mathit{deftype}^*, \\ && \phantom{\{}\mathsf{funcs}~\mathit{funcaddr}^*, \\ && \phantom{\{}\mathsf{globals}~\mathit{globaladdr}^*, \\ && \phantom{\{}\mathsf{tables}~\mathit{tableaddr}^*, \\ && \phantom{\{}\mathsf{mems}~\mathit{memaddr}^*, \\ && \phantom{\{}\mathsf{elems}~\mathit{elemaddr}^*, \\ && \phantom{\{}\mathsf{datas}~\mathit{dataaddr}^*, \\ && \phantom{\{}\mathsf{exports}~\mathit{exportinst}^*\} \end{array}$$

moduleinst

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

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

4.3.6 Function Instances

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

funcinst

1. Return s.funcs.

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

4.3.7 Table Instances

$$\begin{array}{lcl} \mathit{tableinst} & ::= & \{\mathsf{type}~\mathit{tabletype}, \\ & & \;\mathsf{refs}~\mathit{ref}^*\} \end{array}$$

tableinst

1. Return s.tables.

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

4.3.8 Memory Instances

$$\begin{array}{lcl} \mathit{meminst} & ::= & \{\mathsf{type}~\mathit{memtype}, \\ & & \;\mathsf{bytes}~\mathit{byte}^*\} \end{array}$$

meminst

1. Return s.mems.

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

4.3.9 Global Instances

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

globalinst

1. Return s.globals.

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

4.3.10 Element Instances

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

eleminst

1. Return s.elems.

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

4.3.11 Data Instances

$$\mathit{datainst} \quad ::= \quad \{\mathsf{bytes}~\mathit{byte}^*\}$$

datainst

1. Return s.datas.

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

4.3.12 Export Instances

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

4.3.13 External Values

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

$\mathrm{funcs}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be $exter^*_{u0}$.
3. If y_0 is of the case func, then:
   a. Let $(\mathsf{func}\ fa)$ be y_0.
   b. Return $fa\ \mathrm{funcs}(xv^*)$.
4. Let $externval\ xv^*$ be $exter^*_{u0}$.
5. Return $\mathrm{funcs}(xv^*)$.

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

$\mathrm{tables}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be $exter^*_{u0}$.
3. If y_0 is of the case table, then:
   a. Let $(\mathsf{table}\ ta)$ be y_0.
   b. Return $ta\ \mathrm{tables}(xv^*)$.
4. Let $externval\ xv^*$ be $exter^*_{u0}$.
5. Return $\mathrm{tables}(xv^*)$.

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

$\mathrm{mems}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0\ xv^*$ be $exter^*_{u0}$.
3. If y_0 is of the case mem, then:
   a. Let $(\mathsf{mem}\ ma)$ be y_0.
   b. Return $ma\ \mathrm{mems}(xv^*)$.
4. Let $externval\ xv^*$ be $exter^*_{u0}$.
5. Return $\mathrm{mems}(xv^*)$.

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

$\mathrm{globals}(exter^*_{u0})$

1. If $exter^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $y_0~xv^*$ be $exter^*_{u0}$.
3. If y_0 is of the case global, then:
   a. Let $(\mathsf{global}~ga)$ be y_0.
   b. Return $ga~\mathrm{globals}(xv^*)$.
4. Let $externval~xv^*$ be $exter^*_{u0}$.
5. Return $\mathrm{globals}(xv^*)$.

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

4.3.14 Aggregate Instances

$$
\begin{array}{llcl}
\text{(structure instance)} & structinst & ::= & \{\mathsf{type}~deftype, \\
 & & & \quad \mathsf{fields}~fieldval^*\} \\
\text{(array instance)} & arrayinst & ::= & \{\mathsf{type}~deftype, \\
 & & & \quad \mathsf{fields}~fieldval^*\} \\
\text{(field value)} & fieldval & ::= & val \mid packval \\
\text{(packed value)} & packval & ::= & packtype.\mathsf{pack}~pack_{packtype}
\end{array}
$$

arrayinst

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

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

structinst

1. Return s.structs.

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

4.3.15 Stack

Activation Frames

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

4.3.16 Administrative Instructions

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

4.3.17 Configurations

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

4.3.18 Evaluation Contexts

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

4.3.19 Typing

store

1. Return.

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

frame

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

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

$$\frac{}{s \vdash \mathsf{ref.null}\ ht : (\mathsf{ref\ null}\ ht)}\ [\textsc{Ref\_ok-null}] \qquad \frac{}{s \vdash \mathsf{ref.i31}\ i : (\mathsf{ref}\ \epsilon\ \mathsf{i31})}\ [\textsc{Ref\_ok-i31}] \qquad \frac{s.\mathsf{structs}[a].\mathsf{type} = dt}{s \vdash \mathsf{ref.struct}\ a : (\mathsf{ref}\ \epsilon\ dt)}\ [\textsc{Ref\_ok-struct}]$$

4.4 Instructions

4.4.1 Numeric Instructions

$nt.unop$

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

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

$nt.binop$

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

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

$nt.testop$

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

$$(nt.\mathsf{const}\ c_1)\ (nt.testop) \quad \hookrightarrow \quad (\mathsf{i32}.\mathsf{const}\ c) \qquad \text{if } c = testop_{nt}(c_1)$$

$nt.relop$

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

$$(nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.relop) \quad \hookrightarrow \quad (\mathsf{i32}.\mathsf{const}\ c) \qquad \text{if } c = relop_{nt}(c_1,\ c_2)$$

$nt_2.cvtop\_nt_1\_sx^?$

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

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

4.4.2 Vector Instructions

$\mathsf{v128}.\mathit{vvunop}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $\mathit{vvunop}\_\,\mathsf{v128}(c_1)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vvunop}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128}.\mathit{vvunop}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{vvunop}\_\,\mathsf{v128}(c_1) = c$$

$\mathsf{v128}.\mathit{vvbinop}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $\mathit{vvbinop}\_\,\mathsf{v128}(c_1,\ c_2)$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vvbinop}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{v128}.\mathit{vvbinop}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{vvbinop}\_\,\mathsf{v128}(c_1,\ c_2) = c$$

$\mathsf{v128}.\mathit{vvternop}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_3)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
7. Let c be $\mathit{vvternop}\_\,\mathsf{v128}(c_1,\ c_2,\ c_3)$.
8. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vvternop}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (\mathsf{v128.const}\ c_3)\ (\mathsf{v128}.\mathit{vvternop}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathit{vvternop}\_\,\mathsf{v128}(c_1,\ c_2,\ c_3) =$$

v128.any_true

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $\mathrm{ine}_{|\mathsf{v128}|}(c_1, 0)$.
4. Push the value $(\mathsf{i32.const}\ c)$ to the stack.

$$[\textsc{E-vvtestop}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.any\_true}) \quad \hookrightarrow \quad (\mathsf{i32.const}\ c) \qquad \text{if } c = \mathrm{ine}_{|\mathsf{v128}|}(c_1, 0)$$

$\mathsf{i}n\mathsf{x}N$.shuffle i^*

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Assert: Due to validation, for all $(k)^{k<N}$, k is less than $|i^*|$.
6. Let c'^* be $\mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_1)\ \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_2)$.
7. Assert: Due to validation, for all $(k)^{k<N}$, $i^*[k]$ is less than $|c'^*|$.
8. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(c'^*[i^*[k]]^{k<N})$.
9. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vshuffle}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ ((\mathsf{i}n\mathsf{x}N).\mathsf{shuffle}\ i^*) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \begin{array}{l} \text{if } c'^* = \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_1)\ \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_2) \\ \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(c'^*[i^*[k]]^{k<N}) \end{array}$$

$\mathsf{i}n\mathsf{x}N$.splat

1. Assert: Due to validation, a value of value type $\mathrm{unpack}(\mathsf{i}n)$ is on the top of the stack.
2. Pop the value $(nt_0.\mathsf{const}\ c_1)$ from the stack.
3. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(\mathrm{pack}_{\mathsf{i}n}(c_1)^N)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vsplat}](\mathrm{unpack}(\mathsf{i}n).\mathsf{const}\ c_1)\ ((\mathsf{i}n\mathsf{x}N).\mathsf{splat}) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(\mathrm{pack}_{\mathsf{i}n}(c_1)^N)$$

$lanet_{u0} \times N.\mathsf{extract\_lane\_}sx^?_{u1}\ i$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. If $sx^?_{u1}$ is not defined and the type of $lanet_{u0}$ is numtype, then:
   a. Let nt be $lanet_{u0}$.
   b. If i is less than $|\mathrm{lanes}_{nt \times N}(c_1)|$, then:
      1) Let c_2 be $\mathrm{lanes}_{nt \times N}(c_1)[i]$.
      2) Push the value $(nt.\mathsf{const}\ c_2)$ to the stack.
4. If the type of $lanet_{u0}$ is packtype, then:
   a. Let pt be $lanet_{u0}$.
   b. If $sx^?_{u1}$ is defined, then:
      1) Let sx be $sx^?_{u1}$.
      2) If i is less than $|\mathrm{lanes}_{pt \times N}(c_1)|$, then:
         a) Let c_2 be $\mathrm{ext}^{sx}_{|pt|,32}(\mathrm{lanes}_{pt \times N}(c_1)[i])$.
         b) Push the value $(\mathsf{i32.const}\ c_2)$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-vextract\_lane-num}] & (\mathsf{v128.const}\ c_1)\ (\mathsf{vextract\_lane}\ (nt \times N)\ i) & \hookrightarrow & (nt.\mathsf{const}\ c_2) & \text{if } c_2 = \mathrm{lanes}_{nt \times N}(c_1)[i] \\
[\textsc{E-vextract\_lane-pack}] & (\mathsf{v128.const}\ c_1)\ ((pt \times N).\mathsf{extract\_lane\_}sx\ i) & \hookrightarrow & (\mathsf{i32.const}\ c_2) & \text{if } c_2 = \mathrm{ext}^{sx}_{|pt|,32}(\mathrm{lanes}_{pt \times N}(c_1)[i])
\end{array}
$$

$\mathsf{i}n \times N.\mathsf{replace\_lane}\ i$

1. Assert: Due to validation, a value of value type $\mathrm{unpack}(\mathsf{i}n)$ is on the top of the stack.
2. Pop the value $(nt_0.\mathsf{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n \times N}(\mathrm{lanes}_{\mathsf{i}n \times N}(c_1)[[i] = \mathrm{pack}_{\mathsf{i}n}(c_2)])$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$
[\textsc{E-vreplace\_lane}] (\mathsf{v128.const}\ c_1)\ (\mathrm{unpack}(\mathsf{i}n).\mathsf{const}\ c_2)\ ((\mathsf{i}n \times N).\mathsf{replace\_lane}\ i) \hookrightarrow (\mathsf{v128.const}\ c) \quad \text{if } c = \mathrm{lanes}^{-1}_{\mathsf{i}n \times N}(\mathrm{lanes}_{\mathsf{i}n \times}
$$

$sh.vunop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $vunop\_\ sh(c_1)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vunop}](\mathsf{v128.const}\ c_1)\ (sh.vunop) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } c = vunop\_\,sh(c_1)$$

$sh.vbinop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. If $|vbinop\_\,sh(c_1,\ c_2)|$ is 1, then:
   a. Let c be $vbinop\_\,sh(c_1,\ c_2)$.
   b. Push the value $(\mathsf{v128.const}\ c)$ to the stack.
6. If $vbinop\_\,sh(c_1,\ c_2)$ is ϵ, then:
   a. Trap.

$$\begin{array}{lcll}
[\textsc{E-vbinop-val}]\ (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.vbinop) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } vbinop_\,sh(c_1,\ c_2) = c \\
[\textsc{E-vbinop-trap}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.vbinop) & \hookrightarrow & \mathsf{trap} & \text{if } vbinop_\,sh(c_1,\ c_2) = \epsilon
\end{array}$$

$sh.vrelop$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $vrelop\_\,sh(c_1,\ c_2)$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\textsc{E-vrelop}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh.vrelop) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } vrelop\_\,sh(c_1,\ c_2) = c$$

$\mathsf{i}n\mathsf{x}N.vshiftop$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c'^* be $\mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_1)$.
6. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(vshiftop\_\,\mathsf{i}n\mathsf{x}N(c',\ n)^*)$.
7. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$
\begin{array}{llll}
[\textsc{E-vshiftop}] (\mathsf{v128.const}\ c_1)\ (\mathsf{i32.const}\ n)\ ((\mathsf{i}n\mathsf{x}N).\mathit{vshiftop}) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } {c'}^* = \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c_1) \\
 & & & \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(\mathit{vshiftop}\_\ \mathsf{i}n\mathsf{x}N(c',\ n)^*)
\end{array}
$$

$\mathsf{i}n\mathsf{x}N$.all_true

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c)$ from the stack.
3. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c)$.
4. If for all $(ci_1)^*$, ci_1 is not 0, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
5. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

$$
\begin{array}{llll}
[\textsc{E-vtestop-true}]\ (\mathsf{v128.const}\ c)\ ((\mathsf{i}n\mathsf{x}N).\mathsf{all\_true}) & \hookrightarrow & (\mathsf{i32.const}\ 1) & \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c) \\
 & & & \wedge\ (ci_1 \neq 0)^* \\
[\textsc{E-vtestop-false}] (\mathsf{v128.const}\ c)\ ((\mathsf{i}n\mathsf{x}N).\mathsf{all\_true}) & \hookrightarrow & (\mathsf{i32.const}\ 0) & \text{otherwise}
\end{array}
$$

$\mathsf{i}n\mathsf{x}N$.bitmask

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c)$ from the stack.
3. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c)$.
4. Let ci be $\mathrm{inverse}_{of\ _{ibits}}(32, \mathrm{ilt}^{\mathsf{s}}_{|\mathsf{i}n|}(ci_1,\ 0)^*)$.
5. Push the value $(\mathsf{i32.const}\ ci)$ to the stack.

$$
\begin{array}{llll}
[\textsc{E-vbitmask}] (\mathsf{v128.const}\ c)\ ((\mathsf{i}n\mathsf{x}N).\mathsf{bitmask}) & \hookrightarrow & (\mathsf{i32.const}\ ci) & \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}n\mathsf{x}N}(c) \\
 & & & \wedge\ \mathrm{bits}_{\mathsf{i32}}(ci) = \mathrm{ilt}^{\mathsf{s}}_{|\mathsf{i}n|}(ci_1,\ 0)^*
\end{array}
$$

$\mathsf{i}n_2\mathsf{x}N_2$.narrow_$\mathsf{i}n_1\mathsf{x}N_1$_$sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let ci_1^* be $\mathrm{lanes}_{\mathsf{i}n_1\mathsf{x}N_1}(c_1)$.
6. Let ci_2^* be $\mathrm{lanes}_{\mathsf{i}n_1\mathsf{x}N_1}(c_2)$.
7. Let cj_1^* be $\mathrm{narrow}^{sx}_{|\mathsf{i}n_1|,|\mathsf{i}n_2|}\, {ci_1}^*$.
8. Let cj_2^* be $\mathrm{narrow}^{sx}_{|\mathsf{i}n_1|,|\mathsf{i}n_2|}\, {ci_2}^*$.

9. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n_2 \times N_2}(cj_1^*~cj_2^*)$.
10. Push the value $(\mathsf{v128.const}~c)$ to the stack.

$$[\textsc{E-vnarrow}]\,(\mathsf{v128.const}~c_1)~(\mathsf{v128.const}~c_2)~((\mathsf{i}n_2\mathsf{x}N_2).\mathsf{narrow\_i}n_1\mathsf{x}N_1\mathsf{\_}sx) \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \quad \begin{array}{l} \text{if } ci_1^* = \mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_1) \\ \wedge~ ci_2^* = \mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_2) \\ \wedge~ cj_1^* = \mathrm{narrow}^{sx}_{|\mathsf{i}n_1|,|\mathsf{i}n_2|}\,{ci_1}^* \\ \wedge~ cj_2^* = \mathrm{narrow}^{sx}_{|\mathsf{i}n_1|,|\mathsf{i}n_2|}\,{ci_2}^* \\ \wedge~ c = \mathrm{lanes}^{-1}_{\mathsf{i}n_2\times N_2}(cj_1^*~cj_2^*) \end{array}$$

$lanet_{u1}\mathsf{x}N_2.vcvtop\_half^?_{u0}\_lanet_{u2}\mathsf{x}N_1\_sx^?\_zero^?_{u3}$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}~c_1)$ from the stack.
3. If $half^?_{u0}$ is not defined and $zero^?_{u3}$ is not defined, then:
   a. Let $\mathsf{i}n_1$ be $lanet_{u2}$.
   b. Let $\mathsf{i}n_2$ be $lanet_{u1}$.
   c. Let c'^* be $\mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_1)$.
   d. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n_2\times N_2}(\mathrm{vcvtop}(\mathsf{i}n_1\mathsf{x}N_1, \mathsf{i}n_2\mathsf{x}N_2, vcvtop, sx^?, c')^*)$.
   e. Push the value $(\mathsf{v128.const}~c)$ to the stack.
4. If $zero^?_{u3}$ is not defined and $half^?_{u0}$ is defined, then:
   a. Let hf be $half^?_{u0}$.
   b. Let $\mathsf{i}n_1$ be $lanet_{u2}$.
   c. Let $\mathsf{i}n_2$ be $lanet_{u1}$.
   d. Let ci^* be $\mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_1)[\mathrm{halfop}(hf, 0, N_2) : N_2]$.
   e. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n_2\times N_2}(\mathrm{vcvtop}(\mathsf{i}n_1\mathsf{x}N_1, \mathsf{i}n_2\mathsf{x}N_2, vcvtop, sx^?, ci)^*)$.
   f. Push the value $(\mathsf{v128.const}~c)$ to the stack.
5. If $half^?_{u0}$ is not defined and $zero^?_{u3}$ is zero and the type of $lanet_{u2}$ is numtype, then:
   a. Let nt_1 be $lanet_{u2}$.
   b. If the type of $lanet_{u1}$ is numtype, then:
      1) Let nt_2 be $lanet_{u1}$.
      2) Let ci^* be $\mathrm{lanes}_{nt_1\times N_1}(c_1)$.
      3) Let c be $\mathrm{lanes}^{-1}_{nt_2\times N_2}(\mathrm{vcvtop}(nt_1\mathsf{x}N_1, nt_2\mathsf{x}N_2, vcvtop, sx^?, ci)^*~\mathrm{zero}(nt_2)^{N_1})$.
      4) Push the value $(\mathsf{v128.const}~c)$ to the stack.

$$[\textsc{E-vcvtop-normal}]\quad (\mathsf{v128.const}~c_1)~((\mathsf{i}n_2\mathsf{x}N_2).vcvtop\_\epsilon\_\mathsf{i}n_1\mathsf{x}N_1\_sx^?) \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \quad \begin{array}{l} \text{if } c'^* = \mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_1) \\ \wedge~ c = \mathrm{lanes}^{-1}_{\mathsf{i}n_2\times N_2}(\mathrm{vcvtop}(\mathsf{i}n_1\mathsf{x}N \end{array}$$

$$[\textsc{E-vcvtop-half}]\quad (\mathsf{v128.const}~c_1)~((\mathsf{i}n_2\mathsf{x}N_2).vcvtop\_hf\_\mathsf{i}n_1\mathsf{x}N_1\_sx^?) \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \quad \begin{array}{l} \text{if } ci^* = \mathrm{lanes}_{\mathsf{i}n_1\times N_1}(c_1)[\mathrm{halfop}(\\ \wedge~ c = \mathrm{lanes}^{-1}_{\mathsf{i}n_2\times N_2}(\mathrm{vcvtop}(\mathsf{i}n_1\mathsf{x}N \end{array}$$

$$[\textsc{E-vcvtop-zero}]\quad (\mathsf{v128.const}~c_1)~((nt_2\mathsf{x}N_2).vcvtop\_\epsilon\_nt_1\mathsf{x}N_1\_sx^?\_\mathsf{zero}) \quad\hookrightarrow\quad (\mathsf{v128.const}~c) \quad \begin{array}{l} \text{if } ci^* = \mathrm{lanes}_{nt_1\times N_1}(c_1) \\ \wedge~ c = \mathrm{lanes}^{-1}_{nt_2\times N_2}(\mathrm{vcvtop}(nt_1\mathsf{x} \end{array}$$

$sh_1.vextunop\_sh_2\_sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
3. Let c be $\mathrm{vextunop}(sh_1, sh_2, vextunop, sx, c_1)$.
4. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\text{E-VEXTUNOP}](\mathsf{v128.const}\ c_1)\ (sh_1.vextunop\_sh_2\_sx) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathrm{vextunop}(sh_1, sh_2, vextunop, sx, c_1) = c$$

$sh_1.vextbinop\_sh_2\_sx$

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $(\mathsf{v128.const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
5. Let c be $\mathrm{vextbinop}(sh_1, sh_2, vextbinop, sx, c_1, c_2)$.
6. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$[\text{E-VEXTBINOP}](\mathsf{v128.const}\ c_1)\ (\mathsf{v128.const}\ c_2)\ (sh_1.vextbinop\_sh_2\_sx) \quad \hookrightarrow \quad (\mathsf{v128.const}\ c) \qquad \text{if } \mathrm{vextbinop}(sh_1, sh_2, vextbinop, sx$$

4.4.3 Reference Instructions

$\mathsf{ref.func}\ x$

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module.funcs}|$.
3. Push the value $(\mathsf{ref.func}\ z.\mathsf{module.funcs}[x])$ to the stack.

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

ref.is_null

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value ref from the stack.
3. If ref is of the case ref.null, then:
   a. Push the value $(\mathsf{i32.const}\ 1)$ to the stack.
4. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

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

ref.as_non_null

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

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

ref.eq

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value ref_2 from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value 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 the value $(\mathsf{i32.const}~1)$ to the stack.
6. Else if ref_1 is ref_2, then:
   a. Push the value $(\mathsf{i32.const}~1)$ to the stack.
7. Else:
   a. Push the value $(\mathsf{i32.const}~0)$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-ref.eq-null}] & \mathit{ref}_1~\mathit{ref}_2~\mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}~1) & \text{if } \mathit{ref}_1 = (\mathsf{ref.null}~\mathit{ht}_1) \wedge \mathit{ref}_2 = (\mathsf{ref.null}~\mathit{ht}_2) \\
[\textsc{E-ref.eq-true}] & \mathit{ref}_1~\mathit{ref}_2~\mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}~1) & \text{otherwise, if } \mathit{ref}_1 = \mathit{ref}_2 \\
[\textsc{E-ref.eq-false}] & \mathit{ref}_1~\mathit{ref}_2~\mathsf{ref.eq} & \hookrightarrow & (\mathsf{i32.const}~0) & \text{otherwise}
\end{array}
$$

ref.test rt

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt' be $\mathrm{ref}_{\mathit{type}_{\mathit{of}}}(\mathit{ref})$.
5. If rt' matches $\mathrm{inst}_{f.\mathsf{module}}(rt)$, then:
   a. Push the value $(\mathsf{i32.const}~1)$ to the stack.

6. Else:
   a. Push the value $(\mathsf{i32.const}\ 0)$ to the stack.

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

ref.cast rt

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt' be $\mathrm{ref}_{\mathit{type}_{\mathit{of}}}(\mathit{ref})$.
5. If rt' does not match $\mathrm{inst}_{f.\mathsf{module}}(\mathit{rt})$, then:
   a. Trap.
6. Push the value ref to the stack.

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

ref.i31

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
3. Push the value $(\mathsf{ref.i31}\ \mathrm{wrap}_{32,31}(i))$ 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 the value admin_{u0} from the stack.
3. If admin_{u0} is of the case ref.null, then:
   a. Trap.
4. If admin_{u0} is of the case ref.i31_num, then:
   a. Let $(\mathsf{ref.i31}\ i)$ be admin_{u0}.
   b. Push the value $(\mathsf{i32.const}\ \mathrm{ext}_{31,32}^{\mathit{sx}}(i))$ to the stack.

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

$\mathrm{ext}_{structinst}(si^*)$

1. Let f be the current frame.
2. Return $(s[\mathsf{structs} = ..si^*],\, f)$.

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

struct.new x

1. Let z be the current state.
2. Let a be $|z.\mathsf{structs}|$.
3. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
4. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
5. Let $(mut,\ zt)^n$ be y_0.
6. Assert: Due to validation, there are at least n values on the top of the stack.
7. Pop the values val^n from the stack.
8. Let si be $\{\mathsf{type}\ z.\mathsf{types}[x],\ \mathsf{fields}\ \mathrm{pack}_{zt}(val)^n\}$.
9. Push the value $(\mathsf{ref.struct}\ a)$ to the stack.
10. Perform $z[\mathsf{structs} = ..si]$.

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

struct.new_default x

1. Let z be the current state.
2. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
3. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
4. Let $(mut,\ zt)^*$ be y_0.
5. Assert: Due to validation, $|mut^*|$ is $|zt^*|$.
6. Assert: Due to validation, for all $(zt)^*$, $\mathrm{default}_{\mathrm{unpack}(zt)}$ is defined.
7. Let val^* be $\mathrm{default}^*_{\mathrm{unpack}(zt)}$.
8. Assert: Due to validation, $|val^*|$ is $|zt^*|$.
9. Push the values val^* to the stack.

10. Execute the instruction $(\mathsf{struct.new}\ x)$.

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

$\mathsf{struct.get\_}sx^?\ x\ i$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value $admin_{u0}$ from the stack.
4. If $admin_{u0}$ is of the case $\mathsf{ref.null}$, then:
   a. Trap.
5. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
6. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
7. Let $(mut,\ zt)^*$ be y_0.
8. If $admin_{u0}$ is of the case $\mathsf{ref.struct\_addr}$, then:
   a. Let $(\mathsf{ref.struct}\ a)$ be $admin_{u0}$.
   b. If i is less than $|z.\mathsf{structs}[a].\mathsf{fields}|$ and a is less than $|z.\mathsf{structs}|$ and $|mut^*|$ is $|zt^*|$ and i is less than $|zt^*|$, then:
      1) Push the value $\mathrm{unpack}^{sx^?}_{zt^*[i]}(z.\mathsf{structs}[a].\mathsf{fields}[i])$ to the stack.

$$[\textsc{E-struct.get-null}]\quad z; (\mathsf{ref.null}\ ht)\ (\mathsf{struct.get\_}sx^?\ x\ i) \hookrightarrow \mathsf{trap}$$
$$[\textsc{E-struct.get-struct}]\ z; (\mathsf{ref.struct}\ a)\ (\mathsf{struct.get\_}sx^?\ x\ i) \hookrightarrow \mathrm{unpack}^{sx^?}_{zt^*[i]}(z.\mathsf{structs}[a].\mathsf{fields}[i]) \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{struct}\ (mut\ z$$

$\mathsf{struct.set}\ x\ i$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value $admin_{u0}$ from the stack.
6. If $admin_{u0}$ is of the case $\mathsf{ref.null}$, then:
   a. Trap.
7. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case struct.
8. Let $(\mathsf{struct}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
9. Let $(mut,\ zt)^*$ be y_0.
10. If $admin_{u0}$ is of the case $\mathsf{ref.struct\_addr}$, then:
    a. Let $(\mathsf{ref.struct}\ a)$ be $admin_{u0}$.
    b. If $|mut^*|$ is $|zt^*|$ and i is less than $|zt^*|$, then:

1) Perform $z[\mathsf{structs}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt^*[i]}(val)]$.

$$[\textsc{E-struct.set-null}] \quad z; (\mathsf{ref.null}\ ht)\ val\ (\mathsf{struct.set}\ x\ i) \quad \hookrightarrow \quad z; \mathsf{trap}$$
$$[\textsc{E-struct.set-struct}] z; (\mathsf{ref.struct}\ a)\ val\ (\mathsf{struct.set}\ x\ i) \quad \hookrightarrow \quad z[\mathsf{structs}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt^*[i]}(val)]; \epsilon \qquad \text{if } z.\mathsf{types}[x] \approx \mathsf{struct}\ (m$$

array.new x

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value val from the stack.
5. Push the values val^n to the stack.
6. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

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

array.new_default x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
5. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
6. Let $(mut,\ zt)$ be y_0.
7. Assert: Due to validation, $\mathrm{default}_{\mathrm{unpack}(zt)}$ is defined.
8. Let val be $\mathrm{default}_{\mathrm{unpack}(zt)}$.
9. Push the values val^n to the stack.
10. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

$$[\textsc{E-array.new\_default}] z; (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_default}\ x) \quad \hookrightarrow \quad val^n\ (\mathsf{array.new\_fixed}\ x\ n) \qquad \begin{array}{l} \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \\ \wedge\ \mathrm{default}_{\mathrm{unpack}(zt)} = val \end{array}$$

$\mathrm{ext}_{arrayinst}(ai^*)$

1. Let f be the current frame.
2. Return $(s[\mathsf{arrays} = ..ai^*],\ f)$.

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

array.new_fixed x n

1. Let z be the current state.
2. Assert: Due to validation, there are at least n values on the top of the stack.
3. Pop the values val^n from the stack.
4. Let a be $|z.\mathsf{arrays}|$.
5. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
6. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
7. Let $(mut,\ zt)$ be y_0.
8. Let ai be $\{\mathsf{type}\ z.\mathsf{types}[x],\ \mathsf{fields}\ \mathrm{pack}_{zt}(val)^n\}$.
9. Push the value $(\mathsf{ref.array}\ a)$ to the stack.
10. Perform $z[\mathsf{arrays} = ..ai]$.

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

array.new_elem x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If $i + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:
   a. Trap.
7. Let ref^n be $z.\mathsf{elems}[y].\mathsf{refs}[i : n]$.
8. Push the values ref^n to the stack.
9. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

$$[\textsc{E-array.new\_elem-oob}]\ z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_elem}\ x\ y) \quad \hookrightarrow \quad \mathsf{trap} \quad \text{if } i + n > |z.\mathsf{elems}[y].\mathsf{refs}|$$
$$[\textsc{E-array.new\_elem-alloc}]\ z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_elem}\ x\ y) \quad \hookrightarrow \quad ref^n\ (\mathsf{array.new\_fixed}\ x\ n) \\ \text{if } ref^n = z.\mathsf{elems}[y].\mathsf{refs}[i : n]$$

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 the value $(\mathsf{i32.const}\ n)$ from the stack.
3. Assert: Due to validation, a value of value type i32 is on the top of the stack.
4. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
5. If $\mathrm{expand}(\mathrm{type}(x))$ is of the case array, then:
    a. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(\mathrm{type}(x))$.
    b. Let $(mut,\ zt)$ be y_0.
    c. If $i + n \cdot |zt|/8$ is greater than $|\mathrm{data}(y).\mathsf{bytes}|$, then:
        1) Trap.
    d. Let t be $\mathrm{unpack}(zt)$.
    e. Let b^* be $\mathrm{data}(y).\mathsf{bytes}[i : n \cdot |zt|/8]$.
    f. Let gb^* be $\mathrm{group}_{bytes_{by}}(|zt|/8, b^*)$.
    g. Let c^n be $\mathrm{inverse}_{of_{ibytes}}(|zt|, gb)^*$.
    h. Push the values $(t.\mathsf{const}\ c)^n$ to the stack.
    i. Execute the instruction $(\mathsf{array.new\_fixed}\ x\ n)$.

$$
\begin{array}{lcl}
[\textsc{E-array.new\_data-oob}]\ z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
 & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \\
 & & \quad \wedge\ i + n \cdot |zt|/8 > |z.\mathsf{datas}[y].\mathsf{bytes}| \\
[\textsc{E-array.new\_data-num}]\ z; (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.new\_data}\ x\ y) & \hookrightarrow & (\mathrm{unpack}(zt).\mathsf{const}\ \mathrm{unpackconst}(zt, c))^n\ (\mathsf{array.new} \\
 & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) \\
 & & \quad \wedge\ \mathrm{concat}(\mathrm{bytes}_{zt}(c)^n) = z.\mathsf{datas}[y].\mathsf{bytes}[i : n \cdot |
\end{array}
$$

array.get_$sx^?$ x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value $admin_{u0}$ from the stack.
6. If $admin_{u0}$ is of the case ref.null, then:
    a. Trap.
7. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and i is greater than or equal to $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
        1) Trap.
8. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
9. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
10. Let $(mut,\ zt)$ be y_0.

11. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. If i is less than $|z.\mathsf{arrays}[a].\mathsf{fields}|$ and a is less than $|z.\mathsf{arrays}|$, then:
        1) Push the value $\mathrm{unpack}_{zt}^{sx^?}(z.\mathsf{arrays}[a].\mathsf{fields}[i])$ to the stack.

$$
\begin{array}{llcl}
[\textsc{E-array.get-null}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-array.get-oob}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) & \hookrightarrow & \mathsf{trap} \quad \text{if } i \geq |z.\mathsf{arrays}[a].\mathsf{fields}| \\
[\textsc{E-array.get-array}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{array.get\_}sx^?\ x) & \hookrightarrow & \mathrm{unpack}_{zt}^{sx^?}(z.\mathsf{arrays}[a].\mathsf{fields}[i]) \\
 & & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt)
\end{array}
$$

array.set x

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value is on the top of the stack.
7. Pop the value $admin_{u0}$ from the stack.
8. If $admin_{u0}$ is of the case ref.null, then:
    a. Trap.
9. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and i is greater than or equal to $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
        1) Trap.
10. Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x])$ is of the case array.
11. Let $(\mathsf{array}\ y_0)$ be $\mathrm{expand}(z.\mathsf{types}[x])$.
12. Let $(mut,\ zt)$ be y_0.
13. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. Perform $z[\mathsf{arrays}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt}(val)]$.

$$
\begin{array}{llcl}
[\textsc{E-array.set-null}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} \\
[\textsc{E-array.set-oob}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z; \mathsf{trap} \quad \text{if } i \geq |z.\mathsf{arrays}[a].\mathsf{fields}| \\
[\textsc{E-array.set-array}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z[\mathsf{arrays}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt}(val)]; \epsilon \\
 & & & \quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt)
\end{array}
$$

array.len

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value $admin_{u0}$ from the stack.
4. If $admin_{u0}$ is of the case ref.null, then:
   a. Trap.
5. If $admin_{u0}$ is of the case ref.array_addr, then:
   a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
   b. If a is less than $|z.\mathsf{arrays}|$, then:
      1) Push the value $(\mathsf{i32.const}\ |z.\mathsf{arrays}[a].\mathsf{fields}|)$ to the stack.

$$
\begin{array}{llcl}
[\textsc{E-array.len-null}] & z; (\mathsf{ref.null}\ ht)\ \mathsf{array.len} & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-array.len-array}] & z; (\mathsf{ref.array}\ a)\ \mathsf{array.len} & \hookrightarrow & (\mathsf{i32.const}\ |z.\mathsf{arrays}[a].\mathsf{fields}|)
\end{array}
$$

array.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value $admin_{u0}$ from the stack.
10. If $admin_{u0}$ is of the case ref.null, then:
    a. Trap.
11. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
       1) Trap.
    c. If n is 0, then:
       1) Do nothing.
    d. Else:
       1) Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
       2) Push the value $(\mathsf{ref.array}\ a)$ to the stack.
       3) Push the value $(\mathsf{i32.const}\ i)$ to the stack.
       4) Push the value val to the stack.
       5) Execute the instruction $(\mathsf{array.set}\ x)$.

6) Push the value $(\mathsf{ref.array}\ a)$ to the stack.
7) Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
8) Push the value val to the stack.
9) Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
10) Execute the instruction $(\mathsf{array.fill}\ x)$.

$$
\begin{array}{llll}
[\textsc{E-array.fill-null}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow\ \mathsf{trap} & \\
[\textsc{E-array.fill-oob}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow\ \mathsf{trap} & \text{if } i + n > |z.\mathsf{arrays}[a].\mathsf{fields}| \\
[\textsc{E-array.fill-zero}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow\ \epsilon & \text{otherwise, if } n = 0 \\
[\textsc{E-array.fill-succ}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{array.fill}\ x) & \hookrightarrow & \\
& \quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & & \text{otherwise} \\
& \quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i + 1)\ val\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.fill}\ x) & &
\end{array}
$$

ARRAY.COPY

$\mathsf{array.copy}\ x_1\ x_2$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i_2)$ from the stack.
6. Assert: Due to validation, a value is on the top of the stack.
7. Pop the value $admin_{u1}$ from the stack.
8. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
9. Pop the value $(\mathsf{i32.const}\ i_1)$ from the stack.
10. Assert: Due to validation, a value is on the top of the stack.
11. Pop the value $admin_{u0}$ from the stack.
12. If $admin_{u0}$ is of the case $\mathsf{ref.null}$ and the type of $admin_{u1}$ is ref, then:
    a. Trap.
13. If $admin_{u1}$ is of the case $\mathsf{ref.null}$ and the type of $admin_{u0}$ is ref, then:
    a. Trap.
14. If $admin_{u0}$ is of the case $\mathsf{ref.array\_addr}$, then:
    a. Let $(\mathsf{ref.array}\ a_1)$ be $admin_{u0}$.
    b. If $admin_{u1}$ is of the case $\mathsf{ref.array\_addr}$, then:
        1) If a_1 is less than $|z.\mathsf{arrays}|$ and $i_1 + n$ is greater than $|z.\mathsf{arrays}[a_1].\mathsf{fields}|$, then:
            a) Trap.
        2) Let $(\mathsf{ref.array}\ a_2)$ be $admin_{u1}$.
        3) If a_2 is less than $|z.\mathsf{arrays}|$ and $i_2 + n$ is greater than $|z.\mathsf{arrays}[a_2].\mathsf{fields}|$, then:
            a) Trap.
    c. If n is 0, then:

1) If $admin_{u1}$ is of the case ref.array_addr, then:
   a) Do nothing.

d. Else if i_1 is greater than i_2, then:
   1) Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x_2])$ is of the case array.
   2) Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x_2])$.
   3) Let (mut, zt_2) be y_0.
   4) Let (ref.array a_1) be $admin_{u0}$.
   5) If $admin_{u1}$ is of the case ref.array_addr, then:
      a) Let (ref.array a_2) be $admin_{u1}$.
      b) Let $sx^?$ be $\mathrm{sx}(zt_2)$.
      c) Push the value (ref.array a_1) to the stack.
      d) Push the value (i32.const $i_1 + n - 1$) to the stack.
      e) Push the value (ref.array a_2) to the stack.
      f) Push the value (i32.const $i_2 + n - 1$) to the stack.
      g) Execute the instruction (array.get_$sx^?$ x_2).
      h) Execute the instruction (array.set x_1).
      i) Push the value (ref.array a_1) to the stack.
      j) Push the value (i32.const i_1) to the stack.
      k) Push the value (ref.array a_2) to the stack.
      l) Push the value (i32.const i_2) to the stack.
      m) Push the value (i32.const $n - 1$) to the stack.
      n) Execute the instruction (array.copy x_1 x_2).

e. Else:
   1) Assert: Due to validation, $\mathrm{expand}(z.\mathsf{types}[x_2])$ is of the case array.
   2) Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x_2])$.
   3) Let (mut, zt_2) be y_0.
   4) Let (ref.array a_1) be $admin_{u0}$.
   5) If $admin_{u1}$ is of the case ref.array_addr, then:
      a) Let (ref.array a_2) be $admin_{u1}$.
      b) Let $sx^?$ be $\mathrm{sx}(zt_2)$.
      c) Push the value (ref.array a_1) to the stack.
      d) Push the value (i32.const i_1) to the stack.
      e) Push the value (ref.array a_2) to the stack.
      f) Push the value (i32.const i_2) to the stack.
      g) Execute the instruction (array.get_$sx^?$ x_2).
      h) Execute the instruction (array.set x_1).
      i) Push the value (ref.array a_1) to the stack.
      j) Push the value (i32.const $i_1 + 1$) to the stack.
      k) Push the value (ref.array a_2) to the stack.

l) Push the value $(\mathsf{i32.const}\ i_2 + 1)$ to the stack.

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

n) Execute the instruction $(\mathsf{array.copy}\ x_1\ x_2)$.

$$
\begin{array}{llcl}
[\textsc{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} \\
[\textsc{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} \\
[\textsc{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} \\
& \quad \text{if } i_1 + n > |z.\mathsf{arrays}[a_1].\mathsf{fields}| & & \\
[\textsc{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} \\
& \quad \text{if } i_2 + n > |z.\mathsf{arrays}[a_2].\mathsf{fields}| & & \\
[\textsc{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 \\
& \quad \text{otherwise, if } n = 0 & & \\
[\textsc{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{i32.const}\ i_1) & & \\
& (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2) & & \\
& (\mathsf{array.get\_}sx^?\ x_2)\ (\mathsf{array.set}\ x_1) & & \\
& (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1 + 1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2 + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.copy}\ x_1\ x_2) & & \\
& \quad \text{otherwise, if } z.\mathsf{types}[x_2] \approx \mathsf{array}\ (mut\ zt_2) & & \\
& \quad \wedge\ i_1 \leq i_2 \wedge sx^? = \mathrm{sx}(zt_2) & & \\
[\textsc{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{i32.const}\ i_1 + n - 1) & & \\
& (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2 + n - 1) & & \\
& (\mathsf{array.get\_}sx^?\ x_2)\ (\mathsf{array.set}\ x_1) & & \\
& (\mathsf{ref.array}\ a_1)\ (\mathsf{i32.const}\ i_1)\ (\mathsf{ref.array}\ a_2)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.copy}\ x_1\ x_2) & & \\
& \quad \text{otherwise, if } z.\mathsf{types}[x_2] \approx \mathsf{array}\ (mut\ zt_2) & & \\
& \quad \wedge\ sx^? = \mathrm{sx}(zt_2) & &
\end{array}
$$

array.init_elem x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value $admin_{u0}$ from the stack.
10. If $admin_{u0}$ is of the case ref.null, then:
    a. Trap.
11. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
        1) Trap.
12. If $j + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:
    a. If $admin_{u0}$ is of the case ref.array_addr, then:

1) Trap.

b. If n is 0 and j is less than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:

1) Let ref be $z.\mathsf{elems}[y].\mathsf{refs}[j]$.

2) If $admin_{u0}$ is of the case ref.array_addr, then:

a) Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.

b) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

c) Push the value $(\mathsf{i32.const}\ i)$ to the stack.

d) Push the value ref to the stack.

e) Execute the instruction $(\mathsf{array.set}\ x)$.

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

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

h) Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.

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

j) Execute the instruction $(\mathsf{array.init\_elem}\ x\ y)$.

13. Else if n is 0, then:

a. If $admin_{u0}$ is of the case ref.array_addr, then:

1) Do nothing.

14. Else:

a. If j is less than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:

1) Let ref be $z.\mathsf{elems}[y].\mathsf{refs}[j]$.

2) If $admin_{u0}$ is of the case ref.array_addr, then:

a) Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.

b) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

c) Push the value $(\mathsf{i32.const}\ i)$ to the stack.

d) Push the value ref to the stack.

e) Execute the instruction $(\mathsf{array.set}\ x)$.

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

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

h) Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.

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

j) Execute the instruction $(\mathsf{array.init\_elem}\ x\ y)$.

$$\begin{array}{llcl}
[\text{E-ARRAY.INIT_ELEM-NULL}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\text{E-ARRAY.INIT_ELEM-OOB1}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
 & \quad \text{if } i + n > |z.\mathsf{arrays}[a].\mathsf{fields}| & & \\
[\text{E-ARRAY.INIT_ELEM-OOB2}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_elem}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
 & \quad \text{if } j + n > |z.\mathsf{elems}[y].\mathsf{refs}| & & \\
[\text{E-ARRAY.INIT_ELEM-ZERO}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_elem}\ x\ y) & \hookrightarrow & \epsilon \\
 & \quad \text{otherwise, if } n = 0 & & \\
[\text{E-ARRAY.INIT_ELEM-SUCC}] & z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init_elem}\ x\ y) & \hookrightarrow & \\
 & (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ \mathit{ref}\ (\mathsf{array.set}\ x) & & \\
 & (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i+1)\ (\mathsf{i32.const}\ j+1)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{array.init_elem}\ x\ y) & & \\
 & \quad \text{otherwise, if } \mathit{ref} = z.\mathsf{elems}[y].\mathsf{refs}[j] & &
\end{array}$$

array.init_data x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value (i32.const n) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const j) from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value (i32.const i) from the stack.
8. Assert: Due to validation, a value is on the top of the stack.
9. Pop the value $admin_{u0}$ from the stack.
10. If $admin_{u0}$ is of the case ref.null, then:
    a. Trap.
11. If $admin_{u0}$ is of the case ref.array_addr, then:
    a. Let (ref.array a) be $admin_{u0}$.
    b. If a is less than $|z.\mathsf{arrays}|$ and $i + n$ is greater than $|z.\mathsf{arrays}[a].\mathsf{fields}|$, then:
       1) Trap.
12. If $\mathrm{expand}(z.\mathsf{types}[x])$ is not of the case array, then:
    a. If n is 0 and $admin_{u0}$ is of the case ref.array_addr, then:
       1) Do nothing.
13. Else:
    a. Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.
    b. Let (mut, zt) be y_0.
    c. If $admin_{u0}$ is of the case ref.array_addr, then:
       1) If $j + n \cdot |zt|/8$ is greater than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:
          a) Trap.
       2) If n is 0, then:
          a) Do nothing.
       3) Else:
          a) Let (array y_0) be $\mathrm{expand}(z.\mathsf{types}[x])$.

b) Let $(mut,\ zt)$ be y_0.

c) Let $(\mathsf{ref.array}\ a)$ be $admin_{u0}$.

d) Let c be $\mathrm{inverse}_{of_{zbytes}}(zt, z.\mathsf{datas}[y].\mathsf{bytes}[j : |zt|/8])$.

e) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

f) Push the value $(\mathsf{i32.const}\ i)$ to the stack.

g) Push the value $\mathrm{unpack}(zt).\mathsf{const}\ \mathrm{unpackconst}(zt, c)$ to the stack.

h) Execute the instruction $(\mathsf{array.set}\ x)$.

i) Push the value $(\mathsf{ref.array}\ a)$ to the stack.

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

k) Push the value $(\mathsf{i32.const}\ j + |zt|/8)$ to the stack.

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

m) Execute the instruction $(\mathsf{array.init\_data}\ x\ y)$.

$$
\begin{array}{lll}
[\textsc{E-array.init\_data-null}]\ z; (\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-array.init\_data-oob1}]\ z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } i + n > |z.\mathsf{arrays}[a].\mathsf{fields}| & & \\
[\textsc{E-array.init\_data-oob2}]\ z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_data}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
\quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) & & \\
\quad \wedge\ j + n \cdot |zt|/8 > |z.\mathsf{datas}[y].\mathsf{bytes}| & & \\
[\textsc{E-array.init\_data-zero}]\ z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_data}\ x\ y) & \hookrightarrow & \epsilon \\
\quad \text{otherwise, if } n = 0 & & \\
[\textsc{E-array.init\_data-num}]\ z; (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ n)\ (\mathsf{array.init\_data}\ x\ y) & \hookrightarrow & \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ (\mathrm{unpack}(zt).\mathsf{const}\ \mathrm{unpackconst}(zt, c))\ (\mathsf{array.set}\ x) & & \\
\quad (\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i + 1)\ (\mathsf{i32.const}\ j + |zt|/8)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{array.init\_data}\ x\ y) & & \\
\quad \text{otherwise, if } z.\mathsf{types}[x] \approx \mathsf{array}\ (mut\ zt) & & \\
\quad \wedge\ \mathrm{bytes}_{zt}(c) = z.\mathsf{datas}[y].\mathsf{bytes}[j : |zt|/8] & &
\end{array}
$$

extern.convert_any

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $admin_{u0}$ from the stack.
3. If $admin_{u0}$ is of the case ref.null, then:
    a. Push the value $(\mathsf{ref.null}\ \mathsf{extern})$ to the stack.
4. If the type of $admin_{u0}$ is addrref, then:
    a. Let $addrref$ be $admin_{u0}$.
    b. Push the value $(\mathsf{ref.extern}\ addrref)$ to the stack.

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

any.convert_extern

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $admin_{u0}$ from the stack.
3. If $admin_{u0}$ is of the case ref.null, then:
   a. Push the value (ref.null any) to the stack.
4. If $admin_{u0}$ is of the case ref.extern, then:
   a. Let (ref.extern $addrref$) be $admin_{u0}$.
   b. Push the value $addrref$ to the stack.

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

4.4.4 Parametric Instructions

drop

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

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

select $t^{*?}$

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

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

4.4.5 Variable Instructions

$\mathsf{local.get}\ x$

1. Let z be the current state.
2. Assert: Due to validation, $z.\mathsf{locals}[x]$ is defined.
3. Let val be $z.\mathsf{locals}[x]$.
4. Push the value val to the stack.

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

$\mathsf{local.set}\ x$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Perform $z[\mathsf{locals}[x] = val]$.

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

$\mathsf{local.tee}\ x$

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

$$val\ (\mathsf{local.tee}\ x) \quad \hookrightarrow \quad val\ val\ (\mathsf{local.set}\ x)$$

$\mathsf{global.get}\ x$

1. Let z be the current state.
2. Let val be $z.\mathsf{globals}[x].\mathsf{value}$.
3. Push the value val to the stack.

$$z; (\mathsf{global.get}\ x) \quad \hookrightarrow \quad val \qquad \text{if } z.\mathsf{globals}[x].\mathsf{value} = val$$

global.set x

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value val from the stack.
4. Perform $z[\mathsf{globals}[x].\mathsf{value} = val]$.

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

4.4.6 Table Instructions

table.get x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. If i is greater than or equal to $|z.\mathsf{tables}[x].\mathsf{refs}|$, then:
   a. Trap.
5. Push the value $z.\mathsf{tables}[x].\mathsf{refs}[i]$ to the stack.

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

table.set x

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If i is greater than or equal to $|z.\mathsf{tables}[x].\mathsf{refs}|$, then:
   a. Trap.
7. Perform $z[\mathsf{tables}[x].\mathsf{refs}[i] = ref]$.

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

table.size x

1. Let z be the current state.
2. Let n be $|z.\mathsf{tables}[x].\mathsf{refs}|$.
3. Push the value $(\mathsf{i32.const}\ n)$ to the stack.

$$z; (\mathsf{table.size}\ x) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \qquad \text{if } |z.\mathsf{tables}[x].\mathsf{refs}| = n$$

table.grow x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value ref from the stack.
6. Either:
   a. Let ti be $\mathrm{growtable}(z.\mathsf{tables}[x], n, \mathit{ref})$.
   b. Push the value $(\mathsf{i32.const}\ |z.\mathsf{tables}[x].\mathsf{refs}|)$ to the stack.
   c. Perform $z[\mathsf{tables}[x] = \mathit{ti}]$.
7. Or:
   a. Push the value $(\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))$ to the stack.

$$\begin{array}{lcl} z; \mathit{ref}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z[\mathsf{tables}[x] = \mathit{ti}]; (\mathsf{i32.const}\ |z.\mathsf{tables}[x].\mathsf{refs}|) \\ & & \quad \text{if } \mathit{ti} = \mathrm{growtable}(z.\mathsf{tables}[x], n, \mathit{ref}) \\ z; \mathit{ref}\ (\mathsf{i32.const}\ n)\ (\mathsf{table.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1)) \end{array}$$

table.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{tables}[x].\mathsf{refs}|$, then:
   a. Trap.
9. If n is 0, then:
   a. Do nothing.

10. Else:
    a. Push the value $(\mathsf{i32.const}\ i)$ to the stack.
    b. Push the value val to the stack.
    c. Execute the instruction $(\mathsf{table.set}\ x)$.
    d. Push the value $(\mathsf{i32.const}\ i+1)$ to the stack.
    e. Push the value val to the stack.
    f. Push the value $(\mathsf{i32.const}\ n-1)$ to the stack.
    g. Execute the instruction $(\mathsf{table.fill}\ x)$.

$$
\begin{array}{llcll}
[\textsc{E-table.fill-oob}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i + n > |z.\mathsf{tables}[x].\mathsf{refs}| \\
[\textsc{E-table.fill-zero}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
[\textsc{E-table.fill-succ}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{table.fill}\ x) & \hookrightarrow & & \\
& \quad (\mathsf{i32.const}\ i)\ val\ (\mathsf{table.set}\ x) & & & \text{otherwise} \\
& \quad (\mathsf{i32.const}\ i+1)\ val\ (\mathsf{i32.const}\ n-1)\ (\mathsf{table.fill}\ x) & & &
\end{array}
$$

table.copy x y

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i+n$ is greater than $|z.\mathsf{tables}[y].\mathsf{refs}|$, then:
    a. Trap.
9. If $j+n$ is greater than $|z.\mathsf{tables}[x].\mathsf{refs}|$, then:
    a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else:
    a. If j is less than or equal to i, then:
        1) Push the value $(\mathsf{i32.const}\ j)$ to the stack.
        2) Push the value $(\mathsf{i32.const}\ i)$ to the stack.
        3) Execute the instruction $(\mathsf{table.get}\ y)$.
        4) Execute the instruction $(\mathsf{table.set}\ x)$.
        5) Push the value $(\mathsf{i32.const}\ j+1)$ to the stack.
        6) Push the value $(\mathsf{i32.const}\ i+1)$ to the stack.
    b. Else:
        1) Push the value $(\mathsf{i32.const}\ j+n-1)$ to the stack.

2) Push the value $(\mathsf{i32.const}\ i + n - 1)$ to the stack.
3) Execute the instruction $(\mathsf{table.get}\ y)$.
4) Execute the instruction $(\mathsf{table.set}\ x)$.
5) Push the value $(\mathsf{i32.const}\ j)$ to the stack.
6) Push the value $(\mathsf{i32.const}\ i)$ to the stack.

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

d. Execute the instruction $(\mathsf{table.copy}\ x\ y)$.

$$
\begin{array}{lllll}
[\textsc{E-table.copy-oob}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \mathsf{trap} & \\
& \quad \text{if } i + n > |z.\mathsf{tables}[y].\mathsf{refs}| \vee j + n > |z.\mathsf{tables}[x].\mathsf{refs}| & & & \\
[\textsc{E-table.copy-zero}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
[\textsc{E-table.copy-le}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & & \\
& (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & & \text{otherwise, if } j \leq i \\
& (\mathsf{i32.const}\ j + 1)\ (\mathsf{i32.const}\ i + 1)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.copy}\ x\ y) & & & \\
[\textsc{E-table.copy-gt}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.copy}\ x\ y) & \hookrightarrow & & \\
& (\mathsf{i32.const}\ j + n - 1)\ (\mathsf{i32.const}\ i + n - 1)\ (\mathsf{table.get}\ y)\ (\mathsf{table.set}\ x) & & & \text{otherwise} \\
& (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{table.copy}\ x\ y) & & &
\end{array}
$$

$\mathsf{table.init}\ x\ y$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:
   a. Trap.
9. If $j + n$ is greater than $|z.\mathsf{tables}[x].\mathsf{refs}|$, then:
   a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else if i is less than $|z.\mathsf{elems}[y].\mathsf{refs}|$, then:
    a. Push the value $(\mathsf{i32.const}\ j)$ to the stack.
    b. Push the value $z.\mathsf{elems}[y].\mathsf{refs}[i]$ to the stack.
    c. Execute the instruction $(\mathsf{table.set}\ x)$.
    d. Push the value $(\mathsf{i32.const}\ j + 1)$ to the stack.
    e. Push the value $(\mathsf{i32.const}\ i + 1)$ to the stack.
    f. Push the value $(\mathsf{i32.const}\ n - 1)$ to the stack.
    g. Execute the instruction $(\mathsf{table.init}\ x\ y)$.

$$
\begin{array}{llll}
[\textsc{E-table.init-oob}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \mathsf{trap} \\
& \quad \text{if } i + n > |z.\mathsf{elems}[y].\mathsf{refs}| \vee j + n > |z.\mathsf{tables}[x].\mathsf{refs}| & & \\
[\textsc{E-table.init-zero}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \epsilon \qquad \text{otherwise, if } n = 0 \\
[\textsc{E-table.init-succ}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{table.init}\ x\ y) & \hookrightarrow & \\
& (\mathsf{i32.const}\ j)\ z.\mathsf{elems}[y].\mathsf{refs}[i]\ (\mathsf{table.set}\ x) & & \text{otherwise} \\
& (\mathsf{i32.const}\ j+1)\ (\mathsf{i32.const}\ i+1)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{table.init}\ x\ y) & &
\end{array}
$$

$\mathsf{elem.drop}\ x$

1. Let z be the current state.
2. Perform $z[\mathsf{elems}[x].\mathsf{refs} = \epsilon]$.

$$z; (\mathsf{elem.drop}\ x) \quad \hookrightarrow \quad z[\mathsf{elems}[x].\mathsf{refs} = \epsilon]; \epsilon$$

4.4.7 Memory Instructions

$numty_{u0}.\mathsf{load}w^?_{sx_{u1}}\ x\ mo$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. If $w^?_{sx_{u1}}$ is not defined, then:
   a. Let nt be $numty_{u0}$.
   b. If $i + mo.\mathsf{offset} + |nt|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
      1) Trap.
   c. Let c be $\mathrm{inverse}_{of_{nbytes}}(nt, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |nt|/8])$.
   d. Push the value $(nt.\mathsf{const}\ c)$ to the stack.
5. If the type of $numty_{u0}$ is inn, then:
   a. If $w^?_{sx_{u1}}$ is defined, then:
      1) Let y_0 be $w^?_{sx_{u1}}$.
      2) Let $(n,\ sx)$ be y_0.
      3) If $i + mo.\mathsf{offset} + n/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
   b. Let $\mathsf{i}n$ be $numty_{u0}$.
   c. If $w^?_{sx_{u1}}$ is defined, then:
      1) Let y_0 be $w^?_{sx_{u1}}$.
      2) Let $(n,\ sx)$ be y_0.
      3) Let c be $\mathrm{inverse}_{of_{ibytes}}(n, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : n/8])$.
      4) Push the value $(\mathsf{i}n.\mathsf{const}\ \mathrm{ext}^{sx}_{n,|\mathsf{i}n|}(c))$ to the stack.

$$
\begin{array}{llcl}
[\textsc{E-load-num-oob}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ mo) & \hookrightarrow & \mathsf{trap} \\
& & & \quad \text{if } i + mo.\mathsf{offset} + |nt|/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
[\textsc{E-load-num-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{load}\ x\ mo) & \hookrightarrow & (nt.\mathsf{const}\ c) \\
& & & \quad \text{if } \mathrm{bytes}_{nt}(c) = z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |nt|/8] \\
[\textsc{E-load-pack-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i}n.\mathsf{load}n\_sx\ x\ mo) & \hookrightarrow & \mathsf{trap} \\
& & & \quad \text{if } i + mo.\mathsf{offset} + n/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
[\textsc{E-load-pack-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i}n.\mathsf{load}n\_sx\ x\ mo) & \hookrightarrow & (\mathsf{i}n.\mathsf{const}\ \mathrm{ext}_{n,|\mathsf{i}n|}^{sx}(c)) \\
& & & \quad \text{if } \mathrm{bytes}_{\mathsf{i}n}(c) = z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : n/8]
\end{array}
$$

$nt.\mathsf{store}w_{u1}^{?}\ x\ mo$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $numty_{u0}$ is on the top of the stack.
3. Pop the value $(numty_{u0}.\mathsf{const}\ c)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If $numty_{u0}$ is nt, then:
   a. If $i + mo.\mathsf{offset} + |nt|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$ and $w_{u1}^{?}$ is not defined, then:
      1) Trap.
   b. If $w_{u1}^{?}$ is not defined, then:
      1) Let b^* be $\mathrm{bytes}_{nt}(c)$.
      2) Perform $z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |nt|/8] = b^*]$.
7. If the type of $numty_{u0}$ is inn, then:
   a. If $w_{u1}^{?}$ is defined, then:
      1) Let n be $w_{u1}^{?}$.
      2) If $i + mo.\mathsf{offset} + n/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
   b. Let $\mathsf{i}n$ be $numty_{u0}$.
   c. If $w_{u1}^{?}$ is defined, then:
      1) Let n be $w_{u1}^{?}$.
      2) Let b^* be $\mathrm{bytes}_{\mathsf{i}n}(\mathrm{wrap}_{|\mathsf{i}n|,n}(c))$.
      3) Perform $z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : n/8] = b^*]$.

$$
\begin{array}{llcl}
[\textsc{E-store-num-oob}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ mo) & \hookrightarrow & z; \mathsf{trap} \\
& & & \quad \text{if } i + mo.\mathsf{offset} + |nt|/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
[\textsc{E-store-num-val}] & z; (\mathsf{i32.const}\ i)\ (nt.\mathsf{const}\ c)\ (nt.\mathsf{store}\ x\ mo) & \hookrightarrow & z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |nt|/8] = b^*]; \epsilon \\
& & & \quad \text{if } b^* = \mathrm{bytes}_{nt}(c) \\
[\textsc{E-store-pack-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i}n.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ mo) & \hookrightarrow & z; \mathsf{trap} \\
& & & \quad \text{if } i + mo.\mathsf{offset} + n/8 > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
[\textsc{E-store-pack-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{i}n.\mathsf{const}\ c)\ (nt.\mathsf{store}n\ x\ mo) & \hookrightarrow & z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : n/8] = b^*]; \epsilon \\
& & & \quad \text{if } b^* = \mathrm{bytes}_{\mathsf{i}n}(\mathrm{wrap}_{|\mathsf{i}n|,n}(c))
\end{array}
$$

$\mathsf{v128.load}\mathit{vload}^?_{u0}\ x\ mo$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
4. If $i + mo.\mathsf{offset} + |\mathsf{v128}|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$ and $\mathit{vload}^?_{u0}$ is not defined, then:
   a. Trap.
5. If $\mathit{vload}^?_{u0}$ is not defined, then:
   a. Let c be $\mathrm{inverse}_{of_{vbytes}}(\mathsf{v128}, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |\mathsf{v128}|/8])$.
   b. Push the value $(\mathsf{v128.const}\ c)$ to the stack.
6. Else:
   a. Let y_0 be $\mathit{vload}^?_{u0}$.
   b. If y_0 is of the case shape, then:
      1) Let $(M{\times}N\_sx)$ be y_0.
      2) If $i + mo.\mathsf{offset} + M \cdot N/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
      3) If the type of $\mathrm{inverse}_{of_{lsize}}(M \cdot 2)$ is imm, then:
         a) Let $\mathsf{i}n$ be $\mathrm{inverse}_{of_{lsize}}(M \cdot 2)$.
         b) Let j^N be $\mathrm{inverse}_{of_{ibytes}}(M, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} + k \cdot M/8 : M/8])^{k<N}$.
         c) Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n{\times}N}(\mathrm{ext}^{sx}_{M,|\mathsf{i}n|}(j)^N)$.
         d) Push the value $(\mathsf{v128.const}\ c)$ to the stack.
   c. If y_0 is of the case splat, then:
      1) Let (N_splat) be y_0.
      2) If $i + mo.\mathsf{offset} + N/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
      3) Let M be $128/N$.
      4) If the type of $\mathrm{inverse}_{of_{lsize}}(N)$ is imm, then:
         a) Let $\mathsf{i}n$ be $\mathrm{inverse}_{of_{lsize}}(N)$.
         b) Let j be $\mathrm{inverse}_{of_{ibytes}}(N, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : N/8])$.
         c) Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n{\times}M}(j^M)$.
         d) Push the value $(\mathsf{v128.const}\ c)$ to the stack.
   d. If y_0 is of the case zero, then:
      1) Let (N_zero) be y_0.
      2) If $i + mo.\mathsf{offset} + N/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
         a) Trap.
      3) Let j be $\mathrm{inverse}_{of_{ibytes}}(N, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : N/8])$.
      4) Let c be $\mathrm{ext}^{\mathsf{u}}_{N,128}(j)$.
      5) Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-vload-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}\ x\ mo) & \hookrightarrow & \mathsf{trap} & \text{if } i + mo.\mathsf{offset} + |\mathsf{v128}|/8 > |z.\mathsf{mems}[x] \\
[\textsc{E-vload-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.load}\ x\ mo) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } \mathrm{bytes}_{\mathsf{v128}}(c) = z.\mathsf{mems}[x].\mathsf{bytes}[i + m \\
[\textsc{E-vload-shape-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadshape}M\mathsf{x}N\,sx\ x\ mo) & \hookrightarrow & \mathsf{trap} & \text{if } i + mo.\mathsf{offset} + M \cdot N/8 > |z.\mathsf{mems}[\\
[\textsc{E-vload-shape-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadshape}M\mathsf{x}N\,sx\ x\ mo) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } (\mathrm{bytes}_{\mathsf{i}M}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + m \\
 & & & & \wedge\ |\mathsf{i}n| = M \cdot 2 \\
 & & & & \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}N}(\mathrm{ext}^{sx}_{M,|\mathsf{i}n|}(j)^N) \\
[\textsc{E-vload-splat-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadsplat}N\ x\ mo) & \hookrightarrow & \mathsf{trap} & \text{if } i + mo.\mathsf{offset} + N/8 > |z.\mathsf{mems}[x].\mathsf{by} \\
[\textsc{E-vload-splat-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadsplat}N\ x\ mo) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } \mathrm{bytes}_{\mathsf{i}N}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + mo \\
 & & & & \wedge\ N = |\mathsf{i}n| \\
 & & & & \wedge\ M = 128/N \\
 & & & & \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}M}(j^M) \\
[\textsc{E-vload-zero-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadzero}N\ x\ mo) & \hookrightarrow & \mathsf{trap} & \text{if } i + mo.\mathsf{offset} + N/8 > |z.\mathsf{mems}[x].\mathsf{by} \\
[\textsc{E-vload-zero-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.loadzero}N\ x\ mo) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } \mathrm{bytes}_{\mathsf{i}N}(j) = z.\mathsf{mems}[x].\mathsf{bytes}[i + mo \\
 & & & & \wedge\ c = \mathrm{ext}^{\mathsf{u}}_{N,128}(j)
\end{array}
$$

$\mathsf{v128.load}N\mathsf{\_lane}\ x\ mo\ j$

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value $(\mathsf{v128.const}\ c_1)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. If $i + mo.\mathsf{offset} + N/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
   a. Trap.
7. Let M be $|\mathsf{v128}|/N$.
8. If the type of $\mathrm{inverse}_{of_{\,lsize}}(N)$ is imm, then:
   a. Let $\mathsf{i}n$ be $\mathrm{inverse}_{of_{\,lsize}}(N)$.
   b. Let k be $\mathrm{inverse}_{of_{\,ibytes}}(N, z.\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : N/8])$.
   c. Let c be $\mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}M}(\mathrm{lanes}_{\mathsf{i}n\mathsf{x}M}(c_1)[[j] = k])$.
   d. Push the value $(\mathsf{v128.const}\ c)$ to the stack.

$$
\begin{array}{llcll}
[\textsc{E-vload\_lane-oob}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.load}N\mathsf{\_lane}\ x\ mo\ j) & \hookrightarrow & \mathsf{trap} & \text{if } i + mo.\mathsf{offset} + N/8 > |z.\mathsf{m} \\
[\textsc{E-vload\_lane-val}] & z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c_1)\ (\mathsf{v128.load}N\mathsf{\_lane}\ x\ mo\ j) & \hookrightarrow & (\mathsf{v128.const}\ c) & \text{if } \mathrm{bytes}_{\mathsf{i}N}(k) = z.\mathsf{mems}[x].\mathsf{byt} \\
 & & & & \wedge\ N = |\mathsf{i}n| \\
 & & & & \wedge\ M = |\mathsf{v128}|/N \\
 & & & & \wedge\ c = \mathrm{lanes}^{-1}_{\mathsf{i}n\mathsf{x}M}(\mathrm{lanes}_{\mathsf{i}n\mathsf{x}M}(c_1
\end{array}
$$

v128.store x mo

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value (v128.const c) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const i) from the stack.
6. If $i + mo.\mathsf{offset} + |\mathsf{v128}|/8$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
   a. Trap.
7. Let b^* be $\mathrm{bytes}_{\mathsf{v128}}(c)$.
8. Perform $z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |\mathsf{v128}|/8] = b^*]$.

$$\begin{array}{lllll} [\textsc{E-vstore-oob}] z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}\ x\ mo) & \hookrightarrow & z; \mathsf{trap} & & \text{if } i + m \\ [\textsc{E-vstore-val}] z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}\ x\ mo) & \hookrightarrow & z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : |\mathsf{v128}|/8] = b^*]; \epsilon & & \text{if } b^* = \end{array}$$

v128.store$N$_lane x mo j

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value (v128.const c) from the stack.
4. Assert: Due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value (i32.const i) from the stack.
6. If $i + mo.\mathsf{offset} + N$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
   a. Trap.
7. Let M be $128/N$.
8. If the type of $\mathrm{inverse}_{of_{lsize}}(N)$ is imm, then:
   a. Let in be $\mathrm{inverse}_{of_{lsize}}(N)$.
   b. If j is less than $|\mathrm{lanes}_{\mathsf{i}n \times M}(c)|$, then:
      1) Let b^* be $\mathrm{bytes}_{\mathsf{i}N}(\mathrm{lanes}_{\mathsf{i}n \times M}(c)[j])$.
      2) Perform $z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : N/8] = b^*]$.

$$\begin{array}{lll} [\textsc{E-vstore\_lane-oob}] z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ x\ mo\ j) & \hookrightarrow & z; \mathsf{trap} \\ [\textsc{E-vstore\_lane-val}] z; (\mathsf{i32.const}\ i)\ (\mathsf{v128.const}\ c)\ (\mathsf{v128.store}N\mathsf{\_lane}\ x\ mo\ j) & \hookrightarrow & z[\mathsf{mems}[x].\mathsf{bytes}[i + mo.\mathsf{offset} : N/8] = b^*]; \epsilon \end{array}$$

memory.size x

1. Let z be the current state.
2. Let $n \cdot 64\,\mathrm{Ki}$ be $|z.\mathsf{mems}[x].\mathsf{bytes}|$.
3. Push the value $(\mathsf{i32.const}\ n)$ to the stack.

$$z; (\mathsf{memory.size}\ x) \quad \hookrightarrow \quad (\mathsf{i32.const}\ n) \qquad \text{if } n \cdot 64\,\mathrm{Ki} = |z.\mathsf{mems}[x].\mathsf{bytes}|$$

memory.grow x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Either:
   a. Let mi be $\mathrm{growmemory}(z.\mathsf{mems}[x], n)$.
   b. Push the value $(\mathsf{i32.const}\ |z.\mathsf{mems}[x].\mathsf{bytes}|/64\,\mathrm{Ki})$ to the stack.
   c. Perform $z[\mathsf{mems}[x] = mi]$.
5. Or:
   a. Push the value $(\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))$ to the stack.

$$\begin{array}{llcl}
[\textsc{E-memory.grow-succeed}] & z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z[\mathsf{mems}[x] = mi]; (\mathsf{i32.const}\ |z.\mathsf{mems}[x].\mathsf{bytes}|/64\,\mathrm{Ki}) \\
& & & \quad \text{if } mi = \mathrm{growmemory}(z.\mathsf{mems}[x], n) \\
[\textsc{E-memory.grow-fail}] & z; (\mathsf{i32.const}\ n)\ (\mathsf{memory.grow}\ x) & \hookrightarrow & z; (\mathsf{i32.const}\ \mathrm{signed}_{32}^{-1}(-1))
\end{array}$$

memory.fill x

1. Let z be the current state.
2. Assert: Due to validation, a value of value type i32 is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value is on the top of the stack.
5. Pop the value val from the stack.
6. Assert: Due to validation, a value of value type i32 is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
   a. Trap.
9. If n is 0, then:
   a. Do nothing.
10. Else:
    a. Push the value $(\mathsf{i32.const}\ i)$ to the stack.
    b. Push the value val to the stack.

c. Execute the instruction $(\mathsf{i32.store8}\ x)$.

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

e. Push the value val to the stack.

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

g. Execute the instruction $(\mathsf{memory.fill}\ x)$.

$$
\begin{array}{llll}
[\textsc{E-memory.fill-oob}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & \mathsf{trap} & \text{if } i + n > |z.\mathsf{mems}[x].\mathsf{bytes}| \\
[\textsc{E-memory.fill-zero}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
[\textsc{E-memory.fill-succ}] & z; (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.fill}\ x) & \hookrightarrow & & \\
& \quad (\mathsf{i32.const}\ i)\ val\ (\mathsf{i32.store8}\ x) & & & \text{otherwise} \\
& \quad (\mathsf{i32.const}\ i + 1)\ val\ (\mathsf{i32.const}\ n - 1)\ (\mathsf{memory.fill}\ x) & & &
\end{array}
$$

memory.copy $x_1\ x_2$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i_2)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ i_1)$ from the stack.
8. If $i_1 + n$ is greater than $|z.\mathsf{mems}[x_1].\mathsf{bytes}|$, then:

   a. Trap.

9. If $i_2 + n$ is greater than $|z.\mathsf{mems}[x_2].\mathsf{bytes}|$, then:

   a. Trap.

10. If n is 0, then:

    a. Do nothing.

11. Else:

    a. If i_1 is less than or equal to i_2, then:

       1) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.

       2) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.

       3) Execute the instruction $(\mathsf{i32.load}(8,\ \mathsf{u})\ x_2)$.

       4) Execute the instruction $(\mathsf{i32.store8}\ x_1)$.

       5) Push the value $(\mathsf{i32.const}\ i_1 + 1)$ to the stack.

       6) Push the value $(\mathsf{i32.const}\ i_2 + 1)$ to the stack.

    b. Else:

       1) Push the value $(\mathsf{i32.const}\ i_1 + n - 1)$ to the stack.

       2) Push the value $(\mathsf{i32.const}\ i_2 + n - 1)$ to the stack.

       3) Execute the instruction $(\mathsf{i32.load}(8,\ \mathsf{u})\ x_2)$.

       4) Execute the instruction $(\mathsf{i32.store8}\ x_1)$.

5) Push the value $(\mathsf{i32.const}\ i_1)$ to the stack.

6) Push the value $(\mathsf{i32.const}\ i_2)$ to the stack.

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

d. Execute the instruction $(\mathsf{memory.copy}\ x_1\ x_2)$.

$$
\begin{array}{llll}
[\textsc{E-memory.copy-oob}] & z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \mathsf{trap} \\
 & \quad \text{if } i_1 + n > |z.\mathsf{mems}[x_1].\mathsf{bytes}| \vee i_2 + n > |z.\mathsf{mems}[x_2].\mathsf{bytes}| & & \\
[\textsc{E-memory.copy-zero}] & z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \epsilon \qquad \text{otherwise, if } n = 0 \\
[\textsc{E-memory.copy-le}] & z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \\
 & (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.load8\_u}\ x_2)\ (\mathsf{i32.store8}\ x_1) & & \text{otherwise, if } i_1 \leq i_2 \\
 & (\mathsf{i32.const}\ i_1+1)\ (\mathsf{i32.const}\ i_2+1)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{memory.copy}\ x_1\ x_2) & & \\
[\textsc{E-memory.copy-gt}] & z; (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.copy}\ x_1\ x_2) & \hookrightarrow & \\
 & (\mathsf{i32.const}\ i_1+n-1)\ (\mathsf{i32.const}\ i_2+n-1)\ (\mathsf{i32.load8\_u}\ x_2)\ (\mathsf{i32.store8}\ x_1) & & \text{otherwise} \\
 & (\mathsf{i32.const}\ i_1)\ (\mathsf{i32.const}\ i_2)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{memory.copy}\ x_1\ x_2) & &
\end{array}
$$

$\mathsf{memory.init}\ x\ y$

1. Let z be the current state.
2. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
3. Pop the value $(\mathsf{i32.const}\ n)$ from the stack.
4. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
5. Pop the value $(\mathsf{i32.const}\ i)$ from the stack.
6. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
7. Pop the value $(\mathsf{i32.const}\ j)$ from the stack.
8. If $i + n$ is greater than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:
   a. Trap.
9. If $j + n$ is greater than $|z.\mathsf{mems}[x].\mathsf{bytes}|$, then:
   a. Trap.
10. If n is 0, then:
    a. Do nothing.
11. Else if i is less than $|z.\mathsf{datas}[y].\mathsf{bytes}|$, then:
    a. Push the value $(\mathsf{i32.const}\ j)$ to the stack.
    b. Push the value $(\mathsf{i32.const}\ z.\mathsf{datas}[y].\mathsf{bytes}[i])$ to the stack.
    c. Execute the instruction $(\mathsf{i32.store8}\ x)$.
    d. Push the value $(\mathsf{i32.const}\ j+1)$ to the stack.
    e. Push the value $(\mathsf{i32.const}\ i+1)$ to the stack.
    f. Push the value $(\mathsf{i32.const}\ n-1)$ to the stack.
    g. Execute the instruction $(\mathsf{memory.init}\ x\ y)$.

$$
\begin{array}{lllll}
[\textsc{E-memory.init-oob}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & \mathsf{trap} & \\
& \quad \text{if } i + n > |z.\mathsf{datas}[y].\mathsf{bytes}| \lor j + n > |z.\mathsf{mems}[x].\mathsf{bytes}| & & & \\
[\textsc{E-memory.init-zero}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & \epsilon & \text{otherwise, if } n = 0 \\
[\textsc{E-memory.init-succ}] & z; (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ i)\ (\mathsf{i32.const}\ n)\ (\mathsf{memory.init}\ x\ y) & \hookrightarrow & & \\
& (\mathsf{i32.const}\ j)\ (\mathsf{i32.const}\ z.\mathsf{datas}[y].\mathsf{bytes}[i])\ (\mathsf{i32.store8}\ x) & & & \text{otherwise} \\
& (\mathsf{i32.const}\ j+1)\ (\mathsf{i32.const}\ i+1)\ (\mathsf{i32.const}\ n-1)\ (\mathsf{memory.init}\ x\ y) & & &
\end{array}
$$

data.drop x

1. Let z be the current state.
2. Perform $z[\mathsf{datas}[x].\mathsf{bytes} = \epsilon]$.

$$z; (\mathsf{data.drop}\ x) \quad \hookrightarrow \quad z[\mathsf{datas}[x].\mathsf{bytes} = \epsilon]; \epsilon$$

4.4.8 Control Instructions

nop

1. Do nothing.

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

unreachable

1. Trap.

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

$\mathrm{blocktype}(\mathit{block}_{u1})$

1. If block_{u1} is ϵ, then:
   a. Return $\epsilon \to \epsilon$.
2. If block_{u1} is of the case , then:
   a. Let y_0 be block_{u1}.
   b. If y_0 is defined, then:
      1) Let t be y_0.
      2) Return $\epsilon \to t$.
3. Assert: Due to validation, block_{u1} is of the case .
4. Let x be block_{u1}.

5. Assert: Due to validation, $\mathrm{expand}(\mathrm{type}(x))$ is of the case func.
6. Let $(\mathsf{func}\ ft)$ be $\mathrm{expand}(\mathrm{type}(x))$.
7. Return ft.

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

$\mathsf{block}\ bt\ instr^*$

1. Let z be the current state.
2. Let $t_1^m \rightarrow t_2^n$ be $\mathrm{blocktype}_z(bt)$.
3. Assert: Due to validation, there are at least m values on the top of the stack.
4. Pop the values val^m from the stack.
5. Let L be the label whose arity is n and whose continuation is ϵ.
6. Enter L with label $instr^*$ $\mathsf{label\_}$:
   a. Push the values val^m to the stack.

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

$\mathsf{loop}\ bt\ instr^*$

1. Let z be the current state.
2. Let $t_1^m \rightarrow t_2^n$ be $\mathrm{blocktype}_z(bt)$.
3. Assert: Due to validation, there are at least m values on the top of the stack.
4. Pop the values val^m from the stack.
5. Let L be the label whose arity is m and whose continuation is $(\mathsf{loop}\ bt\ instr^*)$.
6. Enter L with label $instr^*$ $\mathsf{label\_}$:
   a. Push the values val^m to the stack.

$$
[\textsc{E-loop}] z; val^m\ (\mathsf{loop}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_m\{\mathsf{loop}\ bt\ instr^*\}\ val^m\ instr^*) \qquad \text{if } \mathrm{blocktype}_z(bt) = t_1^m \rightarrow t_2^n
$$

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

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

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

$\mathsf{br}\ l$

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 $admin_{u0}^*$ from the stack.
5. Exit current context.
6. If l is 0 and $|admin_{u0}^*|$ is greater than or equal to n, then:
   a. Let $val'^*\ val^n$ be $admin_{u0}^*$.
   b. Push the values val^n to the stack.
   c. Execute the sequence $instr'^*$.
7. If l is greater than 0, then:
   a. Let val^* be $admin_{u0}^*$.
   b. Push the values val^* to the stack.
   c. Execute the instruction $(\mathsf{br}\ l-1)$.

$$\begin{array}{llcll}
[\textsc{e-br-zero}] & (\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^n\ instr'^* & \text{if } l = 0 \\
[\textsc{e-br-succ}] & (\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l-1) & \text{if } l > 0
\end{array}$$

$\mathsf{br\_if}\ l$

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

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

br_table l^* l'

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

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

br_on_null l

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

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

br_on_non_null l

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. If val is of the case ref.null, then:
   a. Do nothing.
4. Else:
   a. Push the value val to the stack.
   b. Execute the instruction (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. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt be $\mathrm{ref}_{type_{of}}(ref)$.
5. If rt does not match $\mathrm{inst}_{f.\mathsf{module}}(rt_2)$, then:
   a. Push the value ref to the stack.
6. Else:
   a. Push the value ref to the stack.
   b. Execute the instruction $(\mathsf{br}\ l)$.

$$\begin{array}{lllll}
[\textsc{E-br_on_cast-succeed}] & s; f; ref\ (\mathsf{br_on_cast}\ l\ rt_1\ rt_2) & \hookrightarrow & ref\ (\mathsf{br}\ l) & \text{if } s \vdash ref : rt \\
 & & & & \wedge \{\} \vdash rt \leq \mathrm{inst}_{f.\mathsf{module}}(rt_2) \\
[\textsc{E-br_on_cast-fail}] & s; f; ref\ (\mathsf{br_on_cast}\ l\ rt_1\ rt_2) & \hookrightarrow & ref & \text{otherwise}
\end{array}$$

br_on_cast_fail l rt_1 rt_2

1. Let f be the current frame.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. Let rt be $\mathrm{ref}_{type_{of}}(ref)$.
5. If rt matches $\mathrm{inst}_{f.\mathsf{module}}(rt_2)$, then:
   a. Push the value ref to the stack.
6. Else:
   a. Push the value ref to the stack.
   b. Execute the instruction $(\mathsf{br}\ l)$.

$$\begin{array}{lllll}
[\textsc{E-br_on_cast_fail-succeed}] & s; f; ref\ (\mathsf{br_on_cast_fail}\ l\ rt_1\ rt_2) & \hookrightarrow & ref & \text{if } s \vdash ref : rt \\
 & & & & \wedge \{\} \vdash rt \leq \mathrm{inst}_{f.\mathsf{module}}(rt_2) \\
[\textsc{E-br_on_cast_fail-fail}] & s; f; ref\ (\mathsf{br_on_cast_fail}\ l\ rt_1\ rt_2) & \hookrightarrow & ref\ (\mathsf{br}\ l) & \text{otherwise}
\end{array}$$

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 the values val^n from the stack.
   d. Pop all values val'^* from the stack.
   e. Exit current context.

f. Push the values 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 the values val^* to the stack.

d. Execute the instruction return.

$$\begin{array}{llcl}[\textsc{E-return-frame}] & (\mathsf{frame}_n\{f\}\ val'^*\ val^n\ \mathsf{return}\ instr^*) & \hookrightarrow & val^n \\ [\textsc{E-return-label}] & (\mathsf{label}_n\{instr'^*\}\ val^*\ \mathsf{return}\ instr^*) & \hookrightarrow & val^*\ \mathsf{return}\end{array}$$

call x

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module.funcs}|$.
3. Let a be $z.\mathsf{module.funcs}[x]$.
4. Assert: Due to validation, a is less than $|z.\mathsf{funcs}|$.
5. Push the value $(\mathsf{ref.func}\ a)$ to the stack.
6. Execute the instruction $(\mathsf{call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type})$.

$$z; (\mathsf{call}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ a)\ (\mathsf{call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type}) \qquad \text{if } z.\mathsf{module.funcs}[x] = a$$

call_ref x

1. Let z be the current state.
2. Assert: Due to validation, a value is on the top of the stack.
3. Pop the value ref from the stack.
4. If ref is of the case ref.null, then:

   a. Trap.

5. Assert: Due to validation, ref is of the case ref.func_addr.
6. Let $(\mathsf{ref.func}\ a)$ be ref.
7. If a is less than $|z.\mathsf{funcs}|$, then:

   a. Let fi be $z.\mathsf{funcs}[a]$.

   b. Assert: Due to validation, $fi.\mathsf{code}$ is of the case func.

   c. Let $(\mathsf{func}\ y_0\ y_1\ instr^*)$ be $fi.\mathsf{code}$.

   d. Let $(\mathsf{local}\ t)^*$ be y_1.

   e. Assert: Due to validation, $\mathrm{expand}(fi.\mathsf{type})$ is of the case func.

   f. Let $(\mathsf{func}\ y_0)$ be $\mathrm{expand}(fi.\mathsf{type})$.

   g. Let $t_1^n \rightarrow t_2^m$ be y_0.

   h. Assert: Due to validation, there are at least n values on the top of the stack.

  i. Pop the values val^n from the stack.
  j. Let f be $\{\mathsf{locals}\ val^n\ \mathrm{default}_t^*,\ \mathsf{module}\ fi.\mathsf{module}\}$.
  k. Let F be the activation of f with arity m.
  l. 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_:

$$
\begin{array}{llcl}
[\textsc{E-call\_ref-null}] & z; (\mathsf{ref.null}\ ht)\ (\mathsf{call\_ref}\ y) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-call\_ref-func}] & z; val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{call\_ref}\ y) & \hookrightarrow & (\mathsf{frame}_m\{f\}\ (\mathsf{label}_m\{\epsilon\}\ instr^*)) \\
& & & \quad \mathrm{if}\ z.\mathsf{funcs}[a] = fi \\
& & & \quad \wedge\ fi.\mathsf{type} \approx \mathsf{func}\ (t_1^n \rightarrow t_2^m) \\
& & & \quad \wedge\ fi.\mathsf{code} = \mathsf{func}\ x\ (\mathsf{local}\ t)^*\ (instr^*) \\
& & & \quad \wedge\ f = \{\mathsf{locals}\ val^n\ (\mathrm{default}_t)^*,\ \mathsf{module}\ fi.\mathsf{module}\}
\end{array}
$$

call_indirect x y

1. Execute the instruction $(\mathsf{table.get}\ x)$.
2. Execute the instruction $(\mathsf{ref.cast}\ (\mathsf{ref}\ (\mathsf{null}\ ())\ y))$.
3. Execute the instruction $(\mathsf{call\_ref}\ y)$.

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

return_call x

1. Let z be the current state.
2. Assert: Due to validation, x is less than $|z.\mathsf{module.funcs}|$.
3. Let a be $z.\mathsf{module.funcs}[x]$.
4. Assert: Due to validation, a is less than $|z.\mathsf{funcs}|$.
5. Push the value $(\mathsf{ref.func}\ a)$ to the stack.
6. Execute the instruction $(\mathsf{return\_call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type})$.

$$
[\textsc{E-return\_call}] z; (\mathsf{return\_call}\ x) \quad \hookrightarrow \quad (\mathsf{ref.func}\ a)\ (\mathsf{return\_call\_ref}\ z.\mathsf{funcs}[a].\mathsf{type}) \qquad \mathrm{if}\ z.\mathsf{module.funcs}[x] = a
$$

return_call_ref y

1. Let z be the current state.
2. If the current context is label_, then:
   a. Pop all values val^* from the stack.
   b. Exit current context.
   c. Push the values val^* to the stack.
   d. Execute the instruction $(\mathsf{return\_call\_ref}\ y)$.
3. Else if the current context is frame_, then:
   a. Pop the value $admin_{u0}$ from the stack.
   b. Pop all values $admin_{u1}^*$ from the stack.
   c. Exit current context.
   d. If $admin_{u0}$ is of the case ref.func_addr, then:
      1) Let $(\mathsf{ref.func}\ a)$ be $admin_{u0}$.
      2) If a is less than $|z.\mathsf{funcs}|$, then:
         a) Assert: Due to validation, $\mathrm{expand}(z.\mathsf{funcs}[a].\mathsf{type})$ is of the case func.
         b) Let $(\mathsf{func}\ y_0)$ be $\mathrm{expand}(z.\mathsf{funcs}[a].\mathsf{type})$.
         c) Let $t_1^n \rightarrow t_2^m$ be y_0.
         d) If $|admin_{u1}^*|$ is greater than or equal to n, then:
            1. Let $val'^*\ val^n$ be $admin_{u1}^*$.
            2. Push the values val^n to the stack.
            3. Push the value $(\mathsf{ref.func}\ a)$ to the stack.
            4. Execute the instruction $(\mathsf{call\_ref}\ y)$.
   e. If $admin_{u0}$ is of the case ref.null, then:
      1) Trap.

$$
\begin{array}{lrcl}
[\textsc{E-return\_call\_ref-label}] & z; (\mathsf{label}_k\{instr'^*\}\ val^*\ (\mathsf{return\_call\_ref}\ y)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{return\_call\_ref}\ y) \\
[\textsc{E-return\_call\_ref-frame-null}] & z; (\mathsf{frame}_k\{f\}\ val^*\ (\mathsf{ref.null}\ ht)\ (\mathsf{return\_call\_ref}\ y)\ instr^*) & \hookrightarrow & \mathsf{trap} \\
[\textsc{E-return\_call\_ref-frame-addr}] & z; (\mathsf{frame}_k\{f\}\ val'^*\ val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{return\_call\_ref}\ y)\ instr^*) & \hookrightarrow & val^n\ (\mathsf{ref.func}\ a)\ (\mathsf{call\_ref}\ y) \\
 & & & \quad \text{if } z.\mathsf{funcs}[a].\mathsf{type} \approx \mathsf{func}\ (t_1^n
\end{array}
$$

return_call_indirect x y

1. Execute the instruction $(\mathsf{table.get}\ x)$.
2. Execute the instruction $(\mathsf{ref.cast}\ (\mathsf{ref}\ (\mathsf{null}\ ())\ y))$.
3. Execute the instruction $(\mathsf{return\_call\_ref}\ y)$.

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

4.4.9 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 the values val^* to the stack.

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

4.4.10 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 the values 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 the values val^n to the stack.

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

4.4.11 Expressions

$$z; instr^* \quad \hookrightarrow^* \quad z'; val^* \qquad \text{if } z; instr^* \hookrightarrow^* z'; val^*$$

4.5 Modules

4.5.1 Allocation

$\mathrm{alloctypes}(type^*_{u0})$

1. If $type^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $type'^*\ type$ be $type^*_{u0}$.
3. Assert: Due to validation, $type$ is of the case type.
4. Let $(\mathsf{type}\ rectype)$ be $type$.
5. Let $deftype'^*$ be $\mathrm{alloctypes}(type'^*)$.

6. Let x be $|deftype'^*|$.
7. Let $deftype^*$ be $\mathrm{roll}_x(rectype)[:= deftype'^*]$.
8. Return $deftype'^*\ deftype^*$.

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

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

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

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

$\mathrm{allocfuncs}(mm, func^*_{u0})$

1. If $func^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $func\ func'^*$ be $func^*_{u0}$.
3. Let fa be $\mathrm{allocfunc}(mm, func)$.
4. Let fa'^* be $\mathrm{allocfuncs}(mm, func'^*)$.
5. Return $fa\ fa'^*$.

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

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

1. Let ti be $\{\mathsf{type}\ ((i,\ j),\ rt),\ \mathsf{refs}\ \mathit{ref}^{i}\}$.
2. Let a be $|s.\mathsf{tables}|$.
3. Append ti to the $s.\mathsf{tables}$.
4. Return a.

$$\mathrm{alloctable}(s, [i..j]\ rt, \mathit{ref}) \quad = \quad (s[\mathsf{tables} = ..ti],\ |s.\mathsf{tables}|) \qquad \text{if } ti = \{\mathsf{type}\ ([i..j]\ rt),\ \mathsf{refs}\ \mathit{ref}^{i}\}$$

$\mathrm{alloctables}(\mathit{table}^{*}_{u0}, \mathit{ref}^{*}_{u1})$

1. If table^{*}_{u0} is ϵ and ref^{*}_{u1} is ϵ, then:
   a. Return ϵ.
2. Assert: Due to validation, $|\mathit{ref}^{*}_{u1}|$ is greater than or equal to 1.
3. Let $\mathit{ref}\ \mathit{ref}'^{*}$ be ref^{*}_{u1}.
4. Assert: Due to validation, $|\mathit{table}^{*}_{u0}|$ is greater than or equal to 1.
5. Let $\mathit{tabletype}\ \mathit{tabletype}'^{*}$ be table^{*}_{u0}.
6. Let ta be $\mathrm{alloctable}(\mathit{tabletype}, \mathit{ref})$.
7. Let ta'^{*} be $\mathrm{alloctables}(\mathit{tabletype}'^{*}, \mathit{ref}'^{*})$.
8. Return $ta\ ta'^{*}$.

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

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

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

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

$\mathrm{allocmems}(\mathit{memty}^*_{u0})$

1. If memty^*_{u0} is ϵ, then:
   a. Return ϵ.
2. Let $\mathit{memtype}\ \mathit{memtype}'^*$ be memty^*_{u0}.
3. Let ma be $\mathrm{allocmem}(\mathit{memtype})$.
4. Let ma'^* be $\mathrm{allocmems}(\mathit{memtype}'^*)$.
5. Return $\mathit{ma}\ \mathit{ma}'^*$.

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

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

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

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

$\mathrm{allocglobals}(\mathit{globa}^*_{u0}, \mathit{val}^*_{u1})$

1. If globa^*_{u0} is ϵ, then:
   a. Assert: Due to validation, val^*_{u1} is ϵ.
   b. Return ϵ.
2. Else:
   a. Let $\mathit{globaltype}\ \mathit{globaltype}'^*$ be globa^*_{u0}.
   b. Assert: Due to validation, $|\mathit{val}^*_{u1}|$ is greater than or equal to 1.
   c. Let $\mathit{val}\ \mathit{val}'^*$ be val^*_{u1}.
   d. Let ga be $\mathrm{allocglobal}(\mathit{globaltype}, \mathit{val})$.
   e. Let ga'^* be $\mathrm{allocglobals}(\mathit{globaltype}'^*, \mathit{val}'^*)$.
   f. Return $\mathit{ga}\ \mathit{ga}'^*$.

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

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

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

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

$\mathrm{allocelems}(refty^*_{u0}, ref^*_{u1})$

1. If $refty^*_{u0}$ is ϵ and ref^*_{u1} is ϵ, then:
   a. Return ϵ.
2. Assert: Due to validation, $|ref^*_{u1}|$ is greater than or equal to 1.
3. Let $ref^*\ ref'^{**}$ be ref^*_{u1}.
4. Assert: Due to validation, $|refty^*_{u0}|$ is greater than or equal to 1.
5. Let $rt\ rt'^*$ be $refty^*_{u0}$.
6. Let ea be $\mathrm{allocelem}(rt, ref^*)$.
7. Let ea'^* be $\mathrm{allocelems}(rt'^*, ref'^{**})$.
8. Return $ea\ ea'^*$.

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

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

1. Let di be $\{\mathsf{bytes}\ byte^*\}$.
2. Let a be $|s.\mathsf{datas}|$.
3. Append di to the $s.\mathsf{datas}$.
4. Return a.

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

$\mathrm{allocdatas}(byte^*_{u0})$

1. If $byte^*_{u0}$ is ϵ, then:
   a. Return ϵ.
2. Let $byte^*$ $byte'^{**}$ be $byte^*_{u0}$.
3. Let da be $\mathrm{allocdata}(byte^*)$.
4. Let da'^* be $\mathrm{allocdatas}(byte'^{**})$.
5. Return da da'^*.

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

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

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

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

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

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

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

$\mathrm{instexport}(fa^*, ga^*, ta^*, ma^*, (\mathsf{export}\ name\ exter_{u0}))$

1. If $exter_{u0}$ is of the case func, then:
   a. Let $(\mathsf{func}\ x)$ be $exter_{u0}$.
   b. Return $\{\mathsf{name}\ name,\ \mathsf{value}\ (\mathsf{func}\ fa^*[x])\}$.
2. If $exter_{u0}$ is of the case global, then:
   a. Let $(\mathsf{global}\ x)$ be $exter_{u0}$.
   b. Return $\{\mathsf{name}\ name,\ \mathsf{value}\ (\mathsf{global}\ ga^*[x])\}$.
3. If $exter_{u0}$ is of the case table, then:
   a. Let $(\mathsf{table}\ x)$ be $exter_{u0}$.
   b. Return $\{\mathsf{name}\ name,\ \mathsf{value}\ (\mathsf{table}\ ta^*[x])\}$.
4. Assert: Due to validation, $exter_{u0}$ is of the case mem.
5. Let $(\mathsf{mem}\ x)$ be $exter_{u0}$.
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 $\mathrm{funcs}(externval^*)$.
2. Let ga_{ex}^* be $\mathrm{globals}(externval^*)$.
3. Let ma_{ex}^* be $\mathrm{mems}(externval^*)$.
4. Let ta_{ex}^* be $\mathrm{tables}(externval^*)$.
5. Assert: Due to validation, $module$ is of the case module.
6. Let $(\mathsf{module}\ type^*\ import^*\ func^{n_f}\ y_0\ y_1\ y_2\ y_3\ y_4\ start^?\ export^*)$ be $module$.
7. Let $(\mathsf{data}\ byte^*\ datamode)^{n_d}$ be y_4.
8. Let $(\mathsf{elem}\ reftype\ expr_e^*\ elemmode)^{n_e}$ be y_3.
9. Let $(\mathsf{memory}\ memtype)^{n_m}$ be y_2.
10. Let $(\mathsf{table}\ tabletype\ expr_t)^{n_t}$ be y_1.
11. Let $(\mathsf{global}\ globaltype\ expr_g)^{n_g}$ be y_0.
12. Let dt^* be $\mathrm{alloctypes}(type^*)$.
13. Let fa^* be $|s.\mathsf{funcs}| + i_f^{i_f<n_f}$.
14. Let ga^* be $|s.\mathsf{globals}| + i_g^{i_g<n_g}$.
15. Let ta^* be $|s.\mathsf{tables}| + i_t^{i_t<n_t}$.
16. Let ma^* be $|s.\mathsf{mems}| + i_m^{i_m<n_m}$.
17. Let ea^* be $|s.\mathsf{elems}| + i_e^{i_e<n_e}$.
18. Let da^* be $|s.\mathsf{datas}| + i_d^{i_d<n_d}$.

19. Let xi^* be $\mathrm{instexport}(fa_{ex}^*\ fa^*, ga_{ex}^*\ ga^*, ta_{ex}^*\ ta^*, ma_{ex}^*\ ma^*, export)^*$.
20. Let mm be $\{\mathsf{types}\ dt^*,\ \mathsf{funcs}\ fa_{ex}^*\ fa^*,\ \mathsf{globals}\ ga_{ex}^*\ ga^*,\ \mathsf{tables}\ ta_{ex}^*\ ta^*,\ \mathsf{mems}\ ma_{ex}^*\ ma^*,\ \mathsf{elems}\ ea^*,\ \mathsf{datas}\ da^*,\ \mathsf{expor}$
21. Let y_0 be $\mathrm{allocfuncs}(mm, func^{n_f})$.
22. Assert: Due to validation, y_0 is fa^*.
23. Let y_0 be $\mathrm{allocglobals}(globaltype^{n_g}, val_g^*)$.
24. Assert: Due to validation, y_0 is ga^*.
25. Let y_0 be $\mathrm{alloctables}(tabletype^{n_t}, ref_t^*)$.
26. Assert: Due to validation, y_0 is ta^*.
27. Let y_0 be $\mathrm{allocmems}(memtype^{n_m})$.
28. Assert: Due to validation, y_0 is ma^*.
29. Let y_0 be $\mathrm{allocelems}(reftype^{n_e}, {ref_e^*}^*)$.
30. Assert: Due to validation, y_0 is ea^*.
31. Let y_0 be $\mathrm{allocdatas}({byte^*}^{n_d})$.
32. Assert: Due to validation, y_0 is da^*.
33. Return mm.

$$
\begin{array}{rcll}
\mathrm{allocmodule}(s, \mathit{module}, \mathit{externval}^*, \mathit{val}_g^*, \mathit{ref}_t^*, (\mathit{ref}_e^*)^*) & = & (s_6,\ \mathit{mm}) & \mathsf{if}\ \mathit{module} = \mathsf{module} \\
&&& \qquad\qquad \mathit{type}^* \\
&&& \qquad\qquad \mathit{import}^* \\
&&& \qquad\qquad \mathit{func}^{n_f} \\
&&& \qquad\qquad (\mathsf{global}\ \mathit{globaltype}\ \mathit{expr}_g)^{n_g} \\
&&& \qquad\qquad (\mathsf{table}\ \mathit{tabletype}\ \mathit{expr}_t)^{n_t} \\
&&& \qquad\qquad (\mathsf{memory}\ \mathit{memtype})^{n_m} \\
&&& \qquad\qquad (\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}_e^*\ \mathit{elemmode})^{n_e} \\
&&& \qquad\qquad (\mathsf{data}\ \mathit{byte}^*\ \mathit{datamode})^{n_d} \\
&&& \qquad\qquad \mathit{start}^? \\
&&& \qquad\qquad \mathit{export}^* \\
&&& \wedge\ \mathit{fa}_{ex}^* = \mathrm{funcs}(\mathit{externval}^*) \\
&&& \wedge\ \mathit{ga}_{ex}^* = \mathrm{globals}(\mathit{externval}^*) \\
&&& \wedge\ \mathit{ta}_{ex}^* = \mathrm{tables}(\mathit{externval}^*) \\
&&& \wedge\ \mathit{ma}_{ex}^* = \mathrm{mems}(\mathit{externval}^*) \\
&&& \wedge\ \mathit{fa}^* = |s.\mathsf{funcs}| + i_f^{i_f<n_f} \\
&&& \wedge\ \mathit{ga}^* = |s.\mathsf{globals}| + i_g^{i_g<n_g} \\
&&& \wedge\ \mathit{ta}^* = |s.\mathsf{tables}| + i_t^{i_t<n_t} \\
&&& \wedge\ \mathit{ma}^* = |s.\mathsf{mems}| + i_m^{i_m<n_m} \\
&&& \wedge\ \mathit{ea}^* = |s.\mathsf{elems}| + i_e^{i_e<n_e} \\
&&& \wedge\ \mathit{da}^* = |s.\mathsf{datas}| + i_d^{i_d<n_d} \\
&&& \wedge\ \mathit{xi}^* = \mathrm{instexport}(\mathit{fa}_{ex}^*\ \mathit{fa}^*, \mathit{ga}_{ex}^*\ \mathit{ga}^*, \mathit{ta}_{ex}^*\ \mathit{ta}^*, \\
&&& \wedge\ \mathit{mm} = \{\mathsf{types}\ \mathit{dt}^*, \\
&&& \qquad\quad \mathsf{funcs}\ \mathit{fa}_{ex}^*\ \mathit{fa}^*, \\
&&& \qquad\quad \mathsf{globals}\ \mathit{ga}_{ex}^*\ \mathit{ga}^*, \\
&&& \qquad\quad \mathsf{tables}\ \mathit{ta}_{ex}^*\ \mathit{ta}^*, \\
&&& \qquad\quad \mathsf{mems}\ \mathit{ma}_{ex}^*\ \mathit{ma}^*, \\
&&& \qquad\quad \mathsf{elems}\ \mathit{ea}^*, \\
&&& \qquad\quad \mathsf{datas}\ \mathit{da}^*, \\
&&& \qquad\quad \mathsf{exports}\ \mathit{xi}^*\} \\
&&& \wedge\ \mathit{dt}^* = \mathrm{alloctypes}(\mathit{type}^*) \\
&&& \wedge\ (s_1,\ \mathit{fa}^*) = \mathrm{allocfuncs}(s, \mathit{mm}, \mathit{func}^{n_f}) \\
&&& \wedge\ (s_2,\ \mathit{ga}^*) = \mathrm{allocglobals}(s_1, \mathit{globaltype}^{n_g}, \mathit{val}_g^*) \\
&&& \wedge\ (s_3,\ \mathit{ta}^*) = \mathrm{alloctables}(s_2, \mathit{tabletype}^{n_t}, \mathit{ref}_t^*) \\
&&& \wedge\ (s_4,\ \mathit{ma}^*) = \mathrm{allocmems}(s_3, \mathit{memtype}^{n_m}) \\
&&& \wedge\ (s_5,\ \mathit{ea}^*) = \mathrm{allocelems}(s_4, \mathit{reftype}^{n_e}, (\mathit{ref}_e^*)^*) \\
&&& \wedge\ (s_6,\ \mathit{da}^*) = \mathrm{allocdatas}(s_5, (\mathit{byte}^*)^{n_d})
\end{array}
$$

4.5.2 Instantiation

$\mathrm{inst}_{mm}(rt)$

1. Let dt^* be mm.types.
2. Return $rt[:= dt^*]$.

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

$\mathrm{rundata}((\mathsf{data}\ \mathit{byte}^*\ \mathit{datam}_{u0}), y)$

1. If datam_{u0} is passive, then:
   a. Return ϵ.
2. Assert: Due to validation, datam_{u0} is of the case active.
3. Let $(\mathsf{active}\ x\ \mathit{instr}^*)$ be datam_{u0}.
4. Return $\mathit{instr}^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ |\mathit{byte}^*|)\ (\mathsf{memory.init}\ x\ y)\ (\mathsf{data.drop}\ y)$.

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

$\mathrm{runelem}((\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}^*\ \mathit{elemm}_{u0}), y)$

1. If elemm_{u0} is passive, then:
   a. Return ϵ.
2. If elemm_{u0} is declare, then:
   a. Return $(\mathsf{elem.drop}\ y)$.
3. Assert: Due to validation, elemm_{u0} is of the case active.
4. Let $(\mathsf{active}\ x\ \mathit{instr}^*)$ be elemm_{u0}.
5. Return $\mathit{instr}^*\ (\mathsf{i32.const}\ 0)\ (\mathsf{i32.const}\ |\mathit{expr}^*|)\ (\mathsf{table.init}\ x\ y)\ (\mathsf{elem.drop}\ y)$.

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

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

1. Assert: Due to validation, module is of the case module.
2. Let $(\mathsf{module}\ \mathit{type}^*\ \mathit{import}^*\ \mathit{func}^*\ \mathit{global}^*\ \mathit{table}^*\ \mathit{mem}^*\ \mathit{elem}^*\ \mathit{data}^*\ \mathit{start}^?\ \mathit{export}^*)$ be module.
3. Let n_{d} be $|\mathit{data}^*|$.
4. Let n_E be $|\mathit{elem}^*|$.
5. Let n_{f} be $|\mathit{func}^*|$.
6. Let $(\mathsf{start}\ x)^?$ be $\mathit{start}^?$.
7. Let $(\mathsf{global}\ \mathit{globaltype}\ \mathit{expr}_{\mathsf{g}})^*$ be global^*.
8. Let $(\mathsf{table}\ \mathit{tabletype}\ \mathit{expr}_{\mathsf{t}})^*$ be table^*.
9. Let $(\mathsf{elem}\ \mathit{reftype}\ \mathit{expr}_E^*\ \mathit{elemmode})^*$ be elem^*.
10. Let $\mathit{instr}_{\mathsf{d}}^*$ be $\mathrm{concat}_{\mathrm{rundata}(\mathit{data}^*[j], j)^{j<n_{\mathsf{d}}}}$.
11. Let instr_E^* be $\mathrm{concat}_{\mathrm{runelem}(\mathit{elem}^*[i], i)^{i<n_E}}$.
12. Let $\mathit{mm}_{\mathit{init}}$ be $\{\mathsf{types}\ \mathrm{alloctypes}(\mathit{type}^*),\ \mathsf{funcs}\ \mathrm{funcs}(\mathit{externval}^*)\ |s.\mathsf{funcs}| + i_{\mathsf{f}}^{i_{\mathsf{f}}<n_{\mathsf{f}}},\ \mathsf{globals}\ \mathrm{globals}(\mathit{externval}^*),\ \mathsf{tables}\ \epsilon$
13. Let z be $\{\mathsf{locals}\ \epsilon,\ \mathsf{module}\ \mathit{mm}_{\mathit{init}}\}$.

14. Push the activation of z to the stack.
15. Let val_{g}^* be $\mathrm{eval}_{expr}(expr_{\mathsf{g}})^*$.
16. Pop the activation of z from the stack.
17. Push the activation of z to the stack.
18. Let ref_{t}^* be $\mathrm{eval}_{expr}(expr_{\mathsf{t}})^*$.
19. Pop the activation of z from the stack.
20. Push the activation of z to the stack.
21. Let ${ref_E^*}^*$ be $\mathrm{eval}_{expr}(expr_E)^{**}$.
22. Pop the activation of z from the stack.
23. Let mm be $\mathrm{allocmodule}(module, externval^*, val_{\mathsf{g}}^*, ref_{\mathsf{t}}^*, {ref_E^*}^*)$.
24. Let f be $\{\mathsf{locals}\ \epsilon,\ \mathsf{module}\ mm\}$.
25. Enter the activation of f with arity 0 with label $\mathsf{frame\_}$:
    a. Execute the sequence $instr_E^*$.
    b. Execute the sequence $instr_{\mathsf{d}}^*$.
    c. If x is defined, then:
        1) Let x_0 be x.
        2) Execute the instruction $(\mathsf{call}\ x_0)$.
26. Return mm.

$$
\begin{array}{lcll}
\mathrm{instantiate}(s, module, externval^*) & = & s'; f; instr_E^*\ instr_{\mathsf{d}}^*\ (\mathsf{call}\ x)^? & \text{if } module = \mathsf{module}\ type^*\ import^*\ func^*\ global^*\ tab \\
&&& \wedge\ global^* = (\mathsf{global}\ globaltype\ expr_{\mathsf{g}})^* \\
&&& \wedge\ table^* = (\mathsf{table}\ tabletype\ expr_{\mathsf{t}})^* \\
&&& \wedge\ elem^* = (\mathsf{elem}\ reftype\ expr_E^*\ elemmode)^* \\
&&& \wedge\ start^? = (\mathsf{start}\ x)^? \\
&&& \wedge\ n_{\mathsf{f}} = |func^*| \\
&&& \wedge\ n_E = |elem^*| \\
&&& \wedge\ n_{\mathsf{d}} = |data^*| \\
&&& \wedge\ mm_{init} = \{\mathsf{types}\ \mathrm{alloctypes}(type^*), \\
&&& \qquad\qquad \mathsf{funcs}\ \mathrm{funcs}(externval^*)\ |s.\mathsf{funcs}| + i_{\mathsf{f}}^{i_{\mathsf{f}}} \\
&&& \qquad\qquad \mathsf{globals}\ \mathrm{globals}(externval^*), \\
&&& \qquad\qquad \} \\
&&& \wedge\ z = s; \{\mathsf{module}\ mm_{init}\} \\
&&& \wedge\ (z; expr_{\mathsf{g}} \hookrightarrow^*\ z; val_{\mathsf{g}})^* \\
&&& \wedge\ (z; expr_{\mathsf{t}} \hookrightarrow^*\ z; ref_{\mathsf{t}})^* \\
&&& \wedge\ (z; expr_E \hookrightarrow^*\ z; ref_E)^{**} \\
&&& \wedge\ (s',\ mm) = \mathrm{allocmodule}(s, module, externval^*, v \\
&&& \wedge\ f = \{\mathsf{module}\ mm\} \\
&&& \wedge\ instr_E^* = \mathrm{concat}(\mathrm{runelem}(elem^*[i], i)^{i<n_E}) \\
&&& \wedge\ instr_{\mathsf{d}}^* = \mathrm{concat}(\mathrm{rundata}(data^*[j], j)^{j<n_{\mathsf{d}}})
\end{array}
$$

4.5.3 Invocation

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

1. Let f be $\{\mathsf{locals}~\epsilon,~\mathsf{module}~\{\mathsf{types}~\epsilon,~\mathsf{funcs}~\epsilon,~\mathsf{globals}~\epsilon,~\mathsf{tables}~\epsilon,~\mathsf{mems}~\epsilon,~\mathsf{elems}~\epsilon,~\mathsf{datas}~\epsilon,~\mathsf{exports}~\epsilon\}\}$.
2. Assert: Due to validation, $\mathrm{expand}(s.\mathsf{funcs}[fa].\mathsf{type})$ is of the case func.
3. Let $(\mathsf{func}~y_0)$ be $\mathrm{expand}(s.\mathsf{funcs}[fa].\mathsf{type})$.
4. Let $t_1^n \rightarrow t_2^*$ be y_0.
5. Assert: Due to validation, $\mathrm{funcinst}[fa]$.code is of the case func.
6. Let k be $|t_2^*|$.
7. Enter the activation of f with arity k with label frame_:
   a. Push the values val^n to the stack.
   b. Push the value $(\mathsf{ref.func}~fa)$ to the stack.
   c. Execute the instruction $(\mathsf{call\_ref}~\mathrm{funcinst}[fa].\mathsf{type})$.
8. Pop the values val^k from the stack.
9. Return val^k.

$$\begin{array}{rcll}
\mathrm{invoke}(s, fa, val^n) & = & s; f; val^n~(\mathsf{ref.func}~fa)~(\mathsf{call_ref}~(s; f).\mathsf{funcs}[fa].\mathsf{type}) & \mathrm{if}~f = \{\mathsf{module}~\{\}\} \\
&&& \wedge~(s; f).\mathsf{funcs}[fa].\mathsf{code} = \mathsf{func}~x~local^*~expr \\
&&& \wedge~s.\mathsf{funcs}[fa].\mathsf{type} \approx \mathsf{func}~(t_1^n \rightarrow t_2^*)
\end{array}$$

4.5.4 Address Getters

funcaddr

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

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

4.5.5 Getters

$\mathrm{type}(x)$

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

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

$\mathrm{func}(x)$

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

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

$\mathrm{global}(x)$

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

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

$\mathrm{table}(x)$

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

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

$\mathrm{mem}(x)$

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

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

$\mathrm{elem}(x)$

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

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

$\mathrm{data}(x)$

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

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

$\mathrm{local}(x)$

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

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

4.5.6 Setters

$\mathrm{with}_{local}(x, v)$

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

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

$C[\mathsf{local}[local^*_{u0}] = local^*_{u1}]$

1. If $local^*_{u0}$ is ϵ and $local^*_{u1}$ is ϵ, then:
   a. Return C.
2. Assert: Due to validation, $|local^*_{u1}|$ is greater than or equal to 1.
3. Let $lt_1\ lt^*$ be $local^*_{u1}$.
4. Assert: Due to validation, $|local^*_{u0}|$ is greater than or equal to 1.
5. Let $x_1\ x^*$ be $local^*_{u0}$.
6. Return $C[\mathsf{locals}[x_1] = lt_1][\mathsf{local}[x^*] = lt^*]$.

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

$\mathrm{with}_{global}(x, v)$

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

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

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

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

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

$\mathrm{with}_{tableinst}(x, ti)$

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

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

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

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

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

$\mathrm{with}_{meminst}(x, mi)$

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

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

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

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

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

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

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

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

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

1. Replace $s.\mathsf{arrays}[a].\mathsf{fields}[i]$ with fv.

$$(s; f)[\mathsf{arrays}[a].\mathsf{fields}[i] = fv] \quad = \quad s[\mathsf{arrays}[a].\mathsf{fields}[i] = fv]; f$$

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

1. Replace $s.\mathsf{structs}[a].\mathsf{fields}[i]$ with fv.

$$(s; f)[\mathsf{structs}[a].\mathsf{fields}[i] = fv] \quad = \quad s[\mathsf{structs}[a].\mathsf{fields}[i] = fv]; f$$