NanoWasm Specification

The SpecTec Team

Jul 01, 2025

CONTENTS

NanoWasm is a small language with simple types and instructions.

ABSTRACT SYNTAX

The *abstract syntax* of types is as follows:

$$
\begin{array}{rcl}
mut & ::= & \mathsf{mut} \\
valtype & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
functype & ::= & valtype^* \rightarrow valtype^* \\
globaltype & ::= & mut^? \; valtype
\end{array}
$$

Instructions take the following form:

$$
\begin{array}{rcl}
const & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
instr & ::= & \mathsf{nop} \\
 & \mid & \mathsf{drop} \\
 & \mid & \mathsf{select} \\
 & \mid & valtype.\mathsf{const} \; const \\
 & \mid & \mathsf{local.get} \; localidx \\
 & \mid & \mathsf{local.set} \; localidx \\
 & \mid & \mathsf{global.get} \; globalidx \\
 & \mid & \mathsf{global.set} \; globalidx
\end{array}
$$

The instruction nop does nothing, drop removes an operand from the stack, select picks one of two operands depending on a condition value. The instruction $t.\mathsf{const} \; c$ pushed the constant c to the stack. The remaining instructions access local and global variables.

VALIDATION

NanoWasm instructions are *type-checked* under a context that assigns types to indices:

$$context \quad ::= \quad \{\text{globals } globaltype^*, \text{locals } valtype^*\}$$

2.1 nop

nop is valid with $\epsilon \rightarrow \epsilon$.

$$\overline{C \vdash \text{nop} : \epsilon \rightarrow \epsilon}$$

2.2 drop

drop is valid with $t \rightarrow \epsilon$.

$$\overline{C \vdash \text{drop} : t \rightarrow \epsilon}$$

2.3 select

select is valid with $t\ t$ i32 $\rightarrow t$.

$$\overline{C \vdash \text{select} : t\ t \text{ i32} \rightarrow t}$$

2.4 const

$(t.\text{const } c)$ is valid with $\epsilon \rightarrow t$.

$$\overline{C \vdash t.\text{const } c : \epsilon \rightarrow t}$$

2.5 local.get

$(\text{local.get } x)$ is valid with $\epsilon \rightarrow t$ if:

- $C.\text{locals}[x]$ exists.
- $C.\text{locals}[x]$ is of the form t.

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

2.6 local.set

(local.set x) is valid with $t \rightarrow \epsilon$ if:

- $C.\text{locals}[x]$ exists.

- $C.\text{locals}[x]$ is of the form t.

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

2.7 global.get

(global.get x) is valid with $\epsilon \rightarrow t$ if:

- $C.\text{globals}[x]$ exists.

- $C.\text{globals}[x]$ is of the form $(\text{mut}^? \, t)$.

$$\frac{C.\text{globals}[x] = \text{mut}^? \, t}{C \vdash \text{global.get } x : \epsilon \rightarrow t}$$

2.8 global.set

(global.get x) is valid with $t \rightarrow \epsilon$ if:

- $C.\text{globals}[x]$ exists.

- $C.\text{globals}[x]$ is of the form $(\text{mut } t)$.

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

EXECUTION

NanoWasm execution requires a suitable definition of state and configuration:

$$
\begin{array}{rcl}
addr & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
moduleinst & ::= & \{\mathsf{globals}\ addr^*\} \\
val & ::= & \mathsf{const}\ valtype\ const \\
store & ::= & \{\mathsf{globals}\ val^*\} \\
frame & ::= & \{\mathsf{locals}\ val^*, \mathsf{module}\ moduleinst\} \\
state & ::= & store; frame \\
config & ::= & state; instr^*
\end{array}
$$

We define the following auxiliary functions for accessing and updating the state:

$$
\begin{array}{rcl}
\mathrm{local}((s; f), x) & = & f.\mathsf{locals}[x] \\
\mathrm{global}((s; f), x) & = & s.\mathsf{globals}[f.\mathsf{module}.\mathsf{globals}[x]] \\
\mathrm{update}_{local}((s; f), x, v) & = & s; f[.\mathsf{locals}[x] = v] \\
\mathrm{update}_{global}((s; f), x, v) & = & s[.\mathsf{globals}[f.\mathsf{module}.\mathsf{globals}[x]] = v]; f
\end{array}
$$

With that, execution is defined as follows:

3.1 nop

1. Do nothing.

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

3.2 drop

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

2. Pop the value val from the stack.

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

3.3 select

1. Assert: Due to validation, a value of valtype 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 $c \neq 0$, then:

    a. Push the value val_1 to the stack.

8. Else:

    a. Push the value val_2 to the stack.

$$val_1 \ val_2 \ (\text{i32.const } c) \ \text{select} \quad \hookrightarrow \quad val_1 \quad \text{if } c \neq 0$$
$$val_1 \ val_2 \ (\text{i32.const } c) \ \text{select} \quad \hookrightarrow \quad val_2 \quad \text{otherwise}$$

3.4 local.get x

1. Let z be the current state.

2. Let val be $\text{local}(z, x)$.

3. Push the value val to the stack.

$$z; (\text{local.get } x) \quad \hookrightarrow \quad z; val \quad \text{if } val = \text{local}(z, x)$$

3.5 local.set x

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

2. Pop the value val from the stack.

$$z; val \ (\text{local.set } x) \quad \hookrightarrow \quad z'; \epsilon \quad \text{if } z' = \text{update}_{local}(z, x, val)$$

3.6 global.get x

1. Let z be the current state.

2. Let val be $\text{global}(z, x)$.

3. Push the value val to the stack.

$$z; (\text{global.get } x) \quad \hookrightarrow \quad z; val \quad \text{if } val = \text{global}(z, x)$$

3.7 global.set x

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

2. Pop the value val from the stack.

$$z; val \ (\text{global.set } x) \quad \hookrightarrow \quad z'; \epsilon \quad \text{if } z' = \text{update}_{global}(z, x, val)$$

BINARY FORMAT

The following grammars define the binary representation of NanoWasm programs.

First, constants are represented in LEB format:

$$
\begin{array}{rcll}
\texttt{byte} & ::= & b\text{:}\texttt{0x00} \mid \ldots \mid b\text{:}\texttt{0xFF} & \Rightarrow & b \\
\texttt{u}(N) & ::= & n\text{:}\texttt{byte} & \Rightarrow & n & \text{if } n < 2^7 \wedge n < 2^N \\
& \mid & n\text{:}\texttt{byte}\ m\text{:}\texttt{u}(N-7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & \text{if } n \geq 2^7 \wedge N > 7 \\
\texttt{u32} & ::= & n\text{:}\texttt{u}(32) & \Rightarrow & n \\
\texttt{u64} & ::= & n\text{:}\texttt{u}(64) & \Rightarrow & n \\
\texttt{f}(N) & ::= & b^*\text{:}\texttt{byte}^{N/8} & \Rightarrow & \text{float}(N, b^*) \\
\texttt{f32} & ::= & p\text{:}\texttt{f}(32) & \Rightarrow & p \\
\texttt{f64} & ::= & p\text{:}\texttt{f}(64) & \Rightarrow & p
\end{array}
$$

Types are encoded as follows:

$$
\begin{array}{rcll}
\texttt{valtype} & ::= & \texttt{0x7F} & \Rightarrow & \mathsf{i32} \\
& \mid & \texttt{0x7E} & \Rightarrow & \mathsf{i64} \\
& \mid & \texttt{0x7D} & \Rightarrow & \mathsf{f32} \\
& \mid & \texttt{0x7C} & \Rightarrow & \mathsf{f64} \\
\texttt{mut} & ::= & \texttt{0x00} & \Rightarrow & \epsilon \\
& \mid & \texttt{0x01} & \Rightarrow & \mathsf{mut} \\
\texttt{globaltype} & ::= & t\text{:}\texttt{valtype}\ mut\text{:}\texttt{mut} & \Rightarrow & mut\ t \\
\texttt{resulttype} & ::= & n\text{:}\texttt{u32}\ (t\text{:}\texttt{valtype})^n & \Rightarrow & t^n \\
\texttt{functype} & ::= & \texttt{0x60}\ t_1^*\text{:}\texttt{resulttype}\ t_2^*\text{:}\texttt{resulttype} & \Rightarrow & t_1^* \to t_2^*
\end{array}
$$

Finally, instruction opcodes:

$$
\begin{array}{rcll}
\texttt{globalidx} & ::= & x\text{:}\texttt{u32} & \Rightarrow & x \\
\texttt{localidx} & ::= & x\text{:}\texttt{u32} & \Rightarrow & x \\
\texttt{instr} & ::= & \texttt{0x01} & \Rightarrow & \mathsf{nop} \\
& \mid & \texttt{0x1A} & \Rightarrow & \mathsf{drop} \\
& \mid & \texttt{0x1B} & \Rightarrow & \mathsf{select} \\
& \mid & \texttt{0x20}\ x\text{:}\texttt{localidx} & \Rightarrow & \mathsf{local.get}\ x \\
& \mid & \texttt{0x21}\ x\text{:}\texttt{localidx} & \Rightarrow & \mathsf{local.set}\ x \\
& \mid & \texttt{0x23}\ x\text{:}\texttt{globalidx} & \Rightarrow & \mathsf{global.get}\ x \\
& \mid & \texttt{0x24}\ x\text{:}\texttt{globalidx} & \Rightarrow & \mathsf{global.set}\ x \\
& \mid & \texttt{0x41}\ n\text{:}\texttt{u32} & \Rightarrow & \mathsf{i32.const}\ n \\
& \mid & \texttt{0x42}\ n\text{:}\texttt{u64} & \Rightarrow & \mathsf{i64.const}\ n \\
& \mid & \texttt{0x43}\ p\text{:}\texttt{f32} & \Rightarrow & \mathsf{f32.const}\ p \\
& \mid & \texttt{0x44}\ p\text{:}\texttt{f64} & \Rightarrow & \mathsf{f64.const}\ p
\end{array}
$$