NanoWasm Specification

The SpecTec Team

Feb 06, 2025

CONTENTS

NanoWasm is a small language with simple types and instructions.

CHAPTER

ONE

ABSTRACT SYNTAX

The *abstract syntax* of types is as follows:

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

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.const c pushed the constant c to the stack. The remaining instructions access local and global variables.

CHAPTER

TWO

VALIDATION

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

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

2.1 nop

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

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

2.2 drop

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

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

2.3 select

select is valid with $t\ t\ \mathsf{i32} \ \rightarrow \ t$.

$$\frac{}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \rightarrow t}$$

2.4 const

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

$$\frac{}{C \vdash t.\mathsf{const}\ c : \epsilon \rightarrow t}$$

2.5 local.get

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

- $C.\mathsf{locals}[x]$ exists.
- $C.\mathsf{locals}[x]$ is equal to t.

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

2.6 local.set

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

- $C.\mathsf{locals}[x]$ exists.
- $C.\mathsf{locals}[x]$ is equal to t.

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

2.7 global.get

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

- $C.\mathsf{globals}[x]$ exists.
- $C.\mathsf{globals}[x]$ is equal to $(\mathsf{mut}^?\ t)$.

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

2.8 global.set

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

- $C.\mathsf{globals}[x]$ exists.
- $C.\mathsf{globals}[x]$ is equal to $(\mathsf{mut}\ t)$.

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

CHAPTER

THREE

EXECUTION

NanoWasm execution requires a suitable definition of state and configuration:

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

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

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

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.

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

3.3 select

1. Assert: Due to validation, a value type is on the top of the stack.
2. Pop the value $(\text{i32.const}\ c)$ 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.

$$
\begin{array}{lcll}
val_1\ val_2\ (\text{i32.const}\ c)\ \text{select} & \hookrightarrow & val_1 & \text{if } c \neq 0 \\
val_1\ val_2\ (\text{i32.const}\ c)\ \text{select} & \hookrightarrow & val_2 & \text{otherwise}
\end{array}
$$

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.

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

CHAPTER

FOUR

BINARY FORMAT

The following grammars define the binary representation of NanoWasm programs.

First, constants are represented in LEB format:

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

Types are encoded as follows:

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

Finally, instruction opcodes:

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