SpecTec: A DSL for the Wasm spec

Andreas Rossberg with
Conrad Watt (Cambridge),
Sukyoung Ryu, Dongjun Youn, Jaehyun Lee, Wonho Shin (KAIST)
Philippa Gardner, Henit Mandaliya, Xiaojia Rao (Imperial College)
Joachim Breitner (Freiburg), Matija Pretnar (Ljubljana), Sam Lindley (Edinburgh)

2023/09/12

[PLDI 2017]

(store)	s	::=	$\{\text{inst } inst^*,\ \text{tab } tabinst^*,\ \text{mem } meminst^*\}$
(instances)	$inst$	::=	$\{\text{func } cl^*,\ \text{glob } v^*,\ \text{tab } i^?,\ \text{mem } i^?\}$
	$tabinst$	::=	cl^*
	$meminst$	::=	b^*
(closures)	cl	::=	$\{\text{inst } i,\ \text{code } f\}$ (where f is not an import and has all exports ex^* erased)
(values)	v	::=	$t.\text{const } c$
(administrative operators)	e	::=	$\ldots \mid \textbf{trap} \mid \textbf{call } cl \mid \textbf{label}_n\{e^*\}\ e^*\ \textbf{end} \mid \textbf{local}_n\{i; v^*\}\ e^*\ \textbf{end}$
(local contexts)	L^0	::=	$v^*\ [\_]\ e^*$
	L^{k+1}	::=	$v^*\ \textbf{label}_n\{e^*\}\ L^k\ \textbf{end}\ e^*$

Reduction

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]} \qquad \frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v_0^*; \textbf{local}_n\{i; v^*\}\ e^*\ \textbf{end} \hookrightarrow_j s'; v_0^*; \textbf{local}_n\{i; v'^*\}\ e'^*\ \textbf{end}}$$

$$\boxed{s; v^*; e^* \hookrightarrow_i s; v^*; e^*}$$

$$\begin{array}{rcll}
L^0[\textbf{trap}] & \hookrightarrow & \textbf{trap} & \text{if } L^0 \neq [_] \\
(t.\text{const } c)\ t.unop & \hookrightarrow & t.\text{const } unop_t(c) & \\
(t.\text{const } c_1)\ (t.\text{const } c_2)\ t.binop & \hookrightarrow & t.\text{const } c & \text{if } c = binop_t(c_1, c_2) \\
(t.\text{const } c_1)\ (t.\text{const } c_2)\ t.binop & \hookrightarrow & \textbf{trap} & \text{otherwise} \\
(t.\text{const } c)\ t.testop & \hookrightarrow & \textbf{i32.const } testop_t(c) & \\
(t.\text{const } c_1)\ (t.\text{const } c_2)\ t.relop & \hookrightarrow & \textbf{i32.const } relop_t(c_1, c_2) & \\
(t_1.\text{const } c)\ t_2.\textbf{convert } t_1_sx^? & \hookrightarrow & t_2.\text{const } c' & \text{if } c' = \text{cvt}^{sx^?}_{t_1,t_2}(c) \\
(t_1.\text{const } c)\ t_2.\textbf{convert } t_1_sx^? & \hookrightarrow & \textbf{trap} & \text{otherwise} \\
(t_1.\text{const } c)\ t_2.\textbf{reinterpret } t_1 & \hookrightarrow & t_2.\text{const } \text{const}_{t_2}(\text{bits}_{t_1}(c)) & \\
\textbf{unreachable} & \hookrightarrow & \textbf{trap} & \\
\textbf{nop} & \hookrightarrow & \epsilon & \\
v\ \textbf{drop} & \hookrightarrow & \epsilon & \\
v_1\ v_2\ (\textbf{i32.const } 0)\ \textbf{select} & \hookrightarrow & v_2 & \\
v_1\ v_2\ (\textbf{i32.const } k+1)\ \textbf{select} & \hookrightarrow & v_1 & \\
v^n\ \textbf{block}\ (t_1^n \to t_2^m)\ e^*\ \textbf{end} & \hookrightarrow & \textbf{label}_m\{\epsilon\}\ v^n\ e^*\ \textbf{end} & \\
v^n\ \textbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \textbf{end} & \hookrightarrow & \textbf{label}_n\{\textbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \textbf{end}\}\ v^n\ e^*\ \textbf{end} & \\
(\textbf{i32.const } 0)\ \textbf{if}\ tf\ e_1^*\ \textbf{else}\ e_2^*\ \textbf{end} & \hookrightarrow & \textbf{block}\ tf\ e_2^*\ \textbf{end} & \\
(\textbf{i32.const } k+1)\ \textbf{if}\ tf\ e_1^*\ \textbf{else}\ e_2^*\ \textbf{end} & \hookrightarrow & \textbf{block}\ tf\ e_1^*\ \textbf{end} & \\
\textbf{label}_n\{e^*\}\ v^*\ \textbf{end} & \hookrightarrow & v^* & \\
\textbf{label}_n\{e^*\}\ \textbf{trap end} & \hookrightarrow & \textbf{trap} & \\
\textbf{label}_n\{e^*\}\ L^j[v^n\ (\textbf{br } j)]\ \textbf{end} & \hookrightarrow & v^n\ e^* & \\
(\textbf{i32.const } 0)\ (\textbf{br_if } j) & \hookrightarrow & \epsilon & \\
(\textbf{i32.const } k+1)\ (\textbf{br_if } j) & \hookrightarrow & \textbf{br } j & \\
(\textbf{i32.const } k)\ (\textbf{br_table } j_1^k\ j\ j_2^*) & \hookrightarrow & \textbf{br } j & \\
(\textbf{i32.const } k+n)\ (\textbf{br_table } j_1^k\ j) & \hookrightarrow & \textbf{br } j & \\
s; \textbf{call } j & \hookrightarrow_i & \textbf{call } s_{\text{func}}(i,j) & \\
s; (\textbf{i32.const } j)\ \textbf{call_indirect } tf & \hookrightarrow_i & \textbf{call } s_{\text{tab}}(i,j) & \text{if } s_{\text{tab}}(i,j)_{\text{code}} = (\textbf{func } tf\ \textbf{local } t^*\ e^*) \\
s; (\textbf{i32.const } j)\ \textbf{call_indirect } tf & \hookrightarrow_i & \textbf{trap} & \text{otherwise} \\
v^n\ (\textbf{call } cl) & \hookrightarrow & \textbf{local}_m\{cl_{\text{inst}}; v^n\ (t.\text{const } 0)^k\}\ \textbf{block}\ (\epsilon \to t_2^m)\ e^*\ \textbf{end end} \quad \ldots & \\
\textbf{local}_n\{i; v_l^*\}\ v^n\ \textbf{end} & \hookrightarrow & v^n & \mid \quad \ldots \text{if } cl_{\text{code}} = (\textbf{func } (t_1^n \to t_2^m)\ \textbf{local } t^k\ e^*) \\
\textbf{local}_n\{i; v_l^*\}\ \textbf{trap end} & \hookrightarrow & \textbf{trap} & \\
\textbf{local}_n\{i; v_l^*\}\ L^k[v^n\ \textbf{return}]\ \textbf{end} & \hookrightarrow & v^n & \\
v_1^j\ v\ v_2^k; \textbf{get_local } j & \hookrightarrow & v & \\
v_1^j\ v\ v_2^k; v'\ (\textbf{set_local } j) & \hookrightarrow & v_1^j\ v'\ v_2^k; \epsilon & \\
v\ (\textbf{tee_local } j) & \hookrightarrow & v\ v\ (\textbf{set_local } j) & \\
s; \textbf{get_global } j & \hookrightarrow_i & s_{\text{glob}}(i,j) & \\
s; v\ (\textbf{set_global } j) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \text{glob}(i,j) = v \\
s; (\textbf{i32.const } k)\ (t.\textbf{load } a\ o) & \hookrightarrow_i & t.\text{const } \text{const}_t(b^*) & \text{if } s_{\text{mem}}(i, k+o, |t|) = b^* \\
s; (\textbf{i32.const } k)\ (t.\textbf{load } tp_sx\ a\ o) & \hookrightarrow_i & t.\text{const } \text{const}_t^{sx}(b^*) & \text{if } s_{\text{mem}}(i, k+o, |tp|) = b^* \\
s; (\textbf{i32.const } k)\ (t.\textbf{load } tp_sx^?\ a\ o) & \hookrightarrow_i & \textbf{trap} & \text{otherwise} \\
s; (\textbf{i32.const } k)\ (t.\text{const } c)\ (t.\textbf{store } a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \text{mem}(i, k+o, |t|) = \text{bits}_t^{|t|}(c) \\
s; (\textbf{i32.const } k)\ (t.\text{const } c)\ (t.\textbf{store } tp\ a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \text{mem}(i, k+o, |tp|) = \text{bits}_t^{|tp|}(c) \\
s; (\textbf{i32.const } k)\ (t.\text{const } c)\ (t.\textbf{store } tp^?\ a\ o) & \hookrightarrow_i & \textbf{trap} & \text{otherwise} \\
s; \textbf{current_memory} & \hookrightarrow_i & \textbf{i32.const } |s_{\text{mem}}(i,*)|/64\,\text{Ki} & \\
s; (\textbf{i32.const } k)\ \textbf{grow_memory} & \hookrightarrow_i & s'; \textbf{i32.const } |s_{\text{mem}}(i,*)|/64\,\text{Ki} & \text{if } s' = s \text{ with } \text{mem}(i,*) = s_{\text{mem}}(i,*)\ (0)^{k \cdot 64\,\text{Ki}} \\
s; (\textbf{i32.const } k)\ \textbf{grow_memory} & \hookrightarrow_i & \textbf{i32.const } (-1) &
\end{array}$$

Figure 2. Small-step reduction rules

(contexts) C ::= $\{\text{func } tf^*,\ \text{global } tg^*,\ \text{table } n^?,\ \text{memory } n^?,\ \text{local } t^*,\ \text{label } (t^*)^*,\ \text{return } (t^*)^?\}$

Typing Instructions

$$\boxed{C \vdash e^* : tf}$$

$$\frac{}{C \vdash t.\textbf{const } c : \epsilon \to t} \quad \frac{}{C \vdash t.unop : t \to t} \quad \frac{}{C \vdash t.binop : t\ t \to t} \quad \frac{}{C \vdash t.testop : t \to \text{i32}} \quad \frac{}{C \vdash t.relop : t\ t \to \text{i32}}$$

$$\frac{t_1 \neq t_2 \qquad sx^? = \epsilon \Leftrightarrow (t_1 = \textbf{in} \wedge t_2 = \textbf{in}' \wedge |t_1| < |t_2|) \vee (t_1 = \textbf{fn} \wedge t_2 = \textbf{fn}')}{C \vdash t_1.\textbf{convert } t_2\_sx^? : t_2 \to t_1} \qquad \frac{t_1 \neq t_2 \qquad |t_1| = |t_2|}{C \vdash t_1.\textbf{reinterpret } t_2 : t_2 \to t_1}$$

$$\frac{}{C \vdash \textbf{unreachable} : t_1^* \to t_2^*} \quad \frac{}{C \vdash \textbf{nop} : \epsilon \to \epsilon} \quad \frac{}{C \vdash \textbf{drop} : t \to \epsilon} \quad \frac{}{C \vdash \textbf{select} : t\ t\ \text{i32} \to t}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \text{label } (t_2^m) \vdash e^* : tf}{C \vdash \textbf{block } tf\ e^*\ \textbf{end} : tf} \qquad \frac{tf = t_1^n \to t_2^m \qquad C, \text{label } (t_1^n) \vdash e^* : tf}{C \vdash \textbf{loop } tf\ e^*\ \textbf{end} : tf}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \text{label } (t_2^m) \vdash e_1^* : tf \qquad C, \text{label } (t_2^m) \vdash e_2^* : tf}{C \vdash \textbf{if } tf\ e_1^*\ \textbf{else } e_2^*\ \textbf{end} : t_1^n\ \text{i32} \to t_2^m}$$

$$\frac{C_{\text{label}}(i) = t^*}{C \vdash \textbf{br } i : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\text{label}}(i) = t^*}{C \vdash \textbf{br\_if } i : t^*\ \text{i32} \to t^*} \qquad \frac{(C_{\text{label}}(i) = t^*)^+}{C \vdash \textbf{br\_table } i^+ : t_1^*\ t^*\ \text{i32} \to t_2^*}$$

$$\frac{C_{\text{return}} = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\text{func}}(i) = tf}{C \vdash \textbf{call } i : tf} \qquad \frac{tf = t_1^* \to t_2^* \qquad C_{\text{table}} = n}{C \vdash \textbf{call\_indirect } tf : t_1^*\ \text{i32} \to t_2^*}$$

$$\frac{C_{\text{local}}(i) = t}{C \vdash \textbf{get\_local } i : \epsilon \to t} \qquad \frac{C_{\text{local}}(i) = t}{C \vdash \textbf{set\_local } i : t \to \epsilon} \qquad \frac{C_{\text{local}}(i) = t}{C \vdash \textbf{tee\_local } i : t \to t} \qquad \frac{C_{\text{global}}(i) = \text{mut}^?\ t}{C \vdash \textbf{get\_global } i : \epsilon \to t} \qquad \frac{C_{\text{global}}(i) = \text{mut}\ t}{C \vdash \textbf{set\_global } i : t \to \epsilon}$$

$$\frac{C_{\text{memory}} = n \qquad 2^a \leq (|tp| <)^?|t| \qquad (tp\_sz)^? = \epsilon \vee t = \textbf{im}}{C \vdash t.\textbf{load } (tp\_sz)^?\ a\ o : \text{i32} \to t} \qquad \frac{C_{\text{memory}} = n \qquad 2^a \leq (|tp| <)^?|t| \qquad tp^? = \epsilon \vee t = \textbf{im}}{C \vdash t.\textbf{store } tp^?\ a\ o : \text{i32}\ t \to \epsilon}$$

$$\frac{C_{\text{memory}} = n}{C \vdash \textbf{current\_memory} : \epsilon \to \text{i32}} \qquad \frac{C_{\text{memory}} = n}{C \vdash \textbf{grow\_memory} : \text{i32} \to \text{i32}}$$

$$\frac{}{C \vdash \epsilon : \epsilon \to \epsilon} \qquad \frac{C \vdash e_1^* : t_1^* \to t_2^* \qquad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*} \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}$$

Typing Modules

$$\frac{tf = t_1^* \to t_2^* \qquad C, \text{local } t_1^*\ t^*, \text{label } (t_2^*), \text{return } (t_2^*) \vdash e^* : \epsilon \to t_2^*}{C \vdash ex^*\ \textbf{func } tf\ \textbf{local } t^*\ e^* : ex^*\ tf} \qquad \frac{tg = \text{mut}^?\ t \qquad C \vdash e^* : \epsilon \to t \qquad ex^* = \epsilon \vee tg = t}{C \vdash ex^*\ \textbf{global } tg\ e^* : ex^*\ tg}$$

$$\frac{(C_{\text{func}}(i) = tf)^n}{C \vdash ex^*\ \textbf{table } n\ i^n : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \textbf{memory } n : ex^*\ n}$$

$$\frac{}{C \vdash ex^*\ \textbf{func } tf\ im : ex^*\ tf} \qquad \frac{tg = t}{C \vdash ex^*\ \textbf{global } tg\ im : ex^*\ tg} \qquad \frac{}{C \vdash ex^*\ \textbf{table } n\ im : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \textbf{memory } n\ im : ex^*\ n}$$

$$\frac{\begin{array}{cc}(C \vdash f : ex_{\text{f}}^*\ tf)^* & (C_i \vdash glob_i : ex_{\text{g}}^*\ tg_i)_i^* \\ (C \vdash tab : ex_{\text{t}}^*\ n)^? & (C \vdash mem : ex_{\text{m}}^*\ n)^? \\ (C_i = \{\text{global } tg^{i-1}\})_i^* & C = \{\text{func } tf^*, \text{global } tg^*, \text{table } n^?, \text{memory } n^?\} \\ ex_{\text{f}}^{**}\ ex_{\text{g}}^{**}\ ex_{\text{t}}^{*?}\ ex_{\text{m}}^{*?} \text{ distinct} & \end{array}}{\vdash \textbf{module } f^*\ glob^*\ tab^?\ mem^?}$$

Figure 3. Typing rules

4.2 Soundness

The WebAssembly type system enjoys standard *soundness* properties [41]. Soundness proves that the reduction rules from Section 3 actually cover all execution states that can arise for valid programs. In other words, it proves the absence of undefined behavior in the execution semantics (assuming the auxiliary numeric primitives are well-defined).

In particular, this implies the absence of *type safety* violations such as invalid calls or illegal accesses to locals, it guarantees *memory safety*, and it ensures the inaccessibility of code addresses or the call stack. It also implies that the use of the operand stack is structured and its layout determined statically at all program points, which is crucial for efficient compilation on a register machine. Furthermore, it establishes memory and state *encapsulation* – i.e., abstraction properties on the module and function boundaries, which cannot leak information unless explicitly exported/returned – necessary conditions for user-defined security measures.

Store Typing Before we can state the soundness theorems concretely, we must extend the typing rules to stores and configurations as defined in Figure 2. These rules, shown in Figure 4, are not required for validation, but for generalizing to dynamic computations. They use an additional *store context* S to classify the store. The typing judgement for instructions in Figure 3 is extended to $S; C \vdash e^* : tf$ by implicitly adding S to all rules – S is never modified or used by those rules, but is accessed by the new rules for **call** cl and **local**.

C.return is absent (set to ϵ) when validating an expression that is not a function body. This differs from it being set to the empty result type ($[\epsilon]$), which is the case for functions not returning anything.

call x

- The function C.funcs$[x]$ must be defined in the context.
- Then the instruction is valid with type C.funcs$[x]$.

$$\frac{C.\mathsf{funcs}[x] = [t_1^*] \to [t_2^*]}{C \vdash \mathsf{call}\ x : [t_1^*] \to [t_2^*]}$$

call_indirect x y

- The table C.tables$[x]$ must be defined in the context.
- Let *limits* t be the table type C.tables$[x]$.
- The reference type t must be funcref.
- The type C.types$[y]$ must be defined in the context.
- Let $[t_1^*] \to [t_2^*]$ be the function type C.types$[y]$.
- Then the instruction is valid with type $[t_1^*\ \mathsf{i32}] \to [t_2^*]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}\ \mathsf{funcref} \qquad C.\mathsf{types}[y] = [t_1^*] \to [t_2^*]}{C \vdash \mathsf{call\_indirect}\ x\ y : [t_1^*\ \mathsf{i32}] \to [t_2^*]}$$

3.3.9 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

- The empty instruction sequence is valid with type $[t^*] \to [t^*]$, for any sequence of operand types t^*.

$$\frac{}{C \vdash \epsilon : [t^*] \to [t^*]}$$

Non-empty Instruction Sequence: $\mathit{instr}^*\ \mathit{instr}_N$

- The instruction sequence instr^* must be valid with type $[t_1^*] \to [t_2^*]$, for some sequences of operand types t_1^* and t_2^*.
- The instruction instr_N must be valid with type $[t^*] \to [t_3^*]$, for some sequences of operand types t^* and t_3^*.
- There must be a sequence of operand types t_0^*, such that $t_2^* = t_0^*\ t'^*$ where the type sequence t'^* is as long as t^*.
- For each operand type t_i' in t'^* and corresponding type t_i in t^*, t_i' matches t_i.
- Then the combined instruction sequence is valid with type $[t_1^*] \to [t_0^*\ t_3^*]$.

$$\frac{C \vdash \mathit{instr}^* : [t_1^*] \to [t_0^*\ t'^*] \qquad \vdash [t'^*] \leq [t^*] \qquad C \vdash \mathit{instr}_N : [t^*] \to [t_3^*]}{C \vdash \mathit{instr}^*\ \mathit{instr}_N : [t_1^*] \to [t_0^*\ t_3^*]}$$

br l

1. Assert: due to validation, the stack contains at least $l + 1$ labels.
2. Let L be the l-th label appearing on the stack, starting from the top and counting from zero.
3. Let n be the arity of L.
4. Assert: due to validation, there are at least n values on the top of the stack.
5. Pop the values val^n from the stack.
6. Repeat $l + 1$ times:
   a. While the top of the stack is a value, do:
      i. Pop the value from the stack.
   b. Assert: due to validation, the top of the stack now is a label.
   c. Pop the label from the stack.
7. Push the values val^n to the stack.
8. Jump to the continuation of L.

$$\mathsf{label}_n\{\mathit{instr}^*\}\ B^l[\mathit{val}^n\ (\mathsf{br}\ l)]\ \mathsf{end} \quad \hookrightarrow \quad \mathit{val}^n\ \mathit{instr}^*$$

br_if 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 c from the stack.
3. If c is non-zero, then:
   a. Execute the instruction $(\mathsf{br}\ l)$.
4. Else:
   a. Do nothing.

$$\begin{array}{lcll} (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & (\text{if } c \neq 0) \\ (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & (\text{if } c = 0) \end{array}$$

br_table $l^*\ l_N$

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 smaller than the length of l^*, then:
   a. Let l_i be the label $l^*[i]$.
   b. Execute the instruction $(\mathsf{br}\ l_i)$.
4. Else:
   a. Execute the instruction $(\mathsf{br}\ l_N)$.

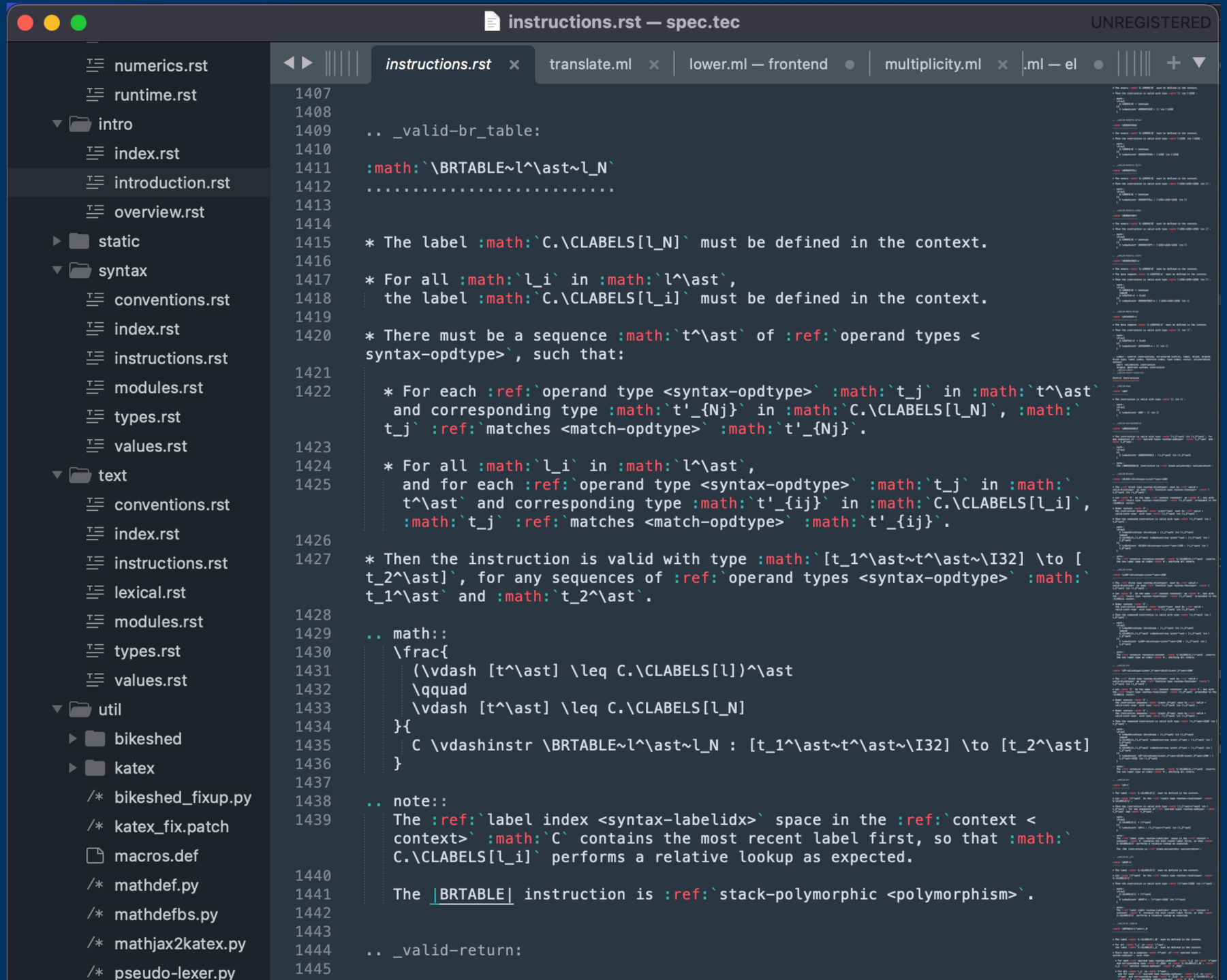
$$\begin{array}{lcll} (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_i) & (\text{if } l^*[i] = l_i) \\ (\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l_N) & \hookrightarrow & (\mathsf{br}\ l_N) & (\text{if } |l^*| \leq i) \end{array}$$

Wasm Spec

Prose essentially is a manual text rendering of the formal rules

…200 pages instead of 2

Written in reStructuredText, an elaborate form of markdown


instructions.rst — spec.tec
UNREGISTERED
numerics.rst
runtime.rst
intro
index.rst
introduction.rst
overview.rst
static
syntax
conventions.rst
index.rst
instructions.rst
modules.rst
types.rst
values.rst
text
conventions.rst
index.rst
instructions.rst
lexical.rst
modules.rst
types.rst
values.rst
util
bikeshed
katex
bikeshed_fixup.py
katex_fix.patch
macros.def
mathdef.py
mathdefbs.py
mathjax2katex.py
pseudo-lexer.py
instructions.rst
translate.ml
lower.ml — frontend
multiplicity.ml
.ml — el
1407
1408
1409 .. _valid-br_table:
1410
1411 :math:`\BRTABLE~l^\ast~l_N`
1412 ....
1413
1414
1415 * The label :math:`C.\CLABELS[l_N]` must be defined in the context.
1416
1417 * For all :math:`l_i` in :math:`l^\ast`,
1418 the label :math:`C.\CLABELS[l_i]` must be defined in the context.
1419
1420 * There must be a sequence :math:`t^\ast` of :ref:`operand types <
syntax-opdtype>`, such that:
1421
1422 * For each :ref:`operand type <syntax-opdtype>` :math:`t_j` in :math:`t^\ast`
and corresponding type :math:`t'_{Nj}` in :math:`C.\CLABELS[l_N]`, :math:`
t_j` :ref:`matches <match-opdtype>` :math:`t'_{Nj}`.
1423
1424 * For all :math:`l_i` in :math:`l^\ast`,
1425 and for each :ref:`operand type <syntax-opdtype>` :math:`t_j` in :math:`
t^\ast` and corresponding type :math:`t'_{ij}` in :math:`C.\CLABELS[l_i]`,
:math:`t_j` :ref:`matches <match-opdtype>` :math:`t'_{ij}`.
1426
1427 * Then the instruction is valid with type :math:`[t_1^\ast~t^\ast~\I32] \to [
t_2^\ast]`, for any sequences of :ref:`operand types <syntax-opdtype>` :math:`
t_1^\ast` and :math:`t_2^\ast`.
1428
1429 .. math::
1430 \frac{
1431 (\vdash [t^\ast] \leq C.\CLABELS[l])^\ast
1432 \qquad
1433 \vdash [t^\ast] \leq C.\CLABELS[l_N]
1434 }{
1435 C \vdashinstr \BRTABLE~l^\ast~l_N : [t_1^\ast~t^\ast~\I32] \to [t_2^\ast]
1436 }
1437
1438 .. note::
1439 The :ref:`label index <syntax-labelidx>` space in the :ref:`context <
context>` :math:`C` contains the most recent label first, so that :math:`
C.\CLABELS[l_i]` performs a relative lookup as expected.
1440
1441 The |BRTABLE| instruction is :ref:`stack-polymorphic <polymorphism>`.
1442
1443
1444 .. _valid-return:
1445

Maintenance

Writing the prose is *extremely* laborious

…took days to produce the formal spec for 1.0, vs. months to write the prose

Every new proposal needs to extend *both* formal and prose rules

Both make for nightmarish code reviews

…Latex math is a write-only language

…reStructuredText is verbose and diff-unfriendly (tables!), repetitive prose puts you to sleep

…no macro facilities in Sphinx (had to hack a plugin, broke *N* times with Sphinx releases)

Merged **Restrict init from stack-polymorphism** #75

Changes from all commits · File filter · Conversations

0 / 2 files viewed

Review changes

Filter changed files

- document/core
  - appendix
    - algorithm.rst
  - valid
    - instructions.rst

28 document/core/valid/instructions.rst

Viewed

```
1259
1260 - * The instruction is valid with any :ref:`valid <valid-instrtype>` type :math:`[t_1^\ast] \to_{x^\ast} [t_2^\ast]`.
1261
1262   .. math::
1263      \frac{
1264 -      C \vdashinstrtype [t_1^\ast] \to_{x^\ast} [t_2^\ast] \ok
1265      }{
1266 -      C \vdashinstr \UNREACHABLE : [t_1^\ast] \to_{x^\ast} [t_2^\ast]
1267      }
1268
1269   .. note::
```

@@ -1365,15 +1365,15 @@ Control Instructions

```
1365
1366   * Let :math:`[t^\ast]` be the :ref:`result type <syntax-resulttype>` :math:`C.\CLABELS[l]`.
1367
1368 + * Then the instruction is valid with type :math:`[t_1^\ast~t^\ast] \to_{x^\ast} [t_2^\ast]`, for any :ref:`valid <valid-instrtype>` type :math:`[t_1^\ast] \to_{x^\ast} [t_2^\ast]`.
1369
1370   .. math::
1371      \frac{
1372        C.\CLABELS[l] = [t^\ast]
1373        \qquad
1374 -      C \vdashinstrtype [t_1^\ast] \to_{x^\ast} [t_2^\ast] \ok
1375      }{
1376 -      C \vdashinstr \BR~l : [t_1^\ast~t^\ast] \to_{x^\ast} [t_2^\ast]
1377      }
1378
1379   .. note::
```

@@ -1422,17 +1422,17 @@ Control Instructions

```
1422     * For all :math:`l_i` in :math:`l^\ast`,
1423       the result type :math:`[t^\ast]` :ref:`matches <match-resulttype>` :math:`C.\CLABELS[l_i]`.
1424
1425 - * Then the instruction is valid with type :math:`[t_1^\ast~t^\ast~\I32] \to_{x^\ast} [t_2^\ast]`, for any :ref:`valid <valid-instrtype>` type :math:`[t_1^\ast]`
```

```
1259
1260 + * The instruction is valid with any :ref:`valid <valid-instrtype>` type of the form :math:`[t_1^\ast] \to [t_2^\ast]`.
1261
1262   .. math::
1263      \frac{
1264 +      C \vdashinstrtype [t_1^\ast] \to [t_2^\ast] \ok
1265      }{
1266 +      C \vdashinstr \UNREACHABLE : [t_1^\ast] \to [t_2^\ast]
1267      }
1268
1269   .. note::
1365
1366   * Let :math:`[t^\ast]` be the :ref:`result type <syntax-resulttype>` :math:`C.\CLABELS[l]`.
1367
1368 + * Then the instruction is valid with any :ref:`valid <valid-instrtype>` type of the form :math:`[t_1^\ast~t^\ast] \to [t_2^\ast]`.
1369
1370   .. math::
1371      \frac{
1372        C.\CLABELS[l] = [t^\ast]
1373        \qquad
1374 +      C \vdashinstrtype [t_1^\ast~t^\ast] \to [t_2^\ast] \ok
1375      }{
1376 +      C \vdashinstr \BR~l : [t_1^\ast~t^\ast] \to [t_2^\ast]
1377      }
1378
1379   .. note::
1422     * For all :math:`l_i` in :math:`l^\ast`,
1423       the result type :math:`[t^\ast]` :ref:`matches <match-resulttype>` :math:`C.\CLABELS[l_i]`.
1424
1425 + * Then the instruction is valid with any :ref:`valid <valid-instrtype>` type of the form :math:`[t_1^\ast~t^\ast~\I32] \to [t_2^\ast]`.
```

Proposals

We currently have

- 8 completed proposals (making up Wasm 2.0)
- 25+ active proposals

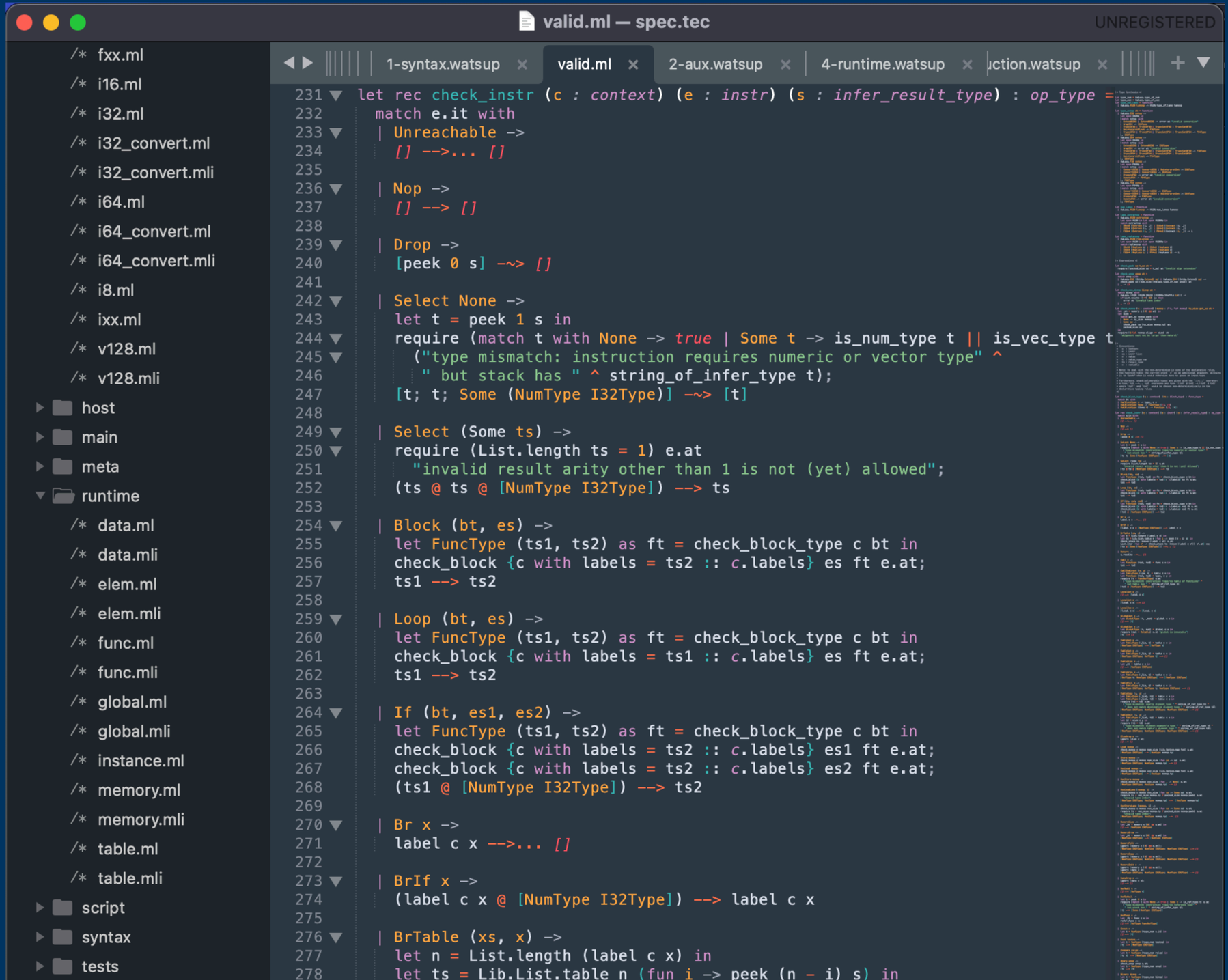
Diffs for proposals range from 1 to 100 Kloc (spec document, interpreter, test suite)

Need to be manually created and code-reviewed

Reference Interpreter

Proposal champions also need to extend the reference interpreter

…essentially, an OCaml rendering of the formal rules


valid.ml — spec.tec
UNREGISTERED
/* fxx.ml
/* i16.ml
/* i32.ml
/* i32_convert.ml
/* i32_convert.mli
/* i64.ml
/* i64_convert.ml
/* i64_convert.mli
/* i8.ml
/* ixx.ml
/* v128.ml
/* v128.mli
host
main
meta
runtime
/* data.ml
/* data.mli
/* elem.ml
/* elem.mli
/* func.ml
/* func.mli
/* global.ml
/* global.mli
/* instance.ml
/* memory.ml
/* memory.mli
/* table.ml
/* table.mli
script
syntax
tests
1-syntax.watsup
valid.ml
2-aux.watsup
4-runtime.watsup
uction.watsup
231 let rec check_instr (c : context) (e : instr) (s : infer_result_type) : op_type
232 match e.it with
233 | Unreachable ->
234 [] -->... []
235
236 | Nop ->
237 [] --> []
238
239 | Drop ->
240 [peek 0 s] -~> []
241
242 | Select None ->
243 let t = peek 1 s in
244 require (match t with None -> true | Some t -> is_num_type t || is_vec_type t
245 ("type mismatch: instruction requires numeric or vector type" ^
246 " but stack has " ^ string_of_infer_type t);
247 [t; t; Some (NumType I32Type)] -~> [t]
248
249 | Select (Some ts) ->
250 require (List.length ts = 1) e.at
251 "invalid result arity other than 1 is not (yet) allowed";
252 (ts @ ts @ [NumType I32Type]) --> ts
253
254 | Block (bt, es) ->
255 let FuncType (ts1, ts2) as ft = check_block_type c bt in
256 check_block {c with labels = ts2 :: c.labels} es ft e.at;
257 ts1 --> ts2
258
259 | Loop (bt, es) ->
260 let FuncType (ts1, ts2) as ft = check_block_type c bt in
261 check_block {c with labels = ts1 :: c.labels} es ft e.at;
262 ts1 --> ts2
263
264 | If (bt, es1, es2) ->
265 let FuncType (ts1, ts2) as ft = check_block_type c bt in
266 check_block {c with labels = ts2 :: c.labels} es1 ft e.at;
267 check_block {c with labels = ts2 :: c.labels} es2 ft e.at;
268 (ts1 @ [NumType I32Type]) --> ts2
269
270 | Br x ->
271 label c x -->... []
272
273 | BrIf x ->
274 (label c x @ [NumType I32Type]) --> label c x
275
276 | BrTable (xs, x) ->
277 let n = List.length (label c x) in
278 let ts = Lib.List.table n (fun i -> peek (n - i) s) in

Mechanisations

Machine-verified translations of formal spec to theorem provers

We currently have two semi-official ones, both done by the WasmCert team

- Isabelle [Watt, CPP 2018]
- Coq [Watt et al., FM 2021]

…some folks also want Agda, Lean, possibly Haskell

Rely on eye-ball correspondence for correctness

```
    reduce_simple [::AI_basic (BI_const v); AI_basic (BI_cvtop t2 CVO_reinterpret t1 None)] [::AI_basic (BI_const (wasm_deserialise (bits v) t2))]

  (** control-flow operations **)
  | rs_unreachable :
    reduce_simple [::AI_basic BI_unreachable] [::AI_trap]
  | rs_nop :
    reduce_simple [::AI_basic BI_nop] [::]
  | rs_drop :
    forall v,
    reduce_simple [::AI_basic (BI_const v); AI_basic BI_drop] [::]
  | rs_select_false :
    forall n v1 v2,
    n = Wasm_int.int_zero i32m ->
    reduce_simple [::AI_basic (BI_const v1); AI_basic (BI_const v2); AI_basic (BI_const (VAL_int32 n)); AI_basic BI_select] [::AI_basic (BI_const v2)]
  | rs_select_true :
    forall n v1 v2,
    n <> Wasm_int.int_zero i32m ->
    reduce_simple [::AI_basic (BI_const v1); AI_basic (BI_const v2); AI_basic (BI_const (VAL_int32 n)); AI_basic BI_select] [::AI_basic (BI_const v1)]
  | rs_block :
      forall vs es n m t1s t2s,
        const_list vs ->
        length vs = n ->
        length t1s = n ->
        length t2s = m ->
        reduce_simple (vs ++ [::AI_basic (BI_block (Tf t1s t2s) es)]) [::AI_label m [::] (vs ++ to_e_list es)]
  | rs_loop :
      forall vs es n m t1s t2s,
        const_list vs ->
        length vs = n ->
        length t1s = n ->
        length t2s = m ->
        reduce_simple (vs ++ [::AI_basic (BI_loop (Tf t1s t2s) es)]) [::AI_label n [::AI_basic (BI_loop (Tf t1s t2s) es)] (vs ++ to_e_list es)]
  | rs_if_false :
      forall n tf e1s e2s,
        n = Wasm_int.int_zero i32m ->
        reduce_simple ([::AI_basic (BI_const (VAL_int32 n)); AI_basic (BI_if tf e1s e2s)]) [::AI_basic (BI_block tf e2s)]
  | rs_if_true :
      forall n tf e1s e2s,
        n <> Wasm_int.int_zero i32m ->
        reduce_simple ([::AI_basic (BI_const (VAL_int32 n)); AI_basic (BI_if tf e1s e2s)]) [::AI_basic (BI_block tf e1s)]
  | rs_label_const :
      forall n es vs,
        const_list vs ->
        reduce_simple [::AI_label n es vs] vs
  | rs_label_trap :
      forall n es,
        reduce_simple [::AI_label n es [::AI_trap]] [::AI_trap]
  | rs_br :
      forall n vs es i LI lh,
```

For every piece of semantics, want to produce at least 3 different formulations:

1. Latex formal rules
2. RST prose description
3. OCaml reference interpreter

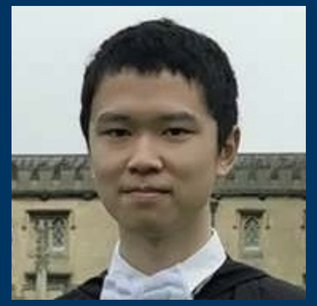
Also, tests…

And mechanizations

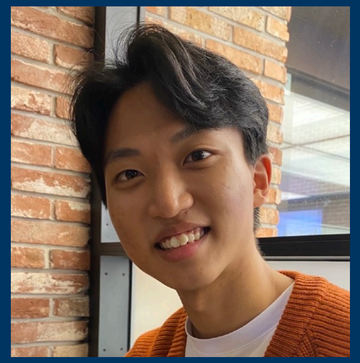
4. Coq inductive definitions
5. Isabelle definitions

We want a language for this!

Dongjun Youn, KAIST

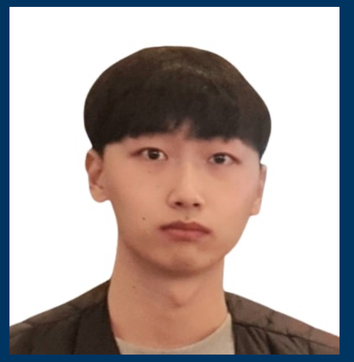


Xiaojia Rao, Imperial



Jaehyun Lee, KAIST

Henit Mandaliya, Imperial



Wonho Shin, KAIST

Joachim Breitner

Sukyoung Ryu, KAIST

Philippa Gardner, Imperial

Conrad Watt, Cambridge

Andreas Rossberg

Matija Pretnar, Ljubljana

Sam Lindley, Edinburgh

Presenting: **SpecTec** (working title)

A DSL for the Wasm Spec

Single source of truth

Easy to write, read, diff, and review; meta-level error checking

Specialised to our use case

Transformable into all the aforementioned representations

…sufficient for math rendering, natural language, and semantic modelling

One frontend – many backends

```
syntax instr hint(desc "instruction") =
    | UNREACHABLE
    | NOP
    | DROP
    | SELECT valtype?
    | BLOCK blocktype instr*
    | LOOP blocktype instr*
    | IF blocktype instr* ELSE instr*
    | BR labelidx
    | BR_IF labelidx
    | BR_TABLE labelidx* labelidx
    | CALL funcidx
    | CALL_INDIRECT tableidx functype
    | RETURN
    | CONST numtype c_numtype           hint(show %.CONST %)
    | UNOP numtype unop_numtype         hint(show %.%)
    | BINOP numtype binop_numtype       hint(show %.%)
    | TESTOP numtype testop_numtype     hint(show %.%)
    | RELOP numtype relop_numtype       hint(show %.%)
    | EXTEND numtype n                  hint(show %.EXTEND#%)
    | CVTOP numtype cvtop numtype sx?   hint(show %.%#_#%#_#%)
    | ...
```

instr	::=	unreachable
	|	nop
	|	drop
	|	select $valtype^?$
	|	block $blocktype$ $instr^*$
	|	loop $blocktype$ $instr^*$
	|	if $blocktype$ $instr^*$ else $instr^*$
	|	br $labelidx$
	|	br_if $labelidx$
	|	br_table $labelidx^*$ $labelidx$
	|	call $funcidx$
	|	call_indirect $tableidx$ $functype$
	|	return
	|	$numtype$.const $c\_numtype$
	|	$numtype.unop\_numtype$
	|	$numtype.binop\_numtype$
	|	$numtype.testop\_numtype$
	|	$numtype.relop\_numtype$
	|	$numtype$.extend$n$
	|	$numtype.cvtop\_numtype\_sx^?$

```
relation Instr_ok: context |- instr : functype  hint(show "T")

rule Instr_ok/nop:
  C |- NOP : epsilon -> epsilon

rule Instr_ok/block:
  C |- BLOCK bt instr* : t_1* -> t_2*
  -- Blocktype_ok: C |- bt : t_1* -> t_2*
  -- InstrSeq_ok: C, LABEL t_2* |- instr* : t_1* -> t_2*

rule Instr_ok/loop:
  C |- LOOP bt instr* : t_1* -> t_2*
  -- Blocktype_ok: C |- bt : t_1* -> t_2*
  -- InstrSeq_ok: C, LABEL t_1* |- instr* : t_1* -> t_2

rule Instr_ok/br:
  C |- BR l : t_1* t* -> t_2*
  -- iff C.LABEL[l] = t*

rule Instr_ok/br_if:
  C |- BR_IF l : t* I32 -> t*
  -- iff C.LABEL[l] = t*

rule Instr_ok/br_table:
  C |- BR_TABLE l* l' : t_1* t* -> t_2*
  -- (Resulttype_sub: |- t* <: C.LABEL[l])*
```

$$\boxed{context \vdash instr : functype}$$

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \to \epsilon}\ [\text{T-NOP}]$$

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

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

$$\frac{C.\mathsf{label}[l] = t^*}{C \vdash \mathsf{br}\ l : t_1^*\ t^* \to t_2^*}\ [\text{T-BR}] \qquad \frac{C.\mathsf{label}[l] = t^*}{C \vdash \mathsf{br\_if}\ l : t^*\ \mathsf{i32} \to t^*}\ [\text{T-BR\_IF}]$$

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

```
relation Step_pure: config ~> config

rule Step_pure/nop:
  NOP  ~>  epsilon

rule Step_pure/block:
  val^k (BLOCK bt instr*)  ~>  (LABEL_n`{epsilon} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/br-zero:
  (LABEL_n`{instr'*} val'* val^n (BR 0) instr*)  ~>  val^n instr'*

rule Step_pure/br-succ:
  (LABEL_n`{instr'*} val* (BR $(l+1)) instr*)  ~>  val* (BR l)

rule Step_pure/br_if-true:
  (CONST I32 c) (BR_IF l)  ~>  (BR l)
  -- iff c =/= 0

rule Step_pure/br_if-false:
  (CONST I32 c) (BR_IF l)  ~>  epsilon
  -- iff c = 0

rule Step_pure/br_table-lt:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l*[i])
  -- iff i < |l*|

rule Step_pure/br_table-le:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l')
  -- iff i >= |l*|
```

$$instr^* \hookrightarrow instr^*$$

$$
\begin{array}{lcll}
\mathsf{nop} & \hookrightarrow & \epsilon & \\
val^k\ (\mathsf{block}\ bt\ instr^*) & \hookrightarrow & (\mathsf{label}_n\{\epsilon\}\ val^k\ instr^*) & \text{if } bt = t_1^k \to t_2^n \\
val^k\ (\mathsf{loop}\ bt\ instr^*) & \hookrightarrow & (\mathsf{label}_n\{\mathsf{loop}\ bt\ instr^*\}\ val^k\ instr^*) & \text{if } bt = t_1^k \to t_2^n \\
(\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ 0)\ instr^*) & \hookrightarrow & val^n\ instr'^* & \\
(\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l+1)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l) & \\
(\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } c \neq 0 \\
(\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & \text{if } c = 0 \\
(\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l^*[i]) & \text{if } i < |l^*| \\
(\mathsf{i32.const}\ i)\ (\mathsf{br\_table}\ l^*\ l') & \hookrightarrow & (\mathsf{br}\ l') & \text{if } i \geq |l^*|
\end{array}
$$

```
relation Step_pure: config ~> config

rule Step_pure/nop:
  NOP  ~>  epsilon

rule Step_pure/block:
  val^k (BLOCK bt instr*)  ~>  (LABEL_n`{epsilon} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/br-zero:
  (LABEL_n`{instr'*} val'* val^n (BR 0) instr*)  ~>  val^n instr'*

rule Step_pure/br-succ:
  (LABEL_n`{instr'*} val* (BR $(l+1)) instr*)  ~>  val* (BR l)

rule Step_pure/br_if-true:
  (CONST I32 c) (BR_IF l)  ~>  (BR l)
  -- iff c =/= 0

rule Step_pure/br_if-false:
  (CONST I32 c) (BR_IF l)  ~>  epsilon
  -- iff c = 0

rule Step_pure/br_table-lt:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l*[i])
  -- iff i < |l*|

rule Step_pure/br_table-le:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l')
  -- iff i >= |l*|
```

nop

1. Do nothing.

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

block bt $instr^*$

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

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

loop bt $instr^*$

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

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

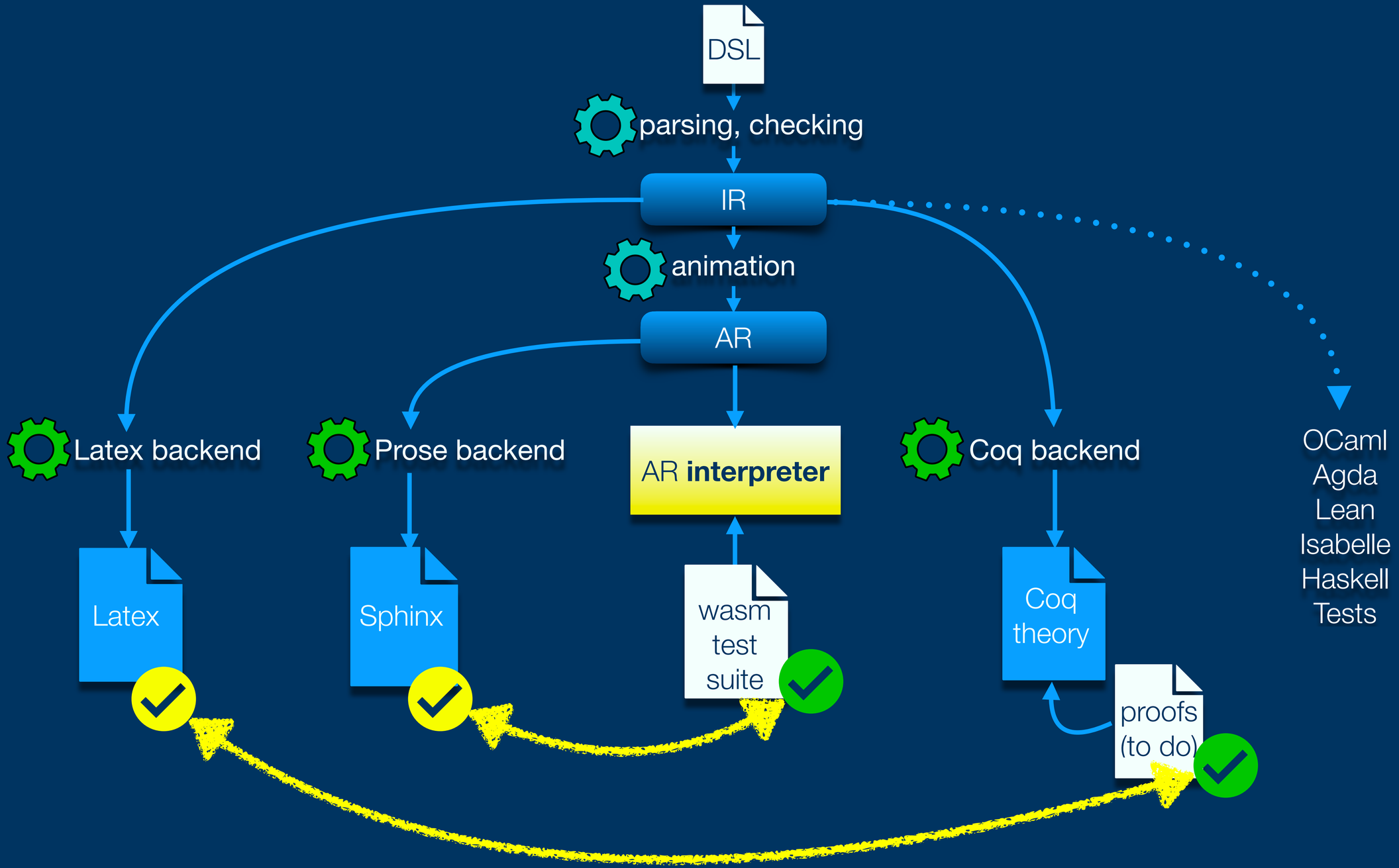
br x_0

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

$$[\text{E-BR-ZERO}](\mathsf{label}_n\{{instr'}^*\}\ {val'}^*\ {val}^n\ (\mathsf{br}\ 0)\ instr^*) \quad \hookrightarrow \quad {val}^n\ {instr'}^*$$
$$[\text{E-BR-SUCC}](\mathsf{label}_n\{{instr'}^*\}\ val^*\ (\mathsf{br}\ l+1)\ instr^*) \quad \hookrightarrow \quad val^*\ (\mathsf{br}\ l)$$

```
relation Step_pure: config ~> config

rule Step_pure/nop:
  NOP  ~>  epsilon

rule Step_pure/block:
  val^k (BLOCK bt instr*)  ~>  (LABEL_n`{epsilon} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

rule Step_pure/br-zero:
  (LABEL_n`{instr'*} val'* val^n (BR 0) instr*)  ~>  val^n instr'*

rule Step_pure/br-succ:
  (LABEL_n`{instr'*} val* (BR $(l+1)) instr*)  ~>  val* (BR l)

rule Step_pure/br_if-true:
  (CONST I32 c) (BR_IF l)  ~>  (BR l)
  -- iff c =/= 0

rule Step_pure/br_if-false:
  (CONST I32 c) (BR_IF l)  ~>  epsilon
  -- iff c = 0

rule Step_pure/br_table-lt:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l*[i])
  -- iff i < |l*|

rule Step_pure/br_table-le:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l')
  -- iff i >= |l*|
```

```
Inductive step_pure : ((list admininstr) * (list admininstr)) -> Prop :=
| step_pure__nop :
  step_pure (
    [admininstr__nop],
    []
  )
| step_pure__block (bt : blocktype) (instr : (list instr)) (k : nat) (n : n)
    (t_1 : (list valtype)) (t_2 : (list valtype)) (val : (list val)) :
  bt = (t_1, t_2) ->
  step_pure (
    (List.app (List.map admininstr_val val) [(admininstr__block (bt, instr))]),
    [(admininstr__label_ (n, [],
      (List.app (List.map admininstr_val val) (List.map admininstr_instr instr))))]
  )
| step_pure__loop (bt : blocktype) (instr : (list instr)) (k : nat) (n : n)
    (t_1 : (list valtype)) (t_2 : (list valtype)) (val : (list val)) :
  bt = (t_1, t_2) ->
  step_pure (
    (List.app (List.map admininstr_val val) [(admininstr__loop (bt, instr))]),
    [(admininstr__label_ (k, [(instr__loop (bt, instr))],
      (List.app (List.map admininstr_val val) (List.map admininstr_instr instr))))]
  )
| step_pure__br_zero (instr : (list instr)) (instr' : (list instr)) (n : n)
    (val : (list val)) (val' : (list val)):
  step_pure (
    [(admininstr__label_ (n, instr',
      (List.app (List.map admininstr_val val')
        (List.app (List.map admininstr_val val)
          (List.app [(admininstr__br 0)] (List.map admininstr_instr instr))))))],
    (List.app (List.map admininstr_val val) (List.map admininstr_instr instr'))
  )
| step_pure__br_succ (instr : (list instr)) (instr' : (list instr)) (l : labelidx) (n : n)
    (val : (list val)) :
  step_pure (
    [(admininstr__label_ (n, instr',
      (List.app (List.map admininstr_val val)
        (List.app [(admininstr__br l + 1)] (List.map admininstr_instr instr)))))],
    (List.app (List.map admininstr_val val) [((*case_exp*)admininstr__br l)])
  )
| step_pure__br_if_true (c : c_numtype) (l : labelidx) :
  c <> 0 ->
  step_pure (
    [(admininstr__const (numtype__i32, c)); (admininstr__br_if l)],
    [(admininstr__br l)]
  )
| step_pure__br_if_false (c : c_numtype) (l : labelidx) :
  c = 0 ->
  step_pure (
    [(admininstr__const (numtype__i32, c)); (admininstr__br_if l)],
    []
  )
| step_pure__br_table_lt (i : nat) (l : (list labelidx)) (l' : labelidx) :
  i < (List.length l) ->
  step_pure (
    [(admininstr__const (numtype__i32, i)); (admininstr__br_table (l, l'))],
    [(admininstr__br (list_get l i))]
  )
| step_pure__br_table_ge (i : nat) (l : (list labelidx)) (l' : labelidx) :
  i >= (List.length l) ->
  step_pure (
    [(admininstr__const (numtype__i32, i)); (admininstr__br_table (l, l'))],
    [(admininstr__br l')]
  )
```


DSL
parsing, checking
IR
animation
AR
Latex backend
Prose backend
AR interpreter
Coq backend
OCaml
Agda
Lean
Isabelle
Haskell
Tests
Latex
Sphinx
wasm
test
suite
Coq
theory
proofs
(to do)

Status

Reduction and validation

Can handle Wasm 2.0 minus SIMD

Numeric primitives still manually implemented

Prose interpreter passes 100% of test suite!

Incremental approach is possible

Also want to generate binary/text grammars later

Possibly reference interpreter and tests

Summary

Should substantially simplify spec authoring

…less redundant work, more diagnostics
…much easier code code reviews
…avoid error-prone and laborious prose

Should substantially improve correctness

…single source of truth
…more sanity checking
…can verify both formal and prose spec

Should help mechanisations to keep up

We could also auto-generate additional tests

Would be a significant advance in the state of the art!

Outtakes

Why now?

We didn’t know how to pull this off 5 years ago

…neither in terms of technology nor in terms of resources nor in terms of needs

Now all the stars have aligned:

Sukyoung Ryu and her team at KAIST have the necessary expertise
in converting between prose and semantics

…SAFE/JSTAR tool for analysing the prose of the EcmaScript spec [ASE 2021]

…the other direction should be much easier!

And they are eager to throw themselves on the Wasm spec now :)

We have a much better idea of what we want and need

Why not Ott/Lem?

Too general, too complex

…want something that's tuned for our use case
…and can quickly be taken up by average proposal writers

Cannot do everything we need

…various aspects of our notation, layout agnosticism
…sufficient domain-specific knowledge
…adjust, change, and extend it quickly and liberally

Unclear long-term fate

…can only afford dependencies that will be available and stable for 20+ years
…would we be able to maintain it?

A DSL specialised to our use case

solving a much simpler problem

can hence solve it faster and with better fit

can be special-cased, tweaked, extended

…and maintained by ourselves

Frontend

Disambiguation of notational overloading and user-defined mixfix notation

Type checking of rules and definitions

Dimension checking for meta variables (x^* etc)

Dependency and recursion analysis

Binder inference

Lowering into a more rigid “IL” that makes all that explicit

Latex Backend

Takes EL and generates Latex

Preprocessor that can “splice” formulas or definitions into a (set of) text files

…generic, but configurations for Latex and reStructured Text (markdown)

“Use” checker (all defs have been included exactly once in the target files)

Prose Backend

Takes EL and generates markdown (with embedded Latex)

Prototype only a week old, still overfitted to a limited set of rules

Mechanisation Backends

Work on Coq and Agda has only started :)

Technical Questions to Tackle

How similar is the problem of prose generation and generating the interpreter?

…perhaps related to proof animation [e.g. Berghofer et al. TPHOL 2009]

How to generically handle the subgrammar of values (injection?, subtyping?)

How to deal with evaluation contexts

…spec makes use of indexed contexts for branches (and handlers)

…notoriously difficult to express directly with inductive types

many more…