SpecTec: A DSL for the Wasm spec

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

2023/10/11

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 $(\mathsf{br}\ l_N)$.

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

```
1191
1192  .. _exec-table.set:
1193
1194  :math:`\TABLESET~x`
1195  ...................
1196
1197  1. Let :math:`F` be the :ref:`current <exec-notation-textual>` :ref:`frame <syntax-frame>`.
1198
1199  2. Assert: due to :ref:`validation <valid-table.set>`, :math:`F.\AMODULE.\MITABLES[x]` exists.
1200
1201  3. Let :math:`a` be the :ref:`table address <syntax-tableaddr>` :math:`F.\AMODULE.\MITABLES[x]`
     .
1202
1203  4. Assert: due to :ref:`validation <valid-table.set>`, :math:`S.\STABLES[a]` exists.
1204
1205  5. Let :math:`\X{tab}` be the :ref:`table instance <syntax-tableinst>` :math:`S.\STABLES[a]`.
1206
1207  6. Assert: due to :ref:`validation <valid-table.set>`, a :ref:`reference value <syntax-ref>`
     is on the top of the stack.
1208
1209  7. Pop the value :math:`\val` from the stack.
1210
1211  8. Assert: due to :ref:`validation <valid-table.set>`, a value of :ref:`value type <
     syntax-valtype>` |I32| is on the top of the stack.
1212
1213  9. Pop the value :math:`\I32.\CONST~i` from the stack.
1214
1215  10. If :math:`i` is not smaller than the length of :math:`\X{tab}.\TIELEM`, then:
1216
1217      a. Trap.
1218
1219  11. Replace the element :math:`\X{tab}.\TIELEM[i]` with :math:`\val`.
1220
1221  .. math::
1222     ~\\[-1ex]
1223     \begin{array}{l}
1224     \begin{array}{lcl@{\qquad}l}
1225     S; F; (\I32.\CONST~i)~\val~(\TABLESET~x) &\stepto& S'; F; \epsilon
1226     \end{array}
1227     \\ \qquad
1228       (\iff S' = S \with \STABLES[F.\AMODULE.\MITABLES[x]].\TIELEM[i] = \val) \\
1229     \begin{array}{lcl@{\qquad}l}
1230     S; F; (\I32.\CONST~i)~\val~(\TABLESET~x) &\stepto& S; F; \TRAP
1231     \end{array}
1232     \\ \qquad
1233       (\otherwise) \\
1234     \end{array}
1235
```

Maintenance

Duplicate work to extend *both* formal and prose rules

Writing the prose is *extremely* laborious

Both make for nightmarish code reviews

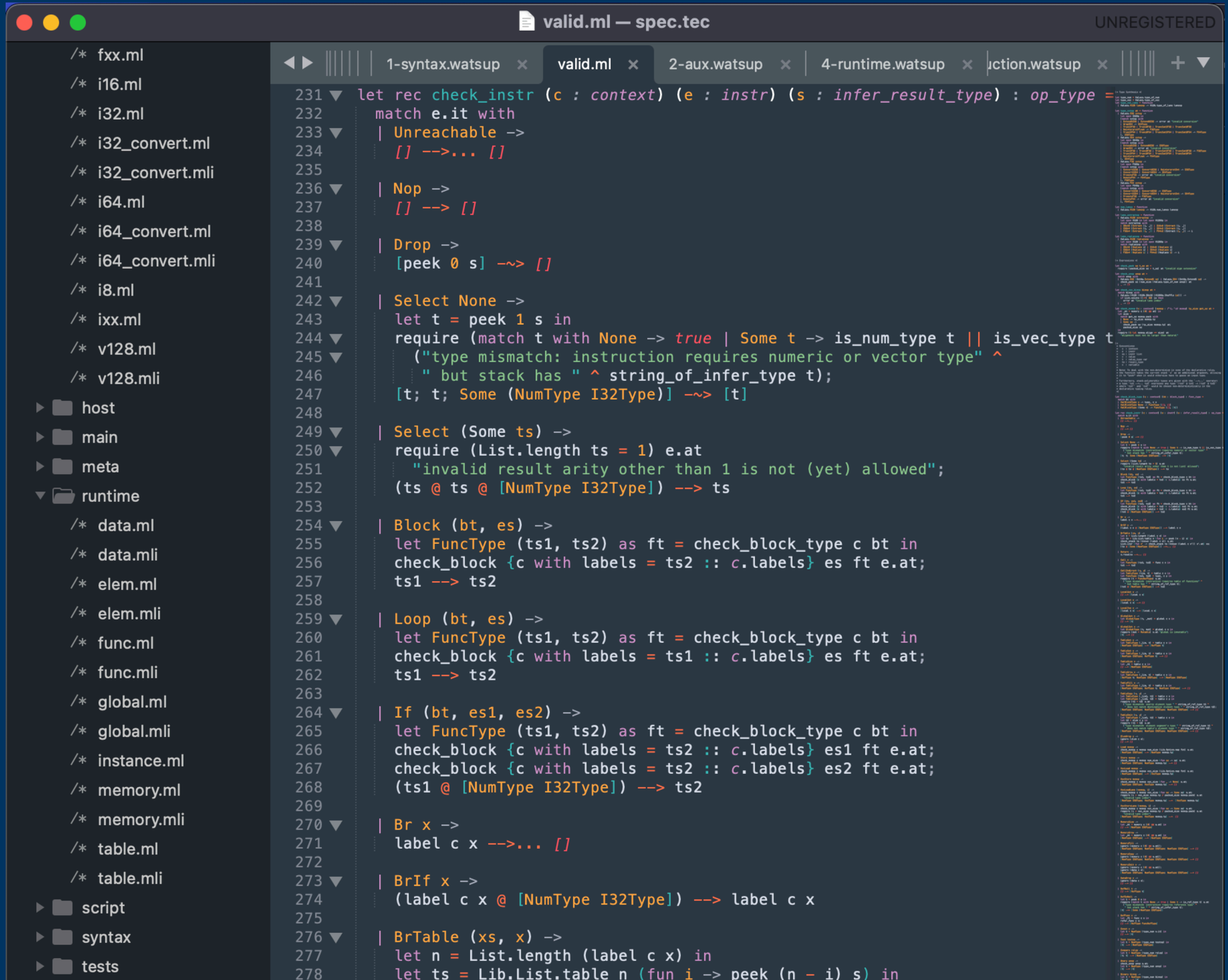
- …Latex math is a write-only language
- …reStructuredText is verbose and diff-unfriendly (tables!)
- …repetitive prose puts reviewer to sleep

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

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…

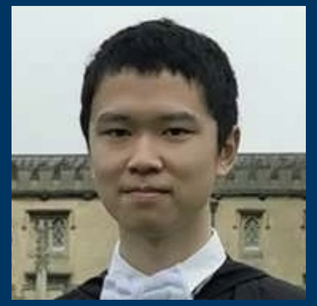
And mechanizations

4. Coq inductive definitions
5. Isabelle definitions

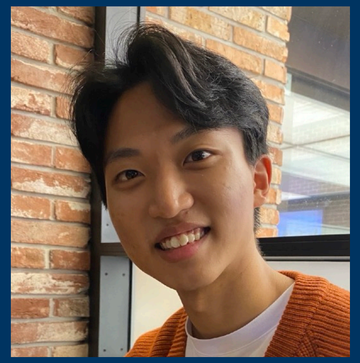
Wasm SpecTec

Wasm SpecTec

Dongjun Youn, KAIST

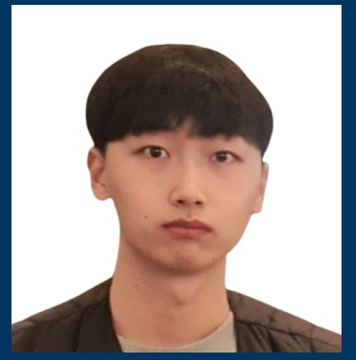


Xiaojia Rao, Imperial



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

```
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*
  -- Instrs_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*
  -- Instrs_ok:    C, LABEL t_1* |- instr* : t_1* -> t_2

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

rule Instr_ok/br_if:
  C |- BR_IF l : t* I32 -> t*
  -- if 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*)
  -- if bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- if 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)
  -- if c =/= 0

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

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

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

$$\boxed{instr^* \hookrightarrow instr^*}$$

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

```
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*)
  -- if bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- if 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)
  -- if c =/= 0

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

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

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

nop

1. Do nothing.

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

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 $instr^*$.

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

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.

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

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.

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

```
grammar Binstr/control: instr =
  | 0x00                                    => UNREACHABLE
  | 0x01                                    => NOP
  | 0x02 bt:Bblocktype (in:Binstr)* 0x0B    => BLOCK bt in*
  | 0x03 bt:Bblocktype (in:Binstr)* 0x0B    => LOOP bt in*
  | 0x04 bt:Bblocktype (in:Binstr)* 0x0B    => IF bt in* epsilon
  | 0x04 bt:Bblocktype (in_1:Binstr)* 0x05 (in_2:Binstr)* 0x0B => IF bt in_1* in_2*
  | 0x0C l:Blabelidx                        => BR l
  | 0x0D l:Blabelidx                        => BR_IF l
  | 0x0E l*:Bvec(Blabelidx) l_N:Blabelidx => BR_TABLE l* l_N
  | 0x0F                                    => RETURN
  | 0x10 x:Bfuncidx                         => CALL x
  | 0x11 y:Btypeidx x:Btableidx             => CALL_INDIRECT x y
  | ...
```

```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

Heap Types
~~~~~~~~~~

Concrete :ref:`Heap types <syntax-heaptype>` are only valid when
the :ref:`type index <syntax-typeidx>` is.

:math:`\absheaptype`
....................

* The heap type is valid.

.. math::
   \frac{
   }{
     C \vdashheaptype \absheaptype \ok
   }

:math:`\typeidx`
................

* The type :math:`C.\CTYPES[\typeidx]` must be defined in the context.

* Then the heap type is valid.

.. math::
   \frac{
     C.\CTYPES[\typeidx] = \deftype
   }{
     C \vdashheaptype \typeidx \ok
   }


.. index:: reference type, heap type
   pair: validation; reference type
   single: abstract syntax; reference type
.. _valid-reftype:

Reference Types
~~~~~~~~~~~~~~~

:ref:`Reference types <syntax-reftype>` are valid when the referenced
:ref:`heap type <syntax-heaptype>` is.

:math:`\REF~\NULL^?~\heaptype`
..............................

* The heap type :math:`\heaptype` must be :ref:`valid <valid-heaptype>`.

* Then the reference type is valid.

.. math::
   \frac{
     C \vdashreftype \heaptype \ok
   }{
     C \vdashreftype \REF~\NULL^?~\heaptype \ok
   }
```

```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

Heap Types
~~~~~~~~~~

Concrete :ref:`Heap types <syntax-heaptype>` are only valid when
the :ref:`type index <syntax-typeidx>` is.

:math:`\absheaptype`
....................

${prose: Heaptype_ok/absheaptype}

${rule: Heaptype_ok/absheaptype}


:math:`\typeidx`
................

${prose: Heaptype_ok/typeidx}

${rule: Heaptype_ok/typeidx}


.. index:: reference type, heap type
   pair: validation; reference type
   single: abstract syntax; reference type
.. _valid-reftype:

Reference Types
~~~~~~~~~~~~~~~

:ref:`Reference types <syntax-reftype>` are valid when the referenced
:ref:`heap type <syntax-heaptype>` is.

:math:`\REF~\NULL^?~\heaptype`
..............................

${prose: Reftype_ok}


${rule: Reftype_ok}
```

```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

Heap Types
~~~~~~~~~~

Concrete :ref:`Heap types <syntax-heaptype>` are only valid when
the :ref:`type index <syntax-typeidx>` is.

:math:`\absheaptype`
....................

* The heap type is valid.

.. math::
   \frac{
   }{
     C \vdashheaptype \absheaptype \ok
   }

:math:`\typeidx`
................

* The type :math:`C.\CTYPES[\typeidx]` must be defined in the context.

* Then the heap type is valid.

.. math::
   \frac{
     C.\CTYPES[\typeidx] = \deftype
   }{
     C \vdashheaptype \typeidx \ok
   }


.. index:: reference type, heap type
   pair: validation; reference type
   single: abstract syntax; reference type
.. _valid-reftype:

Reference Types
~~~~~~~~~~~~~~~

:ref:`Reference types <syntax-reftype>` are valid when the referenced
:ref:`heap type <syntax-heaptype>` is.

:math:`\REF~\NULL^?~\heaptype`
..............................

* The heap type :math:`\heaptype` must be :ref:`valid <valid-heaptype>`.

* Then the reference type is valid.

.. math::
   \frac{
     C \vdashreftype \heaptype \ok
   }{
     C \vdashreftype \REF~\NULL^?~\heaptype \ok
   }
```

```
Heap Types
~~~~~~~~~~

Concrete :ref:`Heap types <syntax-heaptype>` are only valid when
the :ref:`type index <syntax-typeidx>` is.



$${prose: Heaptype_ok/absheaptype}

$${rule: Heaptype_ok/absheaptype}



$${prose: Heaptype_ok/typeidx}

$${rule: Heaptype_ok/typeidx}



Reference Types
~~~~~~~~~~~~~~~

:ref:`Reference types <syntax-reftype>` are valid when the referenced
:ref:`heap type <syntax-heaptype>` is.



$${prose: Reftype_ok}


$${rule: Reftype_ok}
```

Status

Reduction and validation

Decoding and parsing in the works

Can handle Wasm 2.0 minus SIMD (WIP)

Most of Wasm 3.0 prototyped as well

Numeric primitives still manually implemented

Prose interpreter passes 100% of 2.0\SIMD test suite

Summary

Single source of truth

simplify spec authoring

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

improve correctness

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

help mechanisations to keep up

interpreter could analyse coverage of test suite

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

Next Steps?

Considerations…

Can transition incrementally

…e.g., start with Latex & validation

Transition is reversable

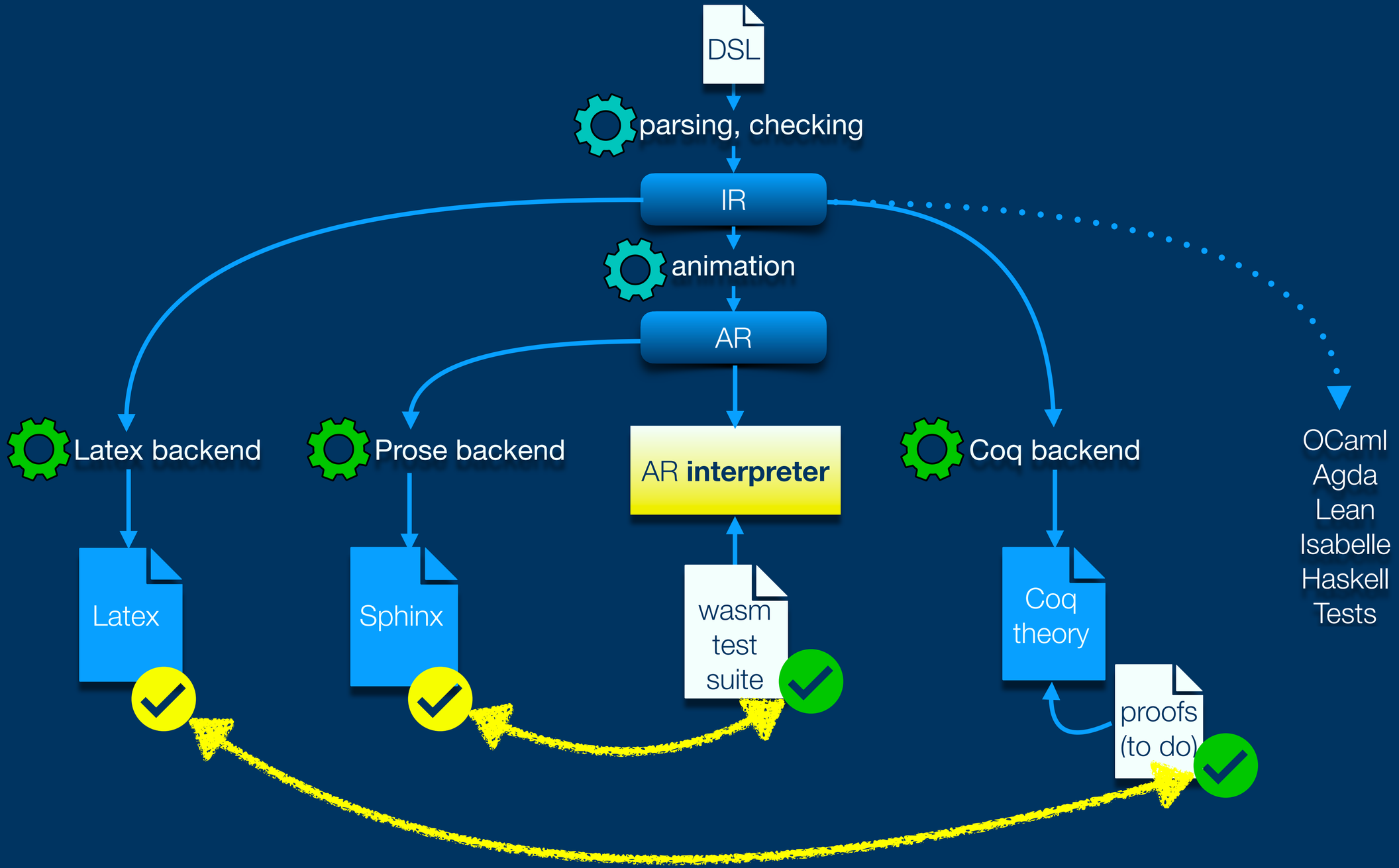
…can always fall back to manually evolving generated rst snapshot

Maintainability risk for tool chain

…non-trivial language changes may require extensions

Wasm 3 could be used as a litmus test?

How to Proceed?


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)

Outtakes

```
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*)
  -- if bt = t_1^k -> t_2^n

rule Step_pure/loop:
  val^k (LOOP bt instr*)  ~>  (LABEL_n`{LOOP bt instr*} val^k instr*)
  -- if 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)
  -- if c =/= 0

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

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

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

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

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…