SpecTec

Update and Announcement

Andreas Rossberg

with Dongjun Youn, Wonho Shin, Jaehyun Lee, Suhyeon Ryu, Hyunhee Kang, Sukyoung Ryu,
Alan Liang, Sam Lindley, Matija Pretnar, Joachim Breitner,
Rao Xiaojia, Diego Cupello, Philippa Gardner, Conrad Watt

Wasm CG 2025/02/13

Spec authoring today

Sphinx restructuredText with embeded Latex

Verbose, laborious, error-prone, terrible for code reviews

No error-checking or macro facilities in Sphinx

Duplicate work to manually write *both* formal and prose rules

+ reference interpreter

+ tests…

+ mechanisations…

a DSL for authoring the Wasm spec

select $(t^*)^?$

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

$$val_1\ val_2\ (\text{i32.const}\ c)\ (\text{select}\ t^?) \hookrightarrow val_1 \quad (\text{if}\ c \neq 0)$$
$$val_1\ val_2\ (\text{i32.const}\ c)\ (\text{select}\ t^?) \hookrightarrow val_2 \quad (\text{if}\ c = 0)$$

execution.watsup

```
rule Step_pure/select-true:
  val_1 val_2 (CONST I32 c) (SELECT t*?)  ~>  val_1
  -- if c =/= 0

rule Step_pure/select-false:
  val_1 val_2 (CONST I32 c) (SELECT t*?)  ~>  val_2
  -- if c = 0
```

execution.rst

```
.. _exec-select:

:math:`\SELECT~(t^\ast)^?`
..........................

1. Assert: due to :ref:`validation <valid-select>`, a :ref:`value <
syntax-value>` of :ref:`value type <syntax-valtype>` |I32| is on
the top of the :ref:`stack <syntax-stack>`.

2. Pop the value :math:`\I32.\CONST~c` from the :ref:`stack <
syntax-stack>`.

3. Assert: due to :ref:`validation <valid-select>`, two more :ref:`
values <syntax-value>` (of the same :ref:`value type <
syntax-valtype>`) are on the top of the stack.

4. Pop the :ref:`value <syntax-value>` :math:`\val_2` from the :ref:
`stack <syntax-stack>`.

5. Pop the :ref:`value <syntax-value>` :math:`\val_1` from the :ref:
`stack <syntax-stack>`.

6. If :math:`c` is not :math:`0`, then:

   a. Push the :ref:`value <syntax-value>` :math:`\val_1` back to
   the :ref:`stack <syntax-stack>`.

7. Else:

   a. Push the :ref:`value <syntax-value>` :math:`\val_2` back to
   the :ref:`stack <syntax-stack>`.

.. math::
   \begin{array}{lcl@{\qquad}l}
   \val_1~\val_2~(\I32\K{.}\CONST~c)~(\SELECT~t^?) &\stepto& \val_1
     & (\iff c \neq 0) \\
   \val_1~\val_2~(\I32\K{.}\CONST~c)~(\SELECT~t^?) &\stepto& \val_2
     & (\iff c = 0) \\
   \end{array}
```

select $(t^*)^?$

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

$$val_1\ val_2\ (\text{i32.const}\ c)\ (\text{select}\ t^?) \hookrightarrow val_1 \quad (\text{if } c \neq 0)$$
$$val_1\ val_2\ (\text{i32.const}\ c)\ (\text{select}\ t^?) \hookrightarrow val_2 \quad (\text{if } c = 0)$$

execution.watsup

```
rule Step_pure/select-true:
  val_1 val_2 (CONST I32 c) (SELECT t*?)  ~>  val_1
  -- if c =/= 0

rule Step_pure/select-false:
  val_1 val_2 (CONST I32 c) (SELECT t*?)  ~>  val_2
  -- if c = 0
```

execution.rst

```
.. _exec-select:

$${rule-prose: exec/select}

$${rule: {Step_pure/select-*}}
```

```
.. _exec-select:

:math:`\SELECT~{({t^\ast})^?}`
..............................

1. Assert: Due to :ref:`validation <valid-select>`, a value of :ref:`number type <
syntax-numtype>` :math:`\I32` is on the top of the stack.

#. Pop the value :math:`(\I32{.}\CONST~c)` from the stack.

#. Assert: Due to :ref:`validation <valid-select>`, a value is on the top of the stack.

#. Pop the value :math:`{\val}_2` from the stack.

#. Assert: Due to :ref:`validation <valid-select>`, a value is on the top of the stack.

#. Pop the value :math:`{\val}_1` from the stack.

#. If :math:`c \neq 0`, then:

   a. Push the value :math:`{\val}_1` to the stack.

#. Else:

   a. Push the value :math:`{\val}_2` to the stack.


.. math::
   \begin{array}[t]{@{}l@{}rcl@{}l@{}}
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) & \stepto
     & {\val}_1 & \quad \mbox{if}~ c \neq 0 \\
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) & \stepto
     & {\val}_2 & \quad \mbox{if}~ c = 0 \\
   \end{array}
```

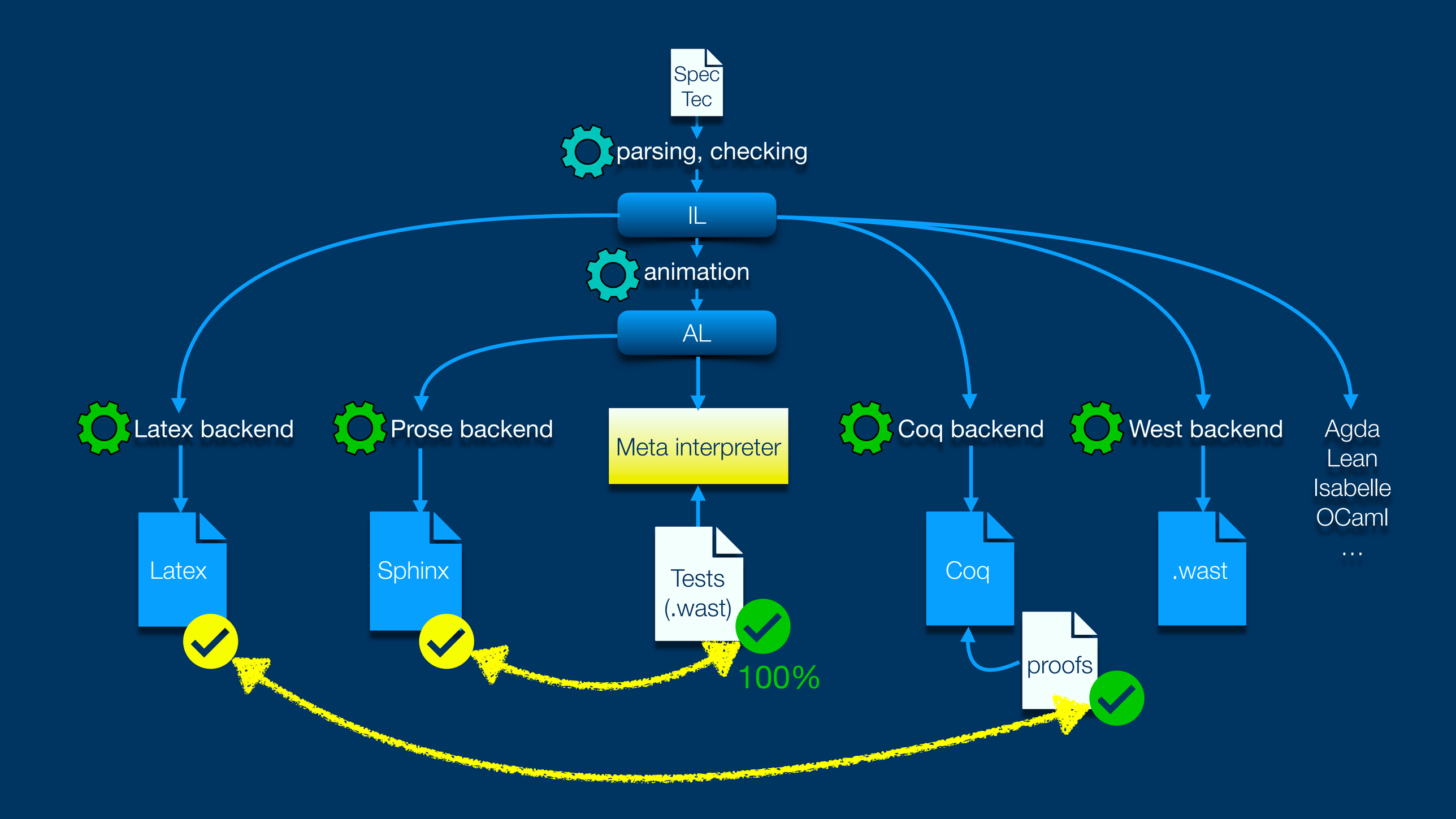
Spec authoring with SpecTec

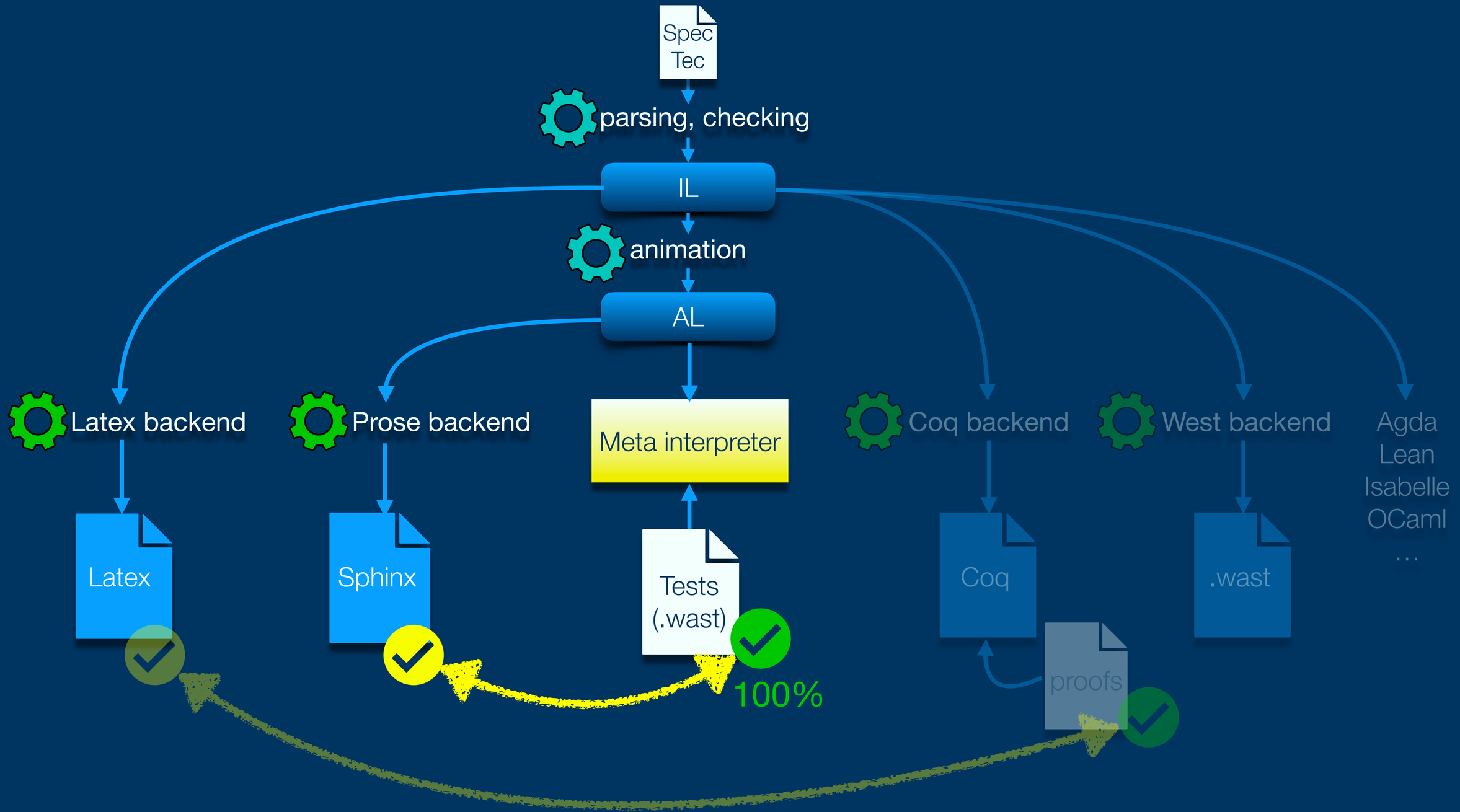
ASCII for the math rules (near WYSIWYG)

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

Single source of truth for auto-generating multiple artefacts:

- …math in Latex
- …prose in Sphinx
- …tests in Wast (ongoing work)
- …mechanised Coq and friends (ongoing work)


Spec
Tec
parsing, checking
IL
animation
AL
Latex backend
Prose backend
Meta interpreter
Coq backend
West backend
Agda
Lean
Isabelle
OCaml
...
Latex
Sphinx
Tests
(.wast)
Coq
.wast
proofs
100%


Spec
Tec
parsing, checking
IL
animation
AL
Latex backend
Prose backend
Meta interpreter
Coq backend
West backend
Agda
Lean
Isabelle
OCaml
...
Latex
Sphinx
Tests
(.wast)
Coq
.wast
proofs
100%

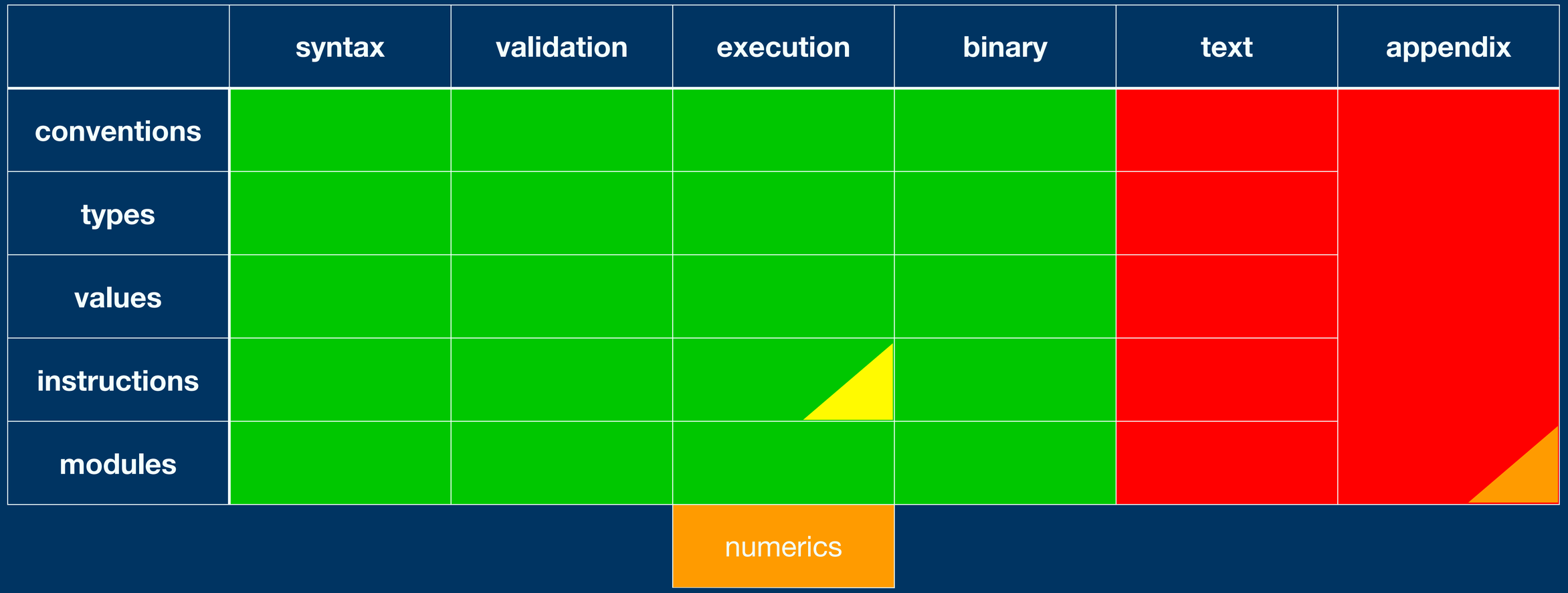
Status

1. building the tool
2. porting the formalism
3. converting the spec document

	syntax	validation	execution	binary	text	appendix
conventions						
types						
values						
instructions						
modules						
			numerics			

	syntax	validation	execution	binary	text	appendix
conventions						
types						
values						
instructions						
modules						
			numerics			

Wasm 2.0 + tail calls + multi memory + function references + garbage collection



Wasm 3.0 = Wasm 2.0 + tail calls + multi memory + function references + garbage collection
+ memory 64 + exception handling + extended constants + annotations + relaxed SIMD

Updates

finished converting primary chapters of the spec document

included all phase 4/5 proposals (modulo threads)

… in sync with wasm-3.0 branch; also did stack switching on the side

almost complete documentation

full generation of cross-references

many fixes and improvements

select $(t^*)^?$

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

$$
\begin{array}{lcll}
val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_1 & (\text{if } c \neq 0) \\
val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_2 & (\text{if } c = 0)
\end{array}
$$

select $(t^*)^?$

1. Assert: Due to validation, a value of number type i32 is on the top of the stack.
2. Pop the value (i32.const c) from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value val_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop the value val_1 from the stack.
7. If $c \neq 0$, then:
    a. Push the value val_1 to the stack.
8. Else:
    a. Push the value val_2 to the stack.

$$
\begin{array}{lcll}
val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ (t^*)^?) & \hookrightarrow & val_1 & \text{if } c \neq 0 \\
val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ (t^*)^?) & \hookrightarrow & val_2 & \text{if } c = 0
\end{array}
$$

Differences

No Use of Evaluation Contexts (for now)

br l — old spec

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.

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

Block Contexts ¶

In order to specify the reduction of branches, the following syntax of *block contexts* is defined, indexed by the count k of labels surrounding a *hole* [_] that marks the place where the next step of computation is taking place:

$$\begin{array}{lcl} B^k & ::= & val\ B^k \mid B^k\ instr \mid \mathsf{handler}_n\{catch^*\}\ B^k\ \mathsf{end} \mid C^k \\ C^0 & ::= & [\_] \\ C^{k+1} & ::= & \mathsf{label}_n\{instr^*\}\ B^k\ \mathsf{end} \end{array}$$

This definition allows to index active labels surrounding a branch or return instruction.

br l — new spec

1. If the first non-value entry of the stack is a label, then:
   a. Let L be the topmost label.
   b. Let n be the arity of L
   c. If $l = 0$, then:
      1. Assert: Due to validation, there are at least n values on the top of the stack.
      2. Pop the values val^n from the stack.
      3. Pop all values val'^* from the top of the stack.
      4. Pop the label L from the stack.
      5. Push the values val^n to the stack.
      6. Jump to the continuation of L.
   d. Else:
      1. Pop all values val^* from the top of the stack.
      2. If $l > 0$, then:
         a. Pop the label L from the stack.
         b. Push the values val^* to the stack.
         c. Execute the instruction $(\mathsf{br}\ l - 1)$.
2. Else if the first non-value entry of the stack is a handler, then:
   a. Pop all values val^* from the top of the stack.
   b. Pop the handler H from the stack.
   c. Push the values val^* to the stack.
   d. Execute the instruction $(\mathsf{br}\ l)$.

$$\begin{array}{rcll} (\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^n\ instr'^* & \text{if } l = 0 \\ (\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l - 1) & \text{if } l > 0 \\ (\mathsf{handler}_n\{catch^*\}\ val^*\ (\mathsf{br}\ l)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l) & \end{array}$$

More Precise Abstract Syntax

Numeric Instructions

old spec

Numeric instructions provide basic operations over numeric values of specific type. These operations closely match respective operations available in hardware.

$$
\begin{array}{rcl}
nn, mm & ::= & 32 \mid 64 \\
sx & ::= & \mathsf{u} \mid \mathsf{s} \\
instr & ::= & \mathsf{i}nn\mathsf{.const}\ unn \mid \mathsf{f}nn\mathsf{.const}\ fnn \\
 & \mid & \mathsf{i}nn\mathsf{.}iunop \mid \mathsf{f}nn\mathsf{.}funop \\
 & \mid & \mathsf{i}nn\mathsf{.}ibinop \mid \mathsf{f}nn\mathsf{.}fbinop \\
 & \mid & \mathsf{i}nn\mathsf{.}itestop \\
 & \mid & \mathsf{i}nn\mathsf{.}irelop \mid \mathsf{f}nn\mathsf{.}frelop \\
 & \mid & \mathsf{i}nn\mathsf{.extend8\_s} \mid \mathsf{i}nn\mathsf{.extend16\_s} \mid \mathsf{i64.extend32\_s} \\
 & \mid & \mathsf{i32.wrap\_i64} \mid \mathsf{i64.extend\_i32}\_sx \mid \mathsf{i}nn\mathsf{.trunc\_f}mm\_sx \\
 & \mid & \mathsf{i}nn\mathsf{.trunc\_sat\_f}mm\_sx \\
 & \mid & \mathsf{f32.demote\_f64} \mid \mathsf{f64.promote\_f32} \mid \mathsf{f}nn\mathsf{.convert\_i}mm\_sx \\
 & \mid & \mathsf{i}nn\mathsf{.reinterpret\_f}nn \mid \mathsf{f}nn\mathsf{.reinterpret\_i}nn \\
 & \mid & \ldots \\
iunop & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\
ibinop & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}\_sx \mid \mathsf{rem}\_sx \\
 & \mid & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr}\_sx \mid \mathsf{rotl} \mid \mathsf{rotr} \\
funop & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \\
fbinop & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \\
itestop & ::= & \mathsf{eqz} \\
irelop & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt}\_sx \mid \mathsf{gt}\_sx \mid \mathsf{le}\_sx \mid \mathsf{ge}\_sx \\
frelop & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge}
\end{array}
$$

Conventions

Occasionally, it is convenient to group operators together according to the following grammar shorthands:

$$
\begin{array}{rcl}
unop & ::= & iunop \mid funop \mid \mathsf{extend}N\mathsf{\_s} \\
binop & ::= & ibinop \mid fbinop \\
testop & ::= & itestop \\
relop & ::= & irelop \mid frelop \\
cvtop & ::= & \mathsf{wrap} \mid \mathsf{extend} \mid \mathsf{trunc} \mid \mathsf{trunc\_sat} \mid \mathsf{convert} \mid \mathsf{demote} \mid \mathsf{promote} \mid \mathsf{reinterpret}
\end{array}
$$

Numeric Instructions

new spec

Numeric instructions provide basic operations over numeric values of specific type. These operations closely match respective operations available in hardware.

$$
\begin{array}{rcll}
sz & ::= & 8 \mid 16 \mid 32 \mid 64 & \\
sx & ::= & \mathsf{u} \mid \mathsf{s} & \\
num_{\mathsf{i}N} & ::= & iN & \\
num_{\mathsf{f}N} & ::= & fN & \\
instr & ::= & \ldots & \\
 & \mid & numtype\mathsf{.const}\ num_{numtype} & \\
 & \mid & numtype.unop_{numtype} & \\
 & \mid & numtype.binop_{numtype} & \\
 & \mid & numtype.testop_{numtype} & \\
 & \mid & numtype.relop_{numtype} & \\
 & \mid & numtype_1.cvtop_{numtype_2,numtype_1}\_numtype_2 & \\
 & \mid & \ldots & \\
unop_{\mathsf{i}N} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \mid \mathsf{extend}sz\mathsf{\_s} & \text{if } sz < N \\
unop_{\mathsf{f}N} & ::= & \mathsf{abs} \mid \mathsf{neg} \mid \mathsf{sqrt} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} & \\
binop_{\mathsf{i}N} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}\_sx \mid \mathsf{rem}\_sx & \\
 & \mid & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr}\_sx \mid \mathsf{rotl} \mid \mathsf{rotr} & \\
binop_{\mathsf{f}N} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} & \\
testop_{\mathsf{i}N} & ::= & \mathsf{eqz} & \\
relop_{\mathsf{i}N} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt}\_sx \mid \mathsf{gt}\_sx \mid \mathsf{le}\_sx \mid \mathsf{ge}\_sx & \\
relop_{\mathsf{f}N} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} & \\
cvtop_{\mathsf{i}N_1,\mathsf{i}N_2} & ::= & \mathsf{extend}\_sx & \text{if } N_1 < N_2 \\
 & \mid & \mathsf{wrap} & \text{if } N_1 > N_2 \\
cvtop_{\mathsf{i}N_1,\mathsf{f}N_2} & ::= & \mathsf{convert}\_sx & \\
 & \mid & \mathsf{reinterpret} & \text{if } N_1 = N_2 \\
cvtop_{\mathsf{f}N_1,\mathsf{i}N_2} & ::= & \mathsf{trunc}\_sx & \\
 & \mid & \mathsf{trunc\_sat}\_sx & \\
 & \mid & \mathsf{reinterpret} & \text{if } N_1 = N_2 \\
cvtop_{\mathsf{f}N_1,\mathsf{f}N_2} & ::= & \mathsf{promote} & \text{if } N_1 < N_2 \\
 & \mid & \mathsf{demote} & \text{if } N_1 > N_2
\end{array}
$$

Notational Tweaks

old spec

- r with field $= A$ denotes the same record as r, except that the contents of the field component is replaced with A

new spec

- $r[.\text{field} = A]$ denotes the same record as r, except that the value of the field component is replaced with A.

Functions ¶

The funcs component of a module defines a vector of *functions* with the following structure:

$$\begin{array}{lcl} \mathit{func} & ::= & \{\text{type } \mathit{typeidx}, \text{locals } \mathit{vec}(\mathit{local}), \text{body } \mathit{expr}\} \\ \mathit{local} & ::= & \{\text{type } \mathit{valtype}\} \end{array}$$

Functions ¶

The *func* section of a module defines a list of *functions* with the following structure:

$$\begin{array}{lcl} \mathit{func} & ::= & \text{func } \mathit{typeidx}\ \mathit{local}^*\ \mathit{expr} \\ \mathit{local} & ::= & \text{local } \mathit{valtype} \end{array}$$

$$\frac{}{C \vdash t.\mathit{relop} : [t\ t] \to [\text{i32}]}$$

$$\frac{}{C \vdash nt.\mathit{relop}_{nt} : nt\ nt \to \text{i32}}$$

$$\frac{\text{expand}(C.\,\text{funcs}[x]) = \text{func}\ [t_1^*] \to [t_2^*]}{C \vdash \text{call}\ x : [t_1^*] \to [t_2^*]}$$

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

Current Limitations

user experience could still be improved

…especially wrt robustness and error reporting in backends

produced prose wording not always ideal (https://github.com/Wasm-DSL/spectec/issues/141)

some math not yet fully unpacked in prose, especially iterations (WIP)

prose and interpreter hard-code many choices that may break when spec changes

…often inevitable, e.g., must convert from reference interpreter's AST

…interpreter requires adjustment for structurally non-trivial spec extensions

…but hope to get rid of more of these over time, e.g., with hint system, meta-parsing

Other News

Prototype translation to Coq with soundness proof for Wasm 1.0

Meta-parser for interpreting SpecTec grammar definitions

Initial investigation of evaluation contexts as a SpecTec feature

SpecTec has been applied experimentally to the P4 network programming language

Test generation!

Generating Tests with West backend

Can fuzz tests guided by complete semantic knowledge

…understands syntax, typing, and execution; knows expected results

Can selectively generate micro tests for individual constructs

…by looking at typing rules

Will be able to systematically generate full combinatorial test matrix for individual features

…e.g., all interesting combinations of types and operands for a given instruction

Generating Tests with West backend

⇒ Much fewer tedious tests to write manually

⇒ Much better coverage

⇒ Generic: automatically supports test generation for future features

Next Steps

Poll from last June

“Adopt SpecTec (once it’s ready) as the toolchain for authoring the spec.”

We think “SpecTec MVP” is ready now, final polishing in progress.

Would like a poll on adoption at one of the next meetings.

Lots of possible “Post-MVP” improvements.

Why now is a good time

All-time low of outstanding proposals that have already entered the spec'ing phase

…minimises friction from switching tool chains.

Getting there was a pain! Need to maintain two very different variants of the spec

…not viable too much longer

Various fixes in the SpecTec version that haven’t made it back to the old spec

…hard to do or too much (redundant) work

Prerequisite for test generation with West

SLA

Continue to fix, improve, and extend the tool

Continue to incrementally convert the remaining parts of the spec

Available to maintain the tool for at least 2 years

- … react to Github issues within reasonable time frame
- … assist proposal authors where necessary
- … knowledge transfer

Invest into building more tooling

- … test generation backend
- … proof mechanisation backend

Post-MVP Features

Replace more hard-coded assumptions in backends with customisable hints

More generation for doc: macro definitions, xref anchors, index entries

More self-contained meta interpreter: meta decoding, validation, numerics

Another meta interpreter operating on reduction rules directly

Meta-theory of SpecTec (SpecTec in SpecTec?)

Repo: https://github.com/wasm-dsl/spectec

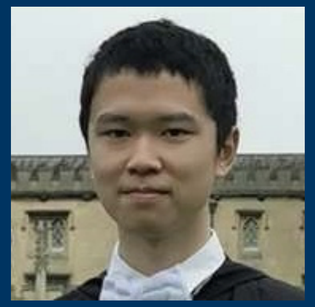
Docs: https://github.com/wasm-dsl/spectec/tree/main/spectec/doc

Spec: https://github.com/wasm-dsl/spectec/tree/main/spectec/spec

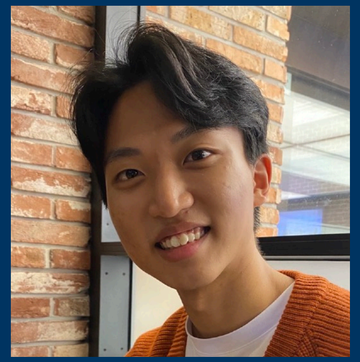
Render: https://wasm-dsl.github.io/spectec/

Outtakes

Dongjun Youn, KAIST

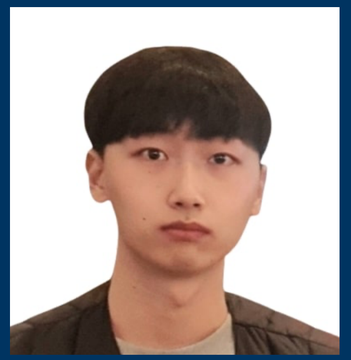


Xiaojia Rao, Imperial



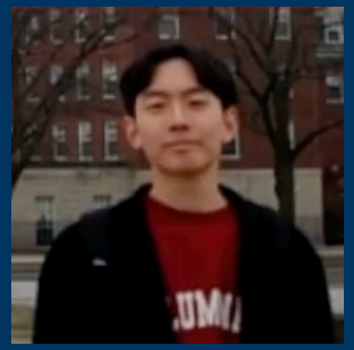
Jaehyun Lee, KAIST

Suhyeon Ryu, KAIST



Wonho Shin, KAIST

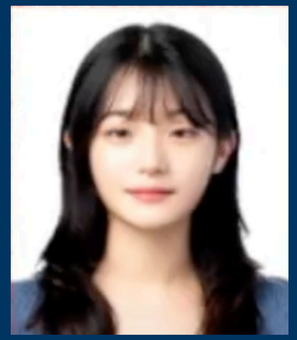
Joachim Breitner, Lean



Hoseong Lee, KAIST

Alan Liang, Edinburgh

Diego Cupello, Imperial



Hyunhee Kang, KAIST

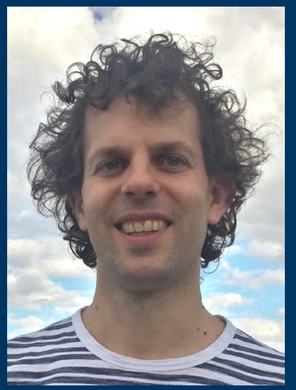
Sukyoung Ryu, KAIST

Philippa Gardner, Imperial

Conrad Watt, Nanyang

Andreas Rossberg

Matija Pretnar, Ljubljana



Sam Lindley, Edinburgh

Pros

prevents many possible spec bugs

… due to automation, error checking, testing with meta-interpreter, easier reviews

faster turn-around for spec’ing new features

more consistent style and layout

test generation and prover backends in the future

it’s more fun :)

non-trivial proposals like GC and stack-switching were expressible out of the box

Risks

1. The output produced by SpecTec is inferior to the hand-written document

   ⇒ we are “close enough”, and surpass it on some aspects (esp. correctness)

2. The user experience of the tool is not good enough

   ⇒ not as good yet as we’d like, maturation will require iteration and user feedback

3. Tool maintenance becomes a burden, and the CG may lack sufficient expertise

   ⇒ definite risk, some backends are not robust against cross-cutting spec changes

   “big feature” proposals will hopefully “saturate” functionality over time

   contingency plan applies, fall back to status quo

Meta Interpreter

Algorithmic prose is derived from user-defined declarative reduction rules

We can interpret “programs” in the prose AST, i.e., the Wasm spec’s execution rules

…and thereby indirectly run actual Wasm with actual Wasm spec

Passes 100% of applicable Wasm test suite

Current Limitations of Meta Interpreter

can do only execution, hooks into reference interpreter for everything else

…parsing, decoding, validation, numerics

still fragile on errors and structural spec changes

…esp. depends on various naming conventions being followed by user

a *lot* of hard-coded knowledge about various aspects of the spec

…often inevitable, e.g., must convert from reference interpreter's AST

requires adjustment for non-trivial spec extensions