Bringing the WebAssembly Standard up to Speed with SpecTec

HYTRADBOI
2025 Feb 28

Dongjun Youn (f52985@kaist.ac.kr)

This presentation builds upon work presented at PLDI'24.

WebAssembly (Wasm)

WebAssembly is a low-level code format
designed for efficient execution
especially on the web

```
int f(int x, int y) {
  return x + y;
}
```

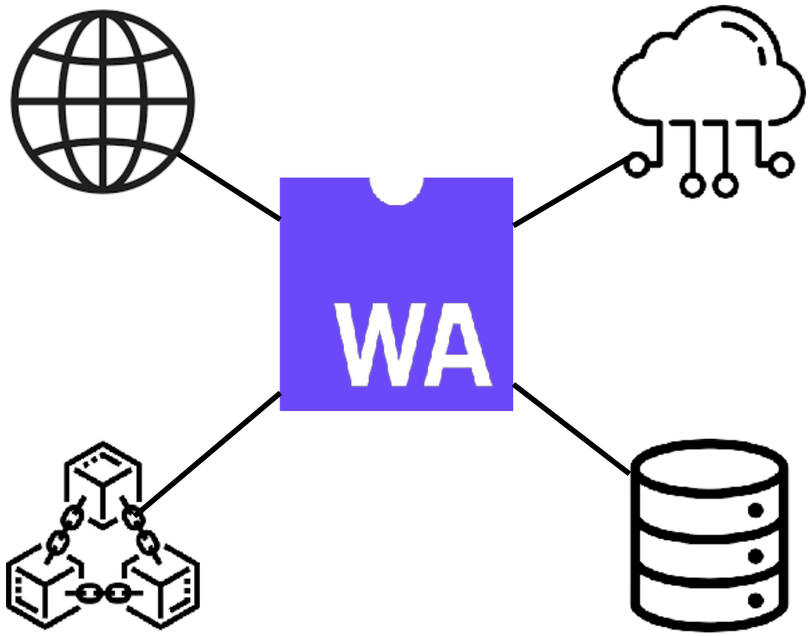
→

```
(func (export "f")
  (param i32 i32) (result i32)
  local.get 0 ;; [ x ]
  local.get 1 ;; [ x , y ]
  i32.add     ;; [ x + y ]
)
```

https://webassembly.github.io/spec/core/intro/introduction.html

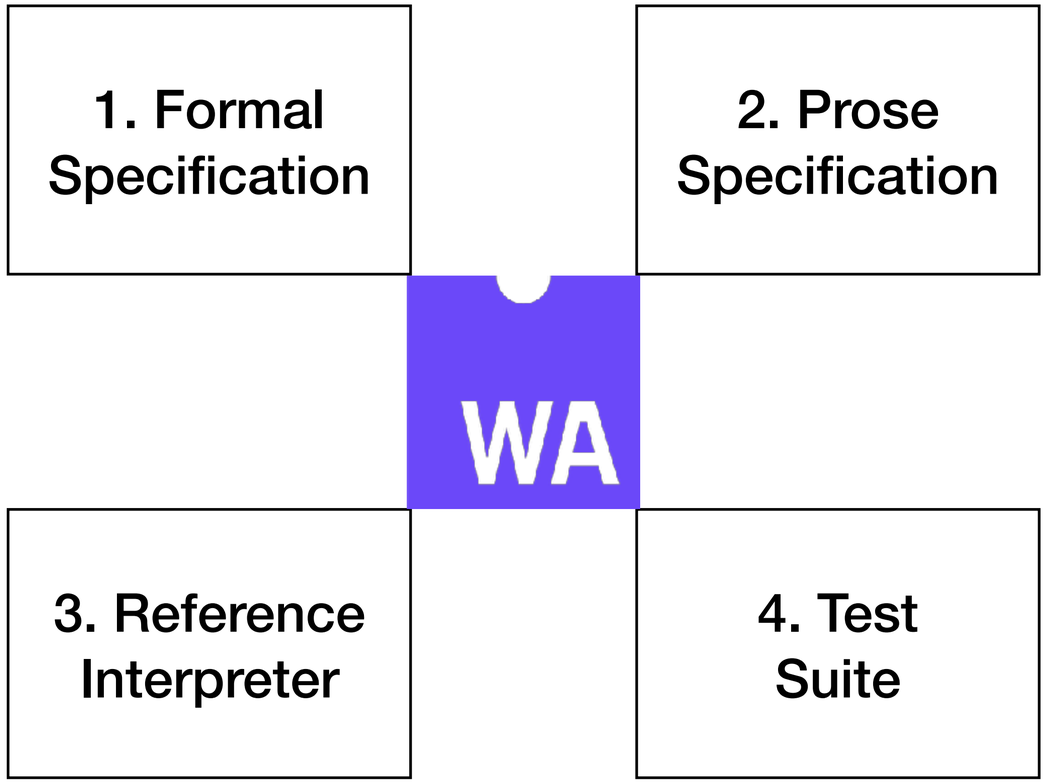
Wasm Beyond Web

Wasm is being used beyond the web,
including in cloud computing, blockchain, and database



Wasm Artifacts

To mitigate the risk of implementation divergence,
Wasm requires four artifacts for a new feature to be standardized



Wasm Artifacts

To mitigate the risk of implementation divergence,
Wasm requires four artifacts for a new feature to be standardized

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{relop} \quad \hookrightarrow \quad (\mathsf{i32.const}\ c)$$
$$(\text{if } c = \mathit{relop}_t(c_1, c_2))$$

<1. Formal Specification (LaTeX)>

t.relop

1. Assert: Two values of value type t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. Let c be the result of computing $\mathit{relop}_t(c_1, c_2)$.
5. Push the value $\mathsf{i32.const}\ c$ to the stack.

<2. Prose Specification (reST)>

Wasm Artifacts

To mitigate the risk of implementation divergence,
Wasm requires four artifacts for a new feature to be standardized

```
(match e', vs with
| Unreachable, vs ->
  vs, [Trapping "unreachable executed" @@ e.at]

| Nop, vs ->
  vs, []
```

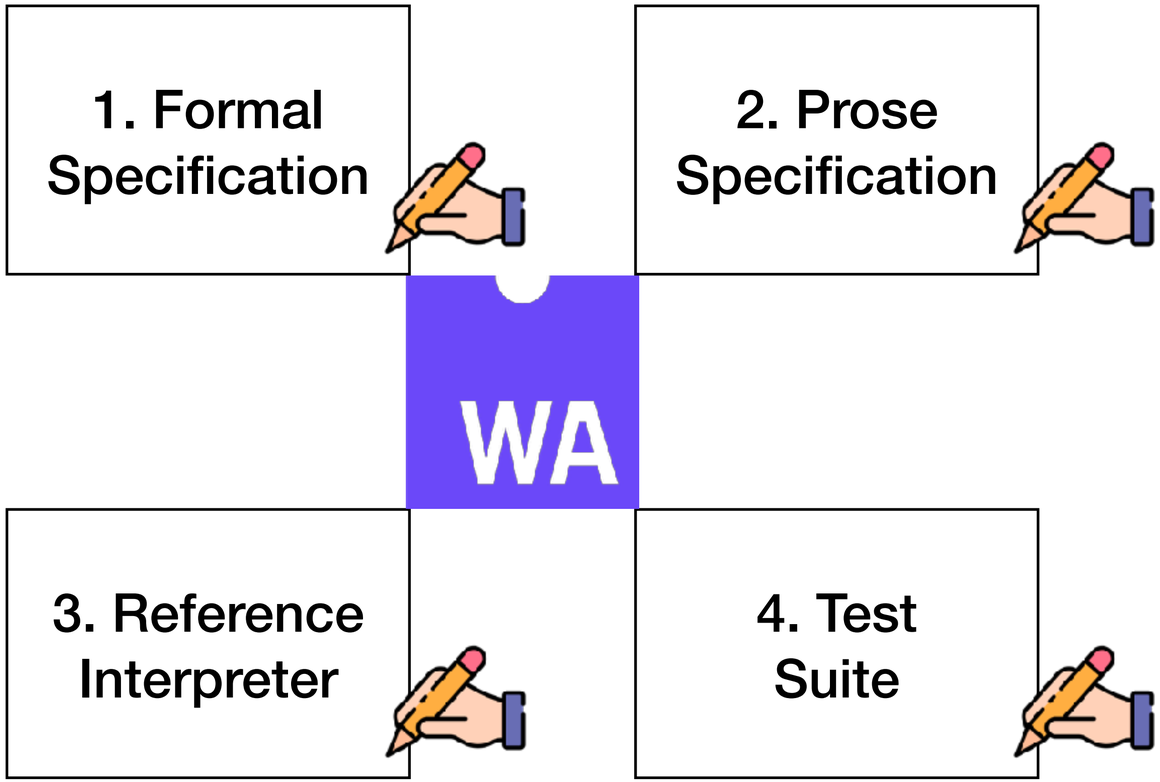
<3. Reference Interpreter (OCaml)>

```
(assert_return (invoke "fac" (i64.const 0)) (i64.const 1))
(assert_return (invoke "fac" (i64.const 1)) (i64.const 1))
(assert_return (invoke "fac" (i64.const 5)) (i64.const 120))
```

<4. Test Suite (wast)>

Wasm Artifacts - Limitation

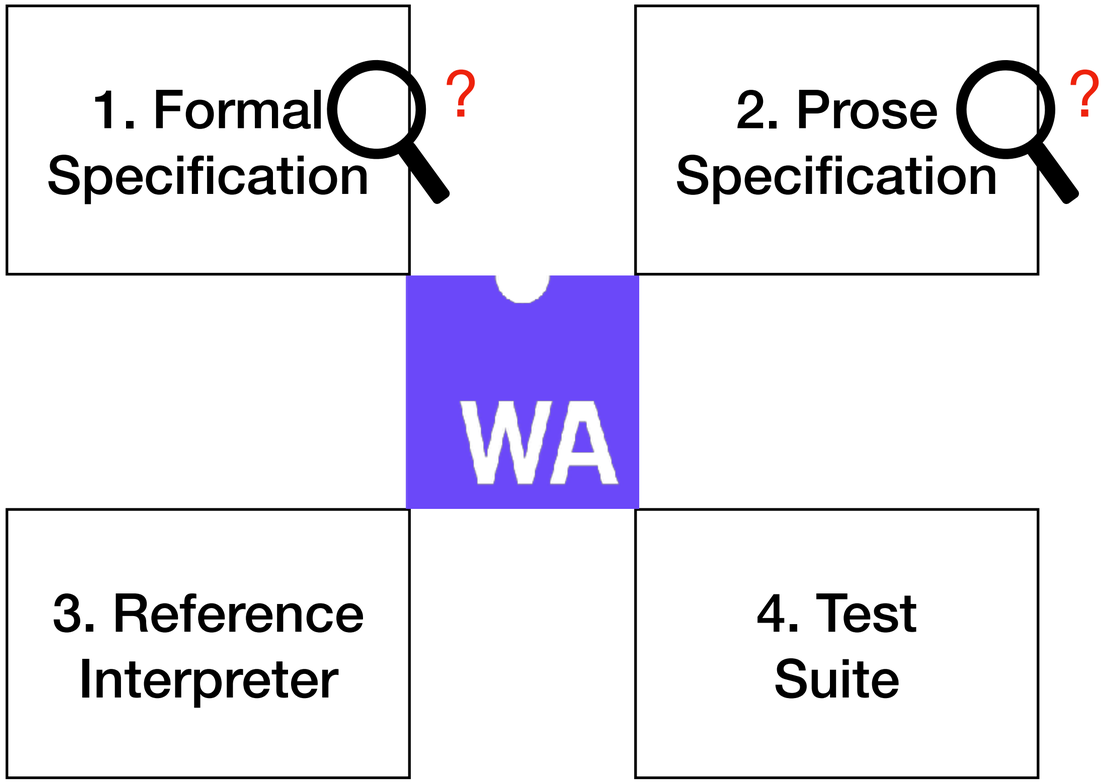
Laborious:
- Manual documentation & maintenance



Wasm Artifacts - Limitation

Laborious:
- Manual documentation & maintenance

Bad readability:
- Specs written in LaTeX / reST
- Code reviews are not user-friendly

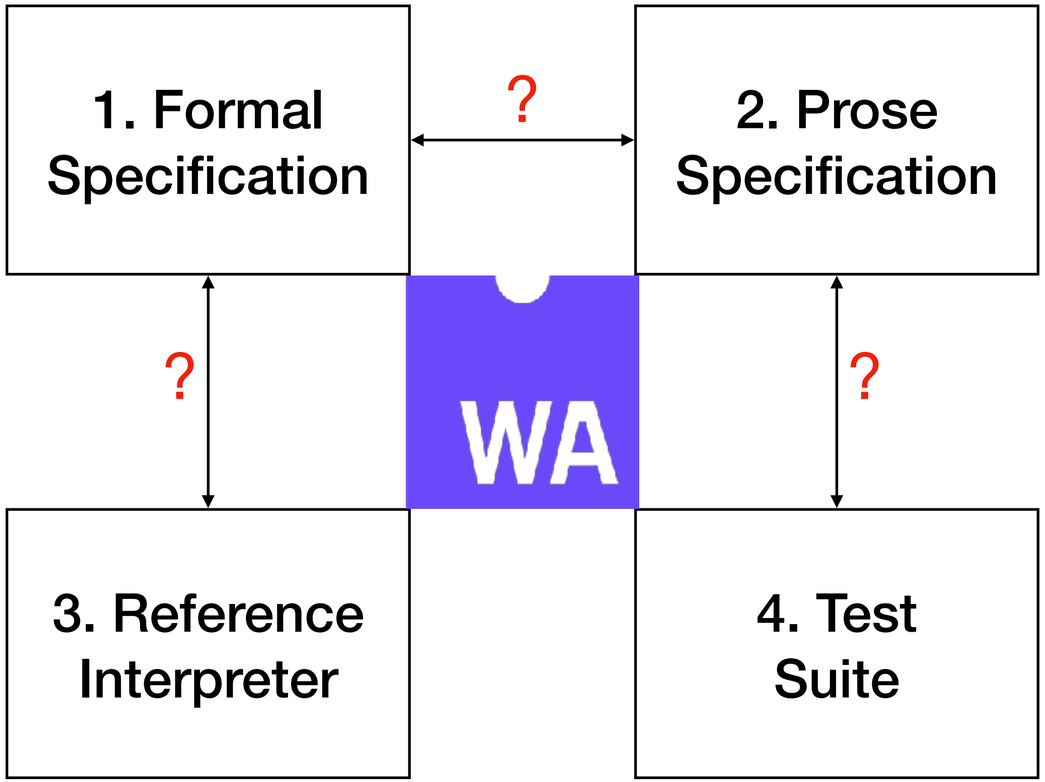


Wasm Artifacts - Limitation

Laborious:
- Manual documentation & maintenance

Bad readability:
- Specs written in LaTeX / reST
- Code reviews are not user-friendly

Error-Prone:
- Correctness / consistency relies on eye-ball correspondence



Wasm - Limitation

```
.. _exec-relop:

:math:`t\K{.}\relop`
....................

1. Assert: due to :ref:`validation <valid-relop>`, two values of :ref:`value type <syntax-
valtype>` :math:`t` are on the top of the stack.
2. Pop the value :math:`t.\CONST~c_2` from the stack.
3. Pop the value :math:`t.\CONST~c_1` from the stack.
4. Let :math:`c` be the result of computing :math:`\relopF_t(c_1, c_2)`.
5. Push the value :math:`\I32.\CONST~c` to the stack.

.. math::
   \begin{array}{lcl@{\qquad}l}
   (t\K{.}\CONST~c_1)~(t\K{.}\CONST~c_2)~t\K{.}\relop &\stepto& (\I32\K{.}\CONST~c)
     & (\iff c = \relopF_t(c_1,c_2)) \\
   \end{array}
```

… Can we do better?

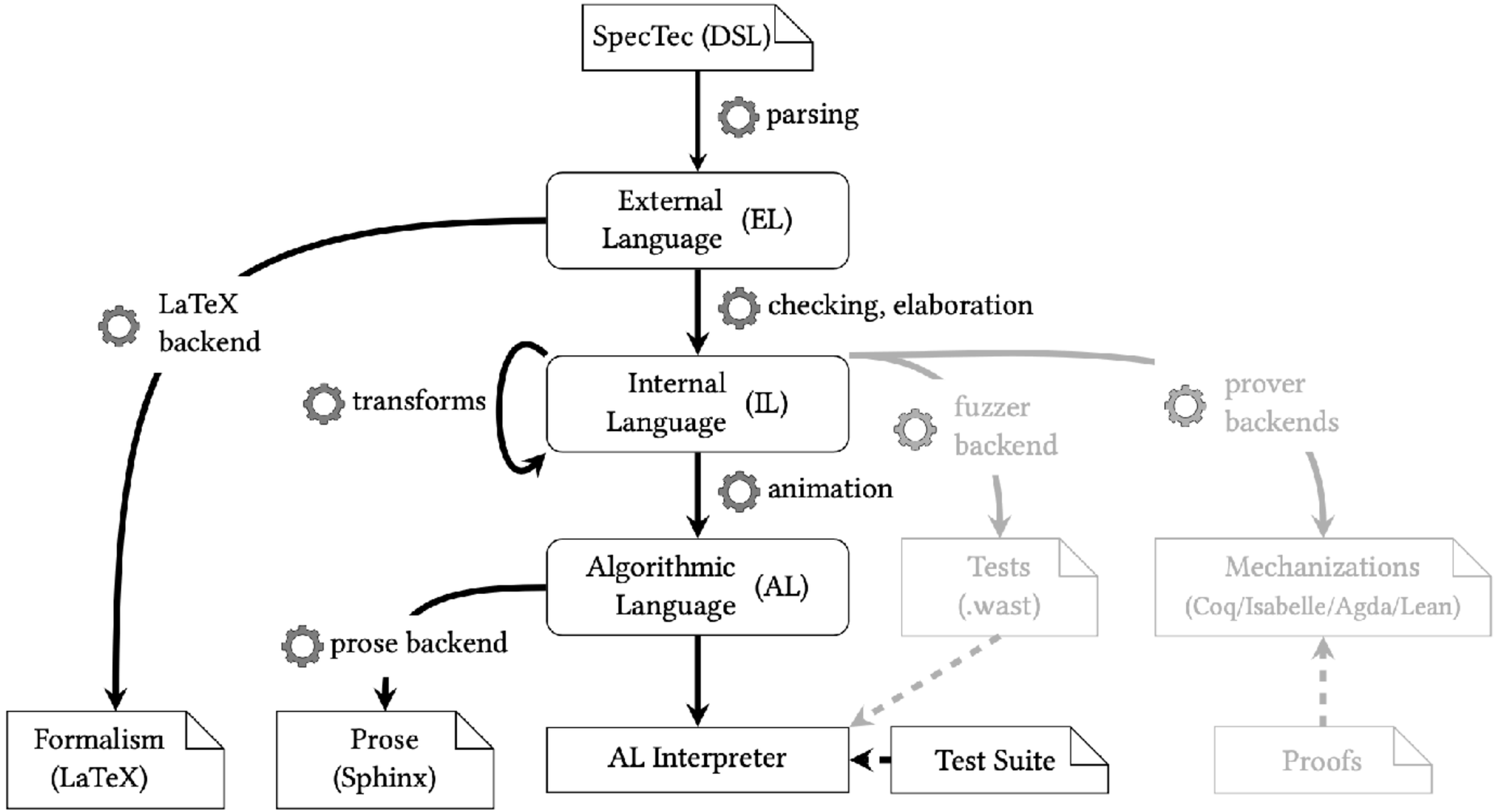
SpecTec (PLDI'24)

Framework for mechanizing specification of Wasm
- Embracing both declarative and algorithmic styles of semantics

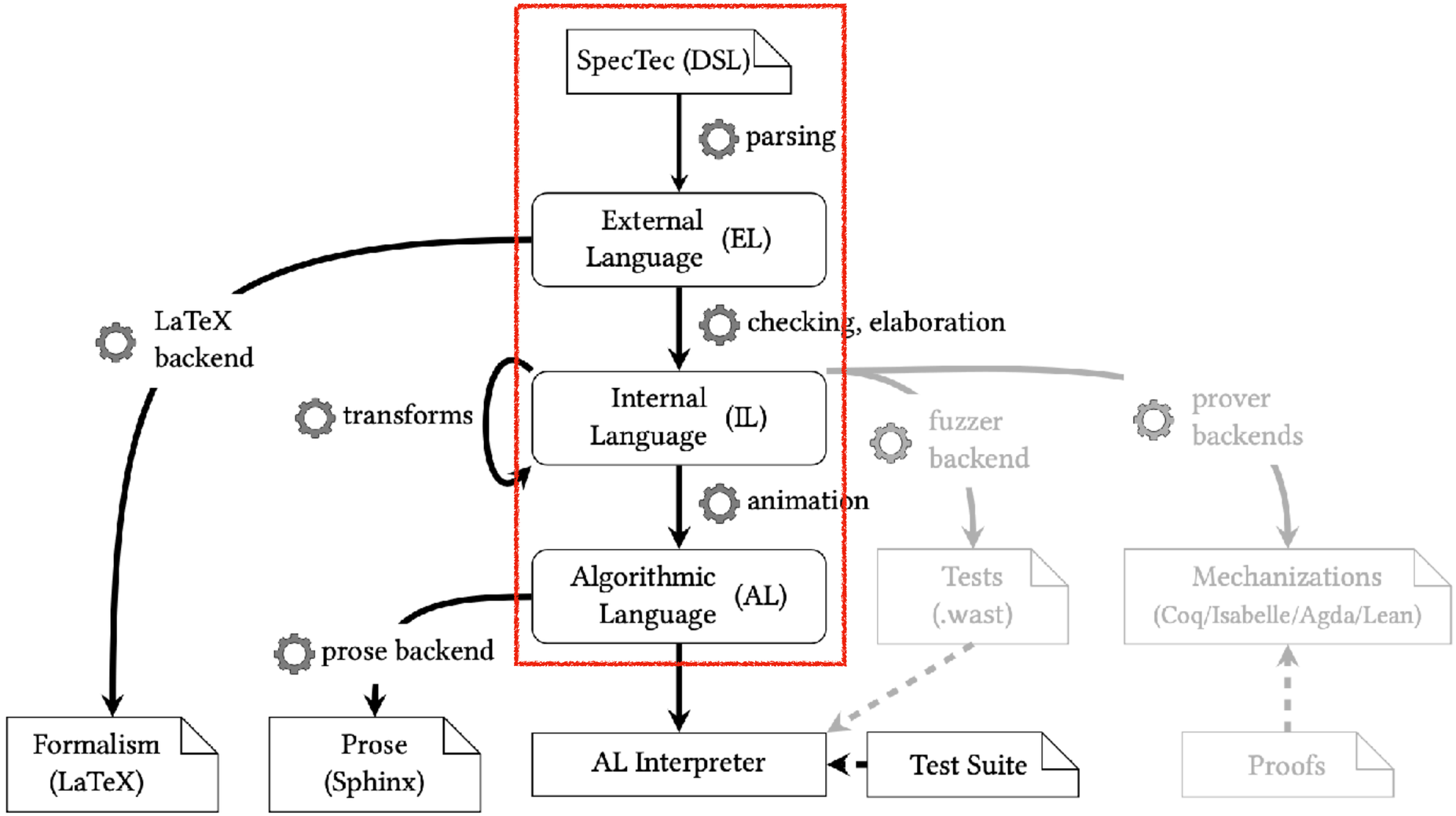
Provides a domain-specific language (DSL)
- Syntax, type system, and execution semantics of Wasm can be easily defined

A single source of truth from which various artifacts can be auto-generated
- LaTeX backend, Prose backend, Interpreter backend, …

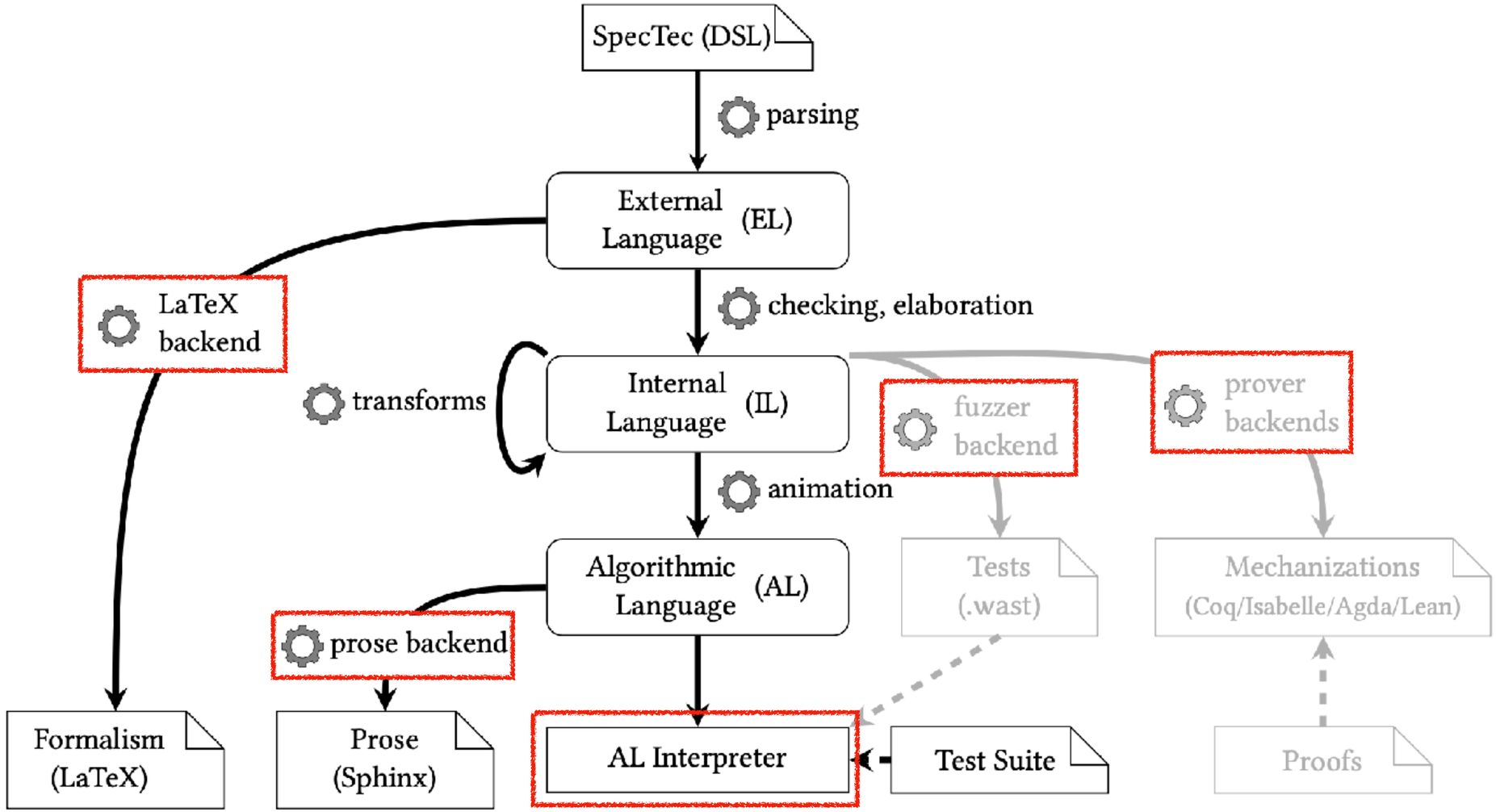
Overview


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

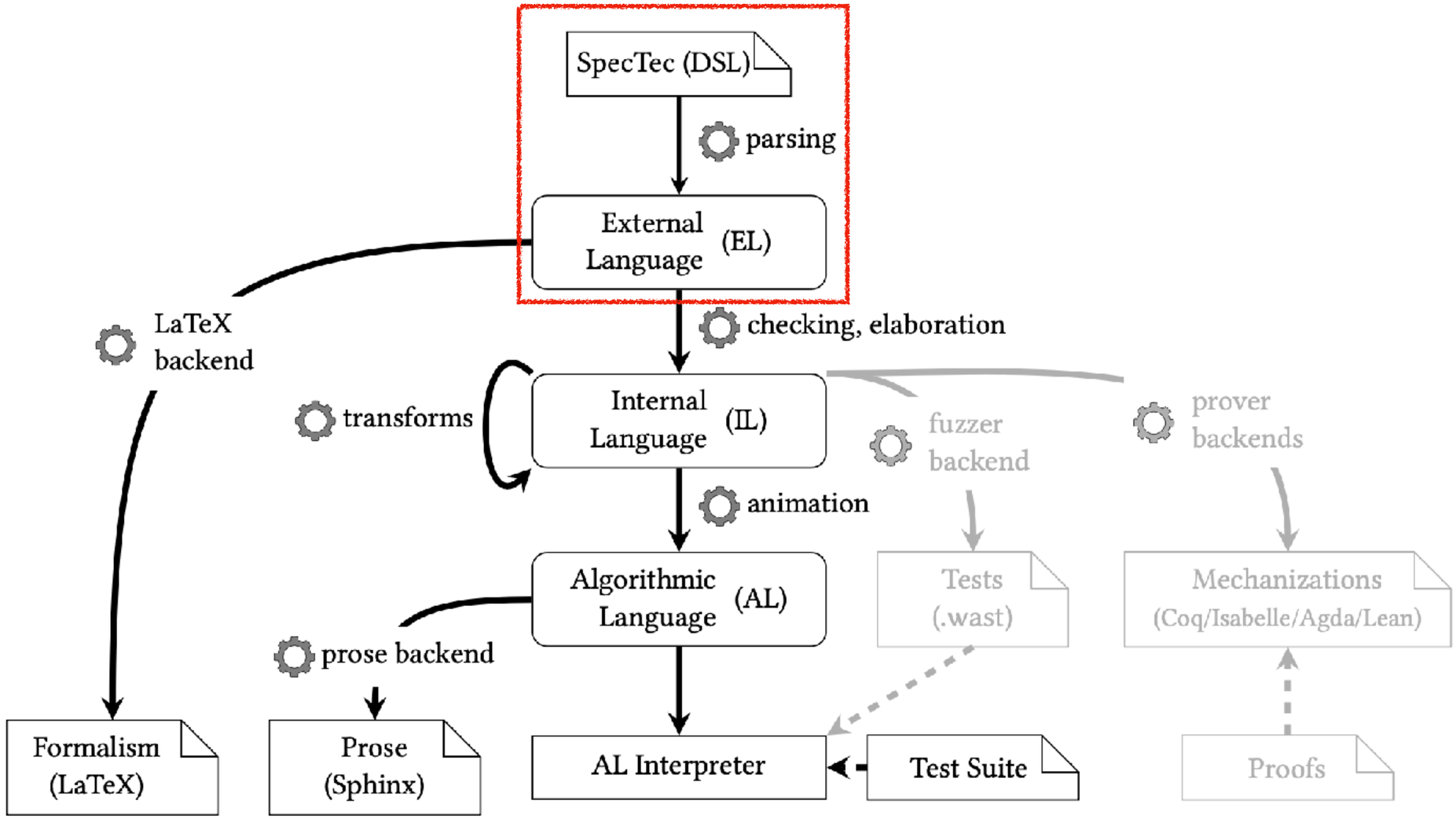
Overview: Backbone



Overview: Backends



DSL → EL


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

DSL

Designed for concisely describing Wasm specification
* Closely resembles pen-and-paper notation
* WYSIWYG - What You See Is What You Get

```
syntax instr = …
  | CONST numtype num_(numtype)
  | UNOP numtype unop_(numtype)
  | BINOP numtype binop_(numtype)
  | TESTOP numtype testop_(numtype)
  | RELOP numtype relop_(numtype)
  | ...
```

<Sytax>

```
rule Instr_ok/relop:
  C |- RELOP nt relop_nt : nt nt -> I32
```

<Relation: Typing Rule>

```
def $relop(numtype, relop, num, num) : num_(I32)
def $relop(inn, EQ, iN_1, iN_2) =
  $ieq($size(inn), iN_1, iN_2)
def $relop(inn, NE, iN_1, iN_2) =
  $ine($size(inn), iN_1, iN_2)
```

<Function>

```
rule Step_pure/relop:
  (CONST nt c_1) (CONST nt c_2) (RELOP nt relop)
~> (CONST I32 c)
  -- if c = $relop(nt, relop, c_1, c_2)
```

<Relation: Execution Semantics>

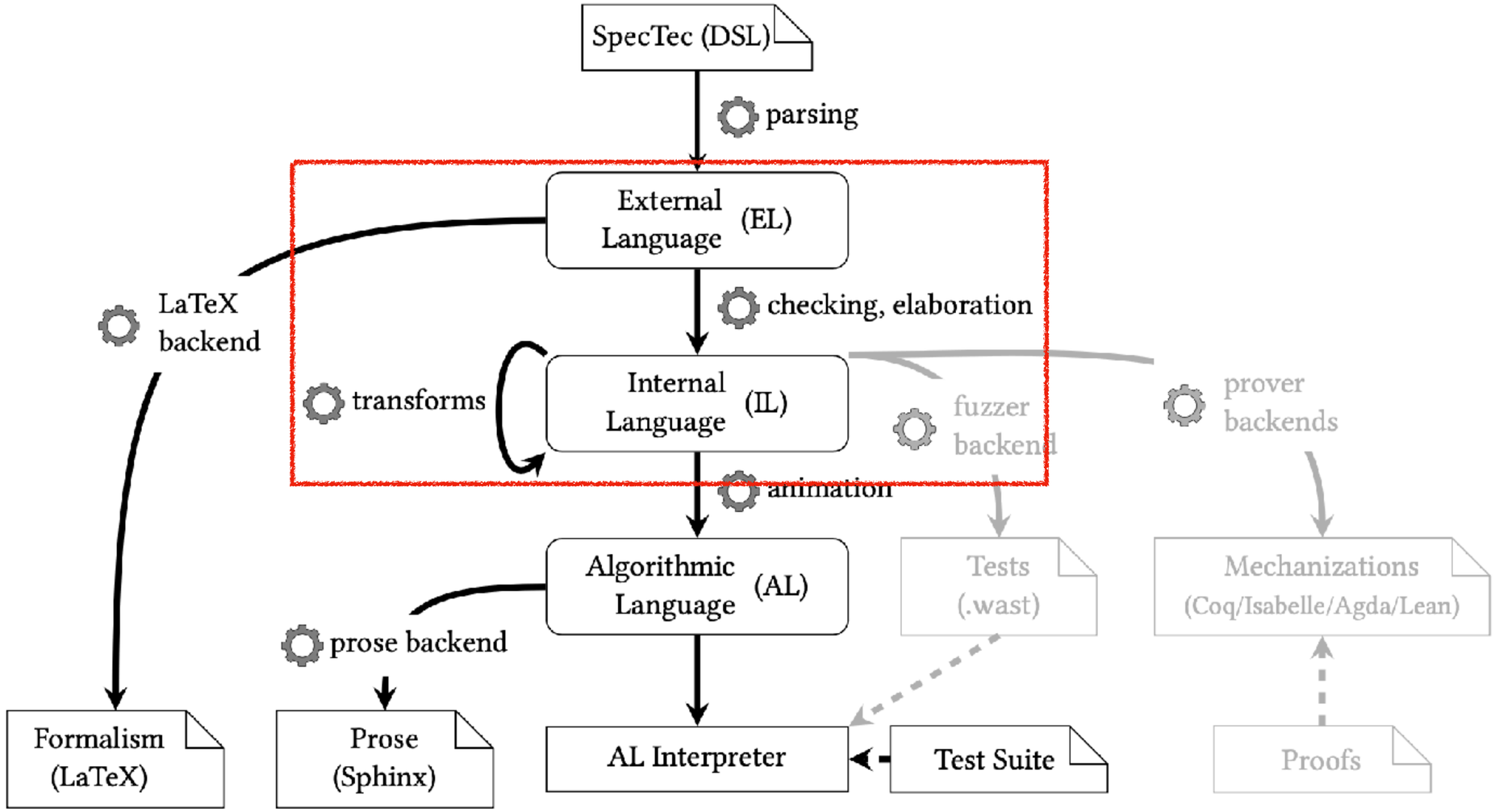
DSL → EL

Definitions in DSL are parsed into a representation called External Language (EL)

Identifier	*id*	::=	…lower-case-identifier …
Atom	*atom*	::=	…upper-case-identifier … | **–>** | **~>** | **:** | **|–** | …
Operator	*unop*	::=	~ | + | –
	binop	::=	/\ | \/ | **<=>** | + | – | * | / | ^ | = | =/= | < | <= | >= | > | …
Iteration	*iter*	::=	**?** | * | + | ^*exp*
Type	*typ*	::=	*id* | **bool** | **nat** | *typ typ* | *typ iter* | *atom* | { (*atom typ*,)* } | (| *atom typ*)+ | (*typ*,*) | …
Expression	*exp*	::=	*id* | *bool* | *num* | *unop exp* | *exp binop exp* | **$***id exp*? | **eps** | *exp exp* | *exp iter* | *exp*[*exp*]
		|	|*exp*| | *atom* | { (*atom exp*,)* } | *exp.atom* | *exp*[*path* = *exp*] | (*exp*,*) | …
Path	*path*	::=	*path*?[*exp*] | *path*?.*atom* | …
Definition	*def*	::=	**syntax** *id* = *typ* | **relation** *id* : *typ* | **rule** *id* (/ *id*)* : *exp* (-- *prem*)*
		|	**def $***id exp*? : *typ* | **def $***id exp*? = *exp* (-- *prem*)* | **var** *id* : *typ*
Premise	*prem*	::=	*id* : *exp* | **if** *exp* | **otherwise** | *prem iter*

<Syntax of EL>

EL → IL


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

EL → IL

EL is elaborated into a more explicit and unambiguous Internal Language (IL)

* Type Checking
* Dimension inference
* Binding and recursion analysis
* Type-driven resolution of notational overloading

✘

```
-- if x* = [1, 2, 3]
-- if y* = [0, 1, 2]
-- if x* < y* + 1
```

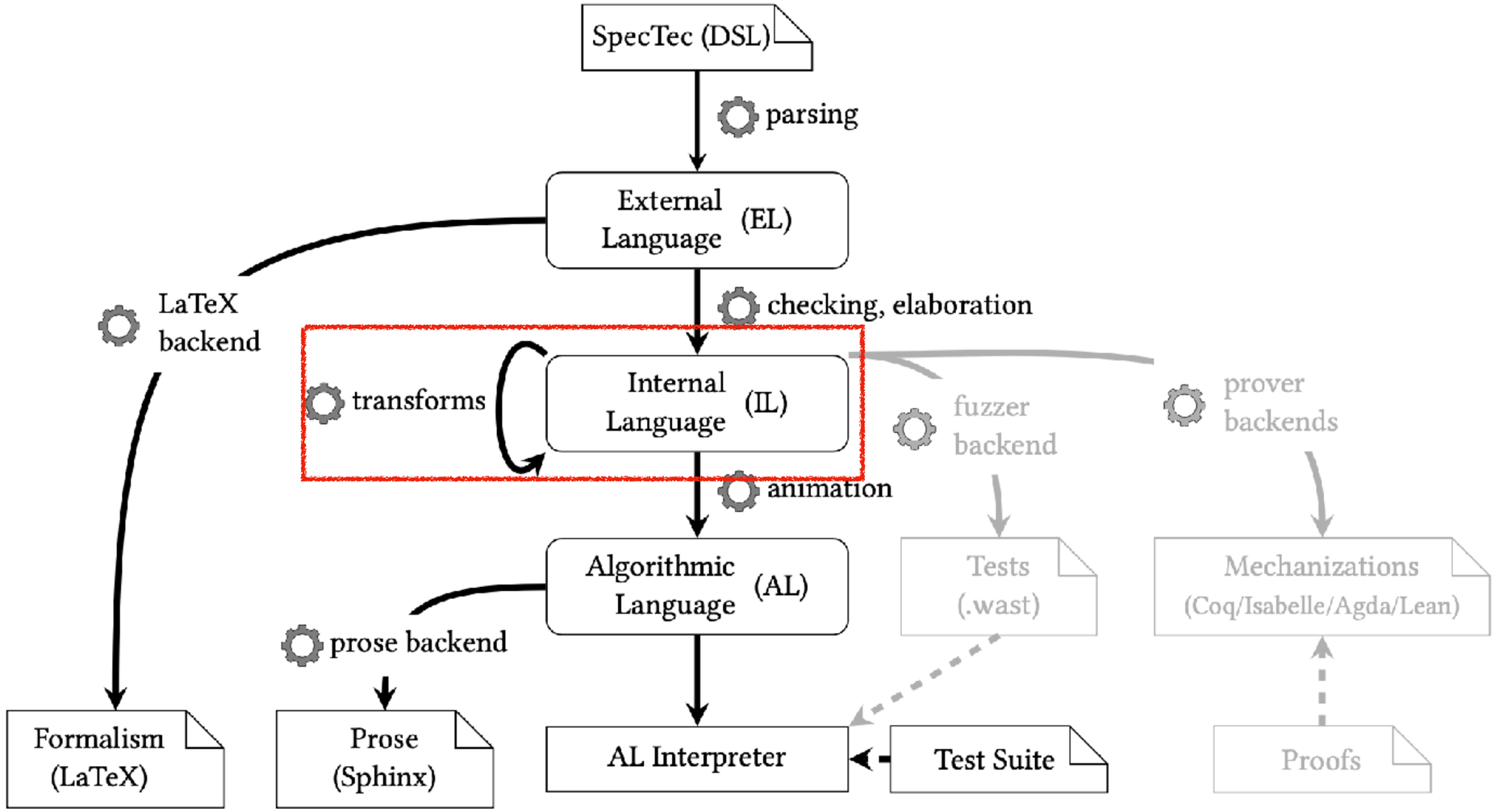
Type checking fail

✔

```
-- if x* = [1, 2, 3]
-- if y* = [0, 1, 2]
-- if (x < y + 1)*
```

Type checking + Dimension inference

IL → IL


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

IL → IL

Runs a number of transformations on the IL for simplifying / explicit notations

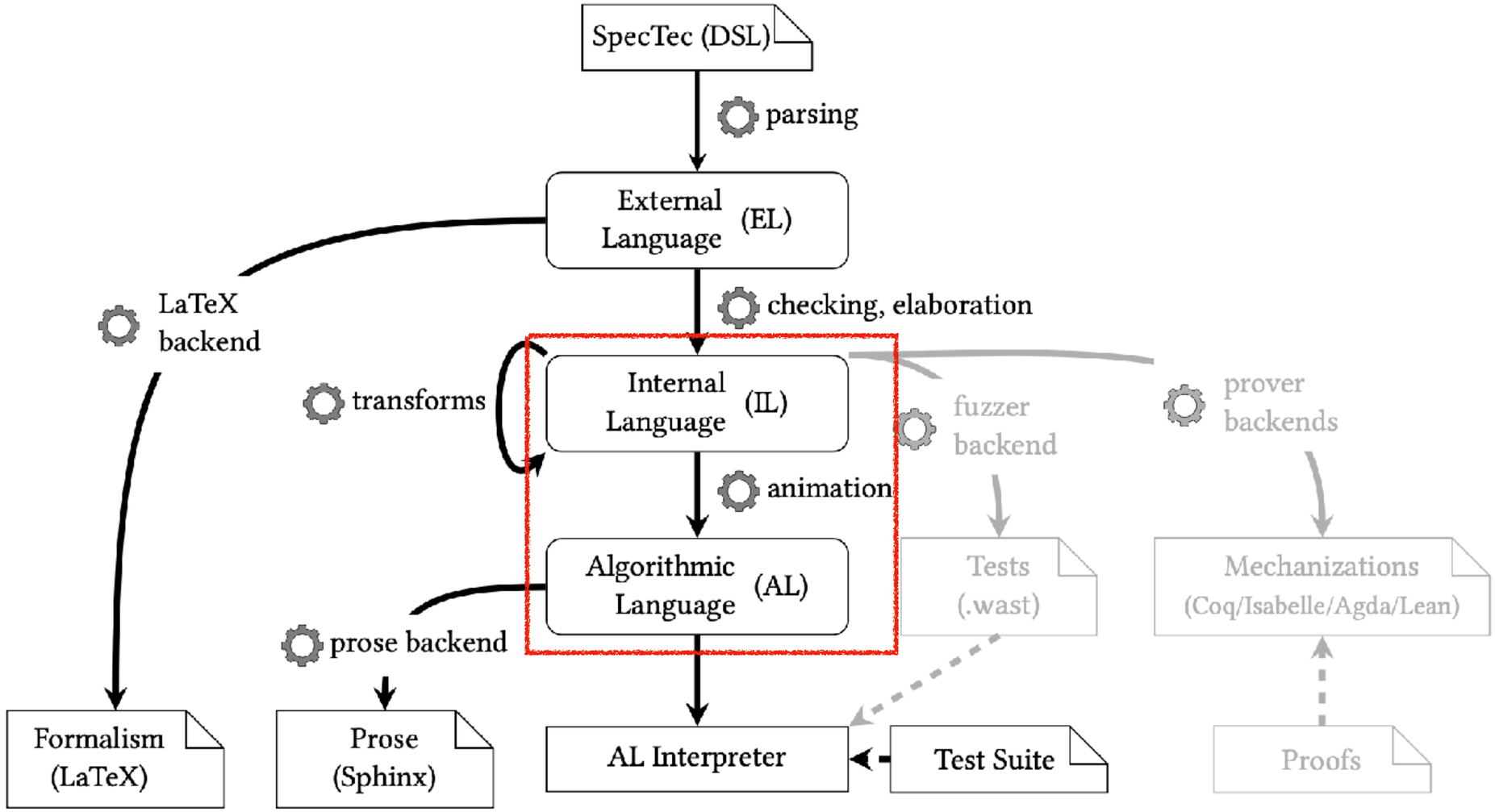
* Infer implicit side conditions
* Lift partial functions types into options
* Introduce auxiliary variables for option types
* Turn subsumption into explicit injections
* Identifying variable bindings + reorder premises

```
-- if x* = [1, 2, 3]
-- if y* = [0, 1, 2]
-- if (x < y + 1)*
```

```
-- if x* = [1, 2, 3]
-- if y* = [0, 1, 2]
-- if (x < y + 1)*
-- if |x*| = |y*|
```

Inferred side condition

IL → AL


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

AL

Algorithmic Language (AL) represents runtime semantics in an imperative style,

from which prose/interpreter can be easily generated

Program	p	::=	a^*
Algorithm	a	::=	algorithm $x(e^*)$ s^*
Statement	s	::=	if c s^* s^* — If c, then: s_1^* Else: s_2^*
		|	enter e e s^* — Enter e_1 with e_2 : s^*.
		|	exit — Exit current context.
		|	assert c — Assert: c.
		|	push e — Push e to the stack.
		|	pop e — Pop e from the stack.
		|	let e e — Let e_1 be e_2.
		|	trap — Trap.
		|	nop — Do nothing.
		|	return $e^?$ — Return $e^?$.
		|	execute e — Execute e.
		|	perform x e^* — Perform $x(e^*)$.
		|	replace e q^* e — Replace $e_1[q^*]$ with e_2.

Condition	c	::=	not c | c $\oplus$ c | iscase e x | w	
Expression	e	::=	x	x
		|	n	n
		|	$\oplus$ e	$\oplus$ e
		|	e $\oplus$ e	e_1 $\oplus$ e_2
		|	$e[q]$	$e[q]$
		|	$e[q^* := e]$	e_1 with q^* = e_2
		|	$\{(x \mapsto e)^*\}$	{ (x : e)* }
		|	$[e^*]$	e^*
		|	e ++ e	e_1 e_2
		|	$\|e\|$	length of e
		|	$(k\ e^*)$	$(k\ e^*)$
		|	$x(e^*)$	$x(e^*)$
		|	w	
Path	q	::=	e | $.x$	

<Syntax of AL>

IL → AL

Translating IL definitions into AL definitions is mostly straightforward,

via simple pattern matching on the execution rule expressed in IL.

```
LHS   ~>   RHS
-- PREM*
```

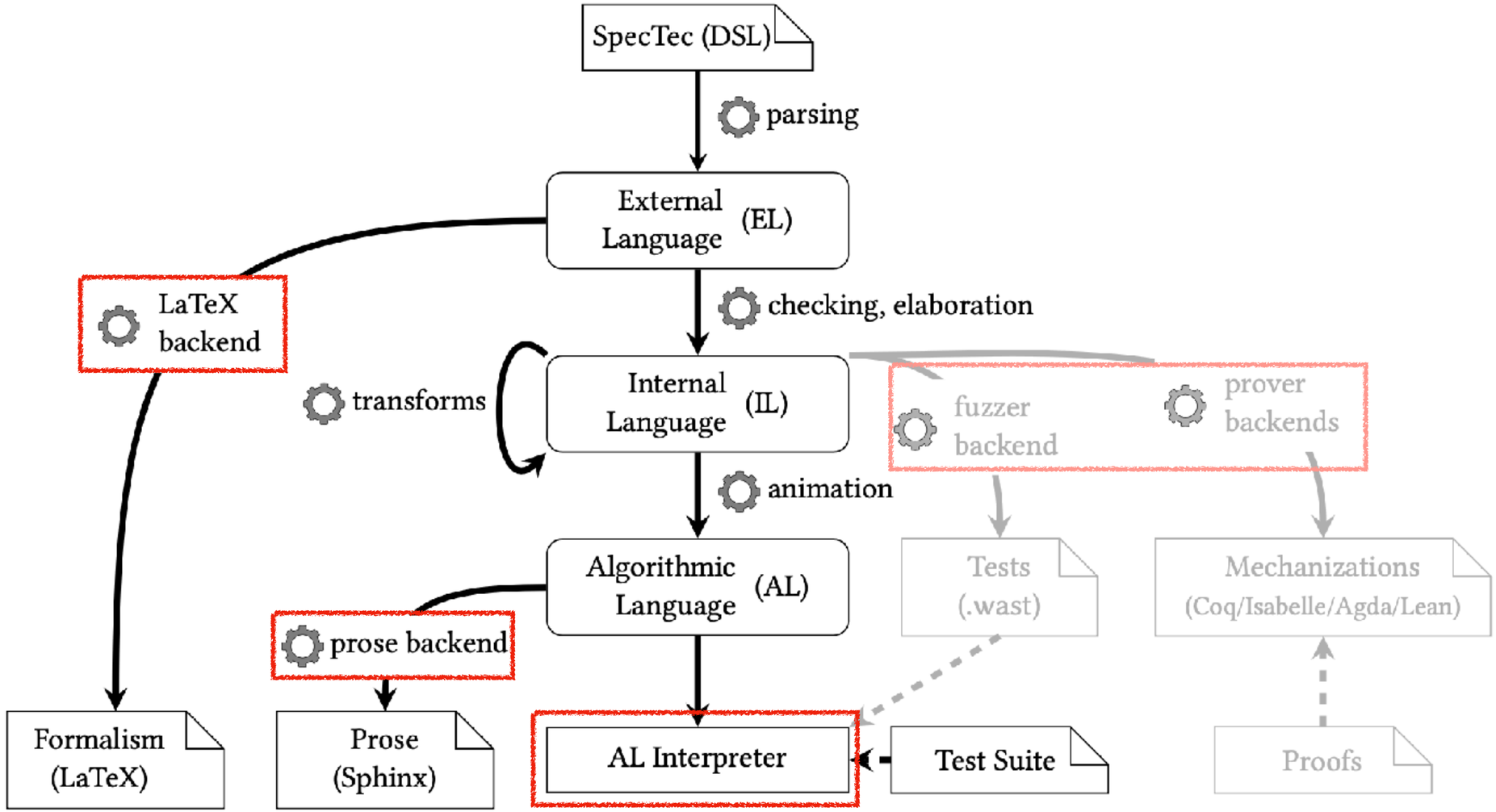
<Preprocessed IL>

→

```
1. Pop … from the stack.
2. Let … be ….
3. If …, then:
   a. Let … be ….
   b. Push … to the stack.
   c. Execute ….
```

<AL>

Backends


SpecTec (DSL)
parsing
External Language (EL)
checking, elaboration
LaTeX backend
transforms
Internal Language (IL)
fuzzer backend
prover backends
animation
Algorithmic Language (AL)
Tests (.wast)
Mechanizations (Coq/Isabelle/Agda/Lean)
prose backend
Formalism (LaTeX)
Prose (Sphinx)
AL Interpreter
Test Suite
Proofs

Formal / Prose Backend

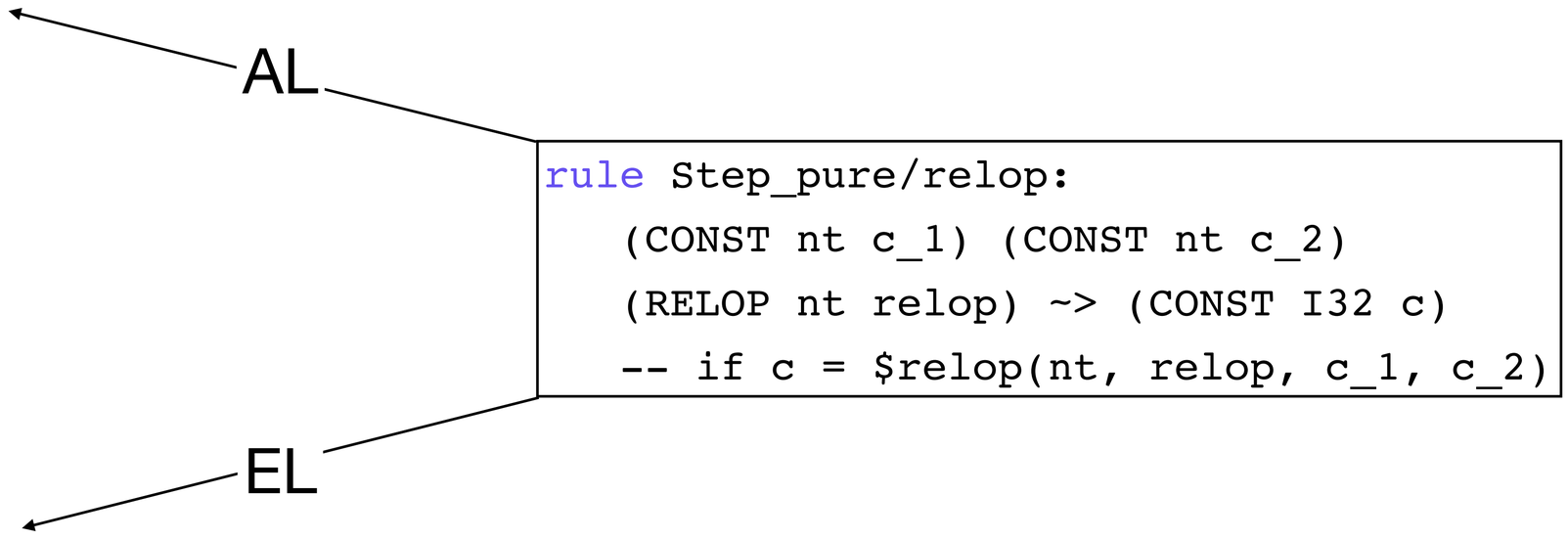
Input:

```
.. _exec-relop:

$${rule-prose: exec/relop}

\

$${rule: Step_pure/relop}
```



Formal / Prose Backend

Output:

```
.. _exec-relop:

:math:`{\mathit{nt}} . {\mathit{relop}}`
.....................................
1. Assert: Due to validation, a value of value type :math:`{\mathit{nt}}` is on the top of the stack.
2. Pop the value :math:`({\mathit{nt}}{.}\mathsf{const}~c_2)` from the stack.
3. Assert: Due to validation, a value of value type :math:`{\mathit{nt}}` is on the top of the stack.
4. Pop the value :math:`({\mathit{nt}}{.}\mathsf{const}~c_1)` from the stack.
5. Let :math:`c` be :math:`{{{\mathit{relop}}}{}_{{\mathit{nt}}}}{(c_1,\, c_2)}`.
6. Push the value :math:`(\mathsf{i{\scriptstyle32}}{.}\mathsf{const}~c)` to the stack.

\

.. math::
   \begin{array}{@{}l@{}rcl@{}l@{}}
   & ({\mathit{nt}}{.}\mathsf{const}~c_1)~({\mathit{nt}}{.}\mathsf{const}~c_2)~({\mathit{nt}} .
{\mathit{relop}}) &\hookrightarrow& (\mathsf{i{\scriptstyle32}}{.}\mathsf{const}~c)
     &\qquad \mbox{if}~c = {{{\mathit{relop}}}{}_{{\mathit{nt}}}}{(c_1,\, c_2)} \\
   \end{array}
```

Formal / Prose Backend

Rendered:

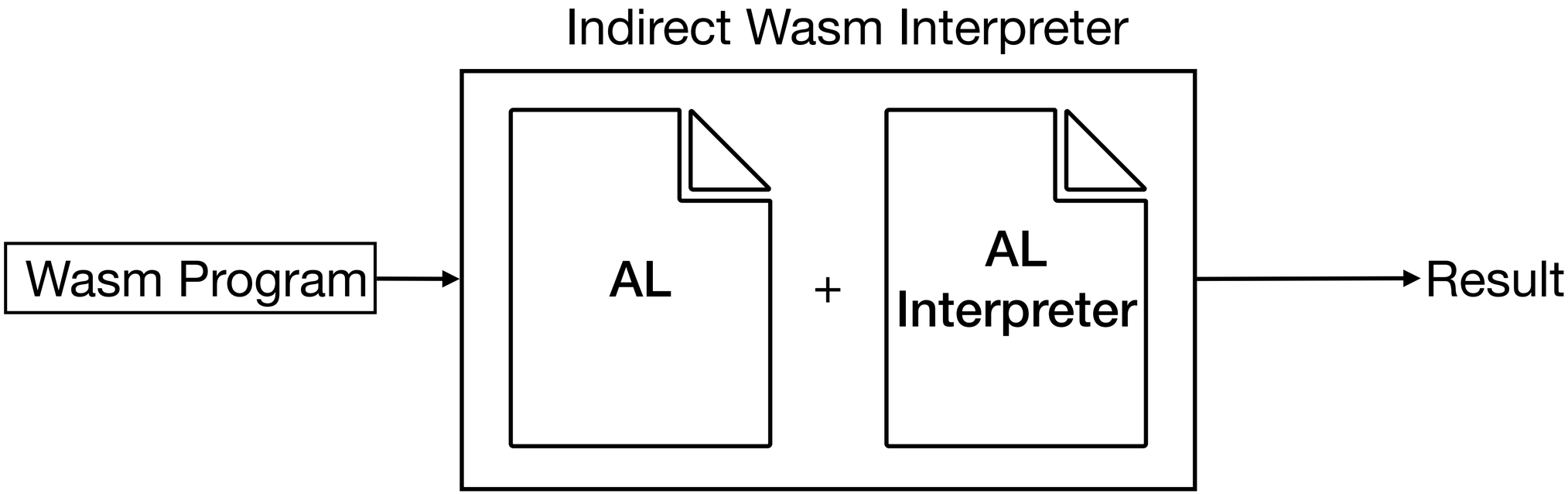
nt.relop

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\mathsf{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop the value $(nt.\mathsf{const}\ c_1)$ from the stack.
5. Let c be $relop_{nt}(c_1,\ c_2)$.
6. Push the value $(\mathsf{i32}.\mathsf{const}\ c)$ to the stack.

$$(nt.\mathsf{const}\ c_1)\ (nt.\mathsf{const}\ c_2)\ (nt.relop) \quad \hookrightarrow \quad (\mathsf{i32}.\mathsf{const}\ c) \qquad \text{if } c = relop_{nt}(c_1,\ c_2)$$

Indirect Interpreter

Interpreting Wasm by interpreting the specification written in AL



Evaluation

1. **Correctness**

   Does SpecTec correctly generate formal and prose specifications, as well as the interpreter backend?

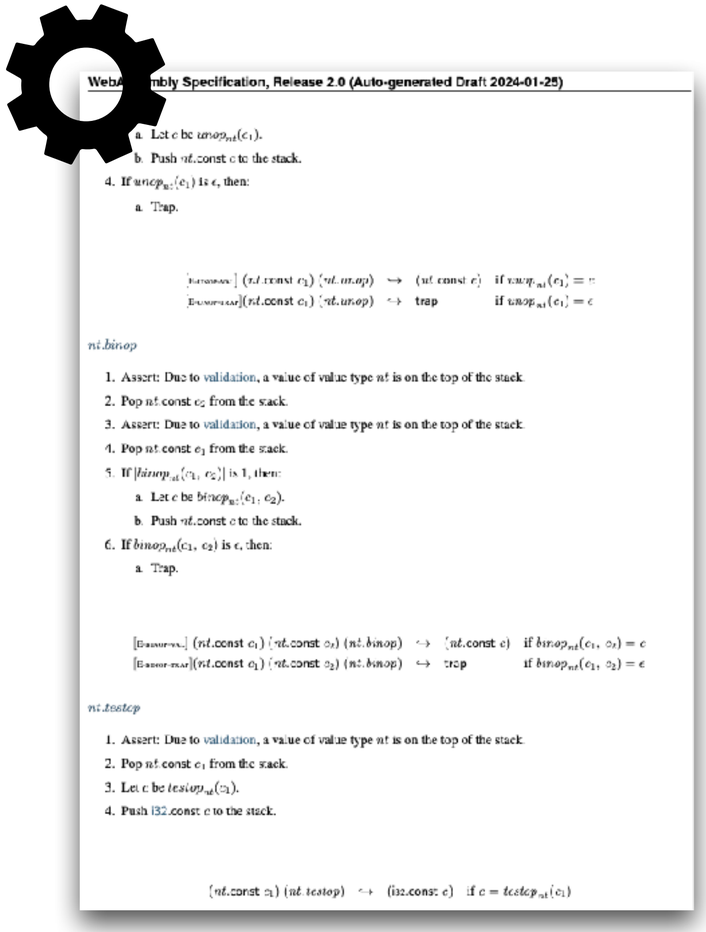
2. **Bug Prevention**

   Can SpecTec prevent bugs during Wasm standard development?

3. **Forward Compatibility**

   Can SpecTec support future language features?

1. Correctness



WebAssembly Specification, Release 2.0 (Auto-generated Draft 2024-01-25)

a. Let c be $unop_{nt}(c_1)$.

b. Push nt.const c to the stack.

4. If $unop_{nt}(c_1)$ is ϵ, then:

a. Trap.

$$\begin{array}{lrll} [\text{Binop-val}] & (nt.\text{const } c_1)\ (nt.unop) & \hookrightarrow & (nt.\text{const } c) \quad \text{if } unop_{nt}(c_1) = c \\ [\text{Binop-trap}] & (nt.\text{const } c_1)\ (nt.unop) & \hookrightarrow & \text{trap} \quad\quad\ \ \text{if } unop_{nt}(c_1) = \epsilon \end{array}$$

nt.binop

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c_2 from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop nt.const c_1 from the stack.
5. If $|binop_{nt}(c_1, c_2)|$ is 1, then:
   a. Let c be $binop_{nt}(c_1, c_2)$.
   b. Push nt.const c to the stack.
6. If $binop_{nt}(c_1, c_2)$ is ϵ, then:
   a. Trap.

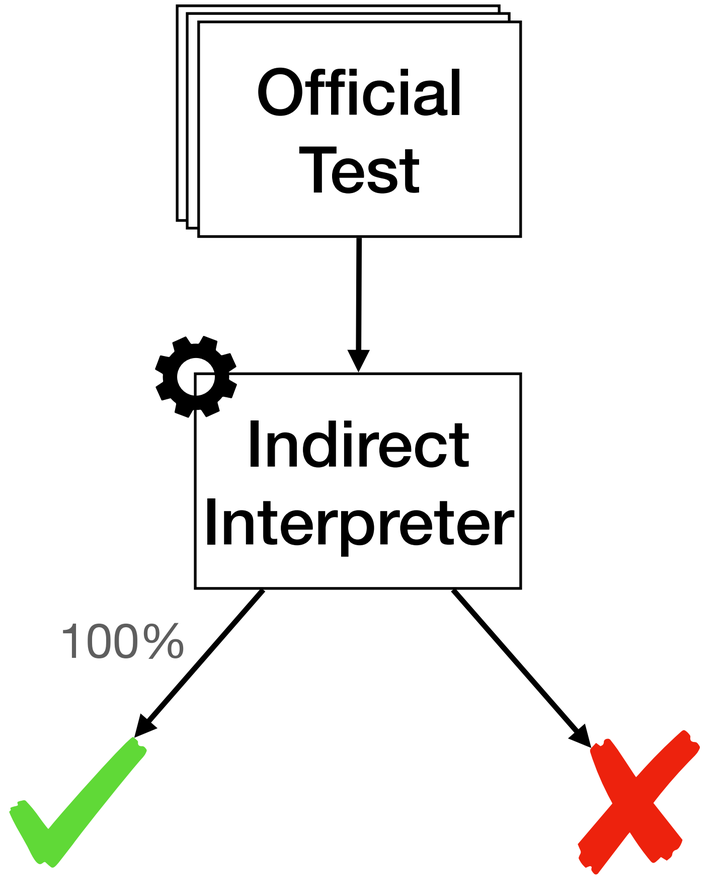
$$\begin{array}{lrll} [\text{Binop-val}] & (nt.\text{const } c_1)\ (nt.\text{const } c_2)\ (nt.binop) & \hookrightarrow & (nt.\text{const } c) \quad \text{if } binop_{nt}(c_1, c_2) = c \\ [\text{Binop-trap}] & (nt.\text{const } c_1)\ (nt.\text{const } c_2)\ (nt.binop) & \hookrightarrow & \text{trap} \quad\quad\ \ \text{if } binop_{nt}(c_1, c_2) = \epsilon \end{array}$$

nt.testop

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop nt.const c_1 from the stack.
3. Let c be $testop_{nt}(c_1)$.
4. Push i32.const c to the stack.

$$(nt.\text{const } c_1)\ (nt.testop) \quad \hookrightarrow \quad (\text{i32.const } c) \quad \text{if } c = testop_{nt}(c_1)$$

Auto-generation of formal / prose specification for Wasm 2.0



Auto-generation of indirect interpreter + 100% pass rate of official test

2. Bug prevention

Injected 13 retrospective spec bugs into the spec written in DSL

→ All bugs were detected at various phase of SpecTec

[spec] Fix variable name typos (#1358)

[spec] Fix missing immediate on table.set (#1441)

[spec] Fix typos in instruction validation rules (#1462)

[spec] Fix naming typo (#1532)

[spec] Fix typo in element execution (#1544)

[spec] Add missing case for declarative elem segments

[spec] Remove an obsolete exec step (#1580)

[spec] Remove stray `x` indices (#1598)

[spec] Add missing value to table.grow reduction rule (#1607)

[spec] Fix reduction rule for label (#1612)

[spec] Add missing access to current frame in prose (#1624)

[spec] Add missing type to elem.drop and store soundness (#1668)

[spec] Pop dummy frame after Invocation (#1691)

3. Forward Compatibility

SpecTec can handle five proposals to be included in Wasm 3.0

+ Found & Reported 10 bugs in the proposals

Phase 4 - Standardize the Feature (WG)

Proposal	Champion
Tail call	Andreas Rossberg
Extended Constant Expressions	Sam Clegg
Typed Function References	Andreas Rossberg
Garbage collection	Andreas Rossberg
Multiple memories	Andreas Rossberg

Pass index argument to `getfield` function #463

Fix `array.get/set` reduction rule #464

[spec] Fix spec for execution of `struct.new`, `array.new_fixed` and `br_on_cast(_fail)` #456

Dimension mismatch in the premise of `array.new_data` reduction rule #476

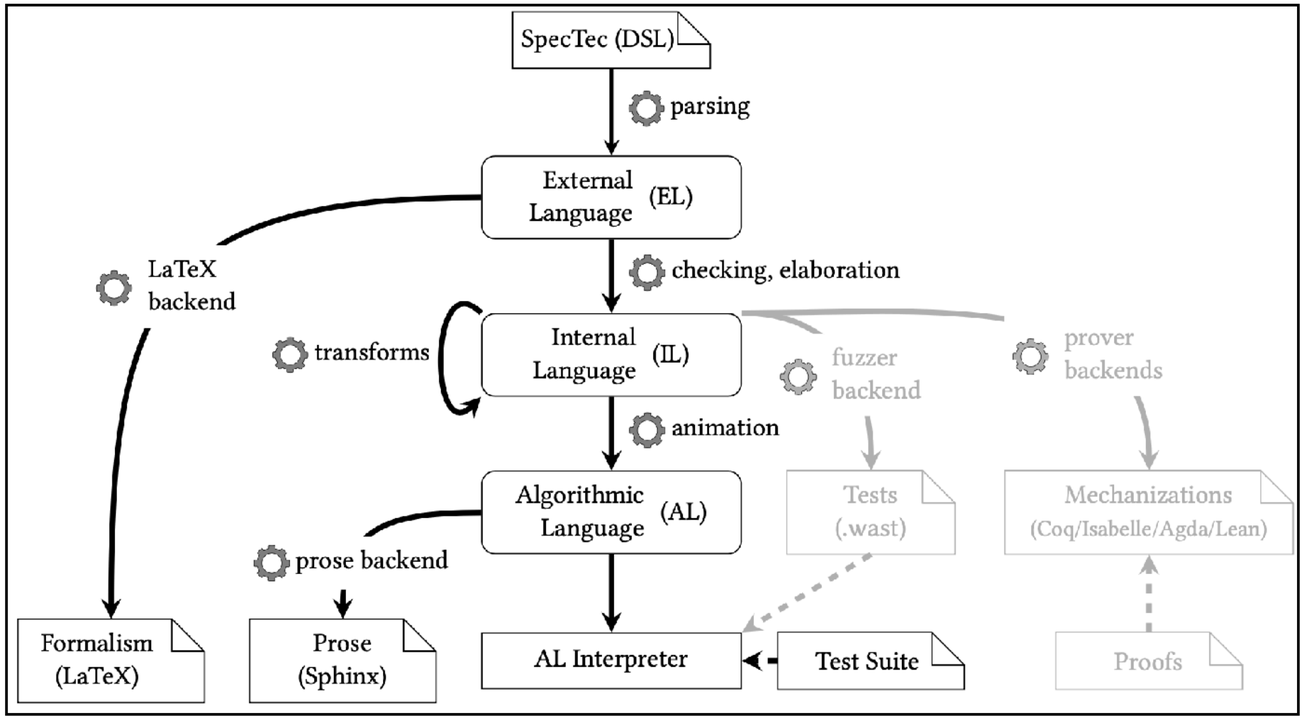
Minor changes on array.new_elem #477

Fix ref.null semantics #478

Add current frame #484

Bringing the WebAssembly Standard up to Speed with SpecTec

https://github.com/Wasm-DSL/spectec




```
rule Step_pure/relop:
   (CONST nt c_1) (CONST nt c_2)
   (RELOP nt relop) ~> (CONST I32 c)
   -- if c = $relop(nt, relop, c_1, c_2)
```

nt.relop

1. Assert: Due to validation, a value of value type nt is on the top of the stack.
2. Pop the value $(nt.\text{const}\ c_2)$ from the stack.
3. Assert: Due to validation, a value of value type nt is on the top of the stack.
4. Pop the value $(nt.\text{const}\ c_1)$ from the stack.
5. Let c be $relop_{nt}(c_1, c_2)$.
6. Push the value $(\text{i32.const}\ c)$ to the stack.

$$(nt.\text{const}\ c_1)\ (nt.\text{const}\ c_2)\ (nt.relop) \hookrightarrow (\text{i32.const}\ c) \quad \text{if } c = relop_{nt}(c_1, c_2)$$