SpecTec

Update and Vote

Andreas Rossberg

with Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu,
Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar,
Rao Xiaojia, Conrad Watt

Wasm CG 2024/06/06

a DSL for authoring the Wasm spec

Spec authoring today

Duplicate work to write *both* formal and prose rules

Writing prose is particularly laborious

Both formats are verbose and terrible for code reviews

Error-prone due to lack of meta-level error checking (syntax, rules, cross-links, etc.)

No macro facilities in Sphinx

Spec authoring with SpecTec

SpecTec is (close to) WYSIWYG ASCII for the math rules

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

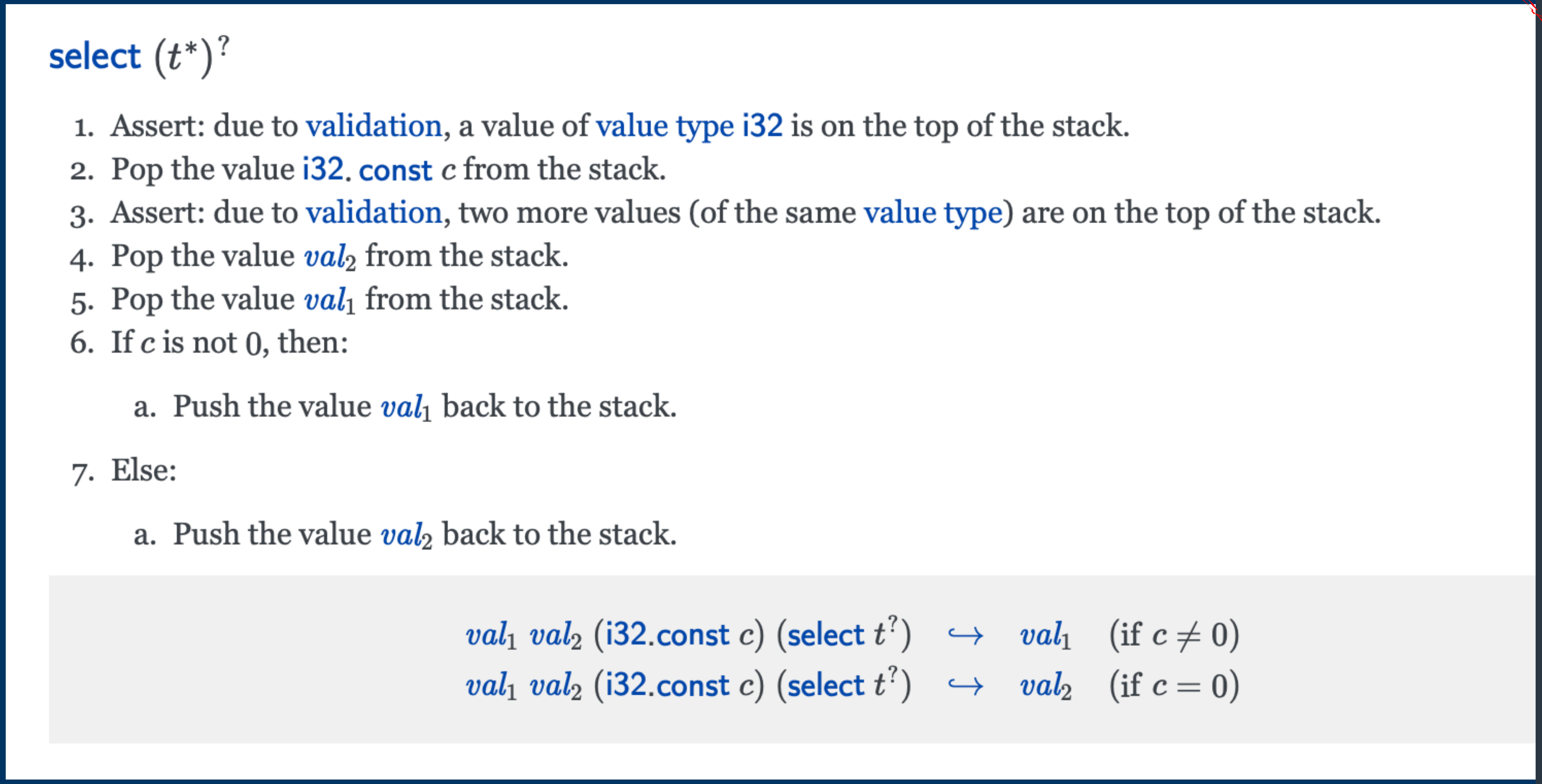
Single source of truth for auto-generating:

…math in Latex

…prose in Sphinx

…mechanised Coq and co (future work)

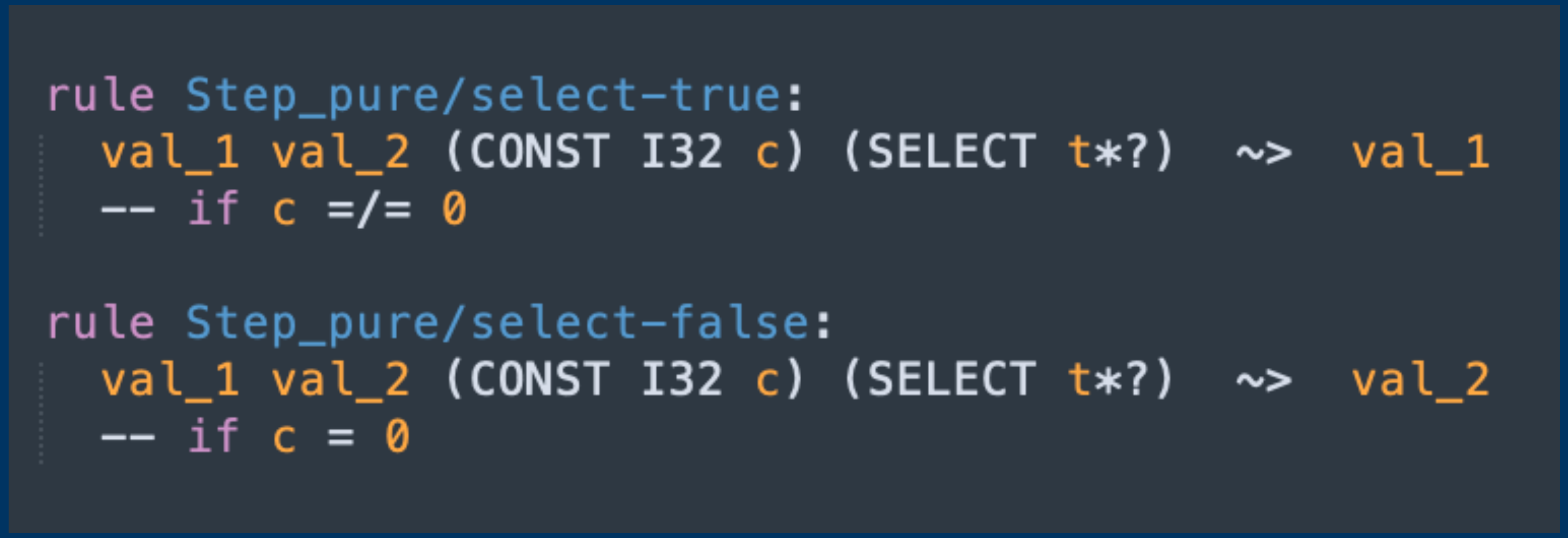
Generated math and prose can be spliced into spec document



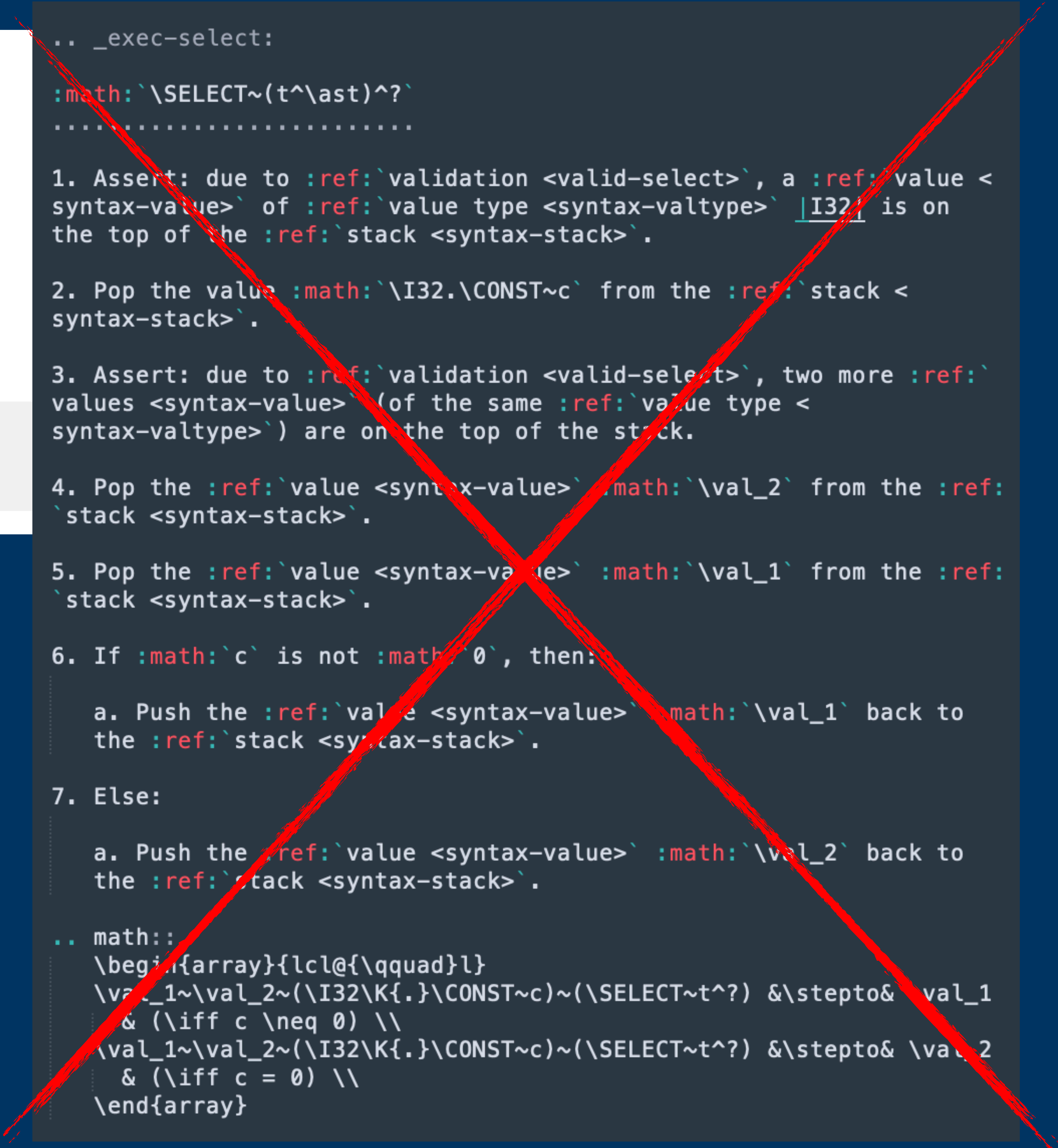
select $(t^*)^?$

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.

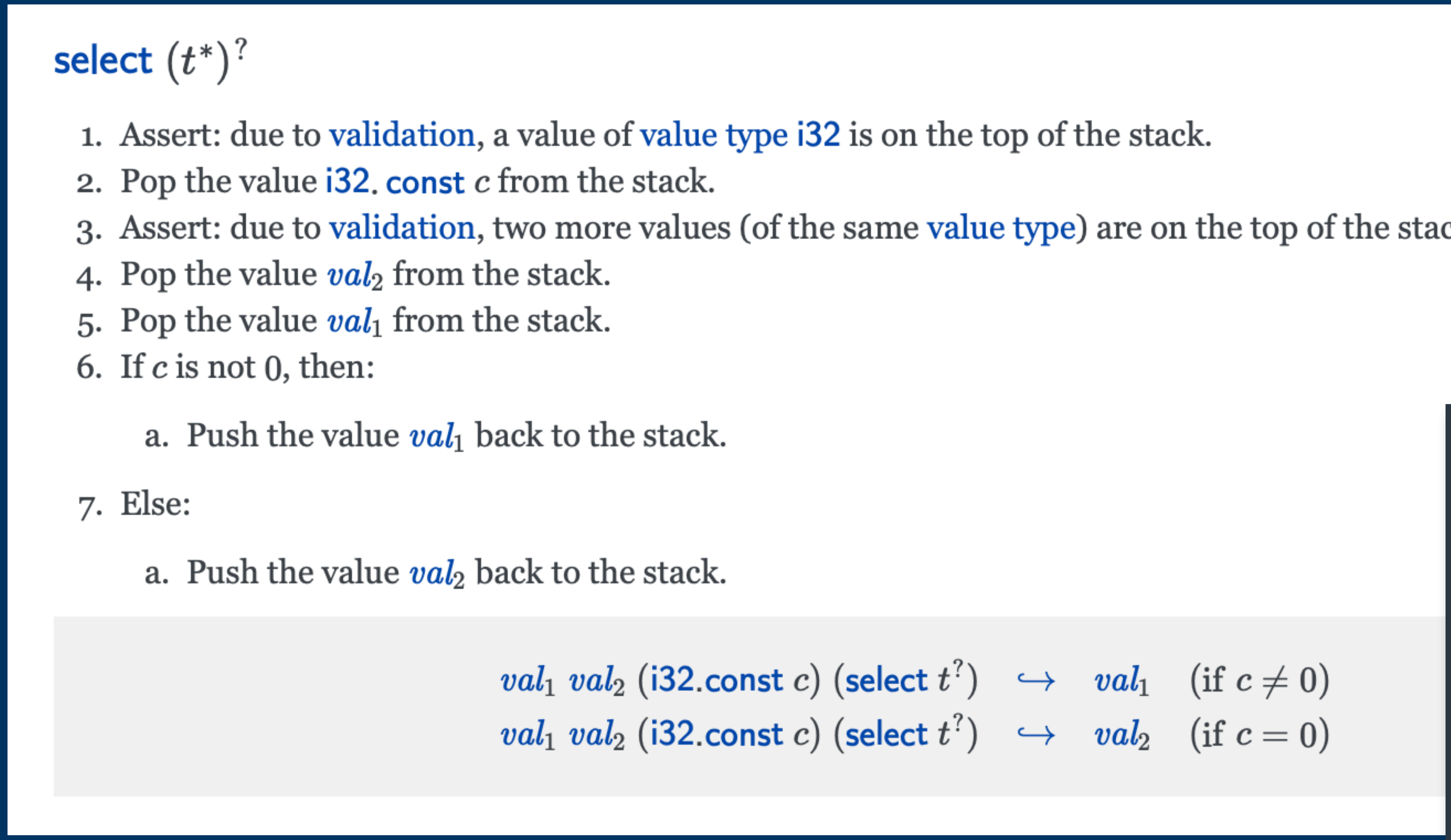
$$\begin{array}{lcl@{\qquad}l} val_1~val_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & val_1 & (\text{if } c \neq 0) \\ val_1~val_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & val_2 & (\text{if } c = 0) \end{array}$$



```
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
```



```
.. _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 $(t^*)^?$

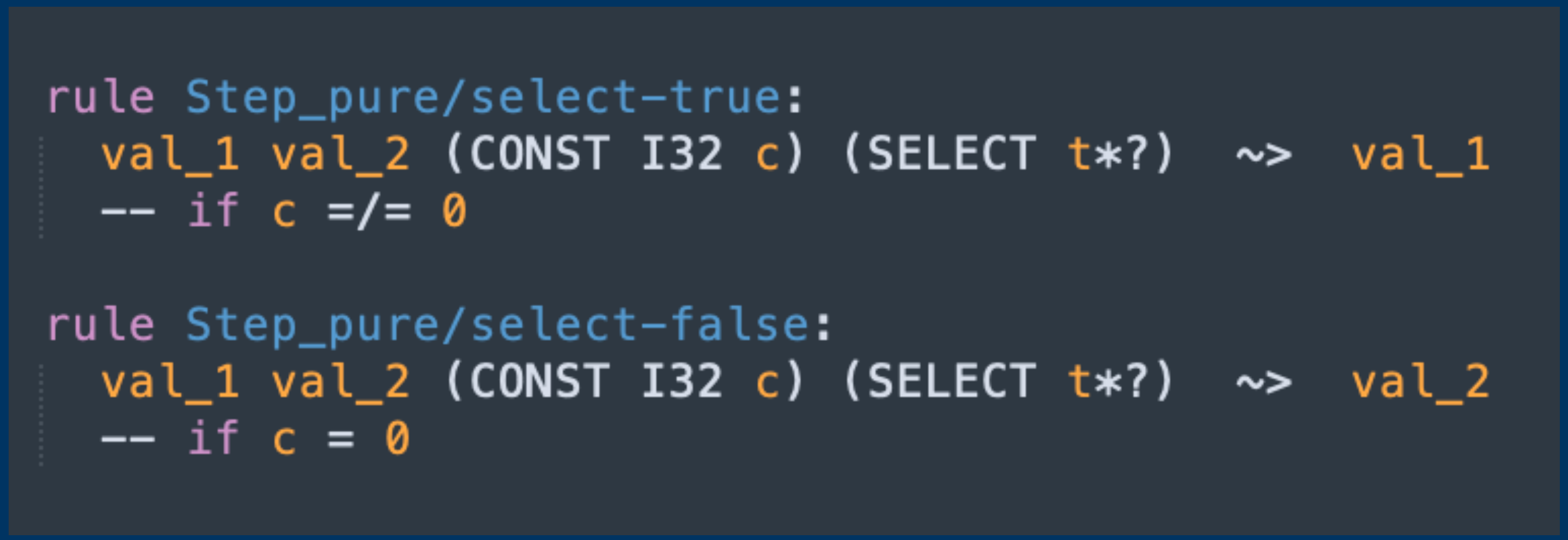
1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value $\mathsf{i32.const}~c$ from the stack.
3. Assert: due to validation, two more values (of the same value type) are on the top of the stack.
4. Pop the value val_2 from the stack.
5. Pop the value val_1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

    a. Push the value val_1 back to the stack.

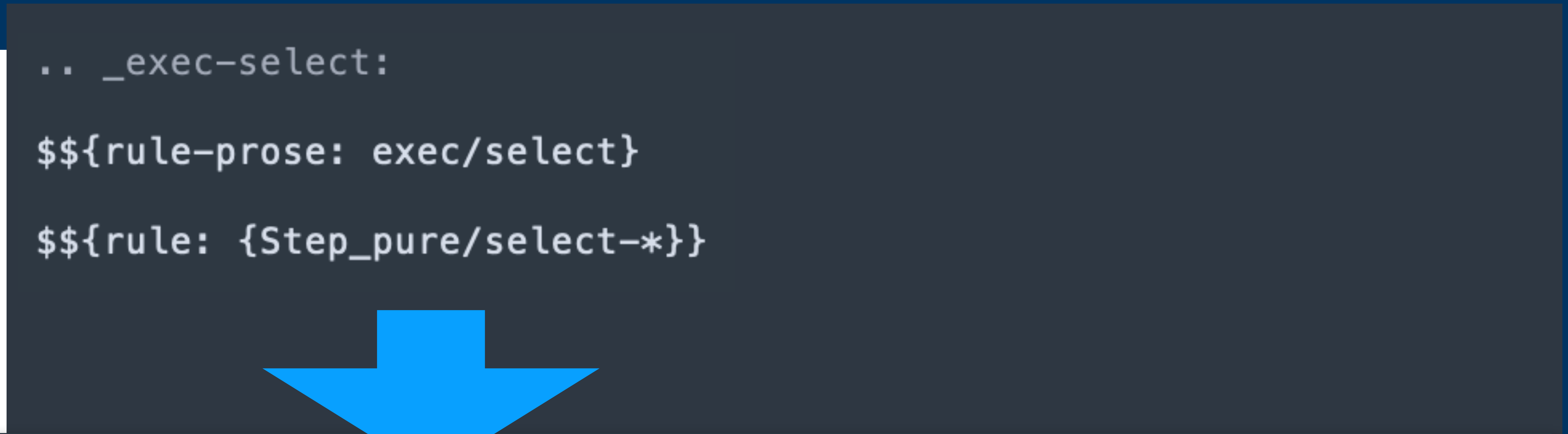
7. Else:

    a. Push the value val_2 back to the stack.

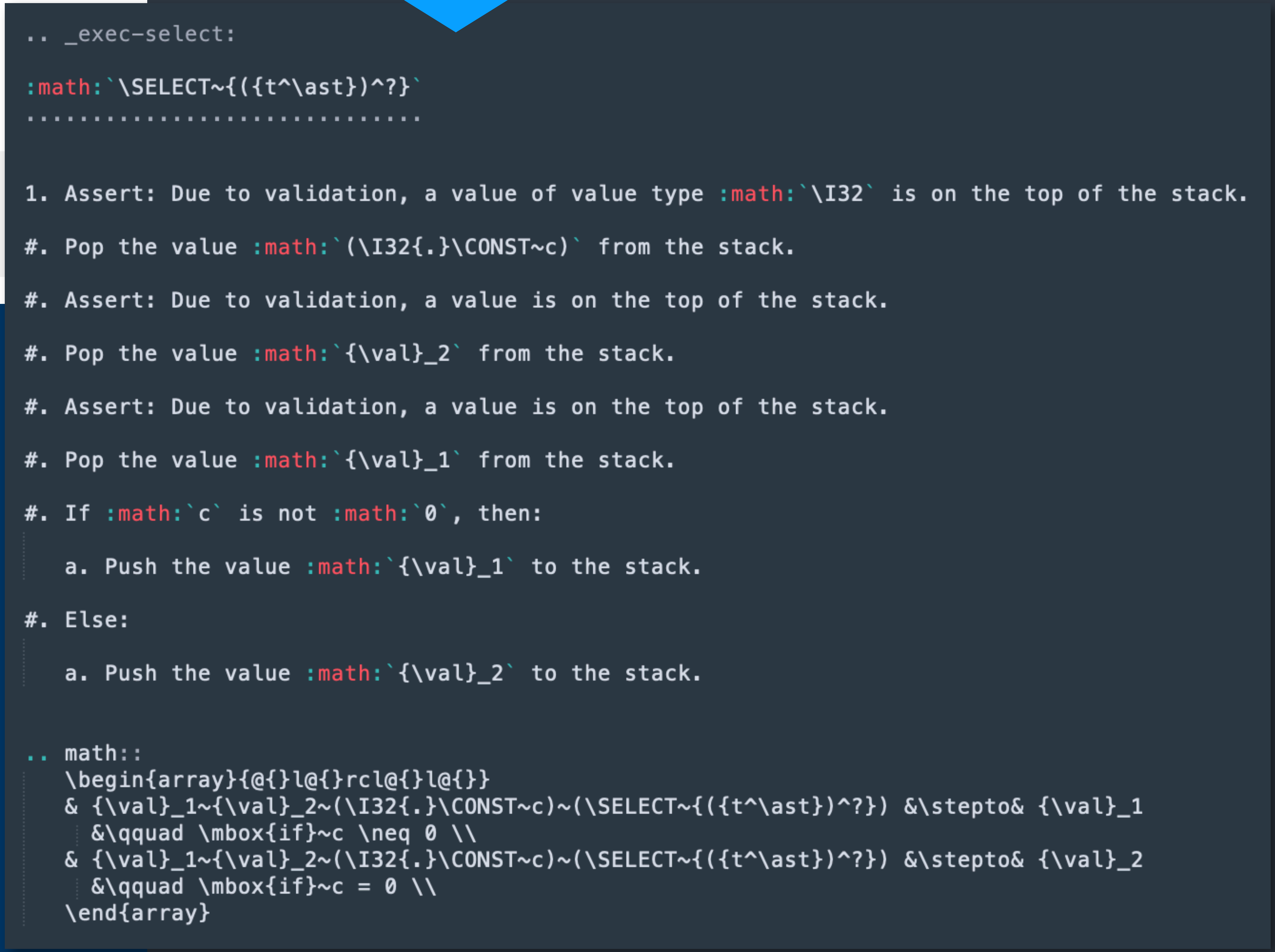
$$\begin{array}{llll}
val_1~val_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & val_1 & (\text{if } c \neq 0) \\
val_1~val_2~(\mathsf{i32.const}~c)~(\mathsf{select}~t^?) & \hookrightarrow & val_2 & (\text{if } c = 0)
\end{array}$$



```
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
```



```
.. _exec-select:

$${rule-prose: exec/select}

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



```
.. _exec-select:

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

1. Assert: Due to validation, a value of value type :math:`\I32` is on the top of the stack.

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

#. Assert: Due to validation, a value is on the top of the stack.

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

#. Assert: Due to validation, a value is on the top of the stack.

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

#. If :math:`c` is not :math:`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}{@{}l@{}rcl@{}l@{}}
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) &\stepto& {\val}_1
     &\qquad \mbox{if}~c \neq 0 \\
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) &\stepto& {\val}_2
     &\qquad \mbox{if}~c = 0 \\
   \end{array}
```

Meta Interpreter

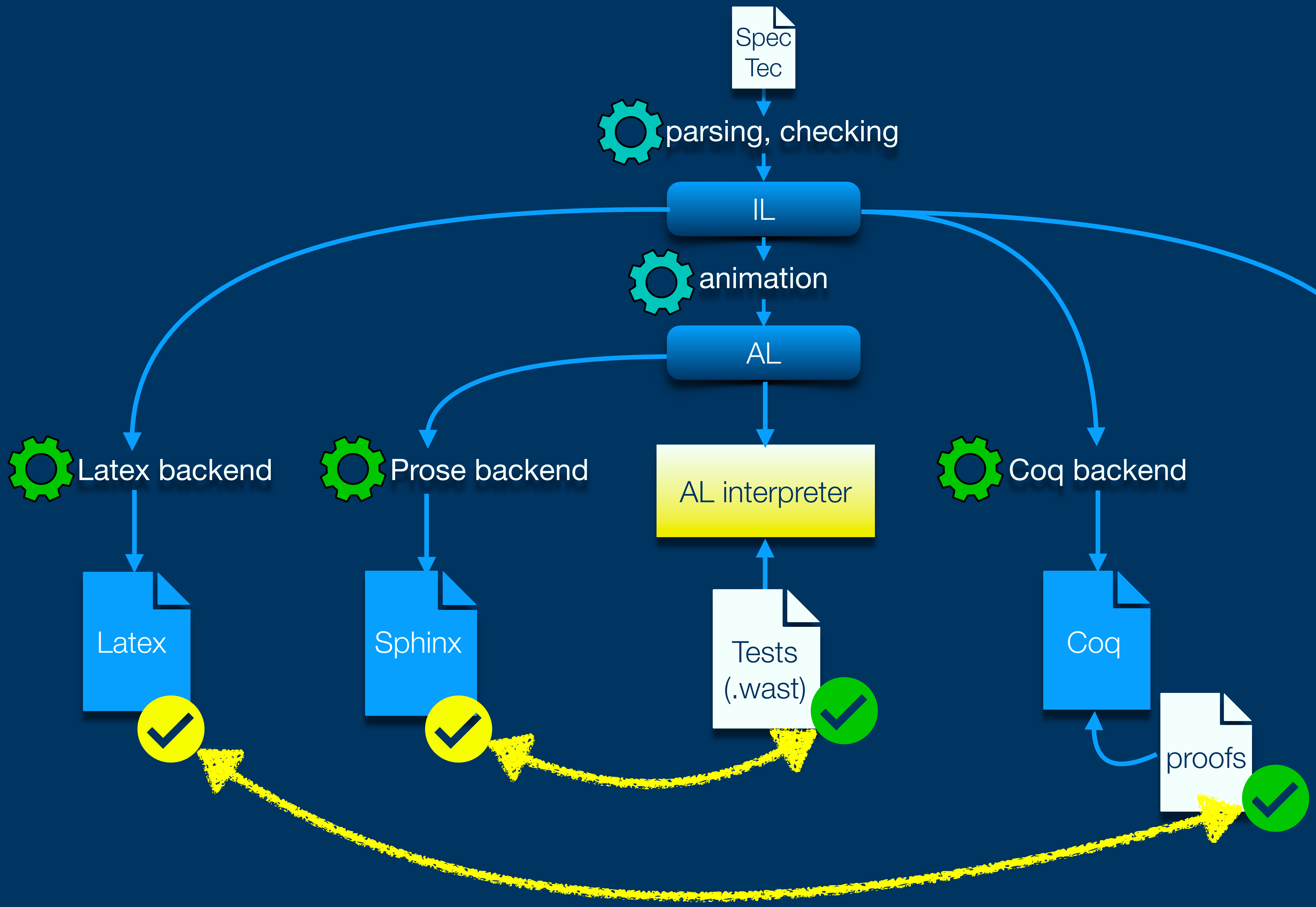
Algorithmic prose is derived from user-defined declarative reduction rules

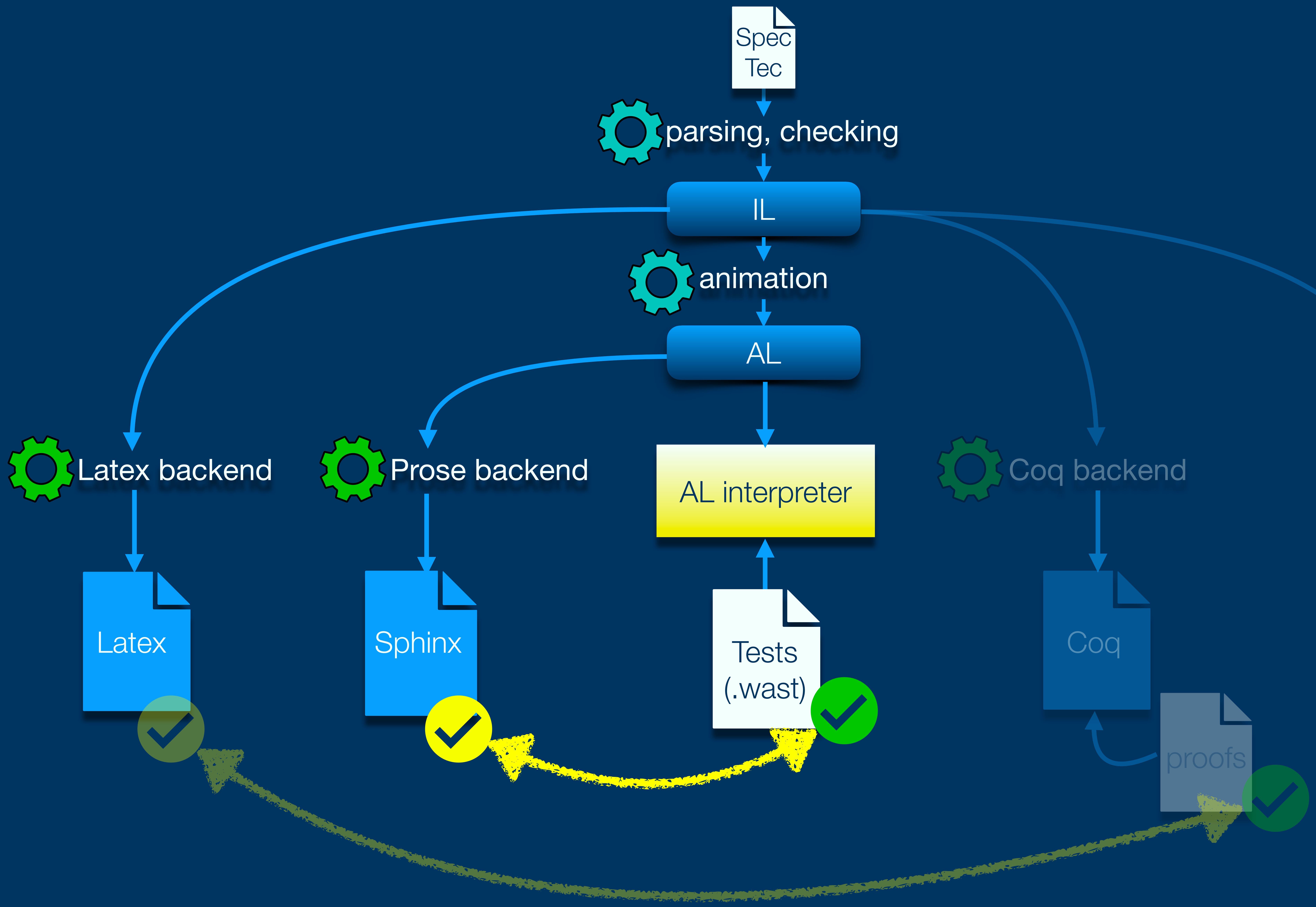
via an intermediate representation that's an AST for the prose

and we can interpret "programs" in this AST, i.e., the Wasm spec

…and thereby indirectly run actual Wasm with actual Wasm spec

Passes 100% of applicable Wasm test suite


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


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

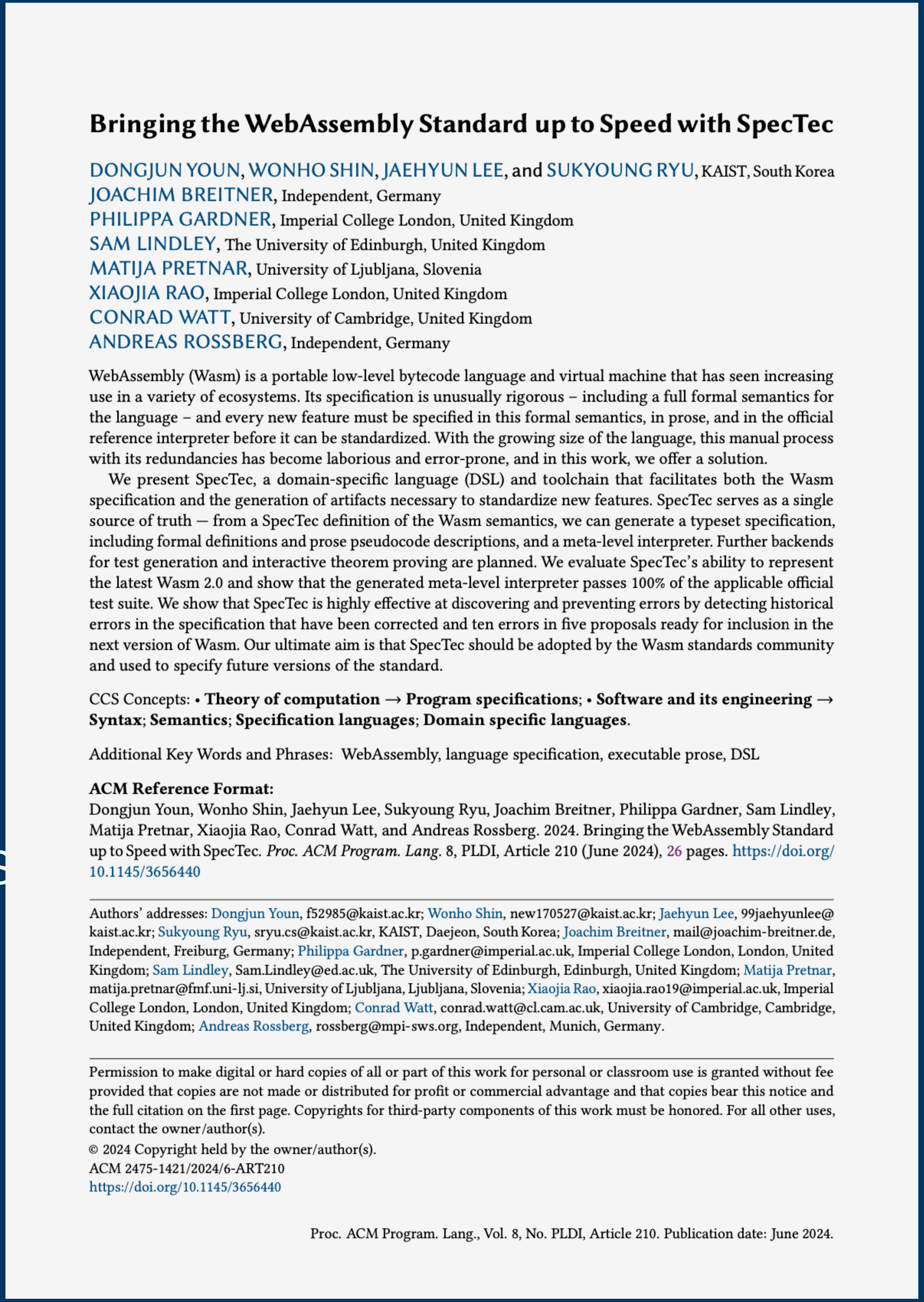
Updates

implemented generation of hyperlinking and customis

fixes and rendering improvements

started writing a tutorial

converted more of the spec document



Bringing the WebAssembly Standard up to Speed with SpecTec

DONGJUN YOUN, WONHO SHIN, JAEHYUN LEE, and SUKYOUNG RYU, KAIST, South Korea
JOACHIM BREITNER, Independent, Germany
PHILIPPA GARDNER, Imperial College London, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom
MATIJA PRETNAR, University of Ljubljana, Slovenia
XIAOJIA RAO, Imperial College London, United Kingdom
CONRAD WATT, University of Cambridge, United Kingdom
ANDREAS ROSSBERG, Independent, Germany

WebAssembly (Wasm) is a portable low-level bytecode language and virtual machine that has seen increasing use in a variety of ecosystems. Its specification is unusually rigorous – including a full formal semantics for the language – and every new feature must be specified in this formal semantics, in prose, and in the official reference interpreter before it can be standardized. With the growing size of the language, this manual process with its redundancies has become laborious and error-prone, and in this work, we offer a solution.

We present SpecTec, a domain-specific language (DSL) and toolchain that facilitates both the Wasm specification and the generation of artifacts necessary to standardize new features. SpecTec serves as a single source of truth — from a SpecTec definition of the Wasm semantics, we can generate a typeset specification, including formal definitions and prose pseudocode descriptions, and a meta-level interpreter. Further backends for test generation and interactive theorem proving are planned. We evaluate SpecTec's ability to represent the latest Wasm 2.0 and show that the generated meta-level interpreter passes 100% of the applicable official test suite. We show that SpecTec is highly effective at discovering and preventing errors by detecting historical errors in the specification that have been corrected and ten errors in five proposals ready for inclusion in the next version of Wasm. Our ultimate aim is that SpecTec should be adopted by the Wasm standards community and used to specify future versions of the standard.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Software and its engineering** → **Syntax**; **Semantics**; **Specification languages**; **Domain specific languages**.

Additional Key Words and Phrases: WebAssembly, language specification, executable prose, DSL

ACM Reference Format:
Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (June 2024), 26 pages. https://doi.org/10.1145/3656440

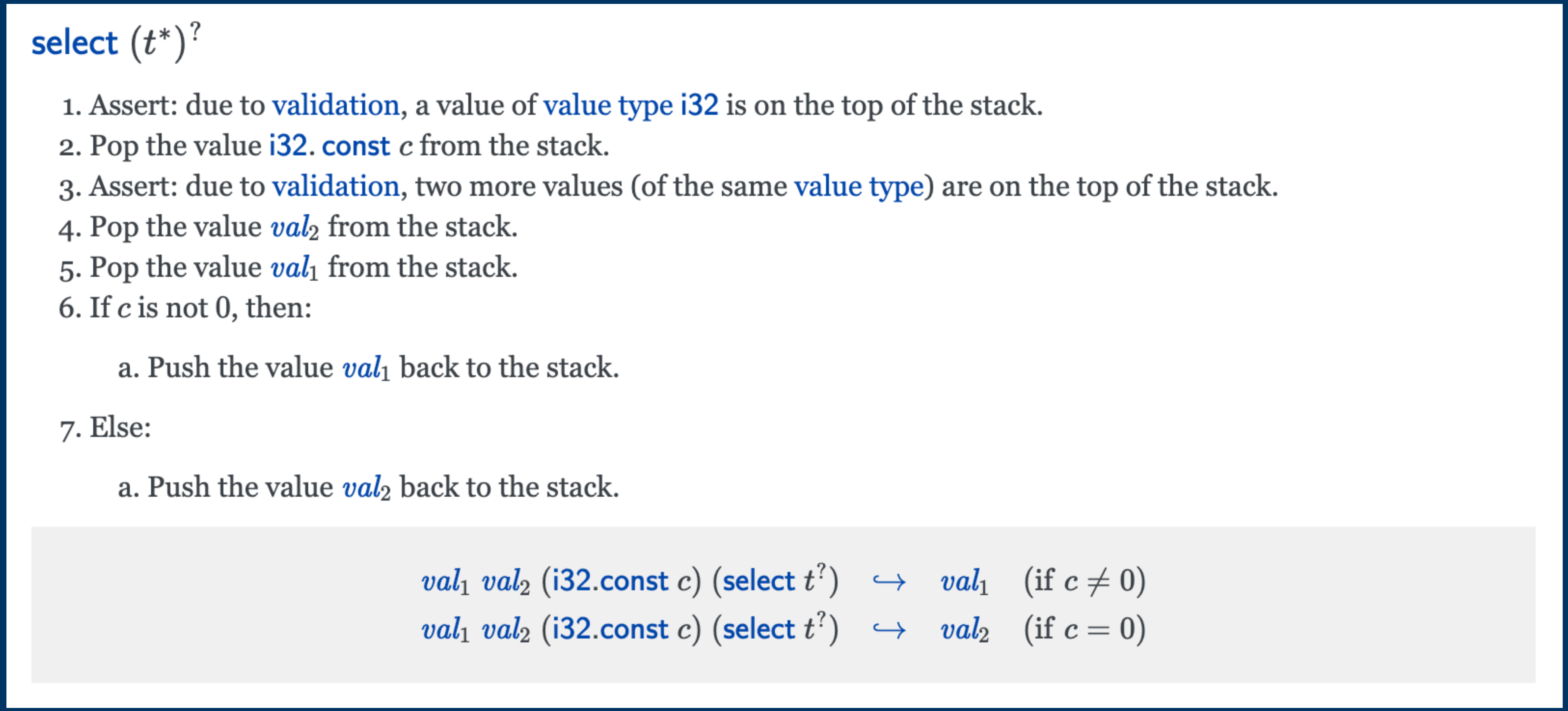
Authors' addresses: Dongjun Youn, f52985@kaist.ac.kr; Wonho Shin, new170527@kaist.ac.kr; Jaehyun Lee, 99jaehyunlee@kaist.ac.kr; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea; Joachim Breitner, mail@joachim-breitner.de, Independent, Freiburg, Germany; Philippa Gardner, p.gardner@imperial.ac.uk, Imperial College London, London, United Kingdom; Sam Lindley, Sam.Lindley@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Matija Pretnar, matija.pretnar@fmf.uni-lj.si, University of Ljubljana, Ljubljana, Slovenia; Xiaojia Rao, xiaojia.rao19@imperial.ac.uk, Imperial College London, London, United Kingdom; Conrad Watt, conrad.watt@cl.cam.ac.uk, University of Cambridge, Cambridge, United Kingdom; Andreas Rossberg, rossberg@mpi-sws.org, Independent, Munich, Germany.

ACM 2475-1421/2024/6-ART210
https://doi.org/10.1145/3656440

https://people.mpi-sws.org/~rossberg/papers/spectec1.pdf

	syntax	validation	execution	binary	text	appendix
conventions	🟩	🟩	🟩	🟩	🟥	🟥
types	🟩	🟨	🟩	🟩	🟥	
values	🟩	🟨	🟨	🟩	🟥	
instructions	🟩	🟨	🟩	🟩	🟥	
modules	🟩	🟨	🟨	🟧	🟥	
			🟥 numerics			

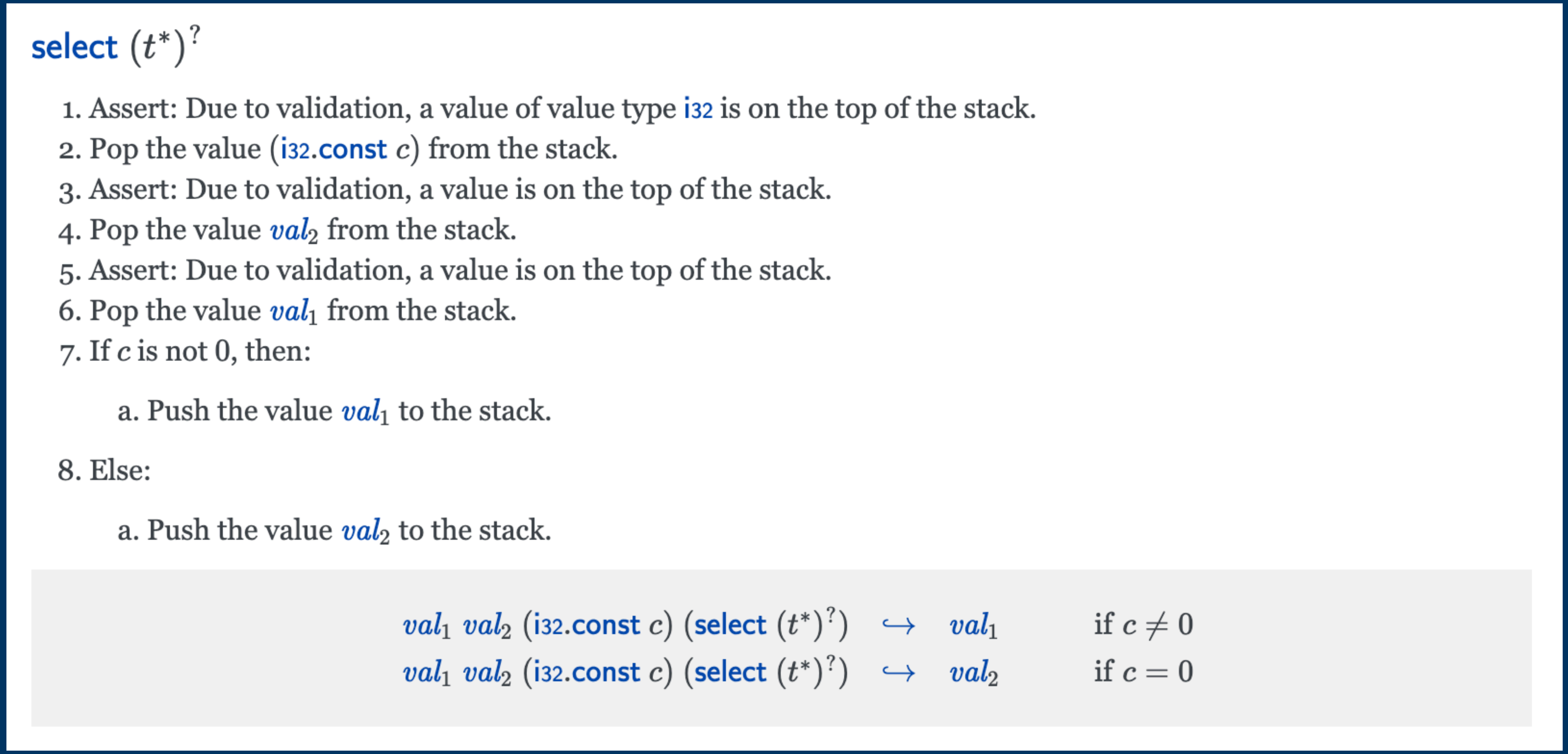
Wasm 2.0 + tail calls + multi memory + function references + garbage collection



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.

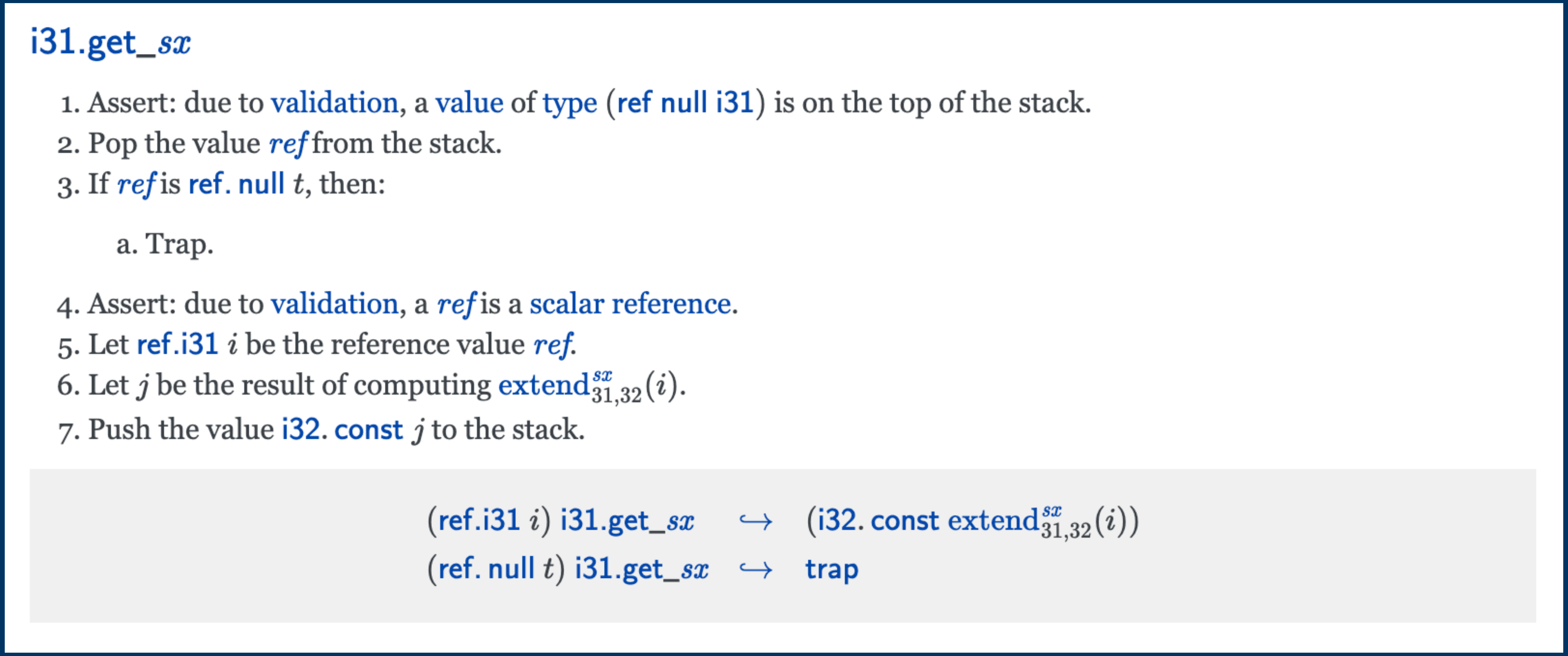
$$\begin{array}{llll} val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_1 & (\text{if } c \neq 0) \\ val_1\ val_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) & \hookrightarrow & val_2 & (\text{if } c = 0) \end{array}$$



select $(t^*)^?$

1. Assert: Due to validation, a value of 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, 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 is not 0, then:
   a. Push the value val_1 to the stack.
8. Else:
   a. Push the value val_2 to the stack.

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



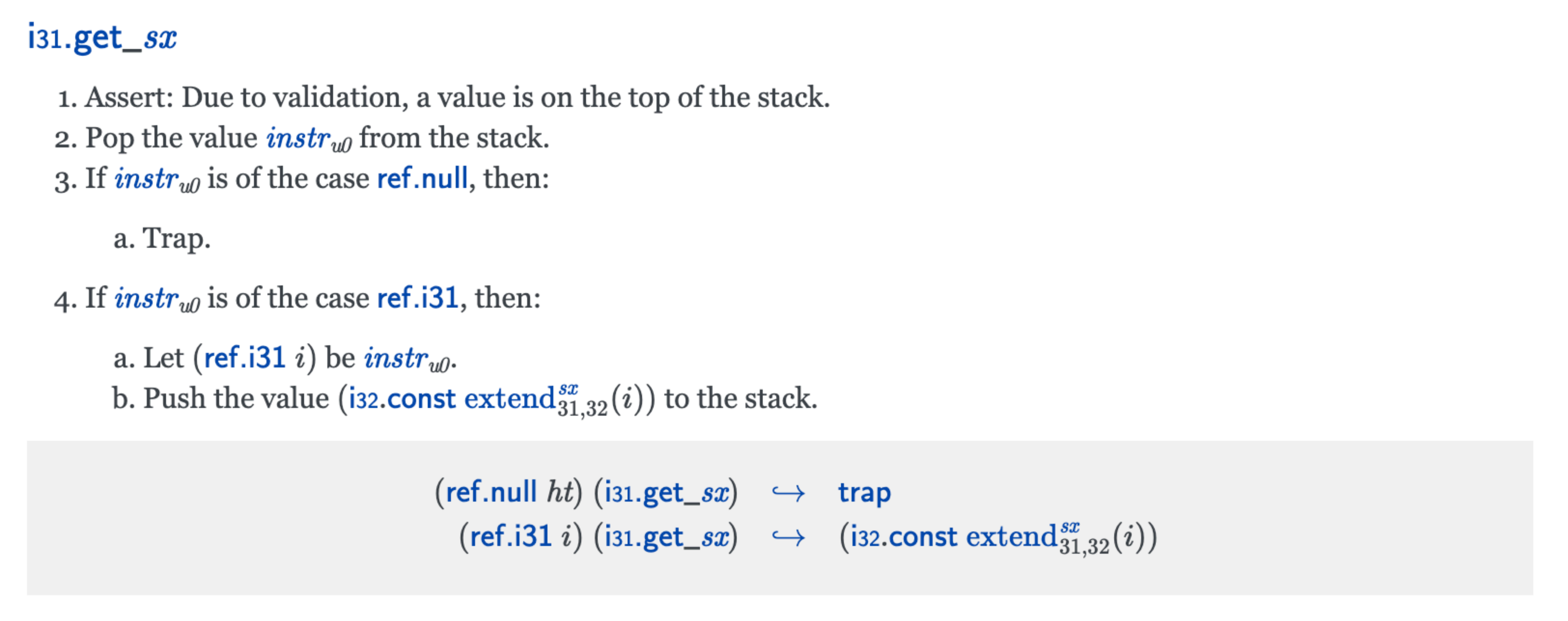
i31.get_sx

1. Assert: due to validation, a value of type (ref null i31) is on the top of the stack.
2. Pop the value ref from the stack.
3. If ref is ref. null t, then:

   a. Trap.

4. Assert: due to validation, a ref is a scalar reference.
5. Let ref.i31 i be the reference value ref.
6. Let j be the result of computing $\mathrm{extend}^{sx}_{31,32}(i)$.
7. Push the value i32. const j to the stack.

$$\begin{array}{lcl} (\mathsf{ref.i31}\ i)\ \mathsf{i31.get\_}sx & \hookrightarrow & (\mathsf{i32.const}\ \mathrm{extend}^{sx}_{31,32}(i)) \\ (\mathsf{ref.null}\ t)\ \mathsf{i31.get\_}sx & \hookrightarrow & \mathsf{trap} \end{array}$$



i31.get_sx

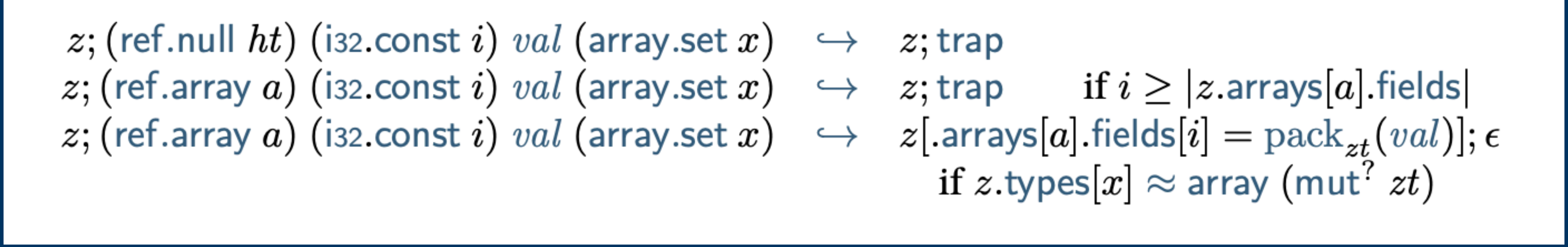
1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value $instr_{u0}$ from the stack.
3. If $instr_{u0}$ is of the case ref.null, then:

   a. Trap.

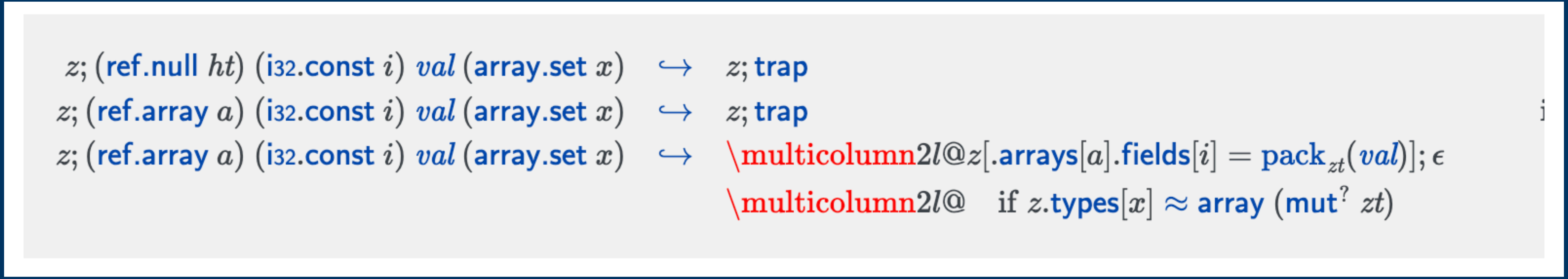
4. If $instr_{u0}$ is of the case ref.i31, then:

   a. Let (ref.i31 i) be $instr_{u0}$.
   b. Push the value (i32.const $\mathrm{extend}^{sx}_{31,32}(i)$) to the stack.

$$\begin{array}{rcl} (\mathsf{ref.null}\ ht)\ (\mathsf{i31.get\_}sx) & \hookrightarrow & \mathsf{trap} \\ (\mathsf{ref.i31}\ i)\ (\mathsf{i31.get\_}sx) & \hookrightarrow & (\mathsf{i32.const}\ \mathrm{extend}^{sx}_{31,32}(i)) \end{array}$$



$$\begin{array}{lcll}
z;(\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z;\mathsf{trap} & \\
z;(\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z;\mathsf{trap} & \text{if } i \geq |z.\mathsf{arrays}[a].\mathsf{fields}| \\
z;(\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & \multicolumn{2}{l}{z[.\mathsf{arrays}[a].\mathsf{fields}[i] = \mathrm{pack}_{zt}(val)];\epsilon} \\
 & & \multicolumn{2}{l}{\quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt)}
\end{array}$$



$$\begin{array}{lcll}
z;(\mathsf{ref.null}\ ht)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z;\mathsf{trap} & \\
z;(\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & z;\mathsf{trap} & \text{i} \\
z;(\mathsf{ref.array}\ a)\ (\mathsf{i32.const}\ i)\ val\ (\mathsf{array.set}\ x) & \hookrightarrow & \backslash\mathrm{multicolumn}2l@z[.\mathsf{arrays}[a].\mathsf{fields}[i] = \mathsf{pack}_{zt}(val)];\epsilon & \\
 & & \backslash\mathrm{multicolumn}2l@\quad \text{if } z.\mathsf{types}[x] \approx \mathsf{array}\ (\mathsf{mut}^?\ zt) &
\end{array}$$

Current Limitations

burndown list: https://github.com/wasm-dsl/spectec/issues/67

user experience still is rather rough

especially wrt robustness and error reporting in backends

output still needs some polishing

tutorial and documentation is WIP

… best intro for now is overview in the PLDI paper

Limitations of Math Renderer

layout control is somewhat whacky

hard-codes knowledge about some notation of relations

… recognises $\vdash$ and $\rightsquigarrow$ to decide rendering, falls back otherwise

Current Limitations of Prose Renderer

conditionals that should be assertions (guaranteed by validation)

…leads to spurious nesting

…interpreter just falls through on missing rules instead of aborting

produced phrasing not always ideal

…naming of generated meta-variables

…noise from helper functions that should be “inlined”

…iterations not unpacked

…missing special cases (e.g., “absent” instead “has length 0” for $x^?$)

hard-codes various definitional choices that can break when spec changes

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

On the up side

found and eliminated various spec bugs

  syntax errors, type errors, omissions, hyperlinks, layout, typos, …

  … detected either by SpecTec error checking

  … or prevented by construction thanks to generation

more consistent style and layout, without manual tuning

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

much faster turn-around for spec'ing new features (when it works…)

it's more fun :)

Open Questions

Better trade-off between WYSIWYG and ambiguity / frontend complexity?

Evaluation contexts (currently, we avoid them)

Reducing amount of hard-coded assumptions in backends

More self-contained meta interpreter

Meta-theory of SpecTec

Practical Questions

Hierarchy of amount of assumptions about s

frontend < Latex backend < prose back

Latter parts more likely to break on spec cha

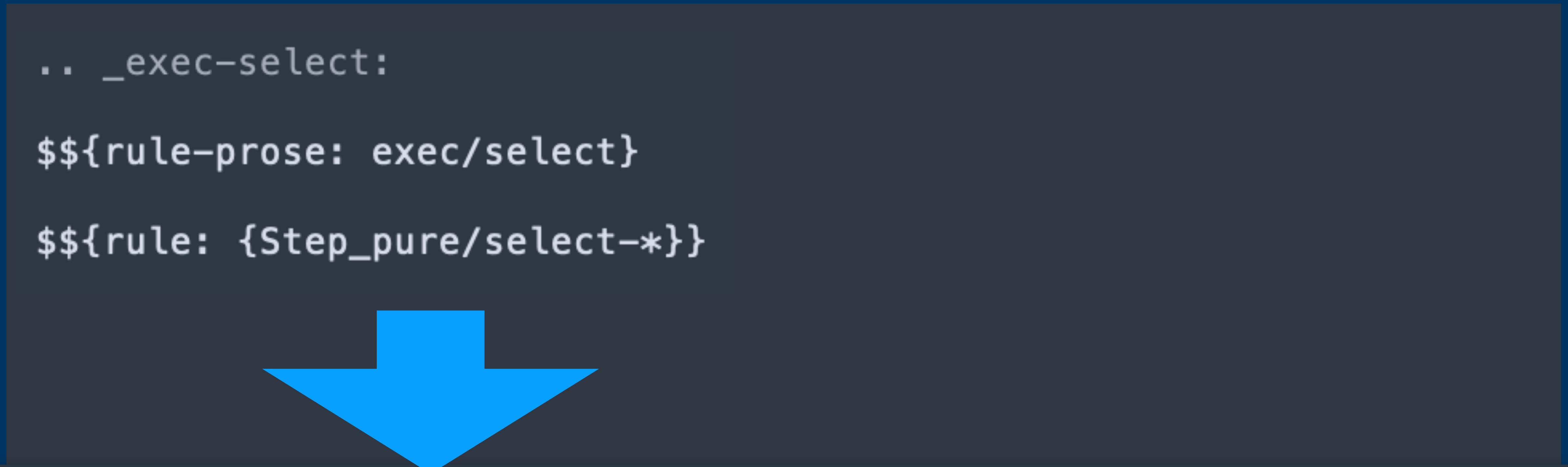
… interpreter particularly brittle and has to

Need process and infrastructure that allow t

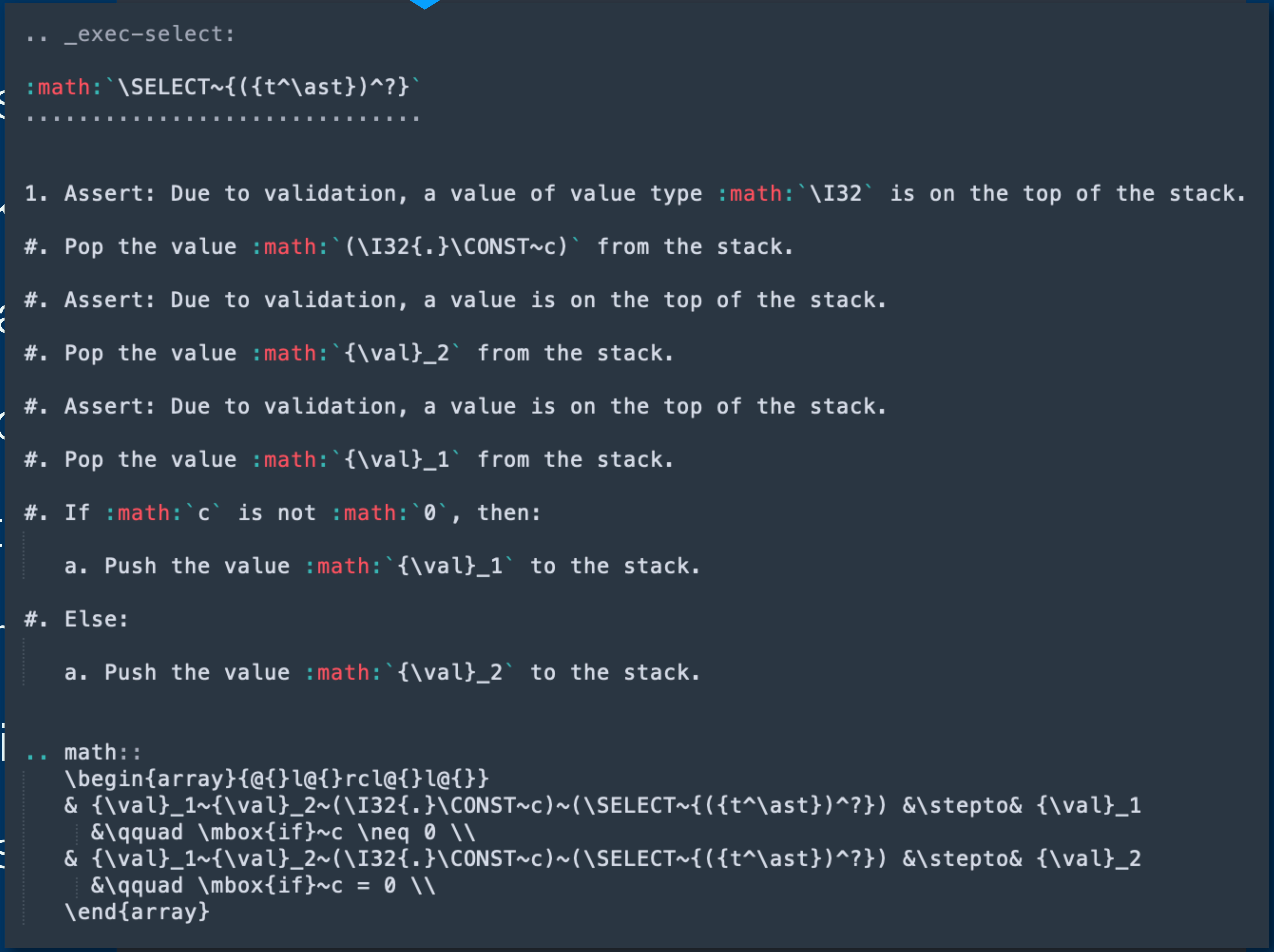
Contingency plan with permanent fallback ir

… interpreter not on critical path, could si

… Latex & prose backend can generate s



```
.. _exec-select:

$${rule-prose: exec/select}

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



```
.. _exec-select:

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

1. Assert: Due to validation, a value of value type :math:`\I32` is on the top of the stack.

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

#. Assert: Due to validation, a value is on the top of the stack.

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

#. Assert: Due to validation, a value is on the top of the stack.

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

#. If :math:`c` is not :math:`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}{@{}l@{}rcl@{}l@{}}
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) &\stepto& {\val}_1
     &\qquad \mbox{if}~c \neq 0 \\
   & {\val}_1~{\val}_2~(\I32{.}\CONST~c)~(\SELECT~{({t^\ast})^?}) &\stepto& {\val}_2
     &\qquad \mbox{if}~c = 0 \\
   \end{array}
```

Levels of Risks

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

   ⇒ we are close, and already 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

What we offer

The current team will finish converting the “3.0” spec

… ETA is ≈6 months

… that includes everything in burndown list on https://github.com/Wasm-DSL/spectec/issues/67

… and most stage 4 proposals (biggest current omission: threads)

The current team will continue to incrementally document, improve/fix, and extend the tool

… including polishing prose output

… improving the user experience

… adding yet missing features such as support for specifying numerics

The current team will be available to maintain the tool for at least 2 years

… and help with knowledge transfer where necessary for a future hand-over

We will invest into building theorem prover backends

… especially porting the existing Coq development and taking it beyond Wasm 2.0

Longer-term

Theorem prover backends, especially Coq

Generate additional parts of the spec document

- …macros, cross-reference anchors, indices

Handle more aspects of the spec in meta interpreter

- …decoding, parsing, validation

Meta interpreter for declarative semantics

Semantics-guided test fuzzing

Why CG sign-off matters for the project

Resources: more than 1 year of engineering work so far; justify further time investment by knowing that it is going to matter.

Impact: applied research has been (and will be) involved; needs some practical impact to be valued and considered publication-worthy.

Funding: some of us need to find funding for continuing this project; for the grant game it again is (even more) important to demonstrate relevance and impact.

Momentum: well-defined semantics has been key to the academic community's growing interest in Wasm; formalisation has to keep up with the language to preserve momentum and keep research relevant to the Wasm community.

Poll

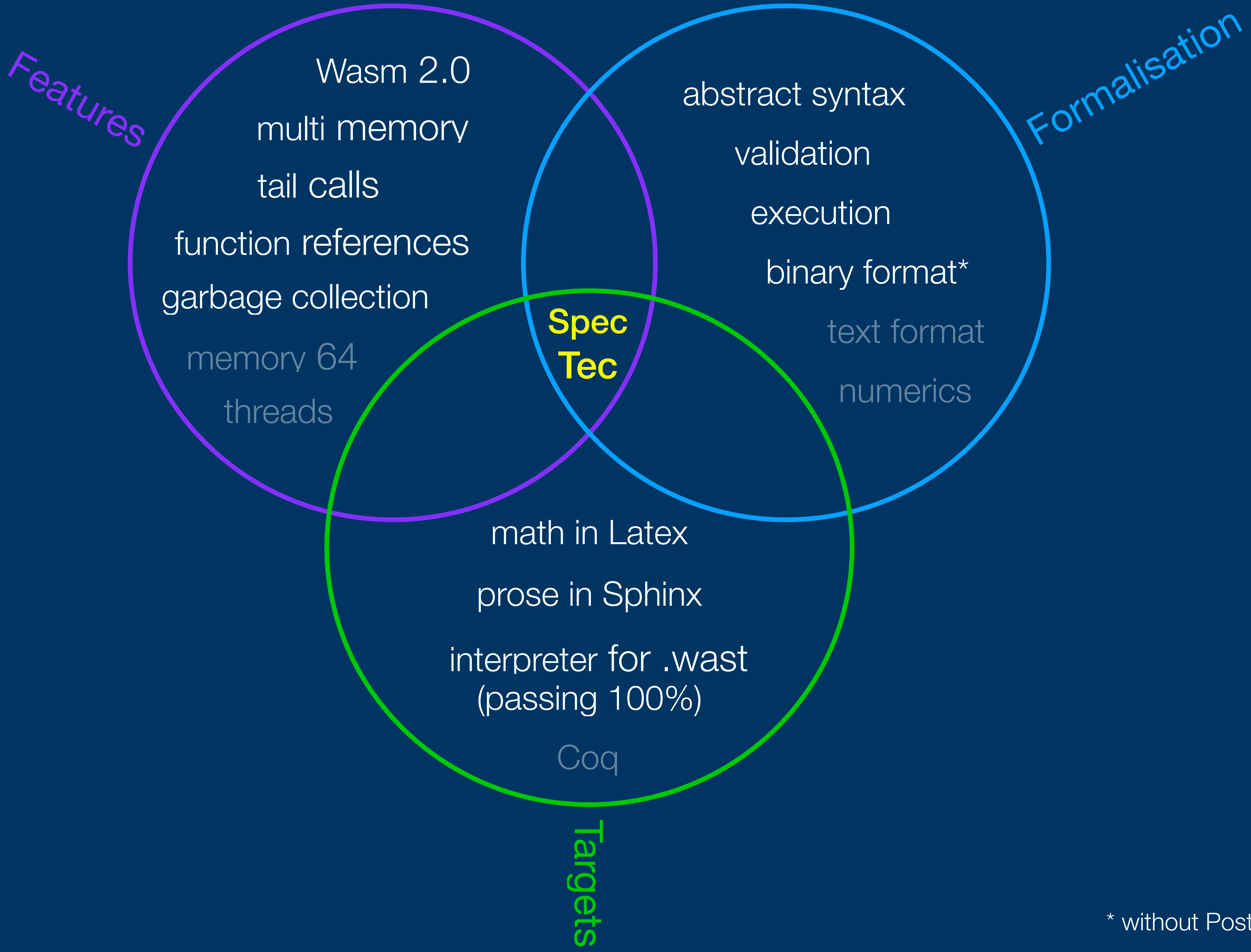
Adopt SpecTec (once it’s ready) as the toolchain for authoring the spec.

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

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

Paper: https://mpi-sws.org/~rossberg/papers/spectec1.pdf

Outtakes



Features

Wasm 2.0

multi memory

tail calls

function references

garbage collection

memory 64

threads

Formalisation

abstract syntax

validation

execution

binary format*

text format

numerics

Spec
Tec

math in Latex

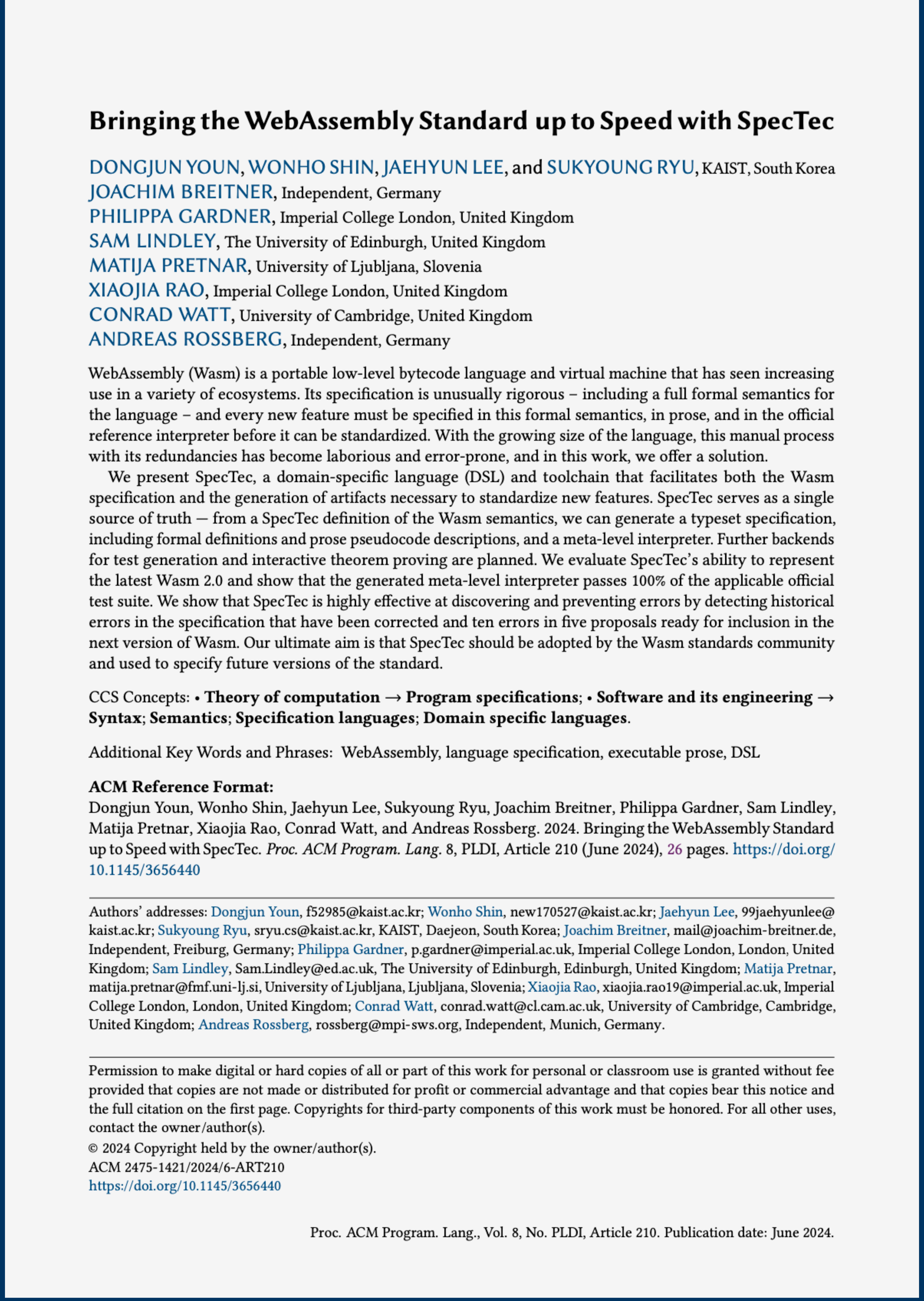
prose in Sphinx

interpreter for .wast
(passing 100%)

Coq

Targets

* without Post-2.0 syntax



Bringing the WebAssembly Standard up to Speed with SpecTec

DONGJUN YOUN, WONHO SHIN, JAEHYUN LEE, and SUKYOUNG RYU, KAIST, South Korea
JOACHIM BREITNER, Independent, Germany
PHILIPPA GARDNER, Imperial College London, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom
MATIJA PRETNAR, University of Ljubljana, Slovenia
XIAOJIA RAO, Imperial College London, United Kingdom
CONRAD WATT, University of Cambridge, United Kingdom
ANDREAS ROSSBERG, Independent, Germany

WebAssembly (Wasm) is a portable low-level bytecode language and virtual machine that has seen increasing use in a variety of ecosystems. Its specification is unusually rigorous – including a full formal semantics for the language – and every new feature must be specified in this formal semantics, in prose, and in the official reference interpreter before it can be standardized. With the growing size of the language, this manual process with its redundancies has become laborious and error-prone, and in this work, we offer a solution.

We present SpecTec, a domain-specific language (DSL) and toolchain that facilitates both the Wasm specification and the generation of artifacts necessary to standardize new features. SpecTec serves as a single source of truth — from a SpecTec definition of the Wasm semantics, we can generate a typeset specification, including formal definitions and prose pseudocode descriptions, and a meta-level interpreter. Further backends for test generation and interactive theorem proving are planned. We evaluate SpecTec's ability to represent the latest Wasm 2.0 and show that the generated meta-level interpreter passes 100% of the applicable official test suite. We show that SpecTec is highly effective at discovering and preventing errors by detecting historical errors in the specification that have been corrected and ten errors in five proposals ready for inclusion in the next version of Wasm. Our ultimate aim is that SpecTec should be adopted by the Wasm standards community and used to specify future versions of the standard.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Software and its engineering** → **Syntax**; **Semantics**; **Specification languages**; **Domain specific languages**.

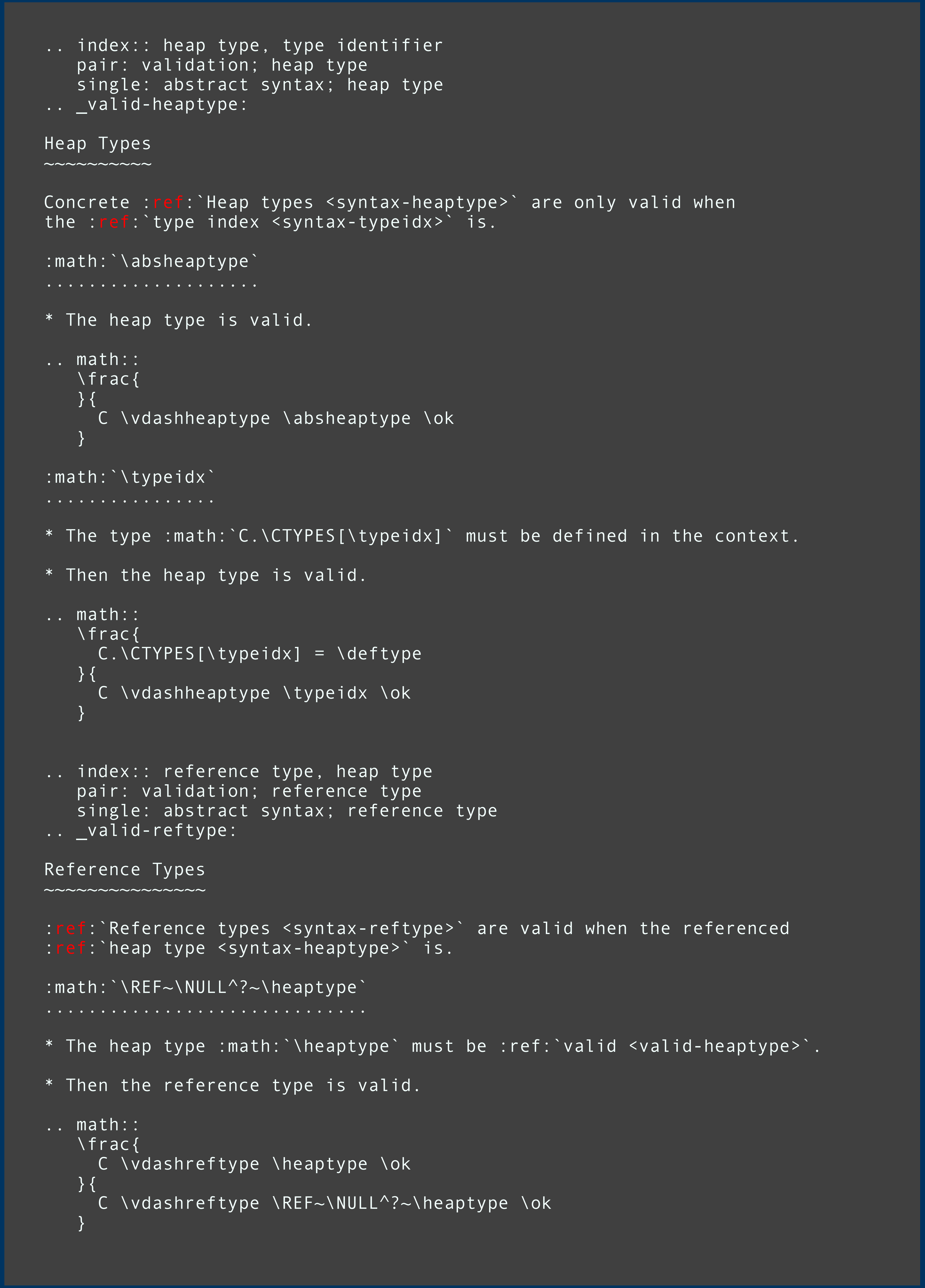
Additional Key Words and Phrases: WebAssembly, language specification, executable prose, DSL

ACM Reference Format:
Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (June 2024), 26 pages. https://doi.org/10.1145/3656440

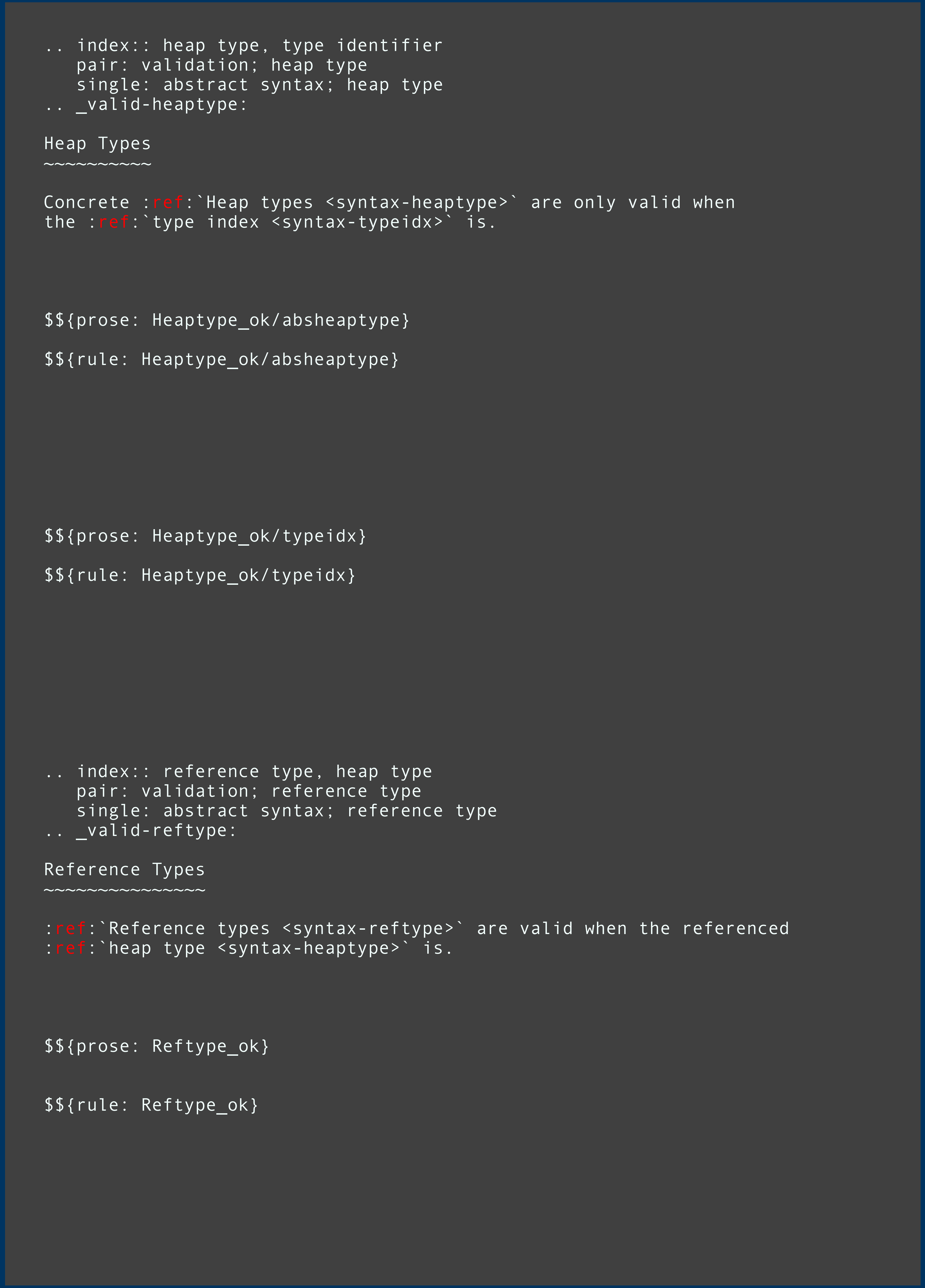
Authors' addresses: Dongjun Youn, f52985@kaist.ac.kr; Wonho Shin, new170527@kaist.ac.kr; Jaehyun Lee, 99jaehyunlee@kaist.ac.kr; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea; Joachim Breitner, mail@joachim-breitner.de, Independent, Freiburg, Germany; Philippa Gardner, p.gardner@imperial.ac.uk, Imperial College London, London, United Kingdom; Sam Lindley, Sam.Lindley@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Matija Pretnar, matija.pretnar@fmf.uni-lj.si, University of Ljubljana, Ljubljana, Slovenia; Xiaojia Rao, xiaojia.rao19@imperial.ac.uk, Imperial College London, London, United Kingdom; Conrad Watt, conrad.watt@cl.cam.ac.uk, University of Cambridge, Cambridge, United Kingdom; Andreas Rossberg, rossberg@mpi-sws.org, Independent, Munich, Germany.

ACM 2475-1421/2024/6-ART210
https://doi.org/10.1145/3656440

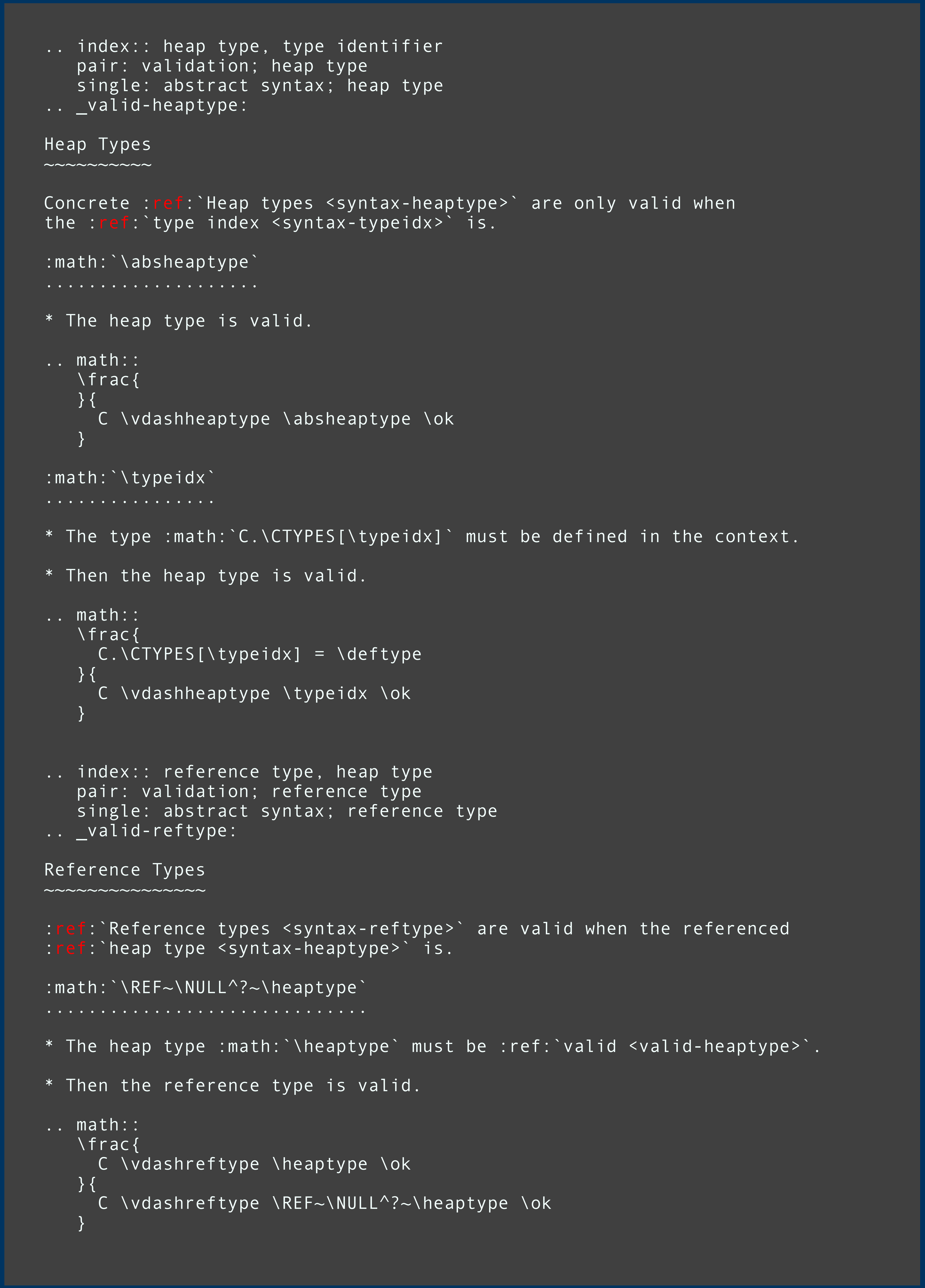
https://people.mpi-sws.org/~rossberg/papers/spectec1.pdf



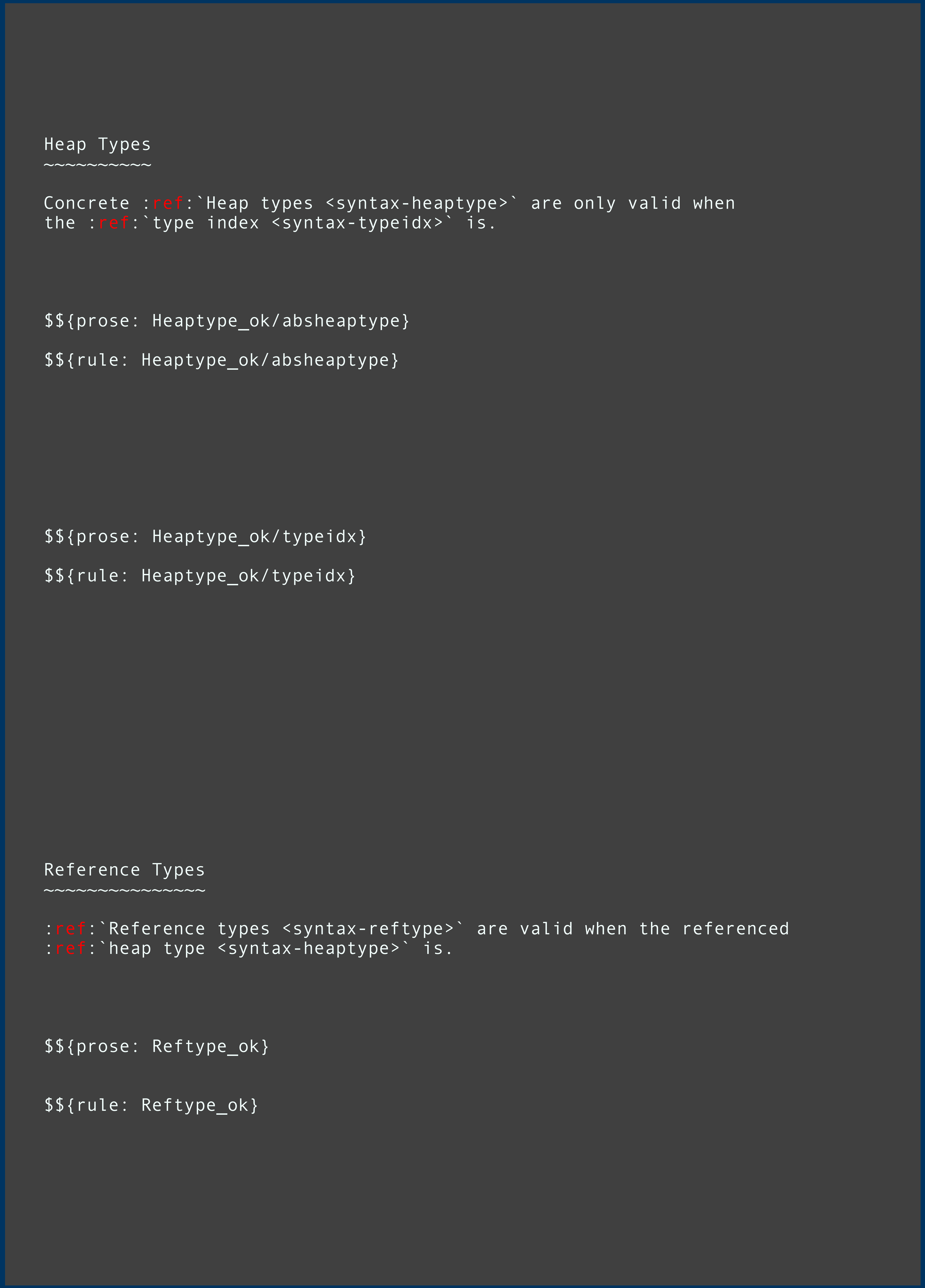
```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

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.



$${prose: Heaptype_ok/absheaptype}

$${rule: Heaptype_ok/absheaptype}









$${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.



$${prose: Reftype_ok}


$${rule: Reftype_ok}
```

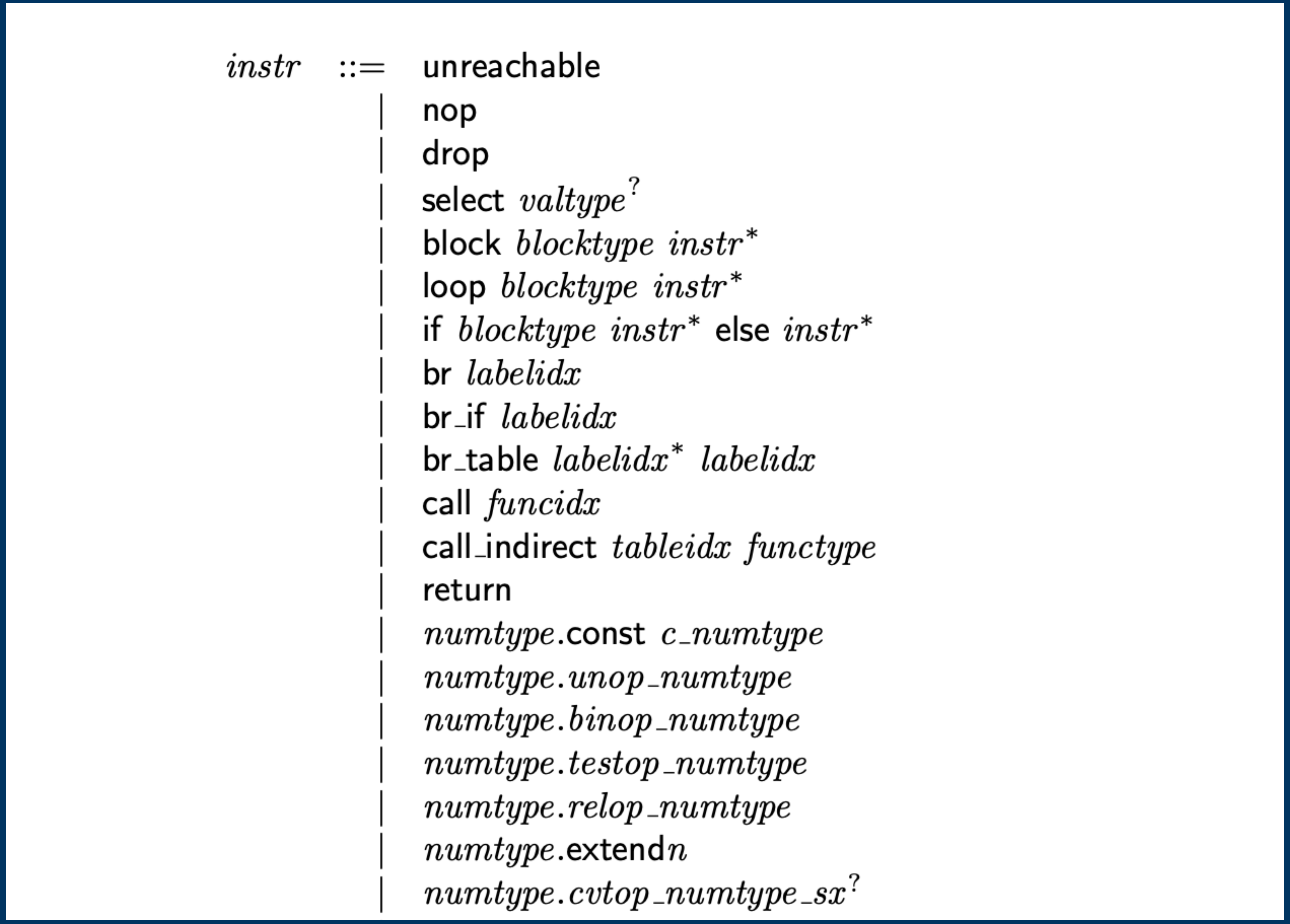


```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

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
   }
```



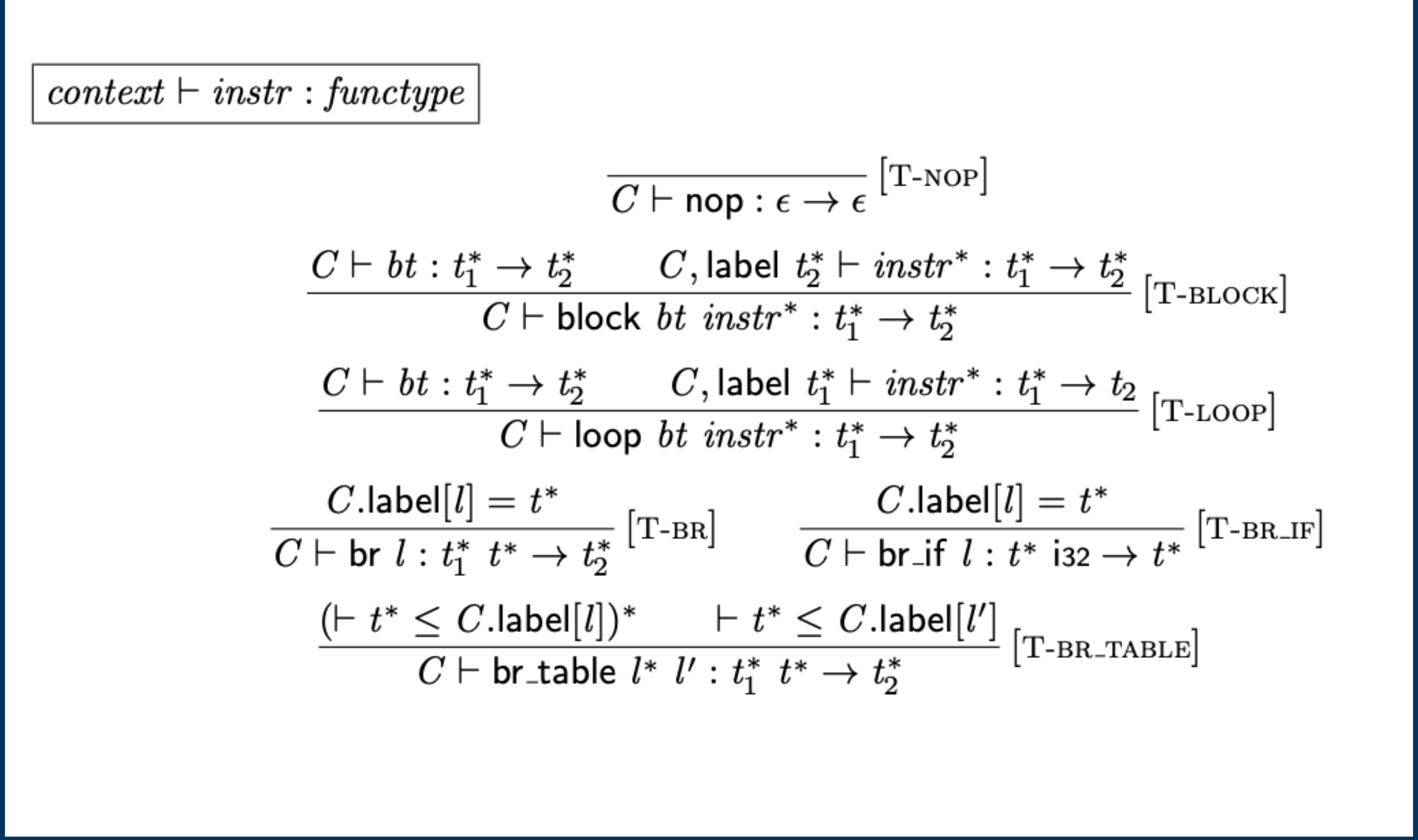
```
Heap Types
~~~~~~~~~~

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}
```

```
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 %.%#_#%#_#%)
    | ...
```



$$
\begin{array}{rrl}
\mathit{instr} & ::= & \mathsf{unreachable} \\
& | & \mathsf{nop} \\
& | & \mathsf{drop} \\
& | & \mathsf{select}\ \mathit{valtype}^? \\
& | & \mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^* \\
& | & \mathsf{loop}\ \mathit{blocktype}\ \mathit{instr}^* \\
& | & \mathsf{if}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^* \\
& | & \mathsf{br}\ \mathit{labelidx} \\
& | & \mathsf{br\_if}\ \mathit{labelidx} \\
& | & \mathsf{br\_table}\ \mathit{labelidx}^*\ \mathit{labelidx} \\
& | & \mathsf{call}\ \mathit{funcidx} \\
& | & \mathsf{call\_indirect}\ \mathit{tableidx}\ \mathit{functype} \\
& | & \mathsf{return} \\
& | & \mathit{numtype}.\mathsf{const}\ \mathit{c\_numtype} \\
& | & \mathit{numtype}.\mathit{unop\_numtype} \\
& | & \mathit{numtype}.\mathit{binop\_numtype} \\
& | & \mathit{numtype}.\mathit{testop\_numtype} \\
& | & \mathit{numtype}.\mathit{relop\_numtype} \\
& | & \mathit{numtype}.\mathsf{extend}\mathit{n} \\
& | & \mathit{numtype}.\mathit{cvtop\_numtype\_sx}^?
\end{array}
$$

```
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])*
```



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

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

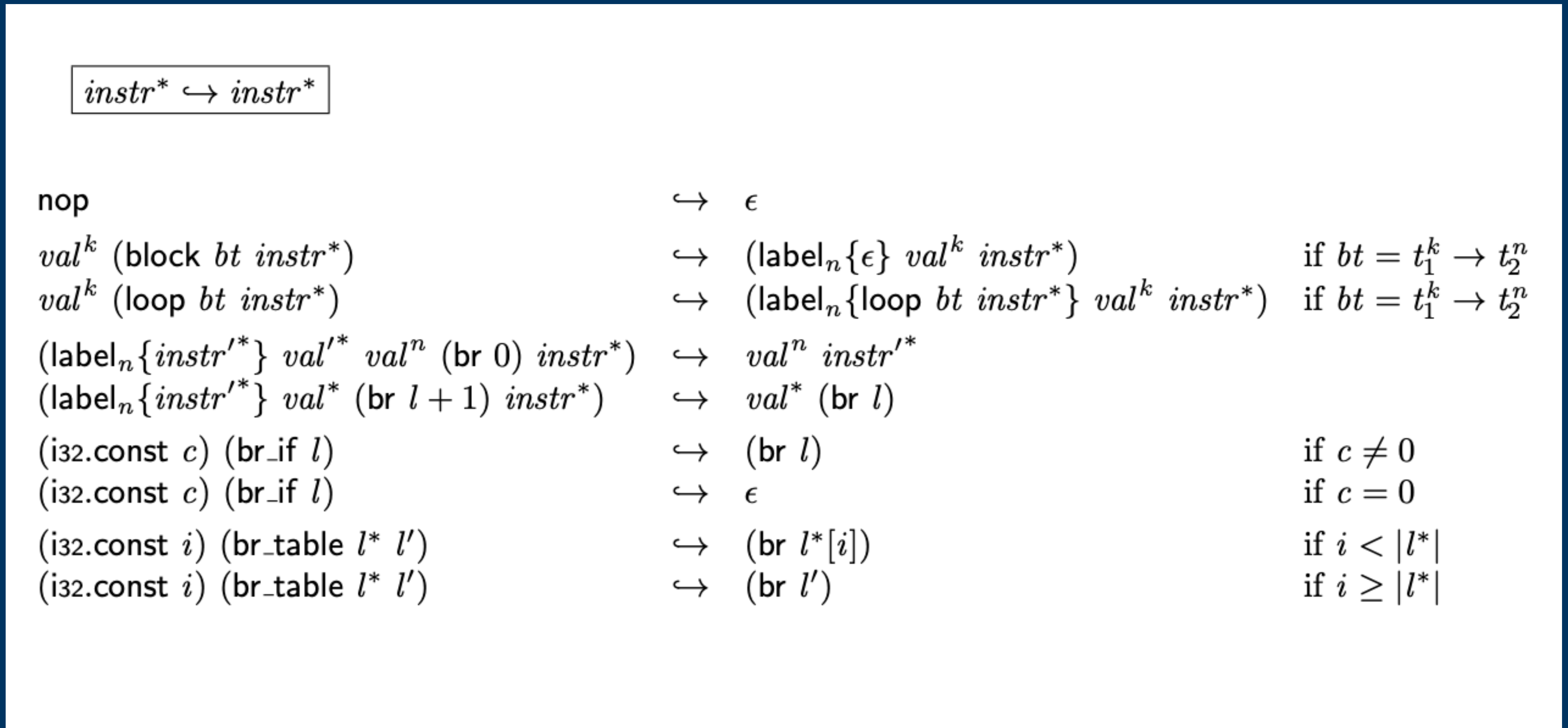
$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad C, \mathsf{label}\ t_2^* \vdash instr^* : t_1^* \to t_2^*}{C \vdash \mathsf{block}\ bt\ instr^* : t_1^* \to t_2^*}\ [\text{T-BLOCK}]$$

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

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

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

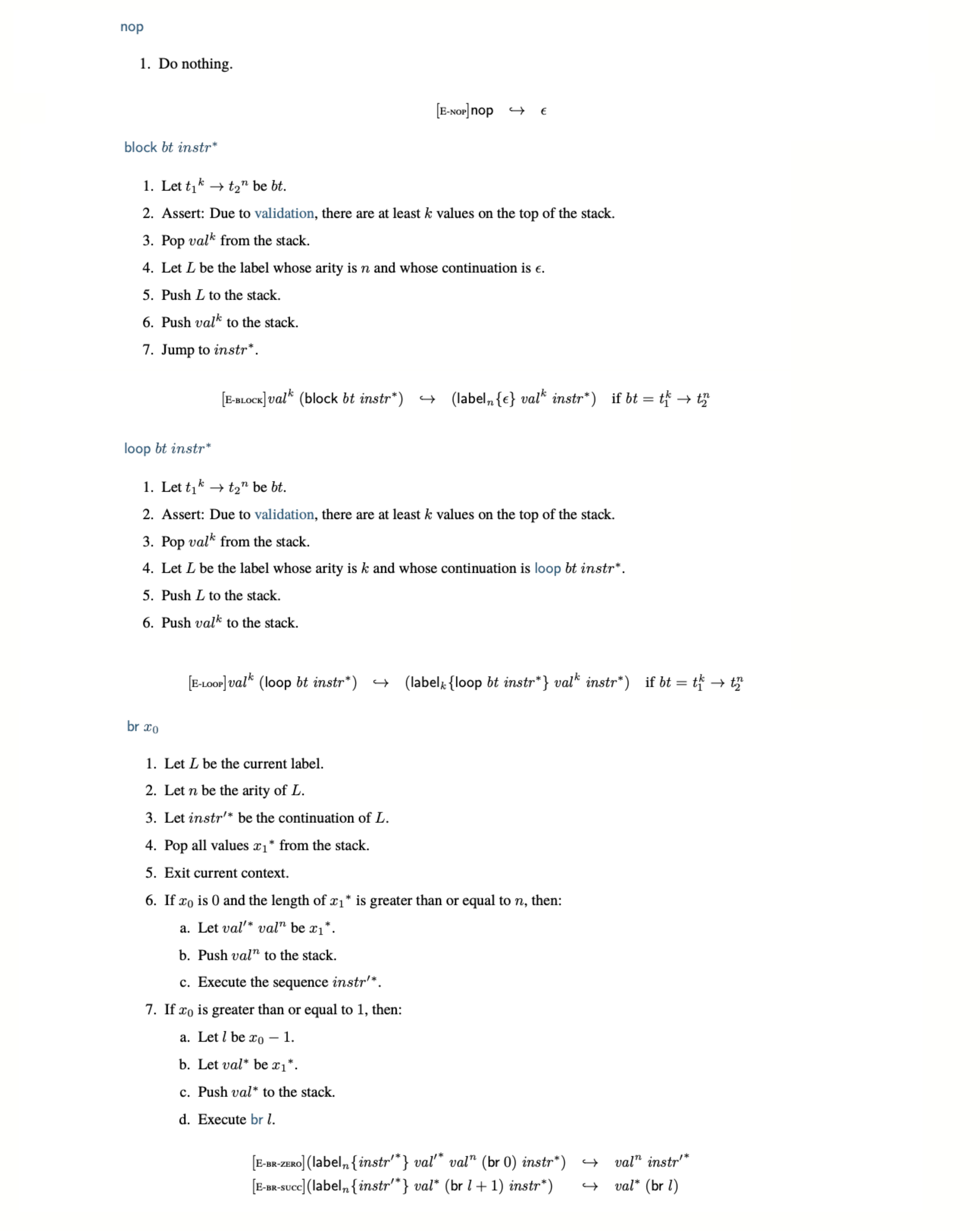
```
relation Step_pure: config ~> config

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*|
```



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

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

```
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.

$$[\textsc{E-nop}]\,\mathsf{nop} \quad \hookrightarrow \quad \epsilon$$

block bt $instr^*$

1. Let ${t_1}^k \rightarrow {t_2}^n$ 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 $instr^*$.

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

loop bt $instr^*$

1. Let ${t_1}^k \rightarrow {t_2}^n$ 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.

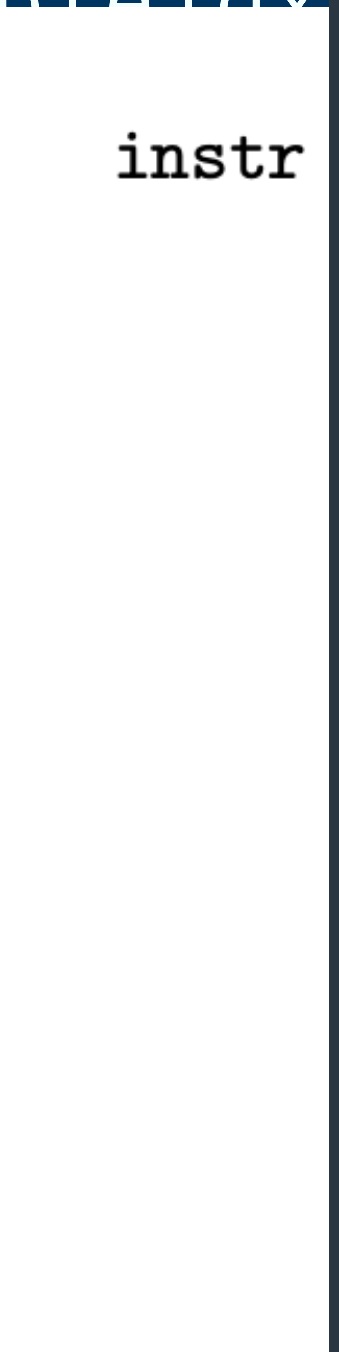
$$[\textsc{E-loop}]\,val^k\ (\mathsf{loop}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_k\{\mathsf{loop}\ bt\ instr^*\}\ val^k\ instr^*) \quad \text{if } bt = t_1^k \rightarrow t_2^n$$

br x_0

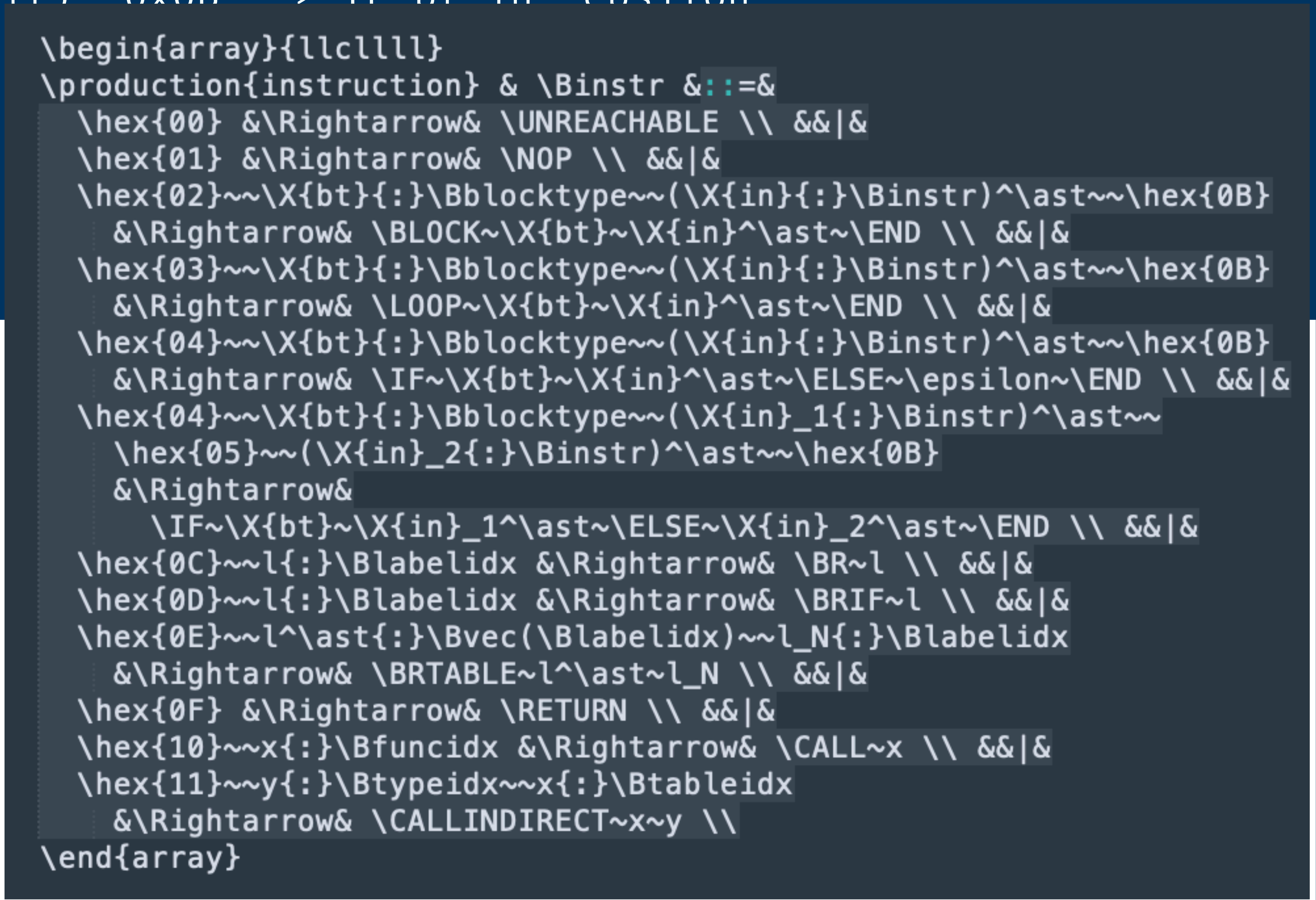
1. Let L be the current label.
2. Let n be the arity of L.
3. Let $instr'^*$ be the continuation of L.
4. Pop all values ${x_1}^*$ from the stack.
5. Exit current context.
6. If x_0 is 0 and the length of ${x_1}^*$ is greater than or equal to n, then:
   a. Let $val'^*\ val^n$ be ${x_1}^*$.
   b. Push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

$$\begin{array}{lcl} [\textsc{E-br-zero}](\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ 0)\ instr^*) & \hookrightarrow & val^n\ instr'^* \\ [\textsc{E-br-succ}](\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l+1)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l) \end{array}$$

```
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:Bi
  | 0x0C l:Blabelidx
  | 0x0D l:Blabelidx
  | 0x0E l*:Bvec(Blabelidx) l_N
  | 0x0F
  | 0x10 x:Bfuncidx
  | 0x11 y:Btypeidx x:Btableidx
  | ...
```



instr



```
\begin{array}{llclll}
\production{instruction} & \Binstr &::=&
  \hex{00} &\Rightarrow& \UNREACHABLE \\ &&|&
  \hex{01} &\Rightarrow& \NOP \\ &&|&
  \hex{02}~~\X{bt}{:}\Bblocktype~~(\X{in}{:}\Binstr)^\ast~~\hex{0B}
    &\Rightarrow& \BLOCK~\X{bt}~\X{in}^\ast~\END \\ &&|&
  \hex{03}~~\X{bt}{:}\Bblocktype~~(\X{in}{:}\Binstr)^\ast~~\hex{0B}
    &\Rightarrow& \LOOP~\X{bt}~\X{in}^\ast~\END \\ &&|&
  \hex{04}~~\X{bt}{:}\Bblocktype~~(\X{in}{:}\Binstr)^\ast~~\hex{0B}
    &\Rightarrow& \IF~\X{bt}~\X{in}^\ast~\ELSE~\epsilon~\END \\ &&|&
  \hex{04}~~\X{bt}{:}\Bblocktype~~(\X{in}_1{:}\Binstr)^\ast~~
    \hex{05}~~(\X{in}_2{:}\Binstr)^\ast~~\hex{0B}
    &\Rightarrow&
      \IF~\X{bt}~\X{in}_1^\ast~\ELSE~\X{in}_2^\ast~\END \\ &&|&
  \hex{0C}~~l{:}\Blabelidx &\Rightarrow& \BR~l \\ &&|&
  \hex{0D}~~l{:}\Blabelidx &\Rightarrow& \BRIF~l \\ &&|&
  \hex{0E}~~l^\ast{:}\Bvec(\Blabelidx)~~l_N{:}\Blabelidx
    &\Rightarrow& \BRTABLE~l^\ast~l_N \\ &&|&
  \hex{0F} &\Rightarrow& \RETURN \\ &&|&
  \hex{10}~~x{:}\Bfuncidx &\Rightarrow& \CALL~x \\ &&|&
  \hex{11}~~y{:}\Btypeidx~~x{:}\Btableidx
    &\Rightarrow& \CALLINDIRECT~x~y \\
\end{array}
```

Bugs found in spec and proposals

type errors

…missing immediates or record fields in rules

semantic errors

…missing stack operands, stack mishandling, index errors

prose errors

…unbound variables, missing steps

editorial errors

…syntax errors, typos, layout errors, wrong hyperlinks, …