SpecTec Update

Andreas Rossberg

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

2024/04/09

Recap: spec authoring today

Duplicate work to write *both* formal and prose rules

Writing prose is particularly laborious

Both terrible for code reviews

…Latex math is a write-only language

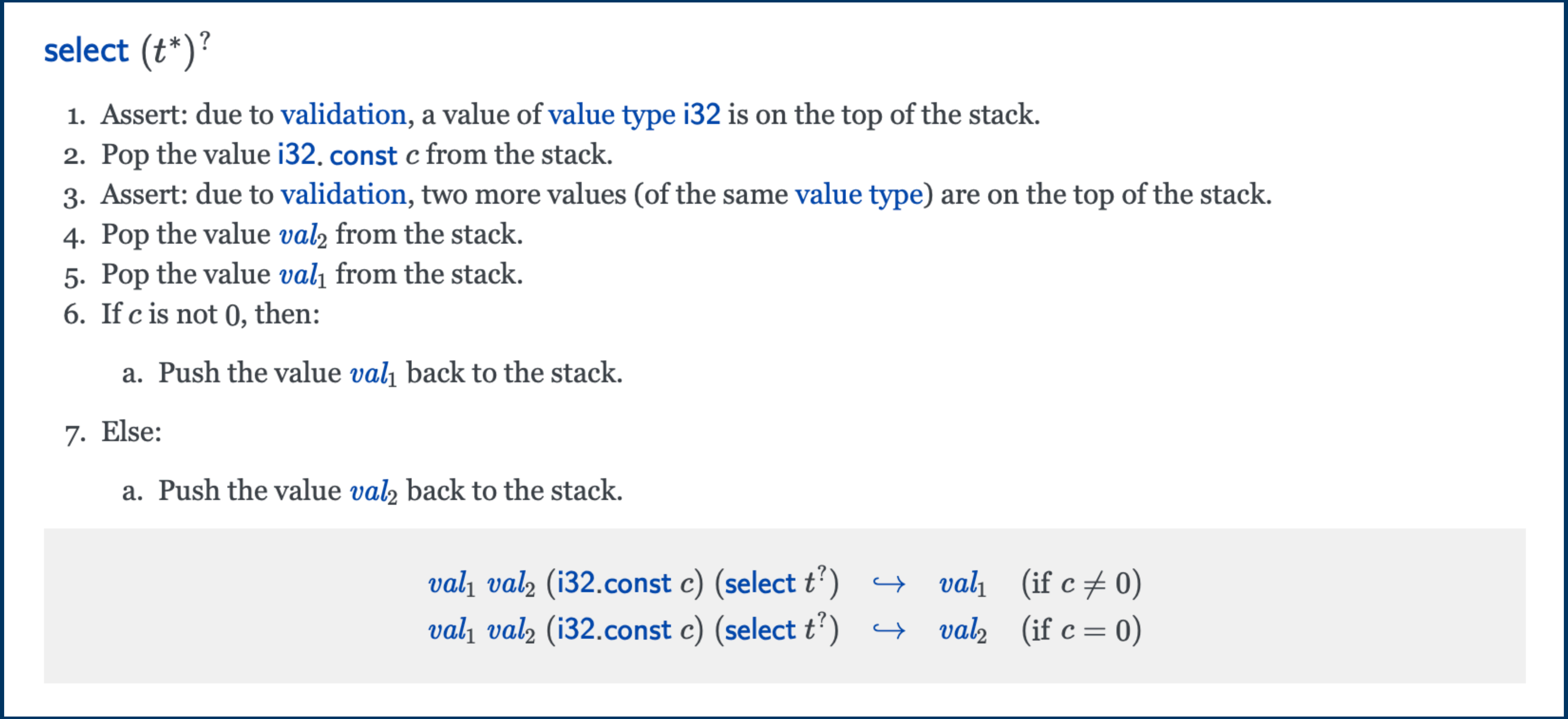
…reStructuredText is verbose, repetitive, and diff-unfriendly (e.g. tables, indentation!)

No macro facilities in Sphinx

…had to build hacky Python scripts and Sphinx plugins on top (broke repeatedly)

SpecTec

a DSL for authoring the Wasm spec



select $(t^*)^?$

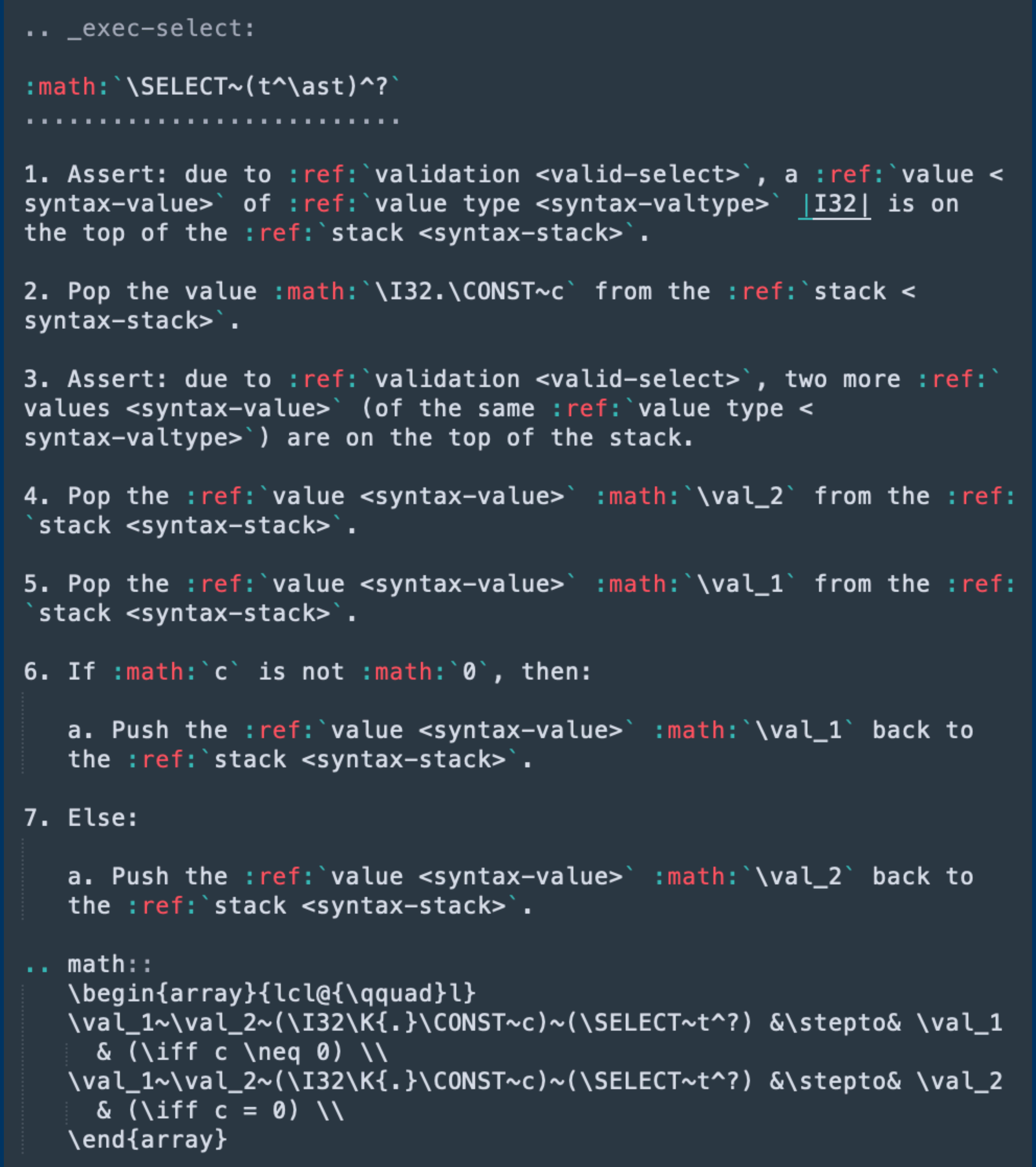
1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32. const c from the stack.
3. Assert: due to validation, two more values (of the same value type) are on the top of the stack.
4. Pop the value val_2 from the stack.
5. Pop the value val_1 from the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

   a. Push the value val_1 back to the stack.

7. Else:

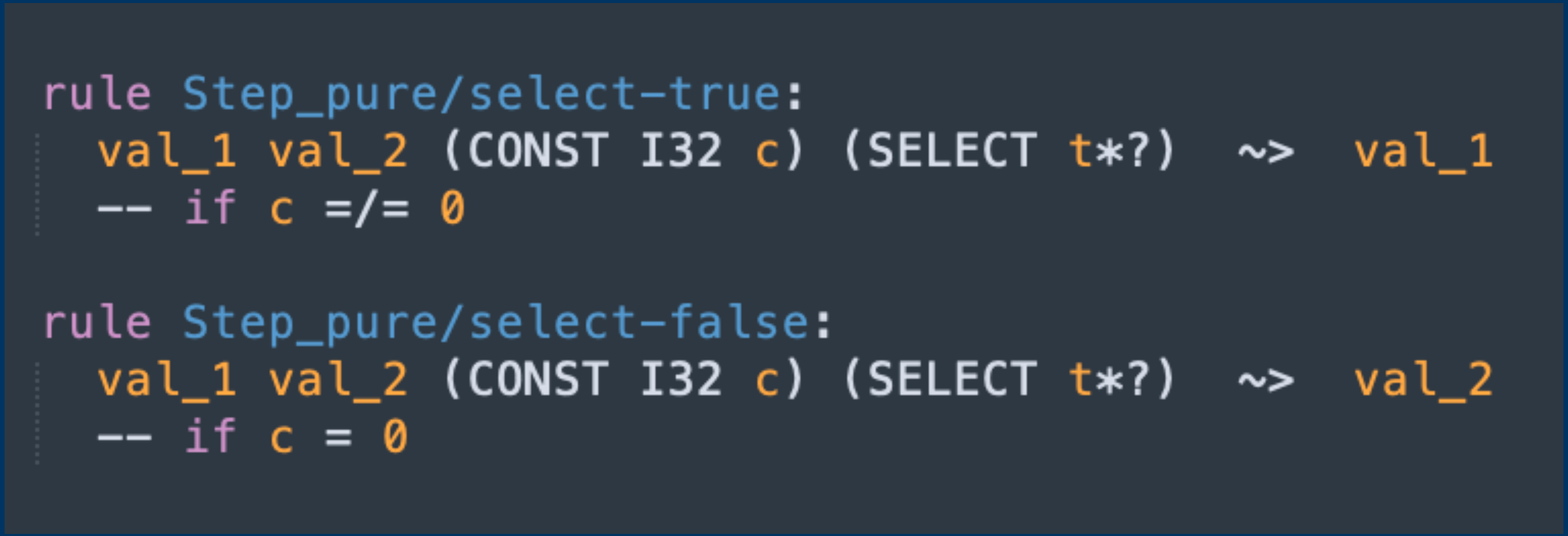
   a. Push the value val_2 back to the stack.

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



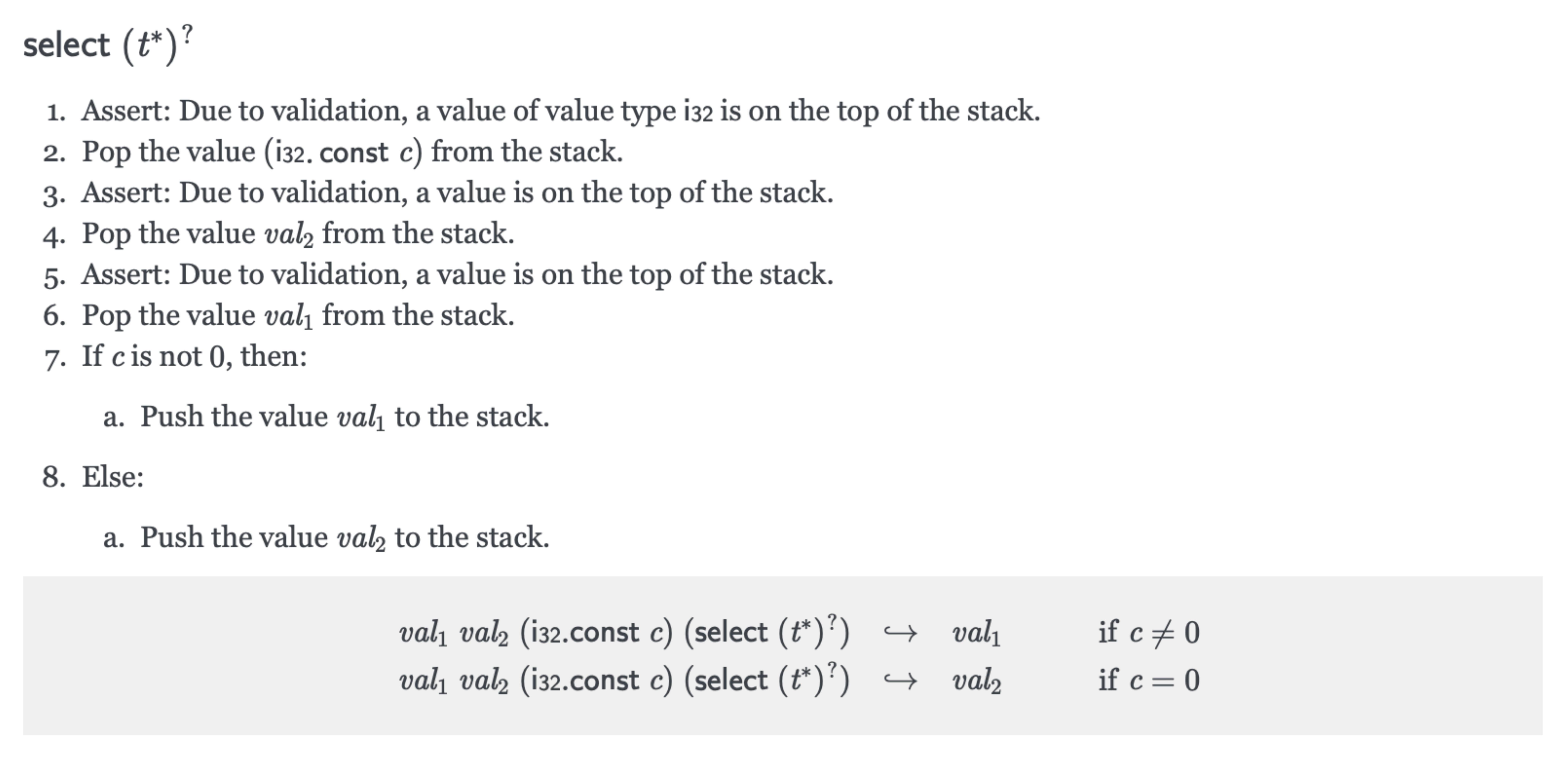
```
.. _exec-select:

: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}
```

execution/instructions.rst



```
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
```

spectec/reduction.watsup



select $(t^*)^?$

1. Assert: Due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $(\mathsf{i32.const}\ c)$ 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.

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

Recap: spec authoring with SpecTec

SpecTec is (mostly) WYSIWYG ASCII for the math rules

single source of truth for auto-generating:

…math in Latex

…prose in Sphinx

…mechanised Coq, Lean, Agda, Isabelle formalisations (future work)

Math and prose can be spliced into spec document

Plus, can run meta-level interpreter executing the prose on tests!

Meta Interpreter

algorithmic prose is derived from user-defined declarative reduction rules

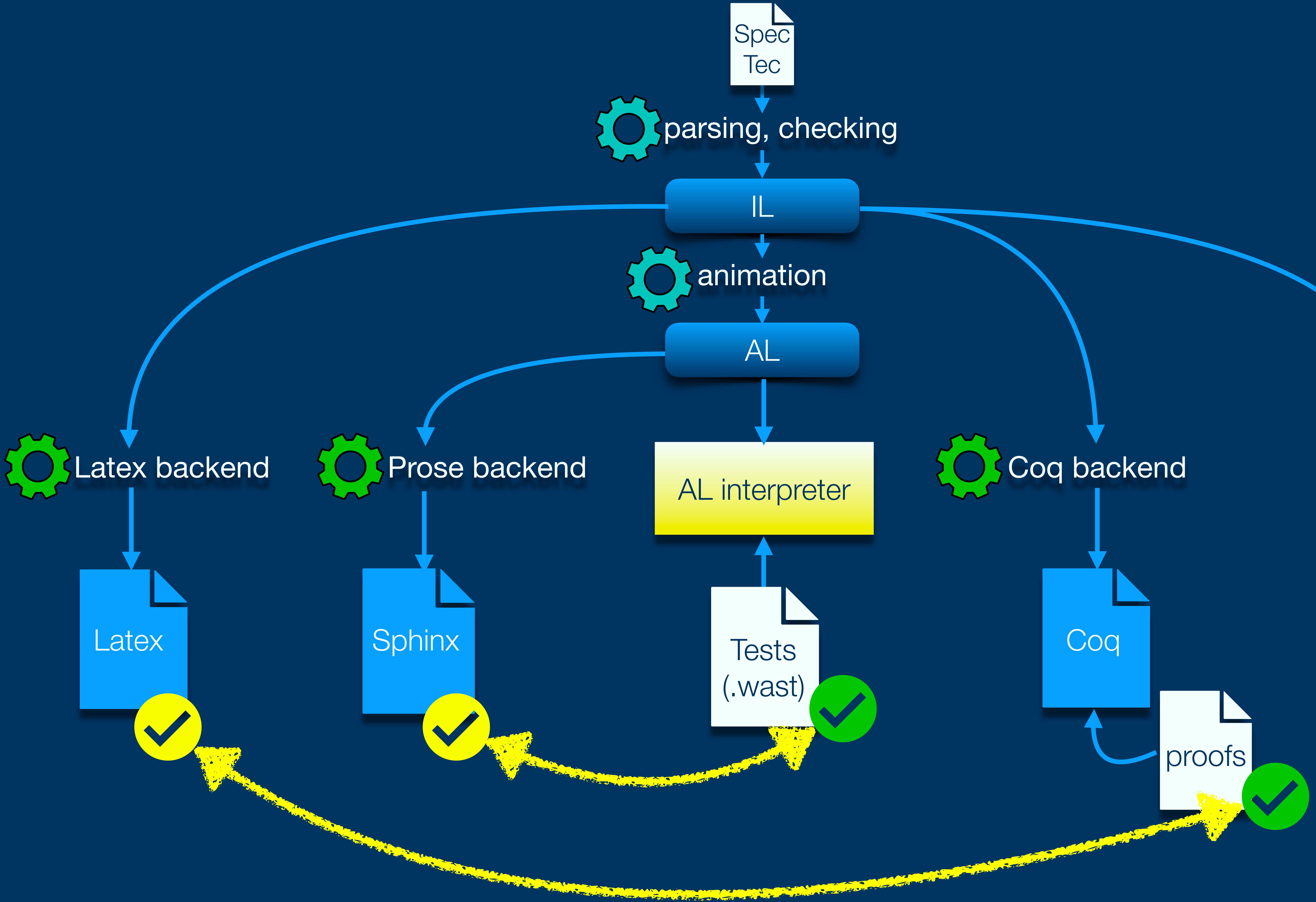
…via a transformation sometimes called *animation*

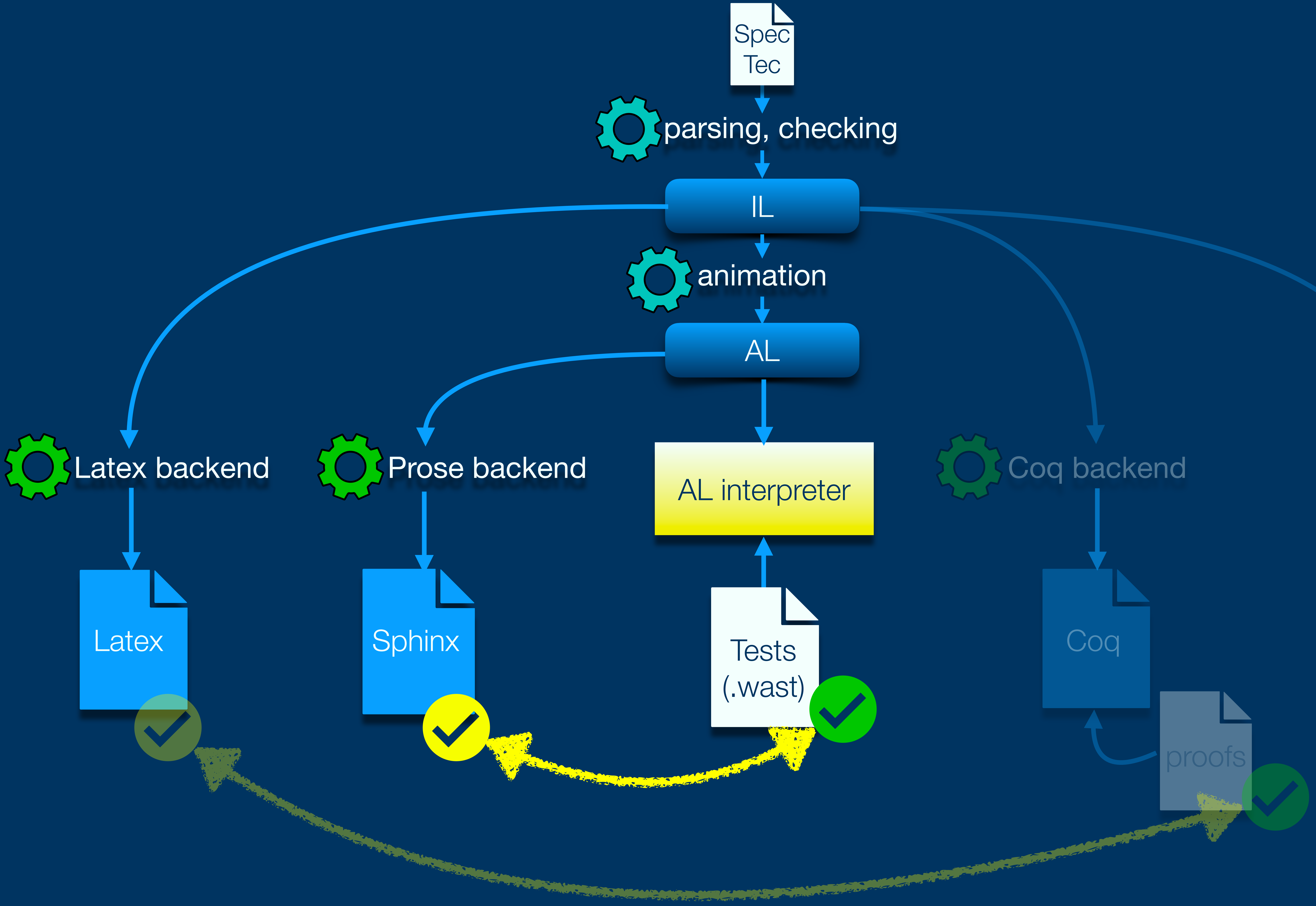
result is an IR called AL (algorithmic language)

…essentially, an AST for the prose

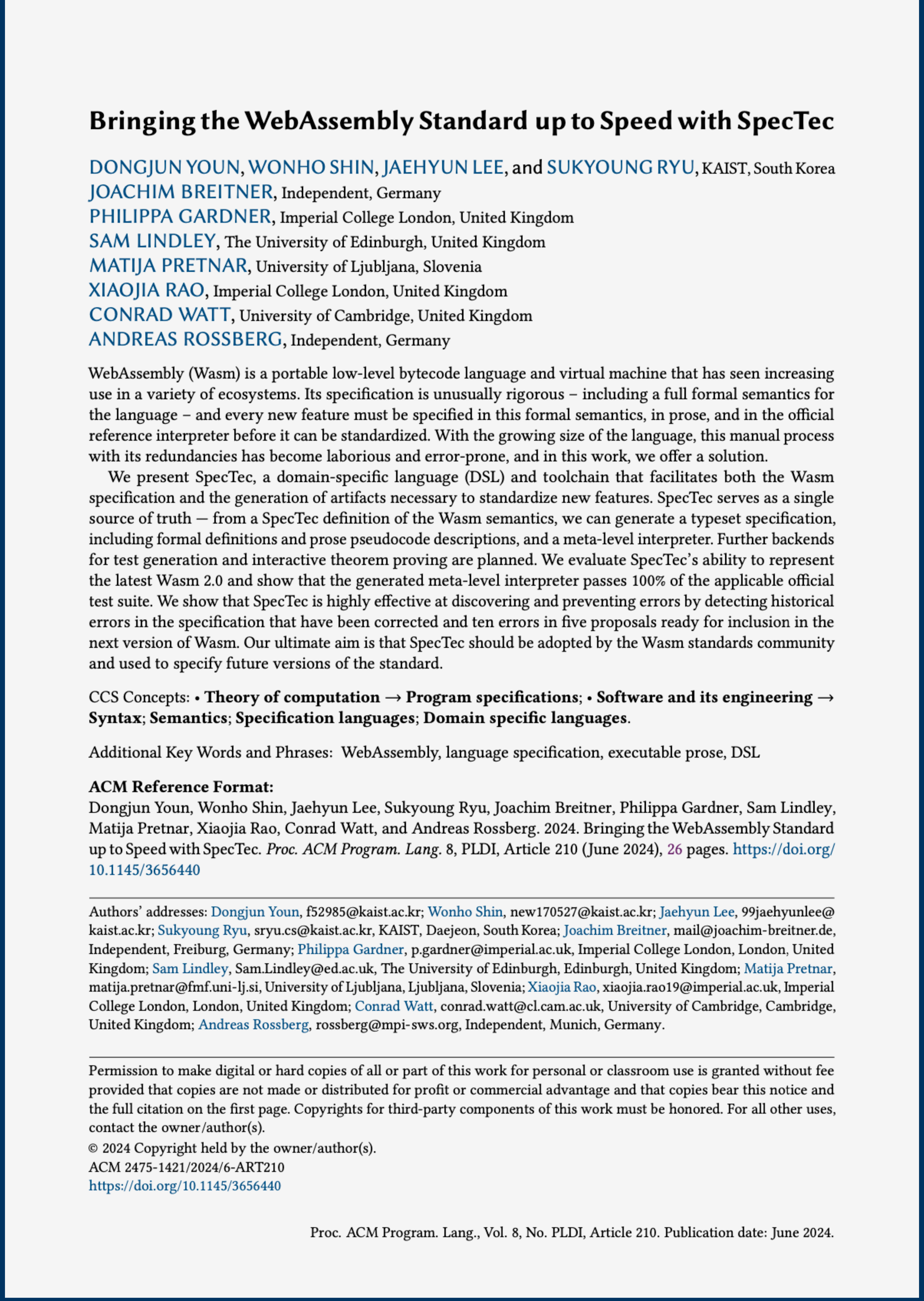
we can interpret this AL

…and thereby indirectly run actual Wasm with actual Wasm spec


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



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.

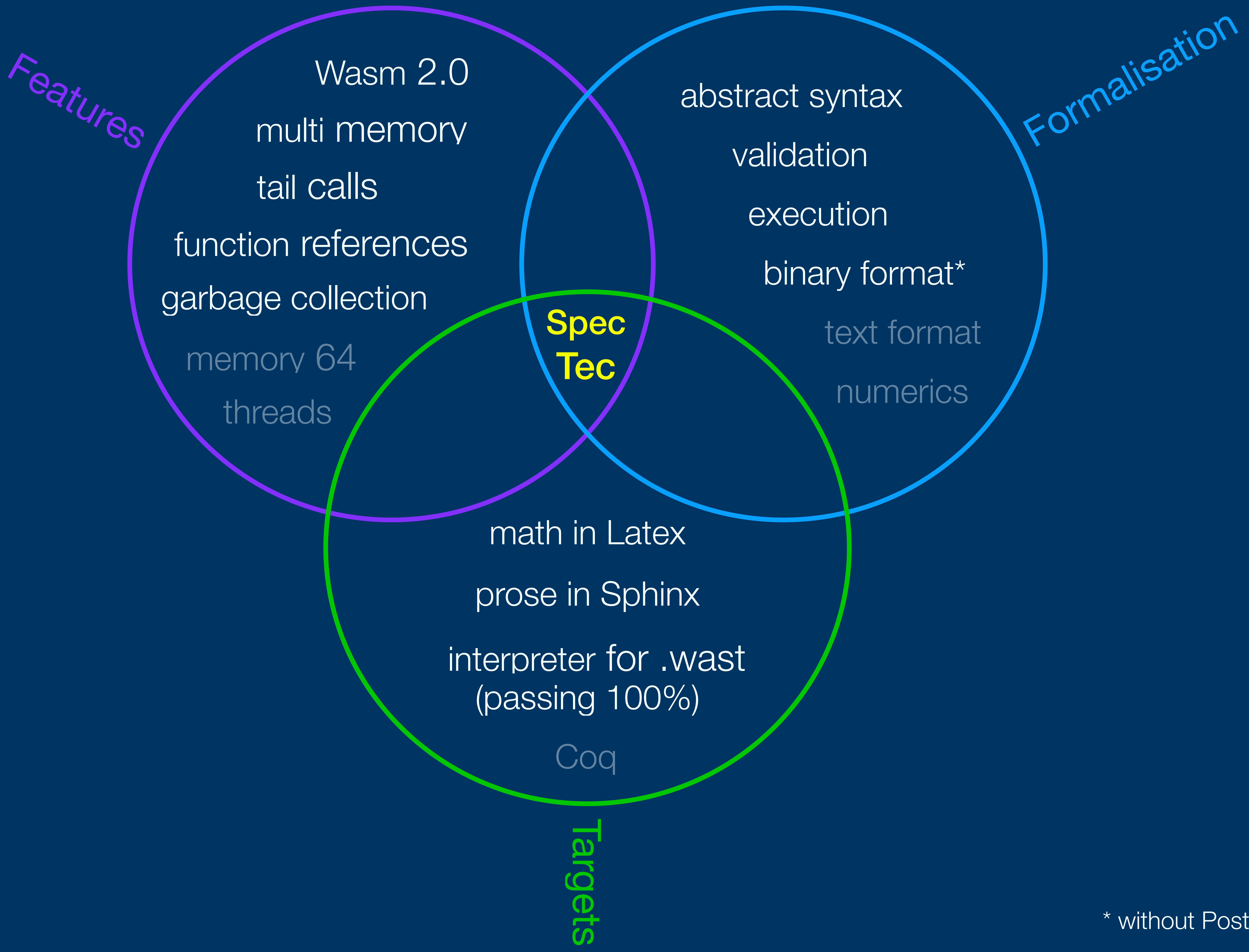
ACM 2475-1421/2024/6-ART210
https://doi.org/10.1145/3656440

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 210. Publication date: June 2024.

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

Code…

Status



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

Targets

math in Latex

prose in Sphinx

interpreter for .wast
(passing 100%)

Coq

* without Post-2.0 syntax

Converting the Document

extended spec build infrastructure with a SpecTec pass

conversion of document can now happen incrementally, replacing text with splices

WIP, so far we have converted:

- all of abstract syntax Chapter (syntax/*.rst)
- instruction execution (exec/instructions.rst) (math and prose)
- instruction validation (valid/instructions.rst) (math only)

Current Limitations

Current Limitations of Math Renderer

hyperlinks not working yet

layout heuristics are slightly hacky

some formulas need more fine-tuning

hard-codes known syntactic shapes of relations (⊢ and ⇝)

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

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

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

validation prose has fallen behind a little

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

Spec Bugs

Bugs found in spec and proposals

type errors

…missing immediates or record fields in rules

semantic errors

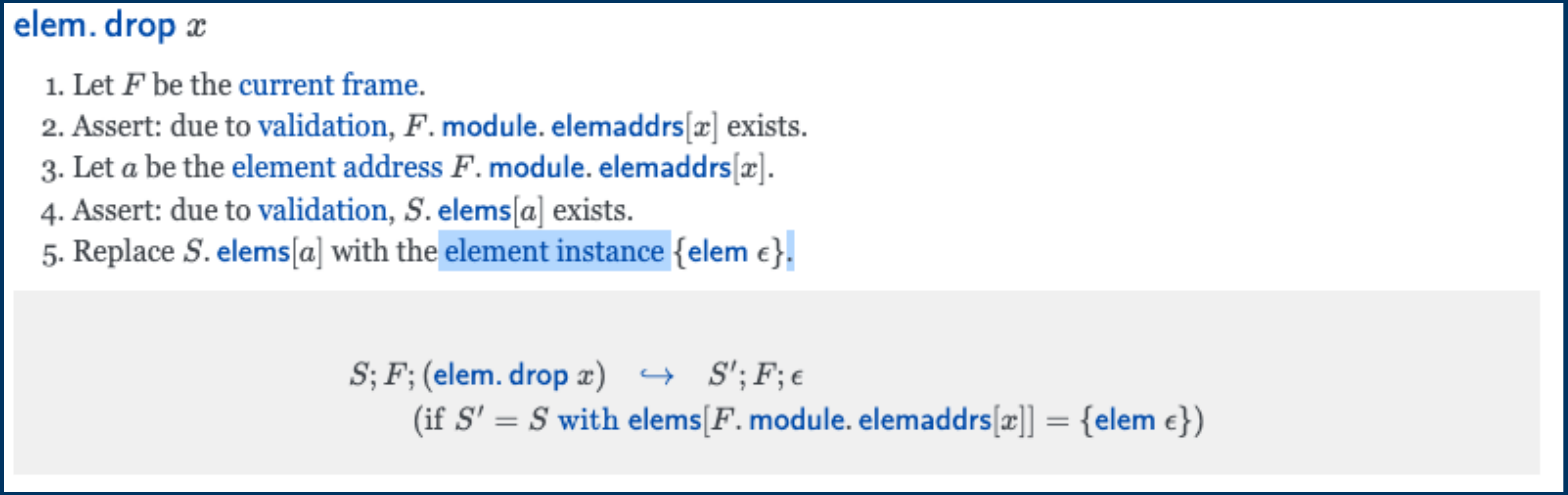
…missing stack operands, stack mishandling, index errors

prose errors

…unbound variables, missing steps

editorial errors

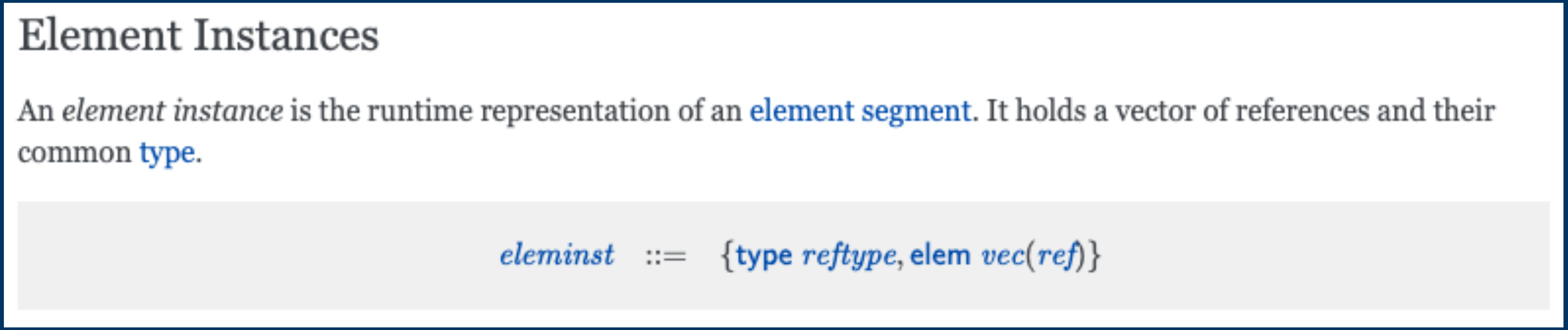
…typos, layout errors, wrong hyperlinks, …



$\mathsf{elem.drop}\ x$

1. Let F be the current frame.
2. Assert: due to validation, $F.\mathsf{module}.\mathsf{elemaddrs}[x]$ exists.
3. Let a be the element address $F.\mathsf{module}.\mathsf{elemaddrs}[x]$.
4. Assert: due to validation, $S.\mathsf{elems}[a]$ exists.
5. Replace $S.\mathsf{elems}[a]$ with the element instance $\{\mathsf{elem}\ \epsilon\}$.

$$\begin{array}{l} S; F; (\mathsf{elem.drop}\ x) \quad \hookrightarrow \quad S'; F; \epsilon \\ \qquad (\text{if } S' = S \text{ with } \mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[x]] = \{\mathsf{elem}\ \epsilon\}) \end{array}$$



Element Instances

An *element instance* is the runtime representation of an element segment. It holds a vector of references and their common type.

$$\mathit{eleminst} \quad ::= \quad \{\mathsf{type}\ \mathit{reftype}, \mathsf{elem}\ \mathit{vec}(\mathit{ref})\}$$

Retroactive bug detection

collected fixed bugs in Wasm 2.0 spec from last 2 years

…and re-injected them into SpecTec formulation

type errors (3)

semantic errors (3)

prose errors (7)

editorial errors (numerous)

…see paper for details

Plan

Current Focus

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

addressing renderer limitations

progress on converting the spec

improve robustness and error reporting

tutorial and documentation

(user experience still is rough at the moment…)

Longer-term

Complete spec conversion

Generate more parts of the spec

…cross-reference anchors, indices, macros

Handle more aspects of the spec in meta interpreter

…decoding, parsing, validation

Improve user experience and stability

Theorem prover backends

Moving Forward

Will have tool and spec in “good shape” by June F2F

…crucial parts of the spec converted and on par; tool usable for early adopters

Hoping for official CG sign-off at that point

…it is a lot of work, can only invest so much upfront

Actual switch to infrastructure can take longer

…and full spec conversion/coverage will be an incremental process anyway

We expect to continue working on SpecTec for at least the next couple of years

…improvements, extensions, maintenance, new ideas

What should be the process?

What should be the acceptance criteria?

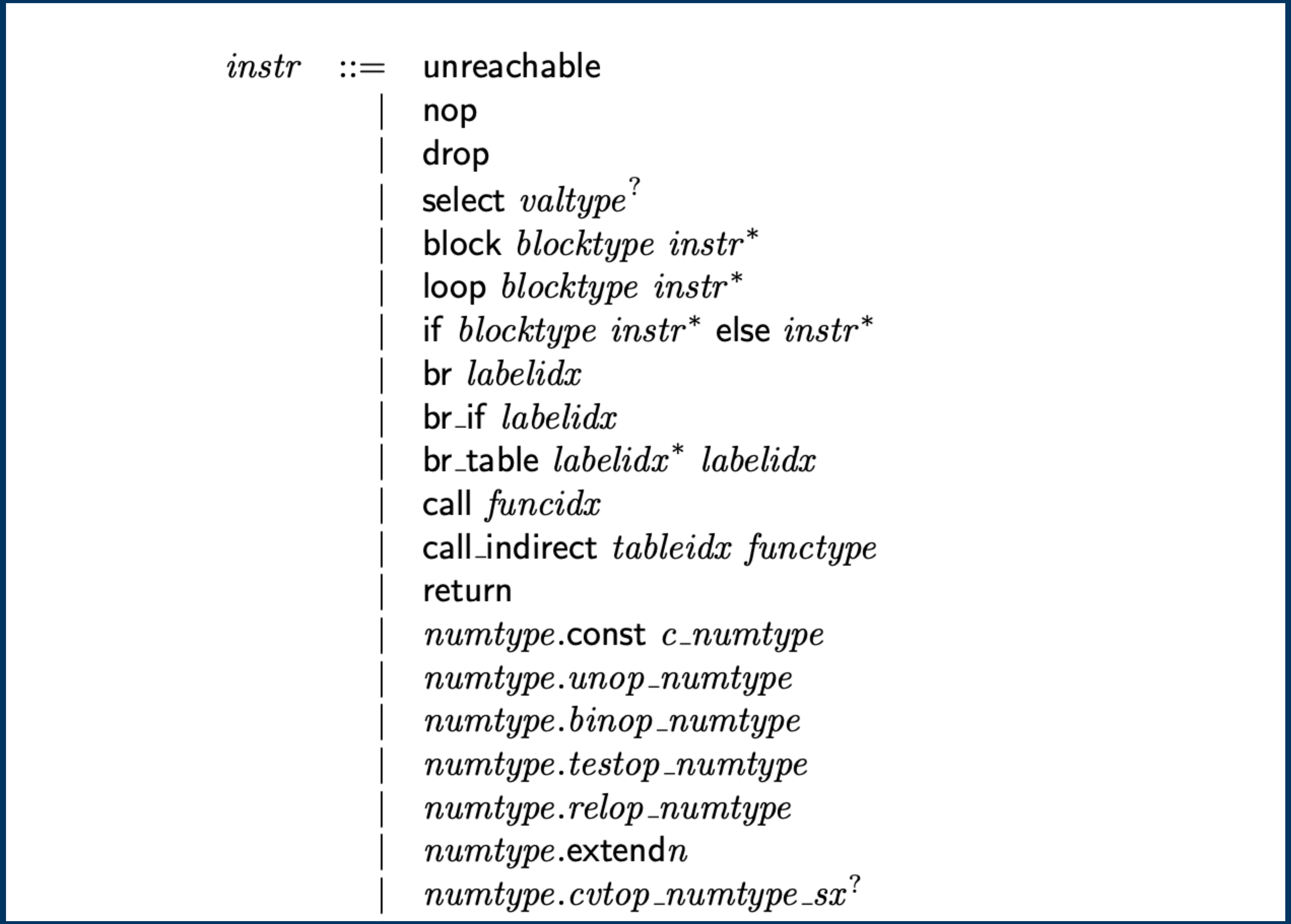
Repo: https://github.com/wasm-dsl/spectec

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

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

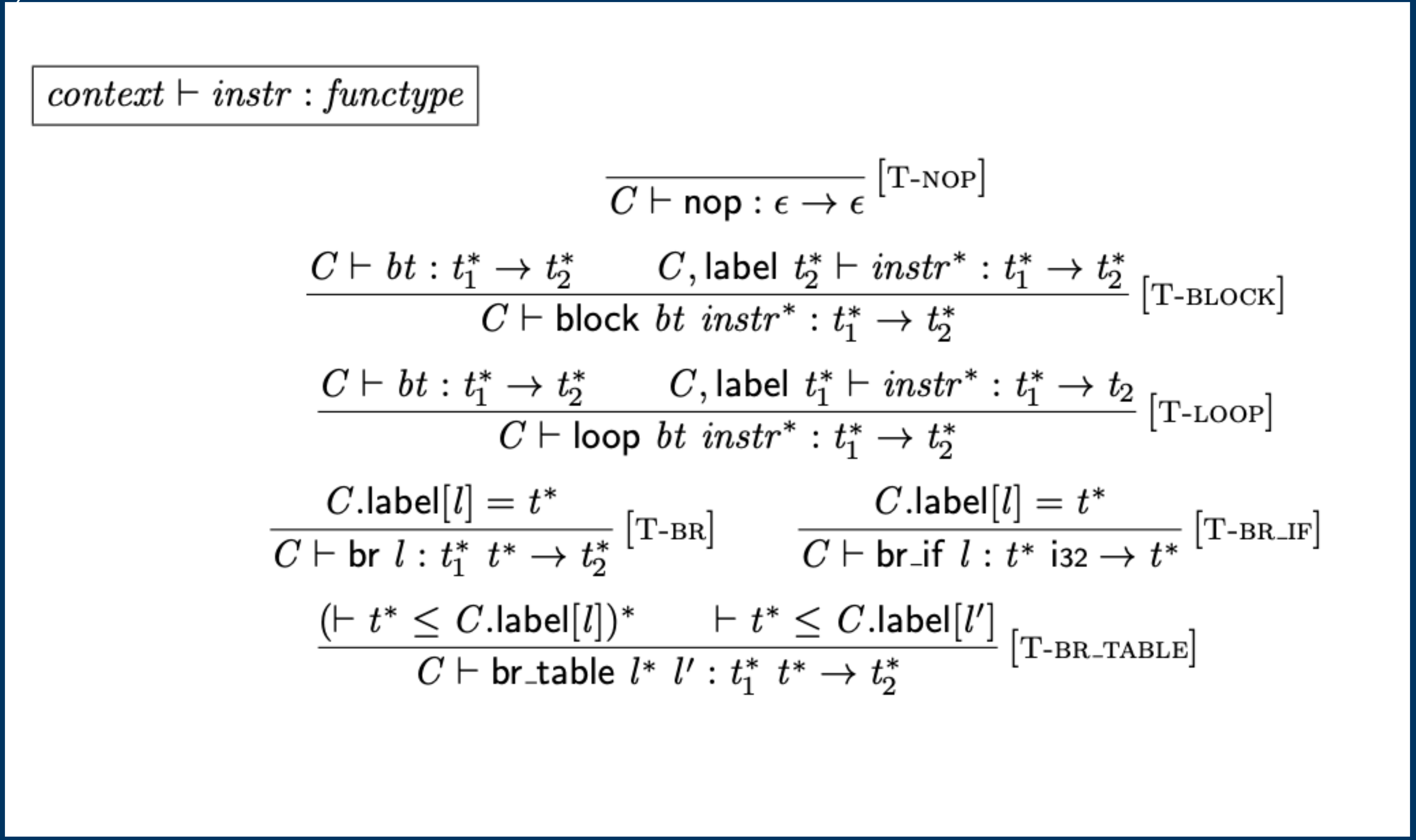
Outtakes

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



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

```
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])*
```



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

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

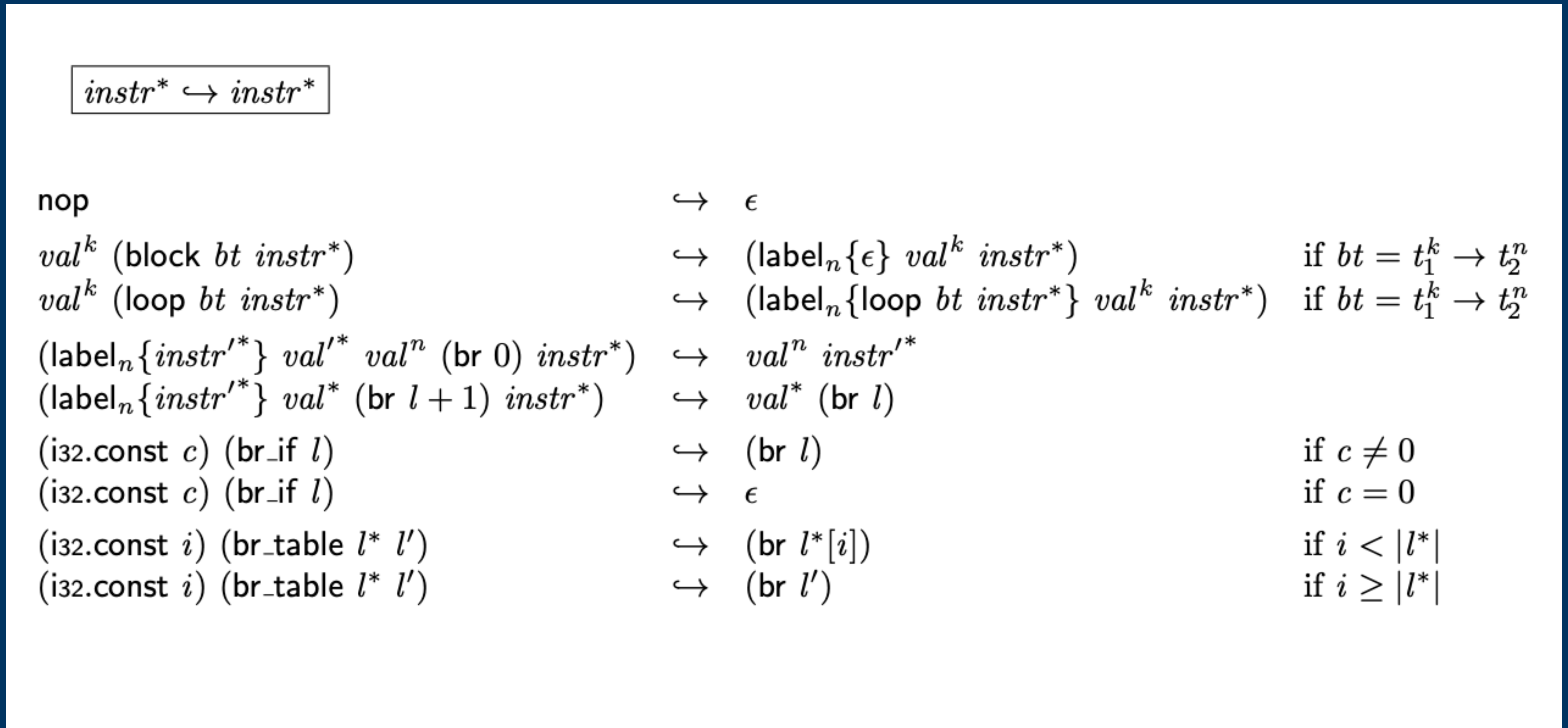
$$\frac{C \vdash \mathit{bt} : t_1^* \to t_2^* \qquad C, \mathsf{label}\ t_2^* \vdash \mathit{instr}^* : t_1^* \to t_2^*}{C \vdash \mathsf{block}\ \mathit{bt}\ \mathit{instr}^* : t_1^* \to t_2^*}\ [\text{T-BLOCK}]$$

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

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

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

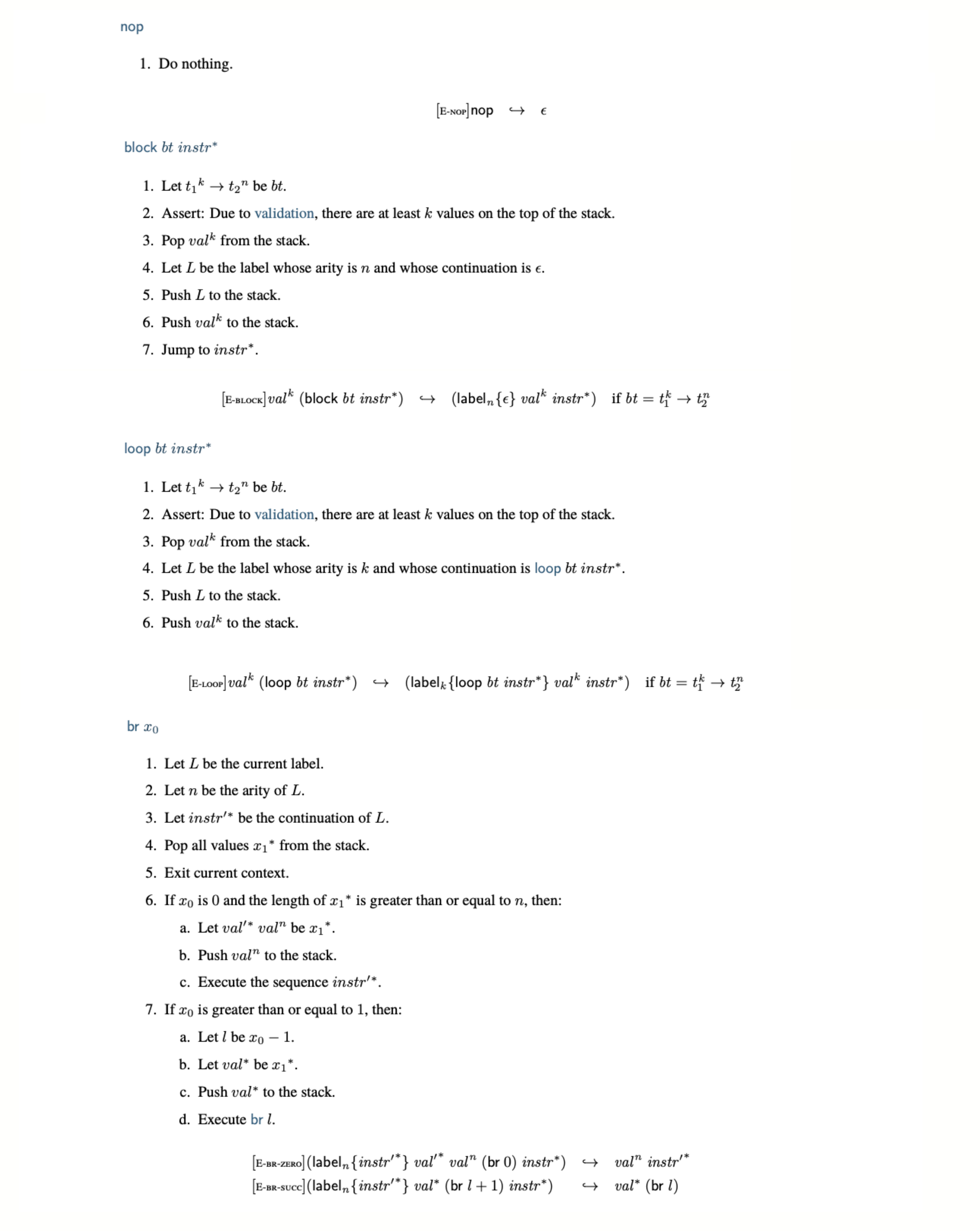
```
relation Step_pure: config ~> config

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*|
```



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

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

```
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.

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

block bt $instr^*$

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

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

loop bt $instr^*$

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

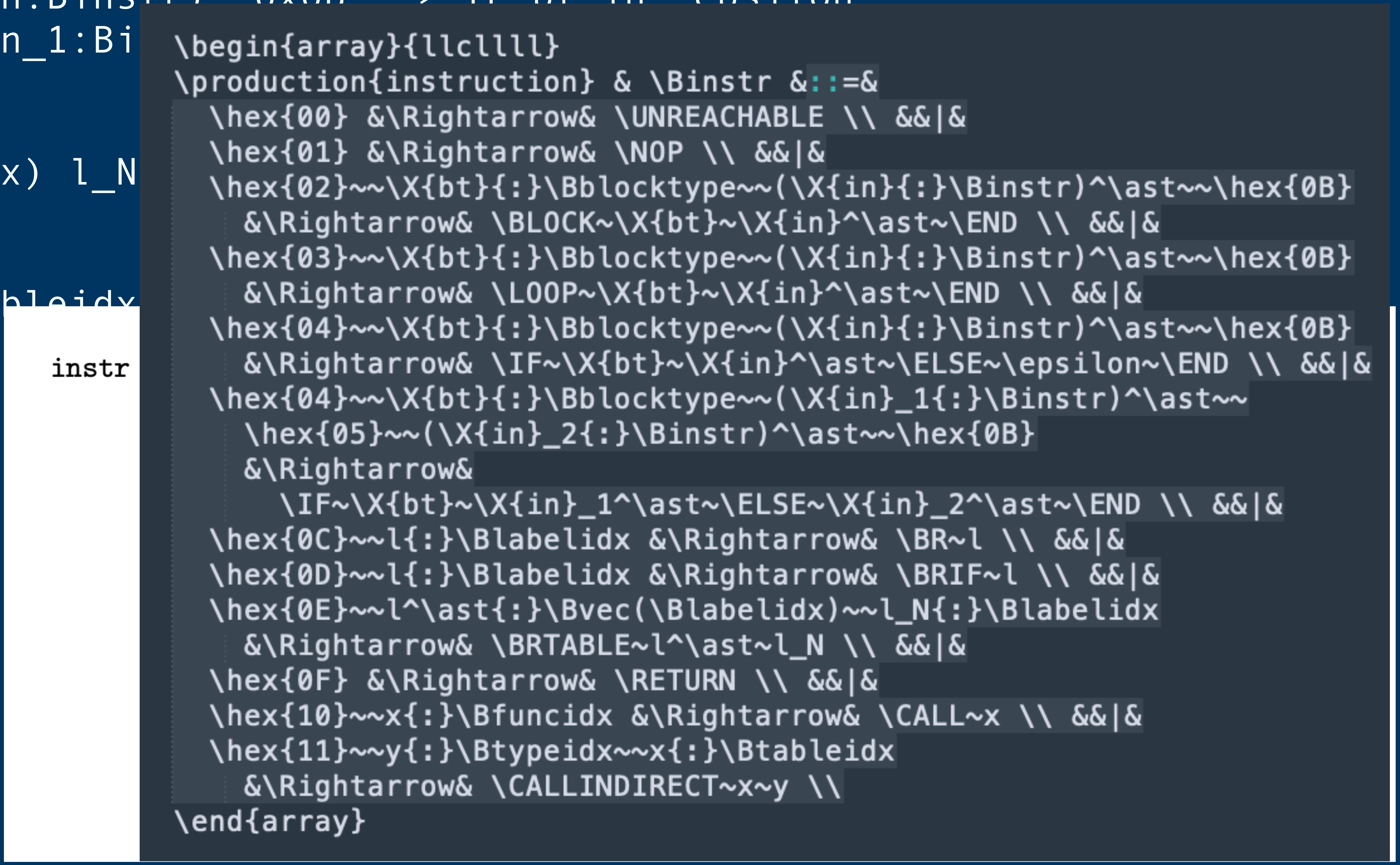
$$[\text{E-LOOP}]\,val^k\ (\mathsf{loop}\ bt\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_k\{\mathsf{loop}\ bt\ instr^*\}\ val^k\ instr^*) \quad \text{if } bt = t_1^k \to t_2^n$$

br x_0

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

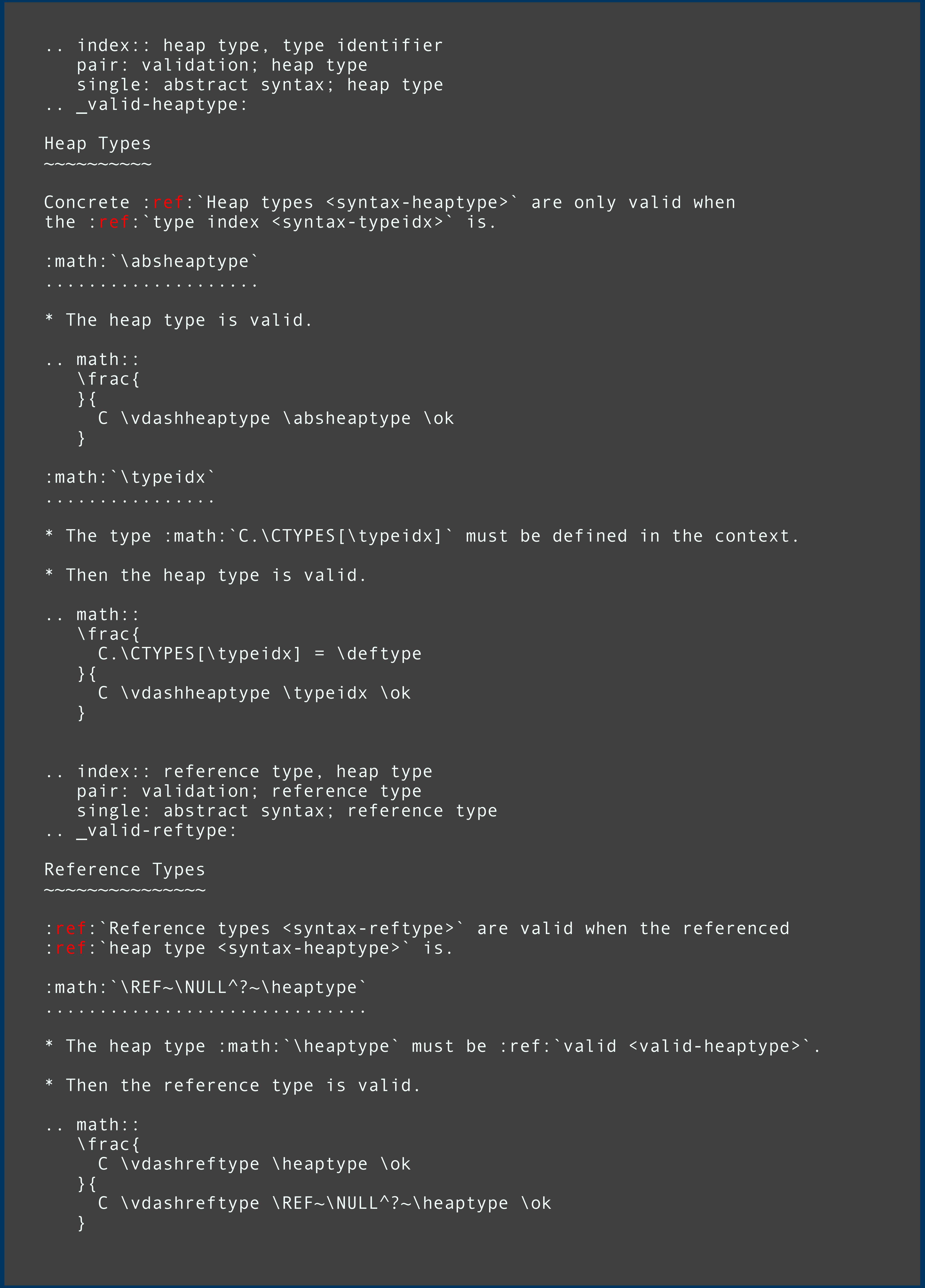
$$\begin{array}{lcl}[\text{E-BR-ZERO}](\mathsf{label}_n\{instr'^*\}\ val'^*\ val^n\ (\mathsf{br}\ 0)\ instr^*) & \hookrightarrow & val^n\ instr'^* \\ [\text{E-BR-SUCC}](\mathsf{label}_n\{instr'^*\}\ val^*\ (\mathsf{br}\ l+1)\ instr^*) & \hookrightarrow & val^*\ (\mathsf{br}\ l)\end{array}$$

```
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:Btab
  | ...
```

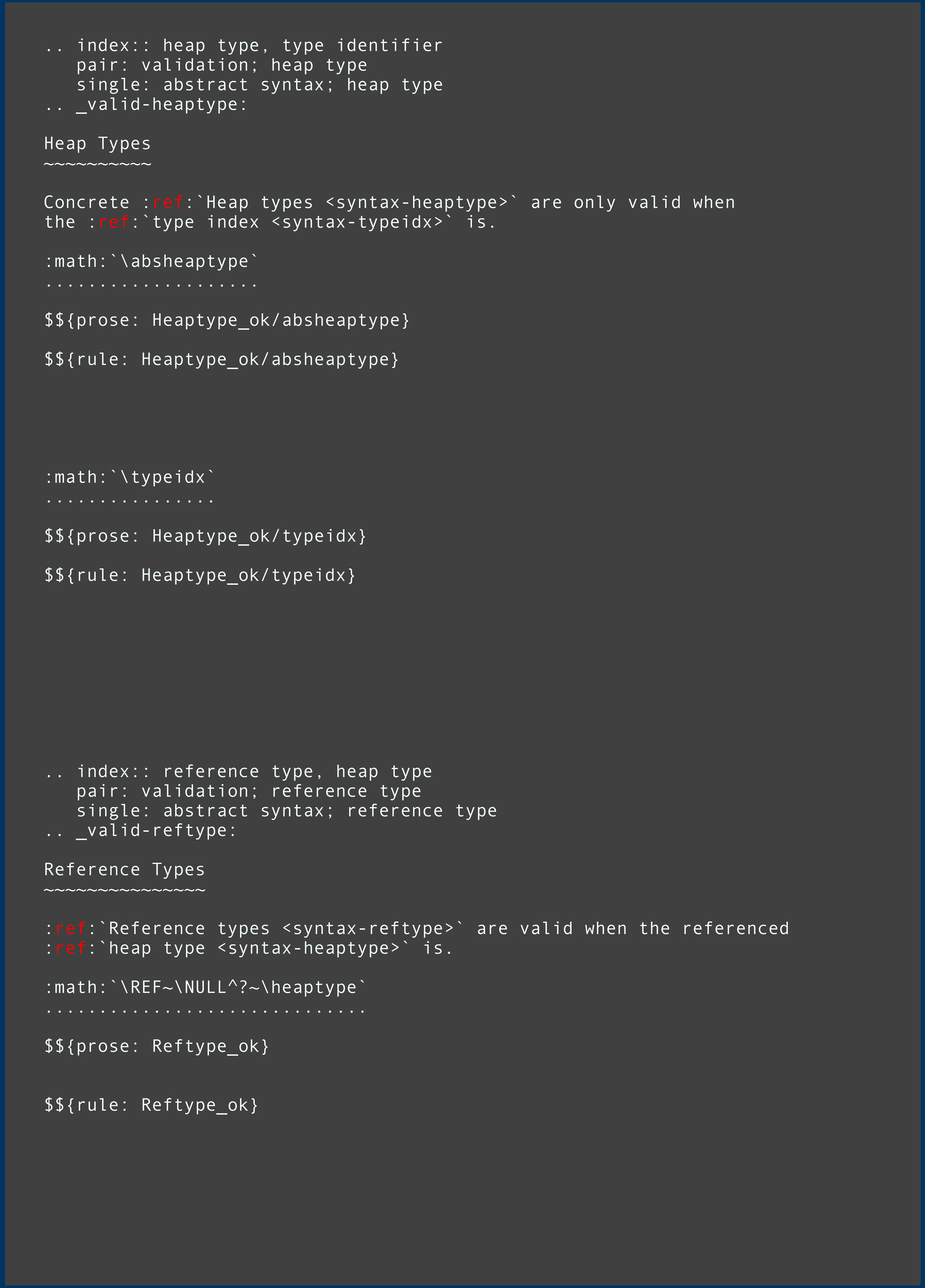


instr

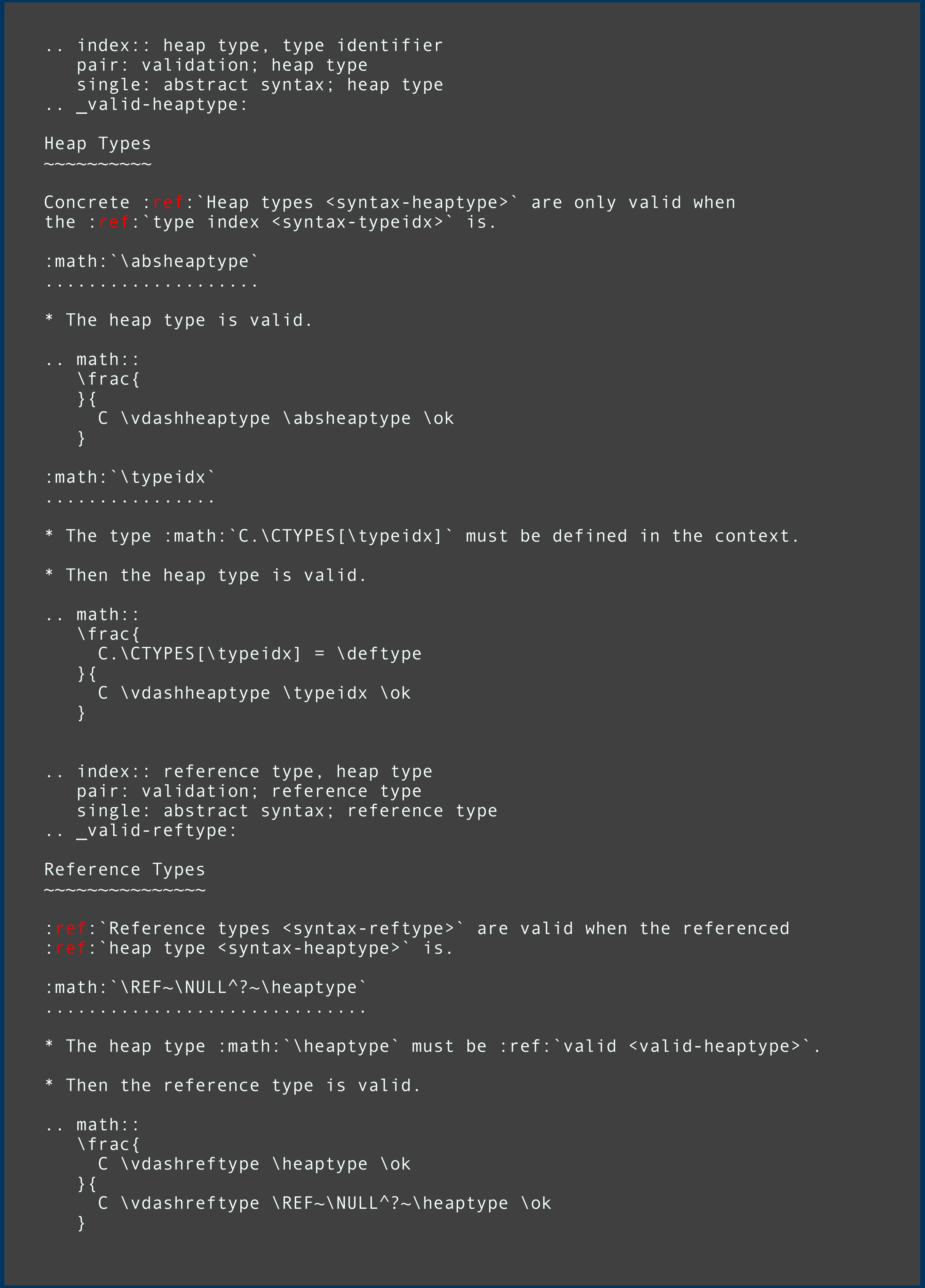
```
\begin{array}{llclll}
\production{instruction} & \Binstr &::=&
  \hex{00} &\Rightarrow& \UNREACHABLE \\ &&|&
  \hex{01} &\Rightarrow& \NOP \\ &&|&
  \hex{02}~~\X{bt}{:}\Bblocktype~~(\X{in}{:}\Binstr)^\ast~~\hex{0B}
    &\Rightarrow& \BLOCK~\X{bt}~\X{in}^\ast~\END \\ &&|&
  \hex{03}~~\X{bt}{:}\Bblocktype~~(\X{in}{:}\Binstr)^\ast~~\hex{0B}
    &\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}
```



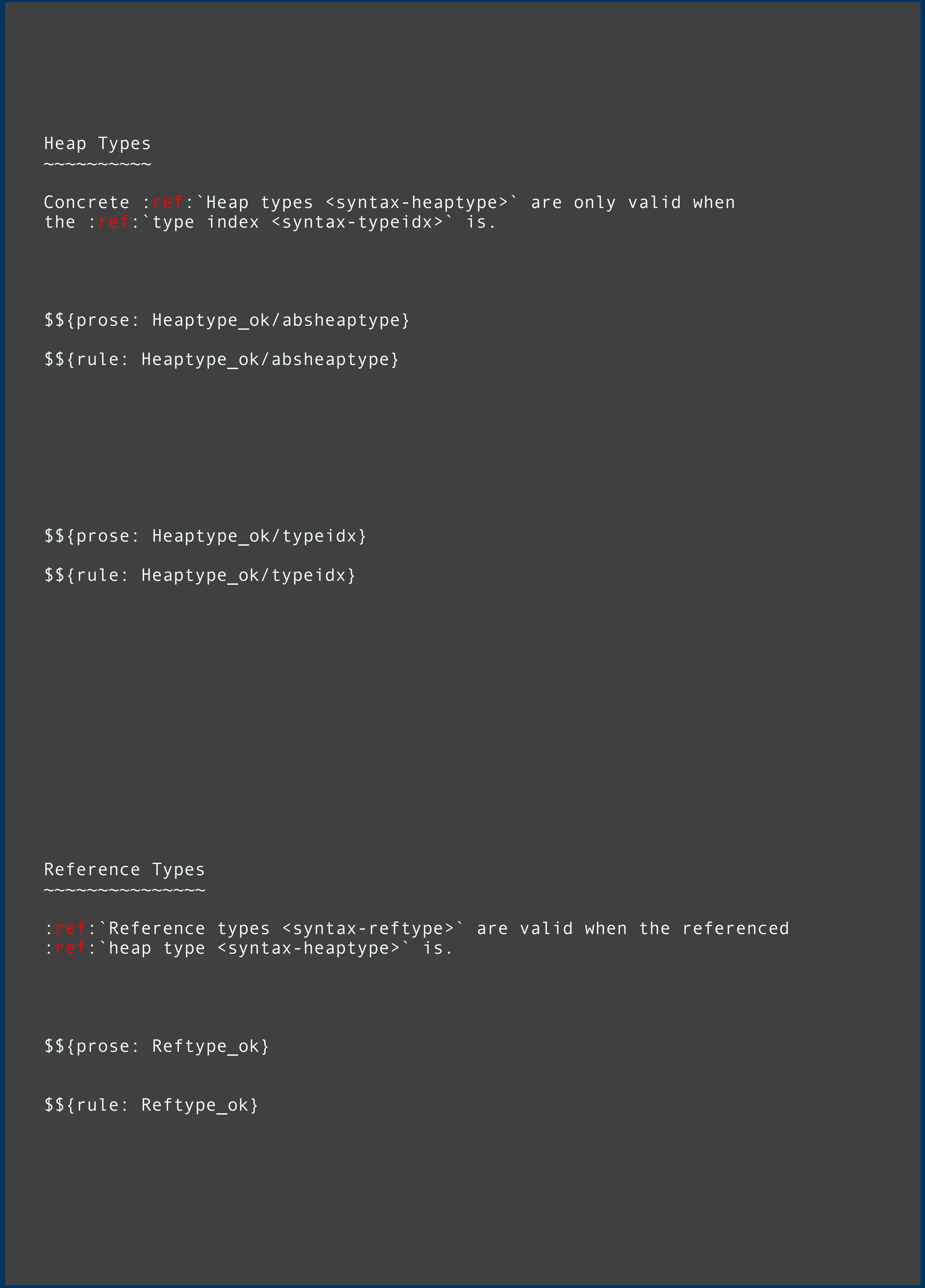
```
.. index:: heap type, type identifier
   pair: validation; heap type
   single: abstract syntax; heap type
.. _valid-heaptype:

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}
```