Who’s afraid of the turnstile?

Andreas Rossberg

BOB 2024/02/15

4.2 IF STATEMENT

IF statements have two forms […]: numerical and logical.

4.2.1 Numerical IF Statements

Numerical IF statements are of the form: IF (expression) n_1,n_2,n_3 where n_1,n_2,n_3 are statement numbers. […] All three statement numbers must be present. The expression may not be complex. […]

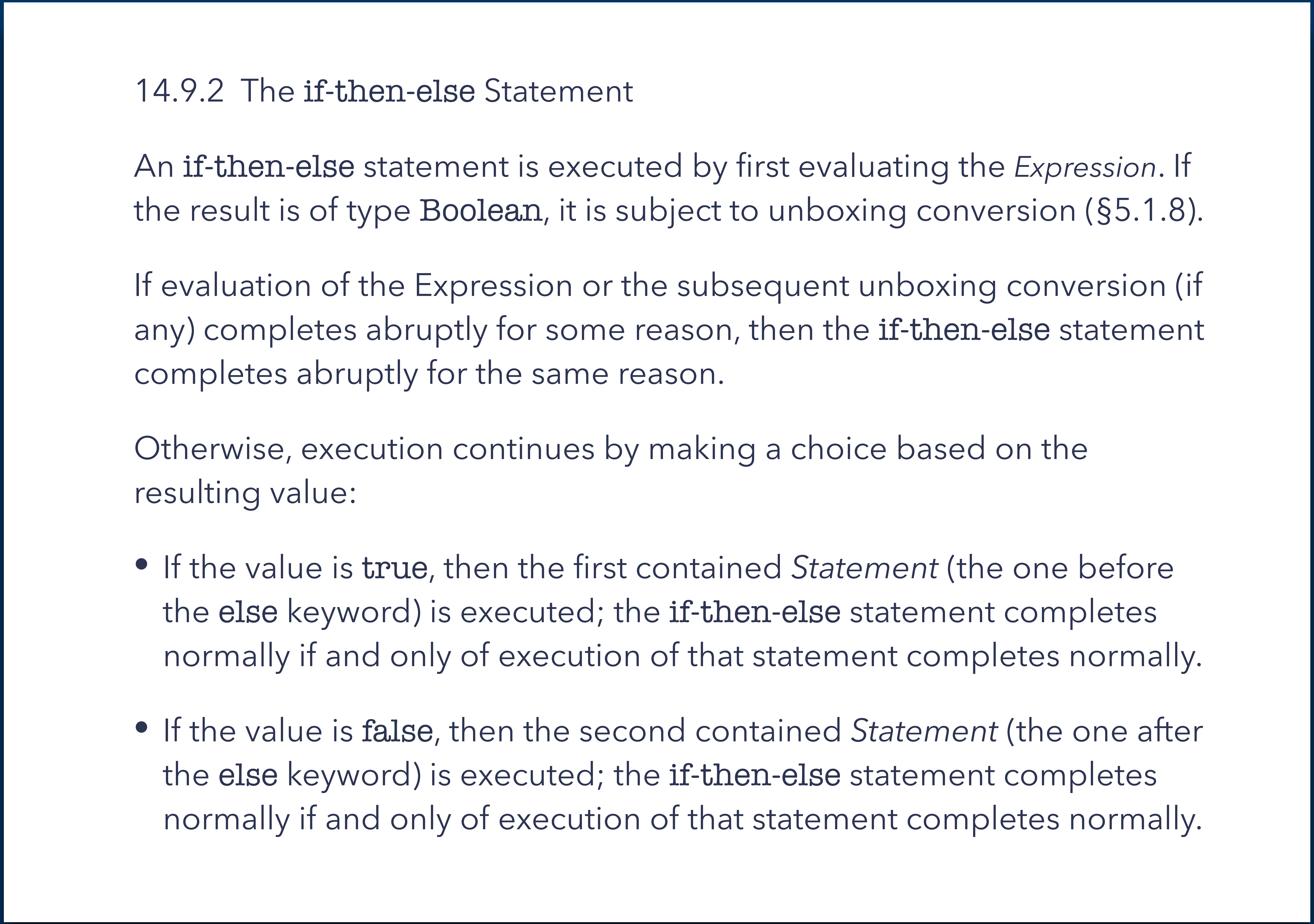
4.2.2 Logical IF Statements

Logical IF statements have the form: IF (expression)S where S is a complete statement. The expression must be logical. S may be any executable statement other than a DO statement or another logical IF statement (see Chapter 2, Section 2.3.2). […]

[FORTRAN IV Reference Manual, 1971]

$$\begin{aligned} \textit{if-stmt} \;::=\;& \mathtt{IF}\ (\ \textit{expr}\ )\ \textit{num}_1\ ,\ \textit{num}_2\ ,\ \textit{num}_3 \\ \mid\;& \mathtt{IF}\ (\ \textit{expr}\ )\ \textit{plain-stmt} \end{aligned}$$

[context-free grammar / BNF, ca. 1958]



14.9.2 The `if-then-else` Statement

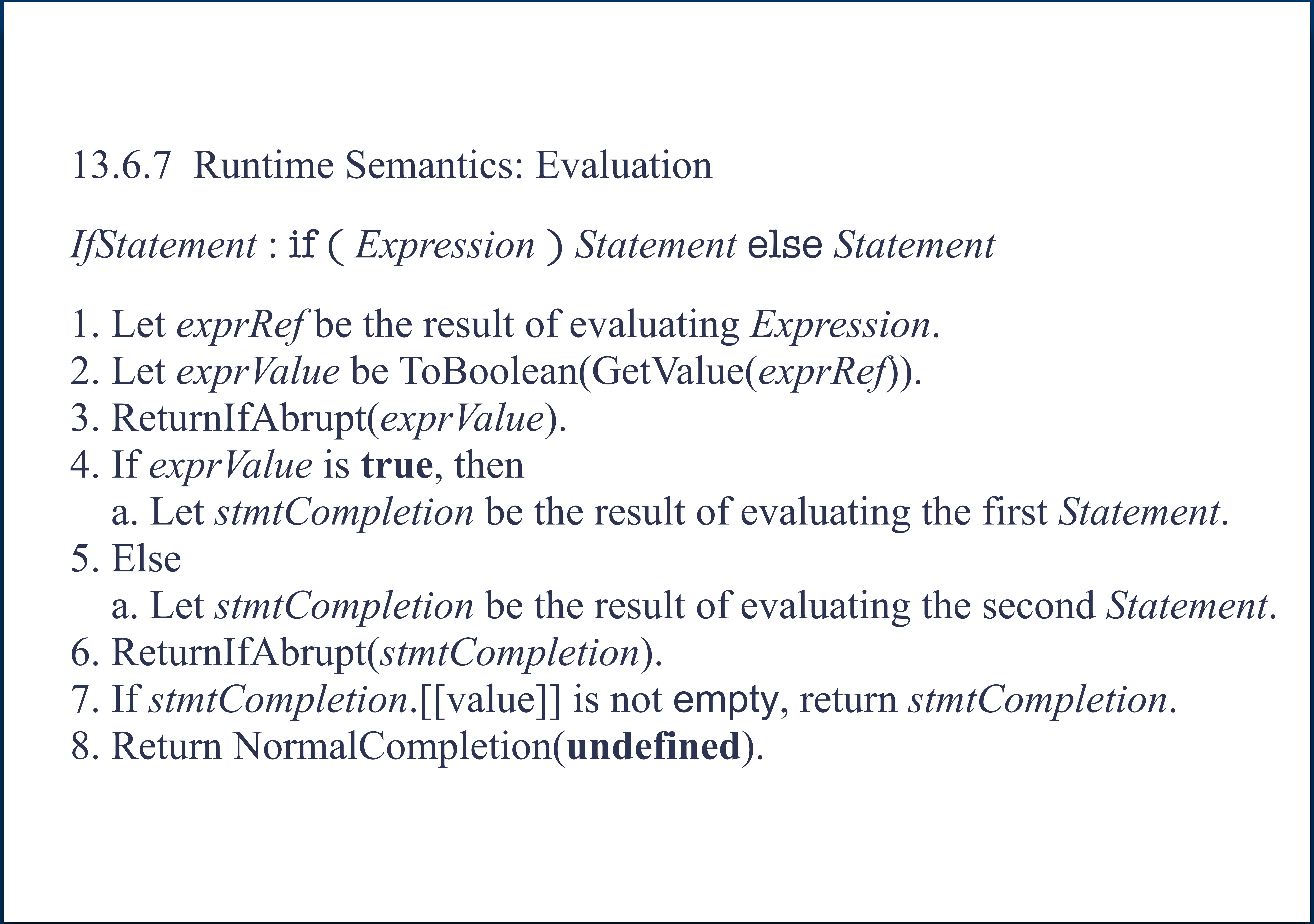
An `if-then-else` statement is executed by first evaluating the *Expression*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, then the `if-then-else` statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the first contained *Statement* (the one before the `else` keyword) is executed; the `if-then-else` statement completes normally if and only of execution of that statement completes normally.
- If the value is `false`, then the second contained *Statement* (the one after the `else` keyword) is executed; the `if-then-else` statement completes normally if and only of execution of that statement completes normally.

[The Java Language Specification, Java SE 21 Edition]



13.6.7 Runtime Semantics: Evaluation

IfStatement : `if` (*Expression*) *Statement* `else` *Statement*

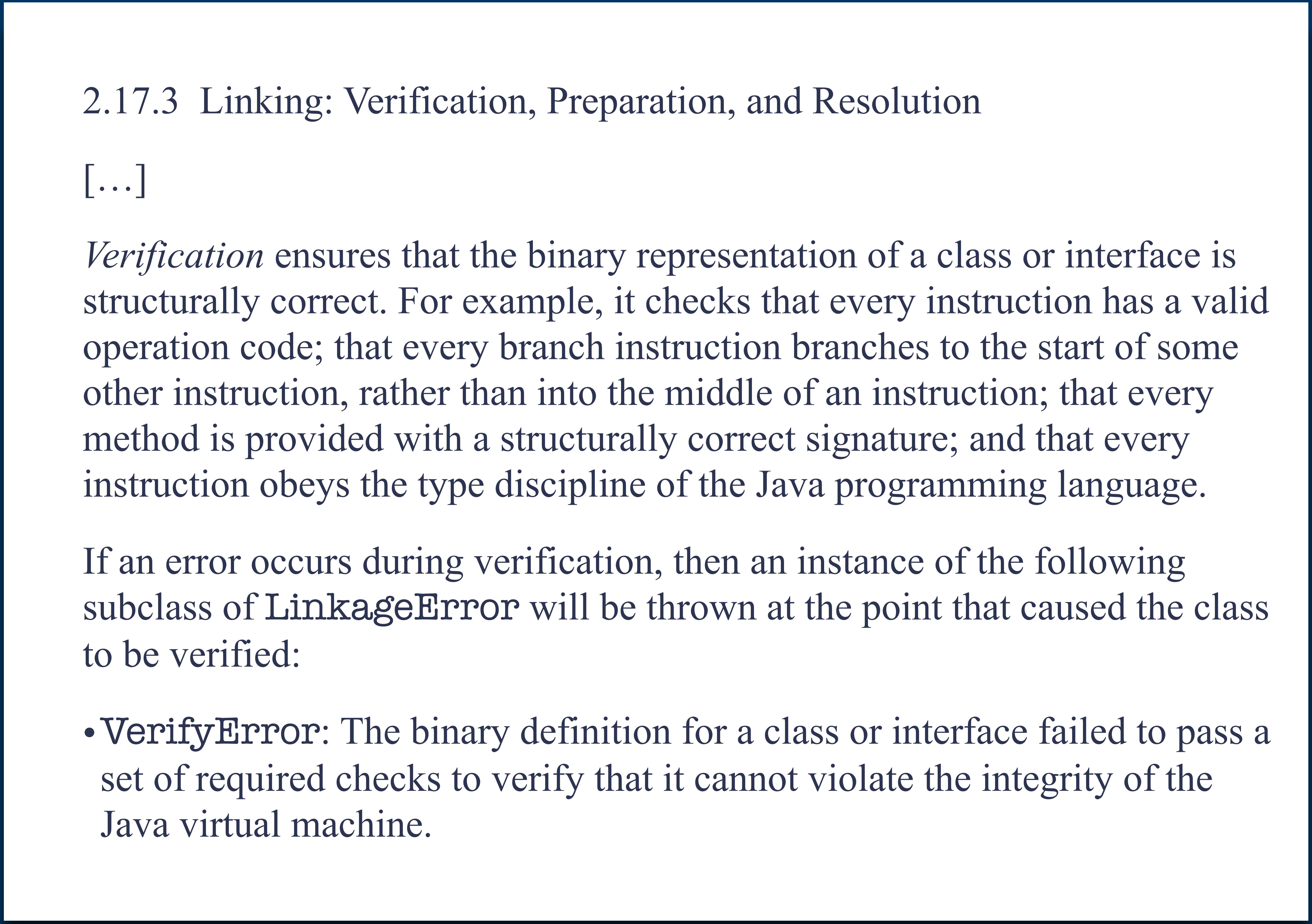
```
1. Let exprRef be the result of evaluating Expression.
2. Let exprValue be ToBoolean(GetValue(exprRef)).
3. ReturnIfAbrupt(exprValue).
4. If exprValue is true, then
   a. Let stmtCompletion be the result of evaluating the first Statement.
5. Else
   a. Let stmtCompletion be the result of evaluating the second Statement.
6. ReturnIfAbrupt(stmtCompletion).
7. If stmtCompletion.[[value]] is not empty, return stmtCompletion.
8. Return NormalCompletion(undefined).
```

[ECMAScript 2015 Language Specification]

$$\textbf{if}\ (\ \textbf{true}\ )\ \mathit{stmt}_1\ \textbf{else}\ \mathit{stmt}_2 \quad \longrightarrow \quad \mathit{stmt}_1$$

$$\textbf{if}\ (\ \textbf{false}\ )\ \mathit{stmt}_1\ \textbf{else}\ \mathit{stmt}_2 \quad \longrightarrow \quad \mathit{stmt}_2$$

[structured operational semantics / SOS, ca. 1980]



2.17.3 Linking: Verification, Preparation, and Resolution

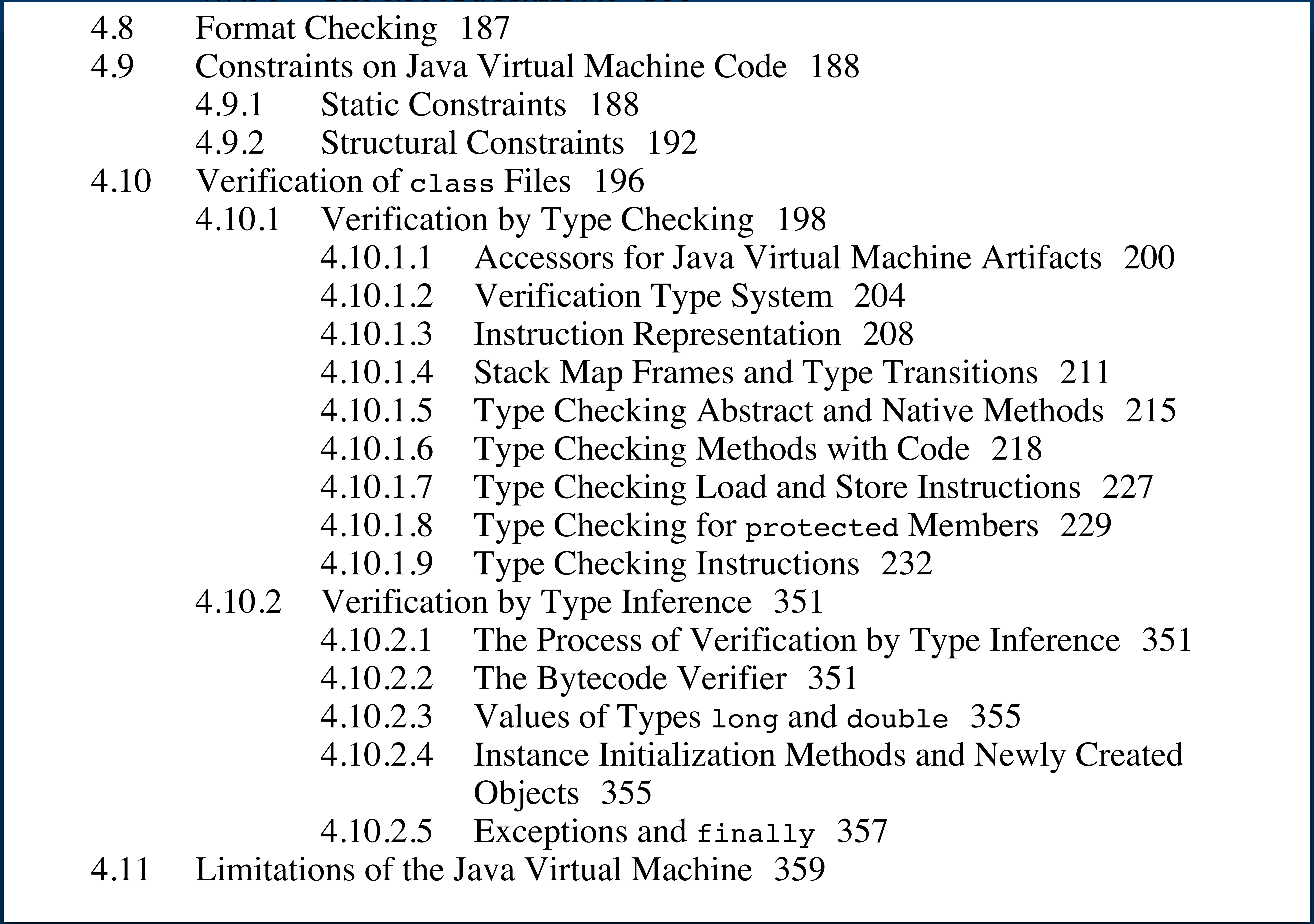
[…]

Verification ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java programming language.

If an error occurs during verification, then an instance of the following subclass of `LinkageError` will be thrown at the point that caused the class to be verified:

- `VerifyError`: The binary definition for a class or interface failed to pass a set of required checks to verify that it cannot violate the integrity of the Java virtual machine.

[The Java Virtual Machine Specification, First Edition]



[The Java Virtual Machine Specification, SE 16 Edition]

Formal Semantics

we can verify that no rule is missing or unreachable

we can verify that the rules are unambiguous

we can verify that it’s well-defined, sound, decidable, deterministic, etc.

we can generate interpreters

we can generate tests

we can prove compilers correct

WebAssembly

Wasm

a low-level, language-independent virtual machine

not a web technology

web

cloud

edge

mobile

embedded

blockchain

desktop

Semantics

- language-independent
- platform-independent
- hardware-independent
- fast to execute
- safe to execute
- deterministic
- easy to reason about

Representation

- compact
- easy to generate
- fast to decode
- fast to validate
- fast to compile
- streamable
- parallelisable

Wasm …is fully formalised 🥹

from the first byte of source code to the last bit of memory at execution

a virtual machine is just a language

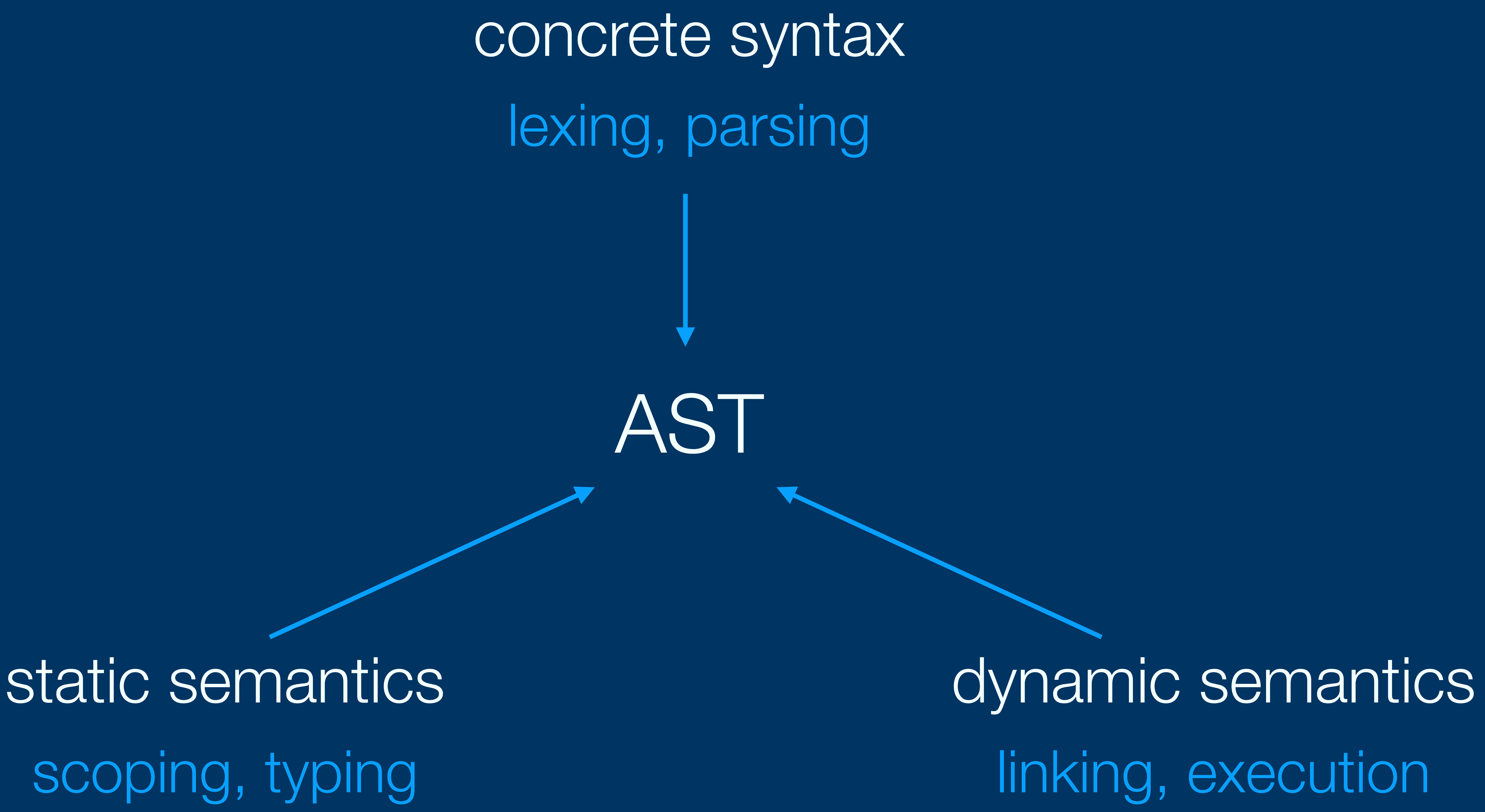
bytecode decoding = parsing

bytecode validation = type checking

bytecode execution = evaluation

…all textbook PL theory techniques apply!

Defining a language



concrete syntax

lexing, parsing

AST

static semantics

scoping, typing

dynamic semantics

linking, execution

abstract syntax

(*value type*) t ::= i32 | i64 | f32 | f64 | v128 | funcref | externref | …
(*function type*) ft ::= $t^* \rightarrow t^*$

$unop$ ::= neg | abs | …
$binop$ ::= add | sub | mul | …
$relop$ ::= eq | ne | lt | gt | le | ge | …
$cvtop$ ::= convert | reinterpret | …

(*instruction*) $instr$ ::= t.const c | $t.unop$ | $t.binop$ | $t.relop$ | $t.cvtop.t$ | …
ref.null | ref.func x | nop | drop | select | unreachable | …
block ft $instr^*$ | loop ft $instr^*$ | if ft $instr^*$ else $instr^*$ | …
br i | br_if i | br_table i^+ | call x | call_indirect ft x | return | …
local.get x | local.set x | local.tee x | global.get x | global.set | …
table.get x | table.set x | table.size x | table.grow x | …
t.load.$n^?$ x | t.store.$n^?$ x | memory.size x | memory.grow x | …

(*function*)	$func$ ::= **func** ft (local t^*) e^*	im ::= **import** “nm” **func** ft	ex ::= **export** “nm” **func** x
(*global*)	$glob$ ::= **global** mut$^?$ t e^*	**import** “nm” **global** mut$^?$ t	**export** “nm” **global** x
(*table*)	tab ::= **table** $[n..m]$ t e^*	**import** “nm” **table** $[n..m]$ t	**export** “nm” **table** x
(*memory*)	mem ::= **memory** $[n..m]$	**import** “nm” **memory** $[n..m]$	**export** “nm” **memory** x

(*module*) mod ::= **module** im^* $func^*$ $glob^*$ tab^* mem^* ex^*

dynamic semantics

stack machine

```
(i32.const 20)
(i32.const 22)   ⟶   (i32.const 42)
(i32.add)
```

$$(t.\text{const } c_1)\ (t.\text{const } c_2)\ (t.\text{add}) \quad \longrightarrow \quad (t.\text{const } c_1 +_t c_2)$$

$$\begin{array}{l}(\text{i32.const } 1)\\ (\text{i32.const } 3)\\ (\text{i32.const } 5)\\ (\text{i32.add})\\ (\text{i32.add})\end{array} \quad \longrightarrow \quad \begin{array}{l}(\text{i32.const } 1)\\ (\text{i32.const } 8)\\ (\text{i32.add})\end{array} \quad \longrightarrow \quad (\text{i32.const } 9)$$

program terminates when reduced entirely to values

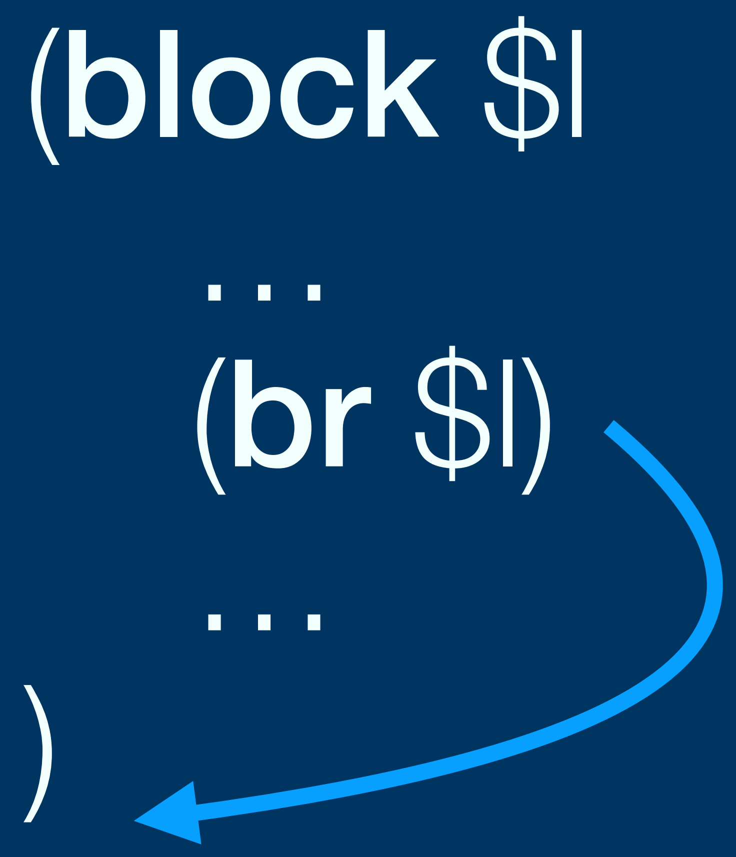
spoiler: soundness proves that this is the only way reduction relation

$$instr^* \quad \longrightarrow \quad instr^*$$

small-step reduction rules
rewrite the program step by step
until it consists only of values

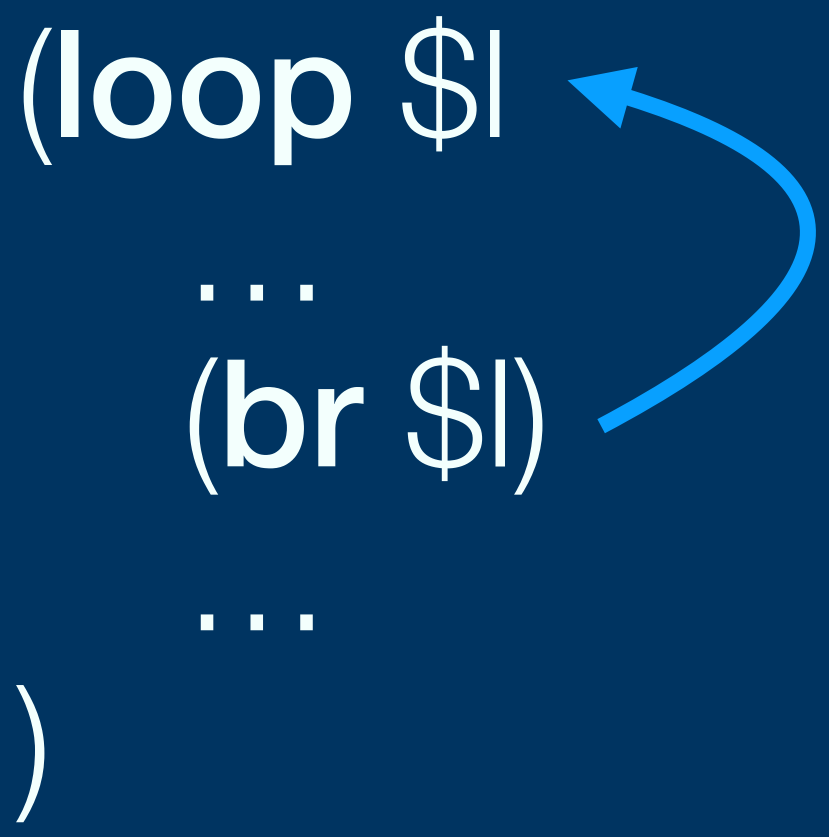
$$instr \;\supseteq\; val \quad ::= \quad t.\mathsf{const}\ c \;\mid\; \mathsf{ref.null} \;\mid\; \mathsf{ref.func}\ x$$

control flow



```
(block $l
    …
    (br $l)
    …
)
```

break



```
(loop $l
    …
    (br $l)
    …
)
```

continue

control flow

```
(block
    ...
    (br i)
    ...
)
```

break

```
(loop
    ...
    (br i)
    ...
)
```

continue

$$
\begin{aligned}
(\textbf{block}\ val^*) &\longrightarrow val^* \\
(\textbf{block}\ val^*\ (\textbf{br}\ 0)\ instr^*) &\longrightarrow \varepsilon \\
(\textbf{block}\ val^*\ (\textbf{br}\ i+1)\ instr^*) &\longrightarrow (\textbf{br}\ i)
\end{aligned}
$$

recall:

$instr \supseteq val ::= t.\mathrm{const}\ c \mid \mathrm{ref.null} \mid \mathrm{ref.func}\ x$

state

$$s \;;\; instr^* \quad \longrightarrow \quad s \;;\; instr^*$$

generalise to reduction over configurations

store s is just more syntax…

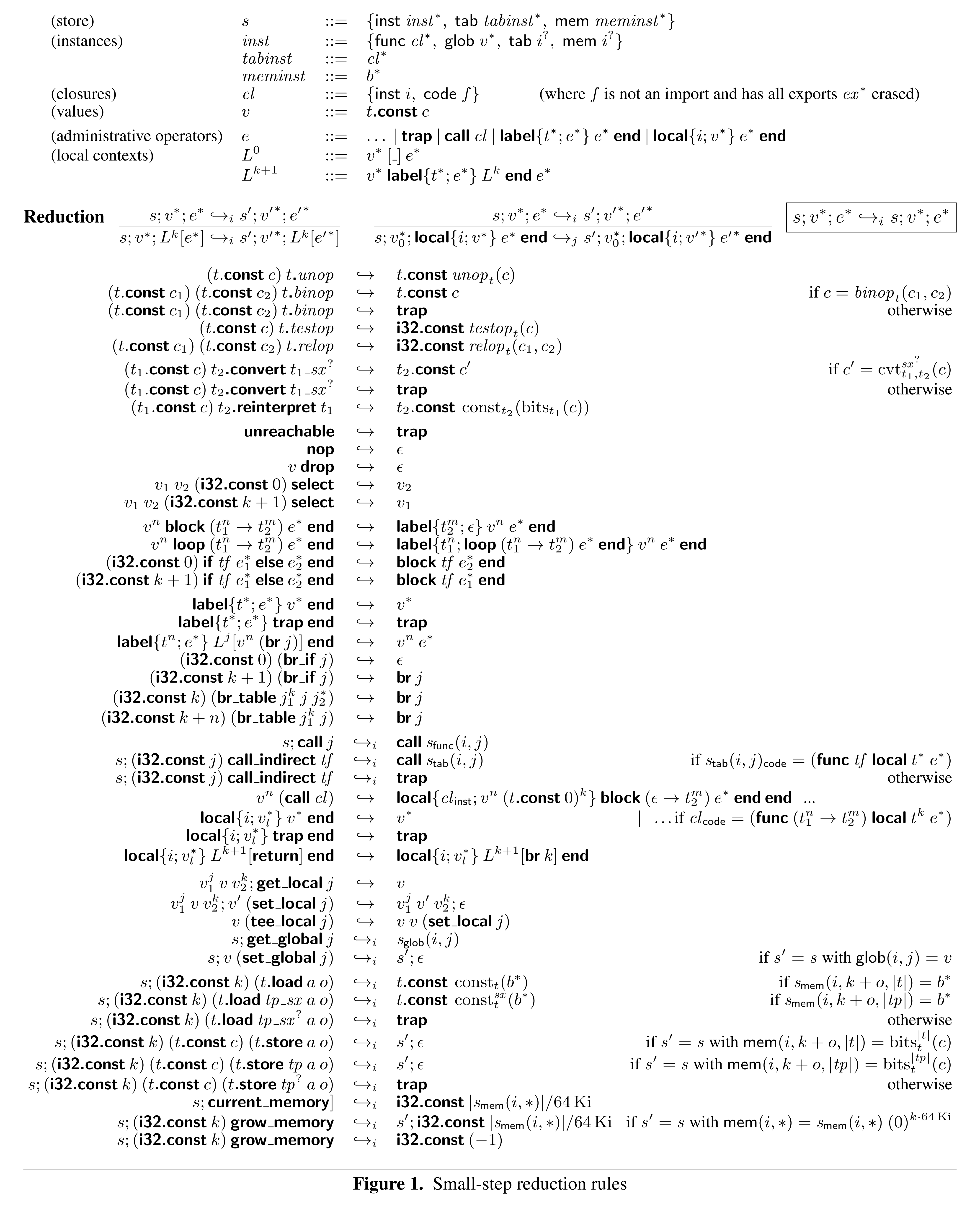
$$s \quad ::= \quad \{\text{globals } val^*, \text{ tables } (val^*)^*, \text{ memories } (byte^*)^*\}$$

$$s \;;\; (\textbf{i32.const}\ i)\ (\textbf{i64.load}\ x) \quad\longrightarrow\quad s \;;\; (\text{i64.const}\ c)$$

$$\text{iff } s.\text{memories}[x][i \,..\, i{+}7] = \mathrm{bytes}_{\text{i64}}(c)$$

$$s \;;\; (\textbf{i32.const}\ i)\ (\textbf{i64.const}\ c)\ (\textbf{i64.store}\ x) \quad\longrightarrow\quad s' \;;\; \varepsilon$$

$$\text{iff } s' = s \text{ with memories}[x][i \,..\, i{+}7] = \mathrm{bytes}_{\text{i64}}(c)$$



$$
\begin{array}{llll}
\text{(store)} & s & ::= & \{\mathsf{inst}\ \mathit{inst}^*,\ \mathsf{tab}\ \mathit{tabinst}^*,\ \mathsf{mem}\ \mathit{meminst}^*\} \\
\text{(instances)} & \mathit{inst} & ::= & \{\mathsf{func}\ \mathit{cl}^*,\ \mathsf{glob}\ v^*,\ \mathsf{tab}\ i^?,\ \mathsf{mem}\ i^?\} \\
 & \mathit{tabinst} & ::= & \mathit{cl}^* \\
 & \mathit{meminst} & ::= & b^* \\
\text{(closures)} & \mathit{cl} & ::= & \{\mathsf{inst}\ i,\ \mathsf{code}\ f\} \qquad \text{(where } f \text{ is not an import and has all exports } \mathit{ex}^* \text{ erased)} \\
\text{(values)} & v & ::= & t.\mathbf{const}\ c \\
\text{(administrative operators)} & e & ::= & \dots \mid \mathbf{trap} \mid \mathbf{call}\ \mathit{cl} \mid \mathbf{label}\{t^*;e^*\}\ e^*\ \mathbf{end} \mid \mathbf{local}\{i;v^*\}\ e^*\ \mathbf{end} \\
\text{(local contexts)} & L^0 & ::= & v^*\ [\_]\ e^* \\
 & L^{k+1} & ::= & v^*\ \mathbf{label}\{t^*;e^*\}\ L^k\ \mathbf{end}\ e^*
\end{array}
$$

Reduction $\dfrac{s;v^*;e^* \hookrightarrow_i s';v'^*;e'^*}{s;v^*;L^k[e^*] \hookrightarrow_i s';v'^*;L^k[e'^*]}$ $\dfrac{s;v^*;e^* \hookrightarrow_i s';v'^*;e'^*}{s;v_0^*;\mathbf{local}\{i;v^*\}\ e^*\ \mathbf{end} \hookrightarrow_j s';v_0^*;\mathbf{local}\{i;v'^*\}\ e'^*\ \mathbf{end}}$ $\boxed{s;v^*;e^* \hookrightarrow_i s;v^*;e^*}$

$$
\begin{array}{rcll}
(t.\mathbf{const}\ c)\ t.\mathit{unop} & \hookrightarrow & t.\mathbf{const}\ \mathit{unop}_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & t.\mathbf{const}\ c & \text{if } c = \mathit{binop}_t(c_1,c_2) \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.\mathit{binop} & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t.\mathbf{const}\ c)\ t.\mathit{testop} & \hookrightarrow & \mathbf{i32.const}\ \mathit{testop}_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.\mathit{relop} & \hookrightarrow & \mathbf{i32.const}\ \mathit{relop}_t(c_1,c_2) & \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1_\mathit{sx}^? & \hookrightarrow & t_2.\mathbf{const}\ c' & \text{if } c' = \mathrm{cvt}^{\mathit{sx}^?}_{t_1,t_2}(c) \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1_\mathit{sx}^? & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{reinterpret}\ t_1 & \hookrightarrow & t_2.\mathbf{const}\ \mathrm{const}_{t_2}(\mathrm{bits}_{t_1}(c)) & \\
\mathbf{unreachable} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{nop} & \hookrightarrow & \epsilon & \\
v\ \mathbf{drop} & \hookrightarrow & \epsilon & \\
v_1\ v_2\ (\mathbf{i32.const}\ 0)\ \mathbf{select} & \hookrightarrow & v_2 & \\
v_1\ v_2\ (\mathbf{i32.const}\ k+1)\ \mathbf{select} & \hookrightarrow & v_1 & \\
v^n\ \mathbf{block}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}\{t_2^m;\epsilon\}\ v^n\ e^*\ \mathbf{end} & \\
v^n\ \mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}\{t_1^n;\mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end}\}\ v^n\ e^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ 0)\ \mathbf{if}\ \mathit{tf}\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ \mathit{tf}\ e_2^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ k+1)\ \mathbf{if}\ \mathit{tf}\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ \mathit{tf}\ e_1^*\ \mathbf{end} & \\
\mathbf{label}\{t^*;e^*\}\ v^*\ \mathbf{end} & \hookrightarrow & v^* & \\
\mathbf{label}\{t^*;e^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{label}\{t^n;e^*\}\ L^j[v^n\ (\mathbf{br}\ j)]\ \mathbf{end} & \hookrightarrow & v^n\ e^* & \\
(\mathbf{i32.const}\ 0)\ (\mathbf{br_if}\ j) & \hookrightarrow & \epsilon & \\
(\mathbf{i32.const}\ k+1)\ (\mathbf{br_if}\ j) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k)\ (\mathbf{br_table}\ j_1^k\ j\ j_2^*) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k+n)\ (\mathbf{br_table}\ j_1^k\ j) & \hookrightarrow & \mathbf{br}\ j & \\
s;\mathbf{call}\ j & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{func}}(i,j) & \\
s;(\mathbf{i32.const}\ j)\ \mathbf{call_indirect}\ \mathit{tf} & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{tab}}(i,j) & \text{if } s_{\mathsf{tab}}(i,j)_{\mathsf{code}} = (\mathbf{func}\ \mathit{tf}\ \mathbf{local}\ t^*\ e^*) \\
s;(\mathbf{i32.const}\ j)\ \mathbf{call_indirect}\ \mathit{tf} & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
v^n\ (\mathbf{call}\ \mathit{cl}) & \hookrightarrow & \mathbf{local}\{\mathit{cl}_{\mathsf{inst}};v^n\ (t.\mathbf{const}\ 0)^k\}\ \mathbf{block}\ (\epsilon \to t_2^m)\ e^*\ \mathbf{end}\ \mathbf{end}\ \ \dots & \\
\mathbf{local}\{i;v_l^*\}\ v^*\ \mathbf{end} & \hookrightarrow & v^* & \mid\ \dots\text{if } \mathit{cl}_{\mathsf{code}} = (\mathbf{func}\ (t_1^n \to t_2^m)\ \mathbf{local}\ t^k\ e^*) \\
\mathbf{local}\{i;v_l^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{local}\{i;v_l^*\}\ L^{k+1}[\mathbf{return}]\ \mathbf{end} & \hookrightarrow & \mathbf{local}\{i;v_l^*\}\ L^{k+1}[\mathbf{br}\ k]\ \mathbf{end} & \\
v_1^j\ v\ v_2^k;\mathbf{get_local}\ j & \hookrightarrow & v & \\
v_1^j\ v\ v_2^k;v'\ (\mathbf{set_local}\ j) & \hookrightarrow & v_1^j\ v'\ v_2^k;\epsilon & \\
v\ (\mathbf{tee_local}\ j) & \hookrightarrow & v\ v\ (\mathbf{set_local}\ j) & \\
s;\mathbf{get_global}\ j & \hookrightarrow_i & s_{\mathsf{glob}}(i,j) & \\
s;v\ (\mathbf{set_global}\ j) & \hookrightarrow_i & s';\epsilon & \text{if } s' = s \text{ with } \mathsf{glob}(i,j) = v \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}_t(b^*) & \text{if } s_{\mathsf{mem}}(i,k+o,|t|) = b^* \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ \mathit{tp}_\mathit{sx}\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}_t^{\mathit{sx}}(b^*) & \text{if } s_{\mathsf{mem}}(i,k+o,|\mathit{tp}|) = b^* \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ \mathit{tp}_\mathit{sx}^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ a\ o) & \hookrightarrow_i & s';\epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i,k+o,|t|) = \mathrm{bits}_t^{|t|}(c) \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ \mathit{tp}\ a\ o) & \hookrightarrow_i & s';\epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i,k+o,|\mathit{tp}|) = \mathrm{bits}_t^{|\mathit{tp}|}(c) \\
s;(\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ \mathit{tp}^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s;\mathbf{current_memory}] & \hookrightarrow_i & \mathbf{i32.const}\ |s_{\mathsf{mem}}(i,*)|/64\,\mathrm{Ki} & \\
s;(\mathbf{i32.const}\ k)\ \mathbf{grow_memory} & \hookrightarrow_i & s';\mathbf{i32.const}\ |s_{\mathsf{mem}}(i,*)|/64\,\mathrm{Ki} & \text{if } s' = s \text{ with } \mathsf{mem}(i,*) = s_{\mathsf{mem}}(i,*)\ (0)^{k\cdot 64\,\mathrm{Ki}} \\
s;(\mathbf{i32.const}\ k)\ \mathbf{grow_memory} & \hookrightarrow_i & \mathbf{i32.const}\ (-1) &
\end{array}
$$

Figure 1. Small-step reduction rules

[PLDI 2017]

static semantics

stack typing

$$\left.\begin{array}{lcl} (\textbf{i32.const}\ 2) & : & \varepsilon \rightarrow \mathrm{i32} \\ (\textbf{i32.const}\ 5) & : & \varepsilon \rightarrow \mathrm{i32} \\ (\textbf{i32.add}) & : & \mathrm{i32}\ \mathrm{i32} \rightarrow \mathrm{i32} \end{array}\right\} \quad : \quad \varepsilon \rightarrow \mathrm{i32}$$

typing judgement

$$instr^* \;:\; t_1^* \to t_2^*$$

t_1^* are the types of values popped from the stack

t_2^* are the types of values pushed to the stack

$$\overline{t.\mathrm{const}\ c \;:\; \varepsilon \to t}$$

axiom

$$\overline{t.\mathrm{add} \;:\; t\ \ t \to t}$$

$$\overline{\varepsilon \;:\; \varepsilon \to \varepsilon}$$

premise

$$\frac{\mathit{instr}_1^* : t_1^* \to t_2^* \qquad\qquad \mathit{instr}_2^* : t_2^* \to t_3^*}{\mathit{instr}_1^*\ \mathit{instr}_2^* \;:\; t_1^* \to t_3^*}$$

deduction rule

conclusion

$$\frac{\mathit{instr}^* : t_1^* \to t_2^*}{\mathit{instr}^* \;:\; t_0^*\ t_1^* \to t_0^*\ t_2^*}$$

context

$$\frac{C.\text{globals}[x] = t}{\text{global.get } x \ : \ \varepsilon \rightarrow t}$$

context C records types of declarations in scope

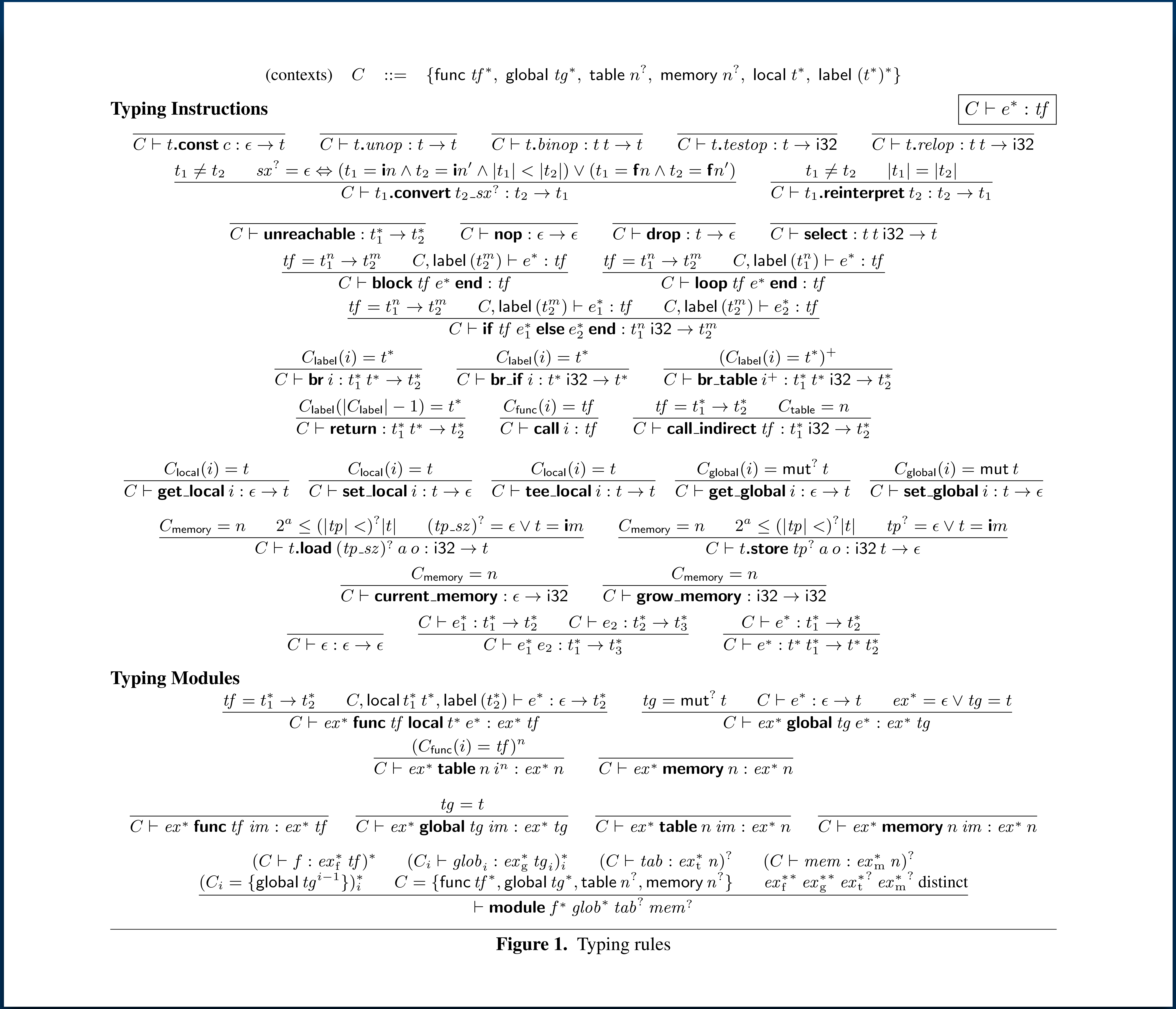
$C \ ::= \ \{\text{globals } t^*, \text{ tables } t[n..m]^*, \text{ memories i8}[n..m]^*, \text{ labels } (t^*)^*\}$

$$C \overset{\text{turnstile}}{\vdash} \mathit{instr}^* : t_1^* \rightarrow t_2^*$$

$$\frac{C.\mathrm{globals}[x] = t}{C \vdash \mathrm{global.get}\ x : \varepsilon \to t}$$

$$\frac{C, \mathrm{labels}\ t_2{}^* \vdash instr^* : t_1{}^* \to t_2{}^* \qquad ft = t_1{}^* \to t_2{}^*}{C \vdash \mathrm{block}\ ft\ \ instr^* : t_1{}^* \to t_2{}^*}$$

$$\frac{C.\mathrm{labels}[l] = t^*}{C \vdash \mathrm{br}\ l : t^* \to \varepsilon}$$



$$(\text{contexts}) \quad C \quad ::= \quad \{\mathsf{func}\ tf^*,\ \mathsf{global}\ tg^*,\ \mathsf{table}\ n^?,\ \mathsf{memory}\ n^?,\ \mathsf{local}\ t^*,\ \mathsf{label}\ (t^*)^*\}$$

Typing Instructions $\boxed{C \vdash e^* : tf}$

$$\frac{}{C \vdash t.\textbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop : t \to t} \qquad \frac{}{C \vdash t.binop : t\ t \to t} \qquad \frac{}{C \vdash t.testop : t \to \mathsf{i32}} \qquad \frac{}{C \vdash t.relop : t\ t \to \mathsf{i32}}$$

$$\frac{t_1 \neq t_2 \qquad sx^? = \epsilon \Leftrightarrow (t_1 = \mathsf{i}n \wedge t_2 = \mathsf{i}n' \wedge |t_1| < |t_2|) \vee (t_1 = \mathsf{f}n \wedge t_2 = \mathsf{f}n')}{C \vdash t_1.\textbf{convert}\ t_2\_sx^? : t_2 \to t_1} \qquad \frac{t_1 \neq t_2 \qquad |t_1| = |t_2|}{C \vdash t_1.\textbf{reinterpret}\ t_2 : t_2 \to t_1}$$

$$\frac{}{C \vdash \textbf{unreachable} : t_1^* \to t_2^*} \qquad \frac{}{C \vdash \textbf{nop} : \epsilon \to \epsilon} \qquad \frac{}{C \vdash \textbf{drop} : t \to \epsilon} \qquad \frac{}{C \vdash \textbf{select} : t\ t\ \mathsf{i32} \to t}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}\,(t_2^m) \vdash e^* : tf}{C \vdash \textbf{block}\ tf\ e^*\ \textbf{end} : tf} \qquad \frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}\,(t_1^n) \vdash e^* : tf}{C \vdash \textbf{loop}\ tf\ e^*\ \textbf{end} : tf}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}\,(t_2^m) \vdash e_1^* : tf \qquad C, \mathsf{label}\,(t_2^m) \vdash e_2^* : tf}{C \vdash \textbf{if}\ tf\ e_1^*\ \textbf{else}\ e_2^*\ \textbf{end} : t_1^n\ \mathsf{i32} \to t_2^m}$$

$$\frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \textbf{br}\ i : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \textbf{br\_if}\ i : t^*\ \mathsf{i32} \to t^*} \qquad \frac{(C_{\mathsf{label}}(i) = t^*)^+}{C \vdash \textbf{br\_table}\ i^+ : t_1^*\ t^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{C_{\mathsf{label}}(|C_{\mathsf{label}}| - 1) = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\mathsf{func}}(i) = tf}{C \vdash \textbf{call}\ i : tf} \qquad \frac{tf = t_1^* \to t_2^* \qquad C_{\mathsf{table}} = n}{C \vdash \textbf{call\_indirect}\ tf : t_1^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{get\_local}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{set\_local}\ i : t \to \epsilon} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{tee\_local}\ i : t \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}^?\ t}{C \vdash \textbf{get\_global}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}\ t}{C \vdash \textbf{set\_global}\ i : t \to \epsilon}$$

$$\frac{C_{\mathsf{memory}} = n \qquad 2^a \leq (|tp| <)^? |t| \qquad (tp\_sz)^? = \epsilon \vee t = \mathsf{i}m}{C \vdash t.\textbf{load}\ (tp\_sz)^?\ a\ o : \mathsf{i32} \to t} \qquad \frac{C_{\mathsf{memory}} = n \qquad 2^a \leq (|tp| <)^? |t| \qquad tp^? = \epsilon \vee t = \mathsf{i}m}{C \vdash t.\textbf{store}\ tp^?\ a\ o : \mathsf{i32}\ t \to \epsilon}$$

$$\frac{C_{\mathsf{memory}} = n}{C \vdash \textbf{current\_memory} : \epsilon \to \mathsf{i32}} \qquad \frac{C_{\mathsf{memory}} = n}{C \vdash \textbf{grow\_memory} : \mathsf{i32} \to \mathsf{i32}}$$

$$\frac{}{C \vdash \epsilon : \epsilon \to \epsilon} \qquad \frac{C \vdash e_1^* : t_1^* \to t_2^* \qquad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*} \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}$$

Typing Modules

$$\frac{tf = t_1^* \to t_2^* \qquad C, \mathsf{local}\ t_1^*\ t^*, \mathsf{label}\,(t_2^*) \vdash e^* : \epsilon \to t_2^*}{C \vdash ex^*\ \textbf{func}\ tf\ \textbf{local}\ t^*\ e^* : ex^*\ tf} \qquad \frac{tg = \mathsf{mut}^?\ t \qquad C \vdash e^* : \epsilon \to t \qquad ex^* = \epsilon \vee tg = t}{C \vdash ex^*\ \textbf{global}\ tg\ e^* : ex^*\ tg}$$

$$\frac{(C_{\mathsf{func}}(i) = tf)^n}{C \vdash ex^*\ \textbf{table}\ n\ i^n : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \textbf{memory}\ n : ex^*\ n}$$

$$\frac{}{C \vdash ex^*\ \textbf{func}\ tf\ im : ex^*\ tf} \qquad \frac{tg = t}{C \vdash ex^*\ \textbf{global}\ tg\ im : ex^*\ tg} \qquad \frac{}{C \vdash ex^*\ \textbf{table}\ n\ im : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \textbf{memory}\ n\ im : ex^*\ n}$$

$$\frac{\begin{array}{cccc} (C \vdash f : ex_{\mathrm{f}}^*\ tf)^* & (C_i \vdash glob_i : ex_{\mathrm{g}}^*\ tg_i)_i^* & (C \vdash tab : ex_{\mathrm{t}}^*\ n)^? & (C \vdash mem : ex_{\mathrm{m}}^*\ n)^? \\ (C_i = \{\mathsf{global}\ tg^{i-1}\})_i^* & C = \{\mathsf{func}\ tf^*, \mathsf{global}\ tg^*, \mathsf{table}\ n^?, \mathsf{memory}\ n^?\} & & ex_{\mathrm{f}}^{**}\ ex_{\mathrm{g}}^{**}\ ex_{\mathrm{t}}^{*?}\ ex_{\mathrm{m}}^{*?}\ \text{distinct} \end{array}}{\vdash \textbf{module}\ f^*\ glob^*\ tab^?\ mem^?}$$

Figure 1. Typing rules

[PLDI 2017]

soundness

Soundness

If $C \vdash instr^* : \varepsilon \to t^*$ and $s : C$,

then $s\ ;\ instr^*$ either diverges,

or $\quad s\ ;\ instr^* \longrightarrow^* s'\ ;\ val^*$

such that $C \vdash val^* : \varepsilon \to t^*$ and $s' : C$.

That is, there is **no undefined behaviour**!

mechanisation

soundness has been machine-verified multiple times

…with theorem provers such as Coq and Isabelle

going next-level:
SpecTec

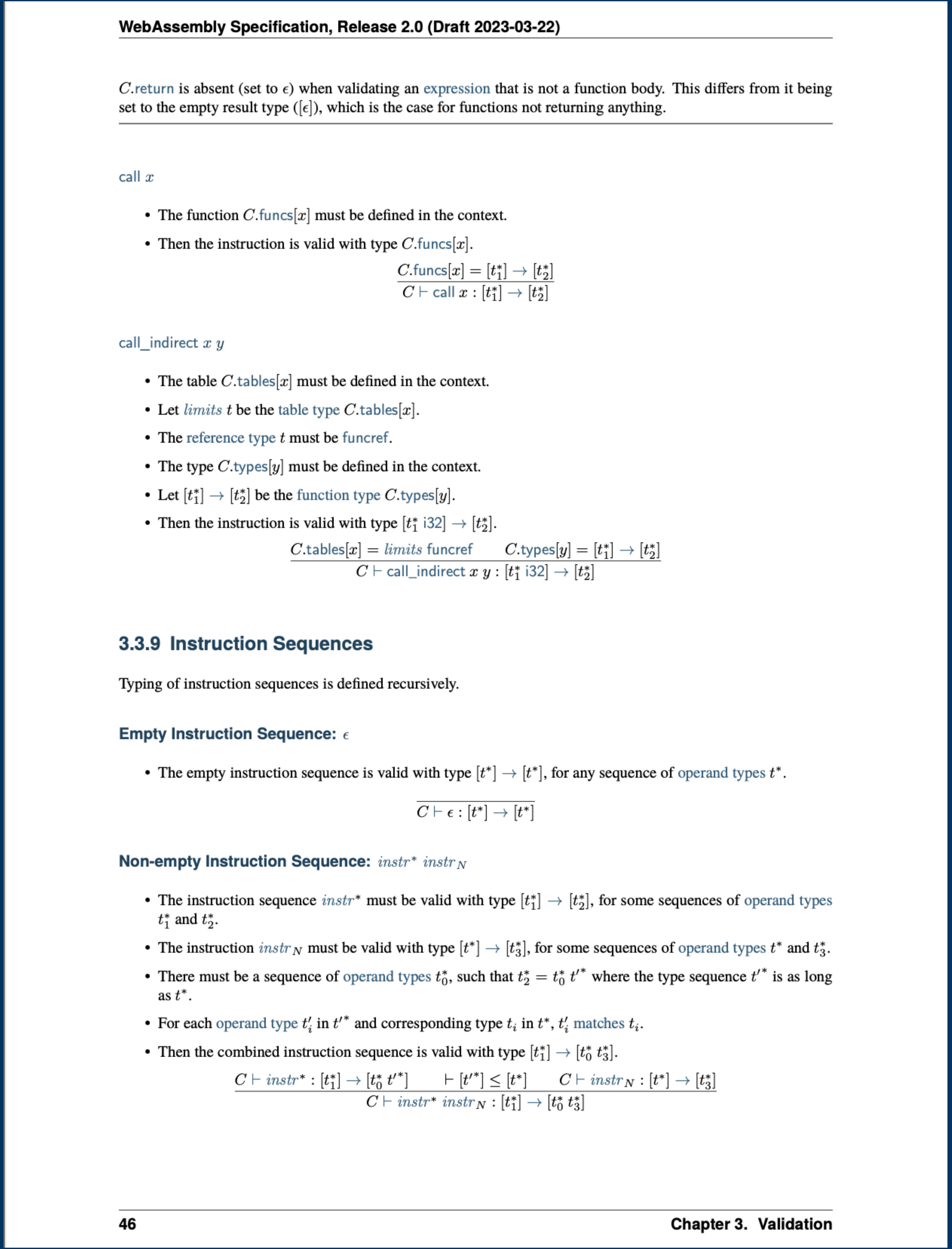
Wasm standard

End-to-end formalisation

… *formal specification* is centerpiece of the official language standard

But for accessibility reasons, folks also want plain English

… *prose specification* also is part of the language standard



$C.\mathsf{return}$ is absent (set to ϵ) when validating an expression that is not a function body. This differs from it being set to the empty result type ($[\epsilon]$), which is the case for functions not returning anything.

$\mathsf{call}\ x$

- The function $C.\mathsf{funcs}[x]$ must be defined in the context.
- Then the instruction is valid with type $C.\mathsf{funcs}[x]$.

$$\frac{C.\mathsf{funcs}[x] = [t_1^*] \to [t_2^*]}{C \vdash \mathsf{call}\ x : [t_1^*] \to [t_2^*]}$$

$\mathsf{call\_indirect}\ x\ y$

- The table $C.\mathsf{tables}[x]$ must be defined in the context.
- Let $\mathit{limits}\ t$ be the table type $C.\mathsf{tables}[x]$.
- The reference type t must be $\mathsf{funcref}$.
- The type $C.\mathsf{types}[y]$ must be defined in the context.
- Let $[t_1^*] \to [t_2^*]$ be the function type $C.\mathsf{types}[y]$.
- Then the instruction is valid with type $[t_1^*\ \mathsf{i32}] \to [t_2^*]$.

$$\frac{C.\mathsf{tables}[x] = \mathit{limits}\ \mathsf{funcref} \qquad C.\mathsf{types}[y] = [t_1^*] \to [t_2^*]}{C \vdash \mathsf{call\_indirect}\ x\ y : [t_1^*\ \mathsf{i32}] \to [t_2^*]}$$

3.3.9 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

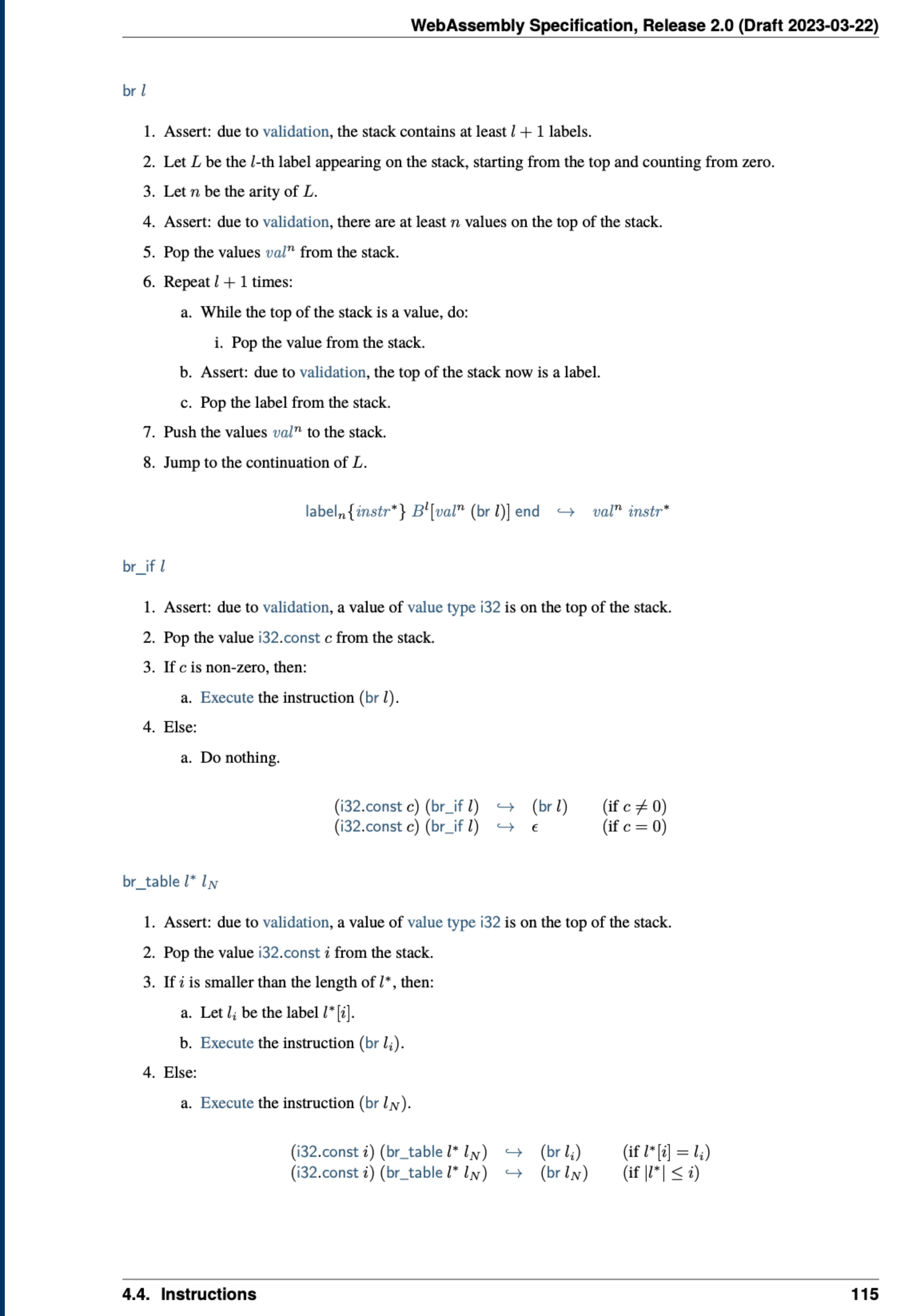
- The empty instruction sequence is valid with type $[t^*] \to [t^*]$, for any sequence of operand types t^*.

$$\frac{}{C \vdash \epsilon : [t^*] \to [t^*]}$$

Non-empty Instruction Sequence: $\mathit{instr}^*\ \mathit{instr}_N$

- The instruction sequence instr^* must be valid with type $[t_1^*] \to [t_2^*]$, for some sequences of operand types t_1^* and t_2^*.
- The instruction instr_N must be valid with type $[t^*] \to [t_3^*]$, for some sequences of operand types t^* and t_3^*.
- There must be a sequence of operand types t_0^*, such that $t_2^* = t_0^*\ t'^*$ where the type sequence t'^* is as long as t^*.
- For each operand type t'_i in t'^* and corresponding type t_i in t^*, t'_i matches t_i.
- Then the combined instruction sequence is valid with type $[t_1^*] \to [t_0^*\ t_3^*]$.

$$\frac{C \vdash \mathit{instr}^* : [t_1^*] \to [t_0^*\ t'^*] \qquad \vdash [t'^*] \leq [t^*] \qquad C \vdash \mathit{instr}_N : [t^*] \to [t_3^*]}{C \vdash \mathit{instr}^*\ \mathit{instr}_N : [t_1^*] \to [t_0^*\ t_3^*]}$$



$\mathsf{br}\ l$

1. Assert: due to validation, the stack contains at least $l+1$ labels.
2. Let L be the l-th label appearing on the stack, starting from the top and counting from zero.
3. Let n be the arity of L.
4. Assert: due to validation, there are at least n values on the top of the stack.
5. Pop the values val^n from the stack.
6. Repeat $l+1$ times:
    a. While the top of the stack is a value, do:
        i. Pop the value from the stack.
    b. Assert: due to validation, the top of the stack now is a label.
    c. Pop the label from the stack.
7. Push the values val^n to the stack.
8. Jump to the continuation of L.

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

$\mathsf{br\_if}\ l$

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. If c is non-zero, then:
    a. Execute the instruction $(\mathsf{br}\ l)$.
4. Else:
    a. Do nothing.

$$\begin{array}{lcll} (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & (\mathrm{if}\ c \neq 0) \\ (\mathsf{i32.const}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & (\mathrm{if}\ c = 0) \end{array}$$

$\mathsf{br\_table}\ l^*\ l_N$

1. Assert: due to validation, a value of value type $\mathsf{i32}$ is on the top of the stack.
2. Pop the value $\mathsf{i32.const}\ i$ from the stack.
3. If i is smaller than the length of l^*, then:
    a. Let l_i be the label $l^*[i]$.
    b. Execute the instruction $(\mathsf{br}\ l_i)$.
4. Else:
    a. Execute the instruction $(\mathsf{br}\ l_N)$.

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

Prose essentially is a manual text rendering of the formal rules

- …200 pages instead of 2, *extremely* laborious
- …plus, for political reasons, the entire spec had to be written in markdown

Both formats are error-prone and make for nightmarish code reviews

- …Latex is a write-only language
- …markdown verbose; diff-unfriendly

proposals

Wasm currently has

- 8 completed proposals (making up Wasm 2.0)
- 25+ active proposals

Every new proposal needs to extend *both* formal and prose rules

Diffs for proposals range from 1 to 100 Kloc (spec document, interpreter, test suite)

Proposal champions also need to extend the reference interpreter

- …essentially, yet another rendering of the formal rules in OCaml

We want a language for this!

A Wasm Spec DSL

Single source of truth

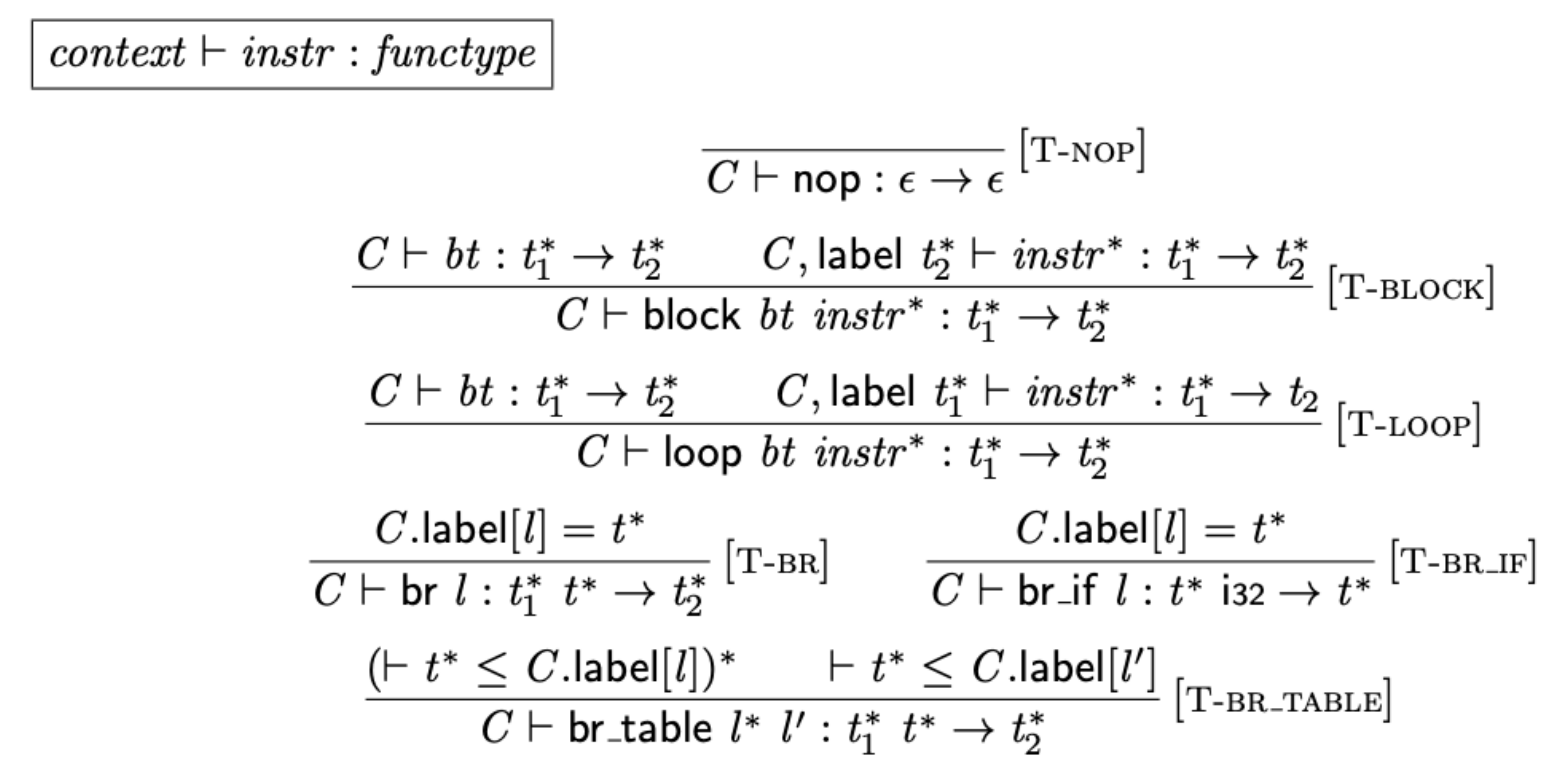
Easy to write, read, diff, and review; meta-level error checking

Transformable into all the aforementioned representations

…sufficient for math rendering, natural language, and semantic modelling

One frontend – many backends

```
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*
  -- InstrSeq_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*
  -- InstrSeq_ok: C, LABEL t_1* |- instr* : t_1* -> t_2

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

rule Instr_ok/br_if:
  C |- BR_IF l : t* I32 -> t*
  -- iff 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])*
```



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

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

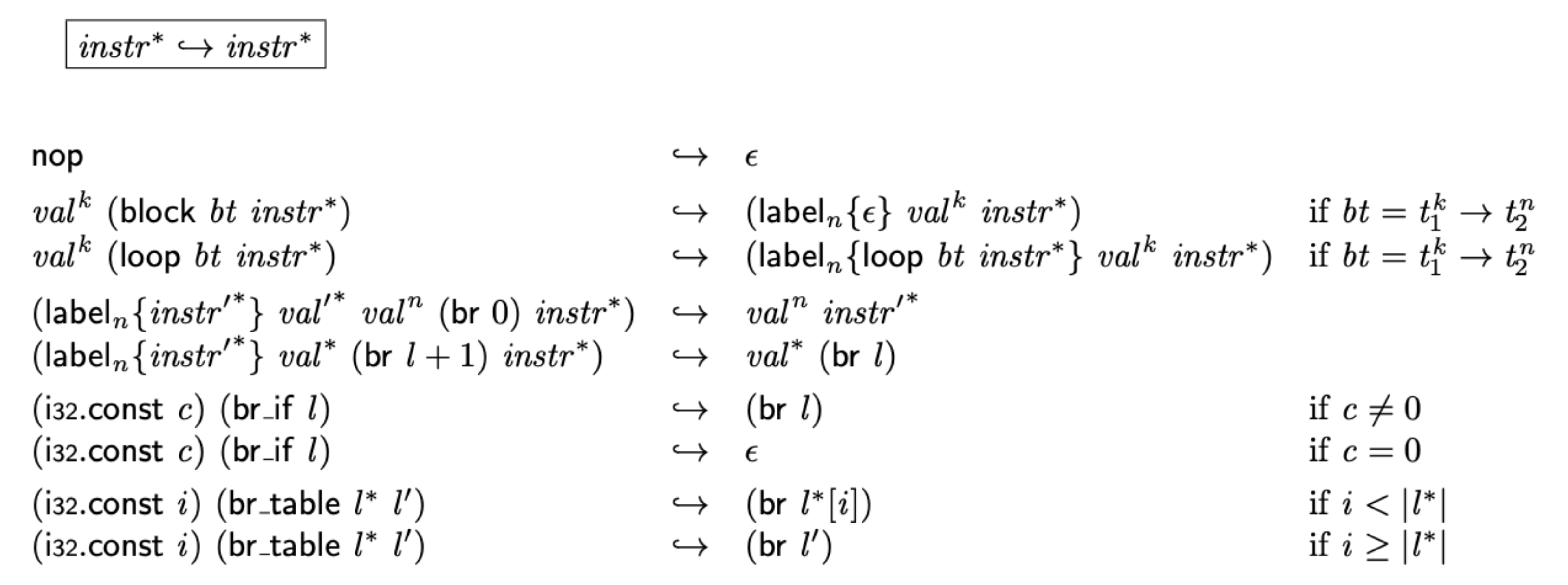
$$\frac{C \vdash bt : t_1^* \to t_2^* \qquad C, \mathsf{label}\ t_2^* \vdash instr^* : t_1^* \to t_2^*}{C \vdash \mathsf{block}\ bt\ instr^* : t_1^* \to t_2^*}\ [\text{T-BLOCK}]$$

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

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

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

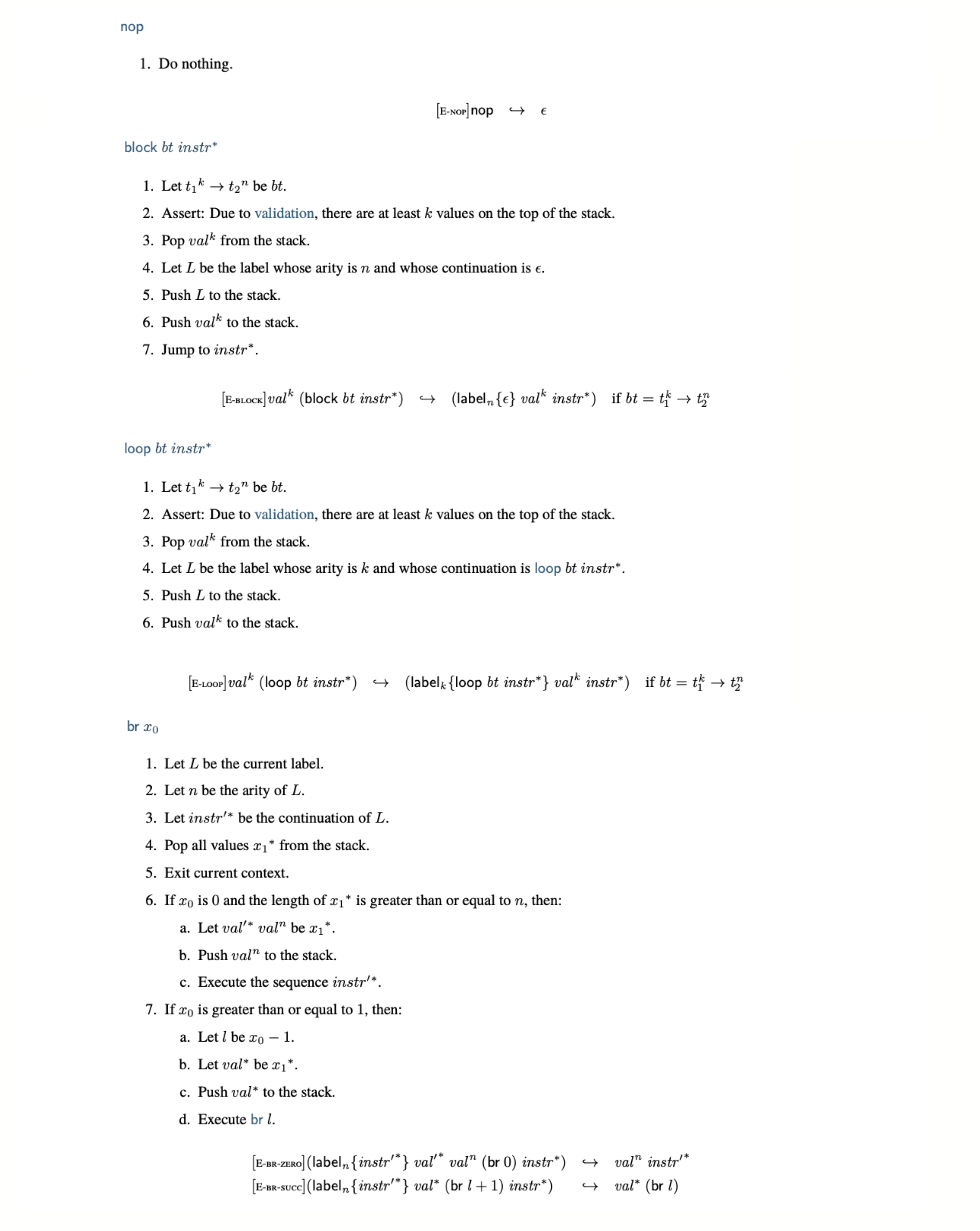
```
relation Step_pure: config ~> config

rule Step_pure/nop:
  NOP  ~>  epsilon

rule Step_pure/block:
  val^k (BLOCK bt instr*)  ~>  (LABEL_n`{epsilon} val^k instr*)
  -- iff bt = t_1^k -> t_2^n

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

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

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

rule Step_pure/br_table-le:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l')
  -- iff 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*)
  -- iff bt = t_1^k -> t_2^n

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

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

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

rule Step_pure/br_table-le:
  (CONST I32 i) (BR_TABLE l* l')  ~>  (BR l')
  -- iff 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 $instr^*$

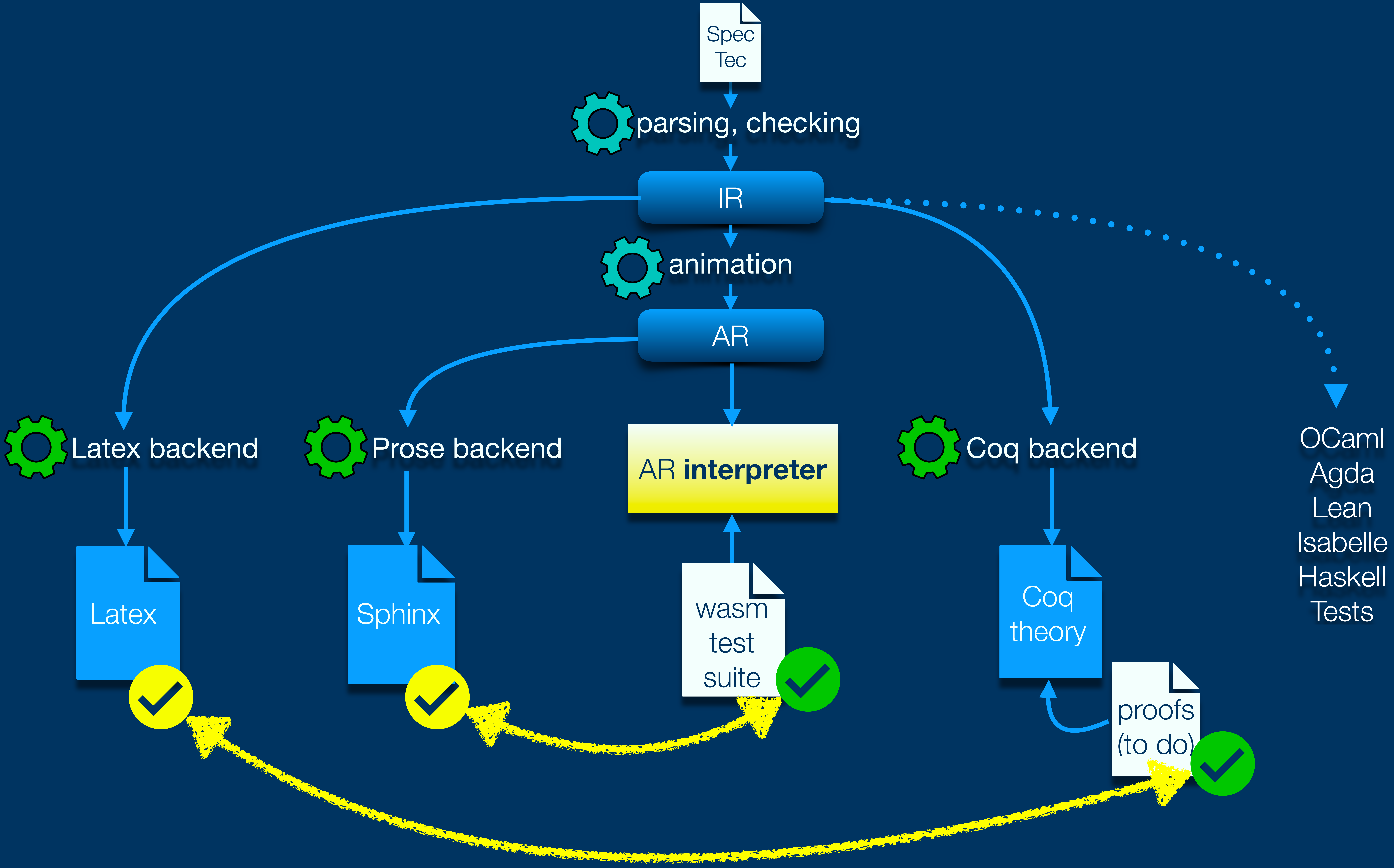
1. Let ${t_1}^k \to {t_2}^n$ be bt.
2. Assert: Due to validation, there are at least k values on the top of the stack.
3. Pop val^k from the stack.
4. Let L be the label whose arity is k and whose continuation is loop bt $instr^*$.
5. Push L to the stack.
6. Push val^k to the stack.

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

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.

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


Spec
Tec
parsing, checking
IR
animation
AR
Latex backend
Prose backend
AR **interpreter**
Coq backend
OCaml
Agda
Lean
Isabelle
Haskell
Tests
Latex
Sphinx
wasm
test
suite
Coq
theory
proofs
(to do)

Summary

spelled Wasm, not ~~WASM~~

not a web technology

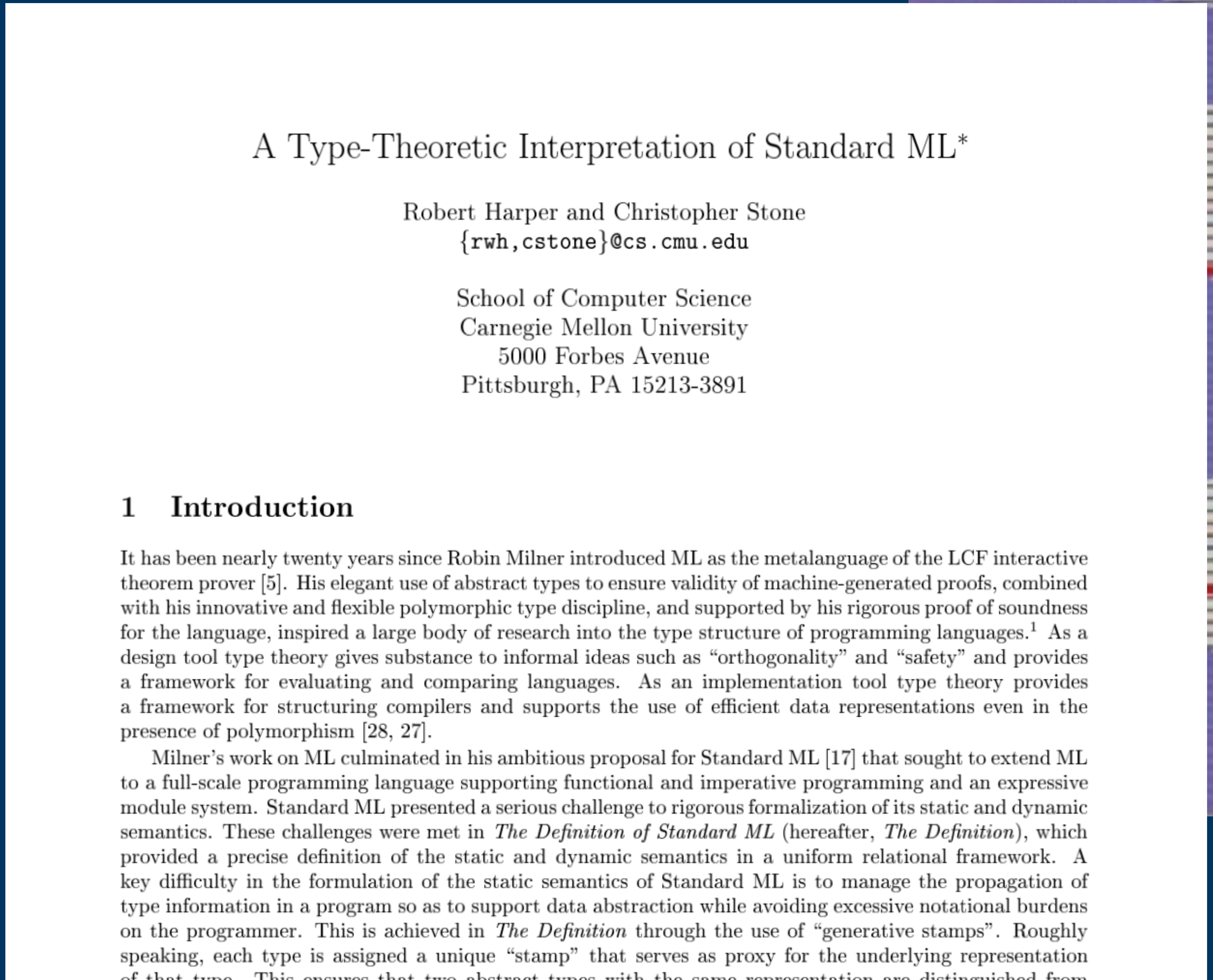
low-level and language-neutral

formally specified end-to-end

nothing new there, just textbook techniques!

formal rigour and machine verification in the mainstream

lead to a cleaner and safer design



A Type-Theoretic Interpretation of Standard ML*

Robert Harper and Christopher Stone
{rwh,cstone}@cs.cmu.edu

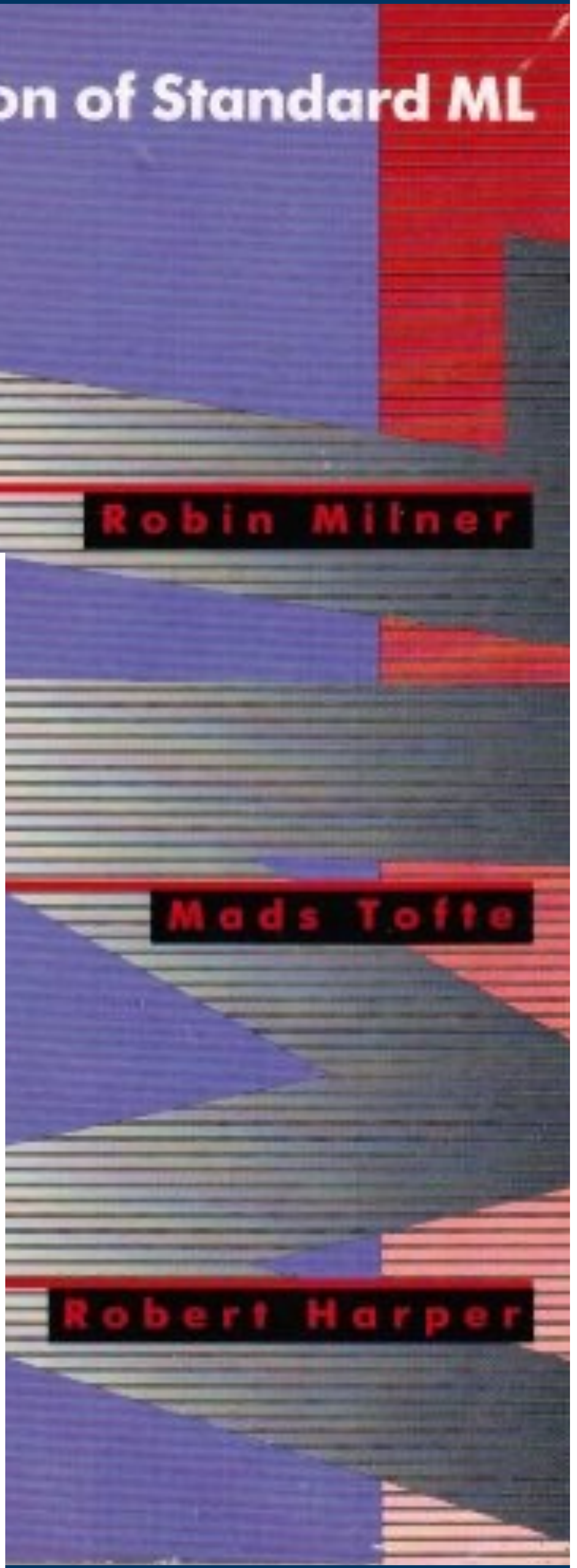
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

1 Introduction

It has been nearly twenty years since Robin Milner introduced ML as the metalanguage of the LCF interactive theorem prover [5]. His elegant use of abstract types to ensure validity of machine-generated proofs, combined with his innovative and flexible polymorphic type discipline, and supported by his rigorous proof of soundness for the language, inspired a large body of research into the type structure of programming languages.[1] As a design tool type theory gives substance to informal ideas such as "orthogonality" and "safety" and provides a framework for evaluating and comparing languages. As an implementation tool type theory provides a framework for structuring compilers and supports the use of efficient data representations even in the presence of polymorphism [28, 27].

Milner's work on ML culminated in his ambitious proposal for Standard ML [17] that sought to extend ML to a full-scale programming language supporting functional and imperative programming and an expressive module system. Standard ML presented a serious challenge to rigorous formalization of its static and dynamic semantics. These challenges were met in *The Definition of Standard ML* (hereafter, *The Definition*), which provided a precise definition of the static and dynamic semantics in a uniform relational framework. A key difficulty in the formulation of the static semantics of Standard ML is to manage the propagation of type information in a program so as to support data abstraction while avoiding excessive notational burdens on the programmer. This is achieved in *The Definition* through the use of "generative stamps". Roughly speaking, each type is assigned a unique "stamp" that serves as proxy for the underlying representation of that type. This ensures that two abstract types with the same representation are distinguished from

---



The Definition of Standard ML

Robin Milner

Mads Tofte

Robert Harper

Towards a Mechanized Metatheory of Standard ML *

Daniel K. Lee Karl Crary Robert Harper
Carnegie Mellon University
{dklee,crary,rwh}@cs.cmu.edu

Abstract

We present an internal language with equivalent expressive power to Standard ML, and discuss its formalization in LF and the machine-checked verification of its type safety in Twelf. The internal language is intended to serve as the target of elaboration in an elaborative semantics for Standard ML in the style of Harper and Stone. Therefore, it includes all the programming mechanisms necessary to implement Standard ML, including translucent modules, abstraction, polymorphism, higher kinds, references, exceptions, recursive types, and recursive functions. Our successful formalization of the proof involved a careful interplay between the precise formulations of the various mechanisms, and required the invention of new representation and proof techniques of general interest.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; Syntax; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Languages, Verification

Keywords Standard ML, language definitions, type safety, mechanized metatheory, logical frameworks, Twelf

1. Introduction

A formal definition of a programming language provides a rigorous, implementation-independent description of the semantics of well-formed programs. By giving a precise meaning to programs a formal definition provides the foundation for building a community of users, for ensuring compatibility of implementations, and for proving properties of the language and programs written in it. But a formal definition does not stand on its own, but must be supported by a body of metatheory that establishes both its internal consistency and coherence with external expectations.

The formal definition of a full-scale programming language can easily run into hundreds of pages, as exemplified by The Definition of Standard ML [21]. Verifying the metatheory of such a language taxes, or even exceeds, human capabilities. Absent complete verification, the best alternative is to employ well-established methods, such as type systems and operational semantics, supported by small case-studies that expose pitfalls. But even using these best practices, errors and inconsistencies arise that are not easily discovered. Moreover, as languages evolve, so must the metatheory that supports it, introducing further opportunities for error.

A promising approach to reducing error is to use mechanized verification tools to ease the burden of proving properties of language definitions. Ideally, a language definition would come equipped with a body of metatheory that is mechanically checked against the definition and that can be extended as need and interest demands. With the development of powerful tools such as mechanical theorem provers and logical frameworks, it is becoming feasible to put this idea into practice. For example, Klein and Nipkow [17] have recently used the Isabelle theorem prover [23] to formalize a large part of the Java programming language and to prove type safety for it.

In this paper we report on the use of the Twelf implementation [27] of the LF logical framework [12] to verify the type safety of the full Standard ML programming language. To our knowledge this is the first mechanical verification of safety for a language of this scale. The first mechanical formalizations of significant subsets of The Definition of Standard ML were performed independently by Syme [36] and VanInwegen and Gunter [39] using HOL [10] for the purpose of establishing determinicity of evaluation. An attempt by VanInwegen [38] to prove type safety was partially successful, but ran into difficulties with the formalism of The Definition of Standard ML, the immaturity of verification tools and methodology at that time, and the unsoundness of the language itself. Our approach draws on intervening experience with logical frameworks [12, 27] and with formalizing language definitions using type-theoretic techniques [16]. Perhaps the most significant lesson to be drawn from VanInwegen's and our experience is that language definitions must be formulated with mechanical verification of metatheory in mind. The formulation of the definition provides the framework for verification, but the demands of verification must also be permitted to influence the definition. Just as programs ought to be written in conjunction with proofs of their key properties, so too must language definitions be developed hand-in-hand with their verification.

2. Overview

Our approach is based on the type-theoretic definition of Standard ML given by Harper and Stone [16]. The Harper-Stone semantics divides the definition of the language into two aspects:

1. *Elaboration*, which translates the *external language*, the abstract syntax of Standard ML, into the *internal language*, a well-behaved type theory based on the translucent sums formalism of Harper and Lillibridge [14]. Elaboration performs type reconstruction, overloading resolution, equality compilation, pattern compilation, and coercive signature matching, resulting in a well-typed term of the internal language.

---

* This work is supported by the National Science Foundation under grant ITR/SY+SI 0121633 and a Graduate Research Fellowship, and by a grant from the Alfred P. Sloan foundation.

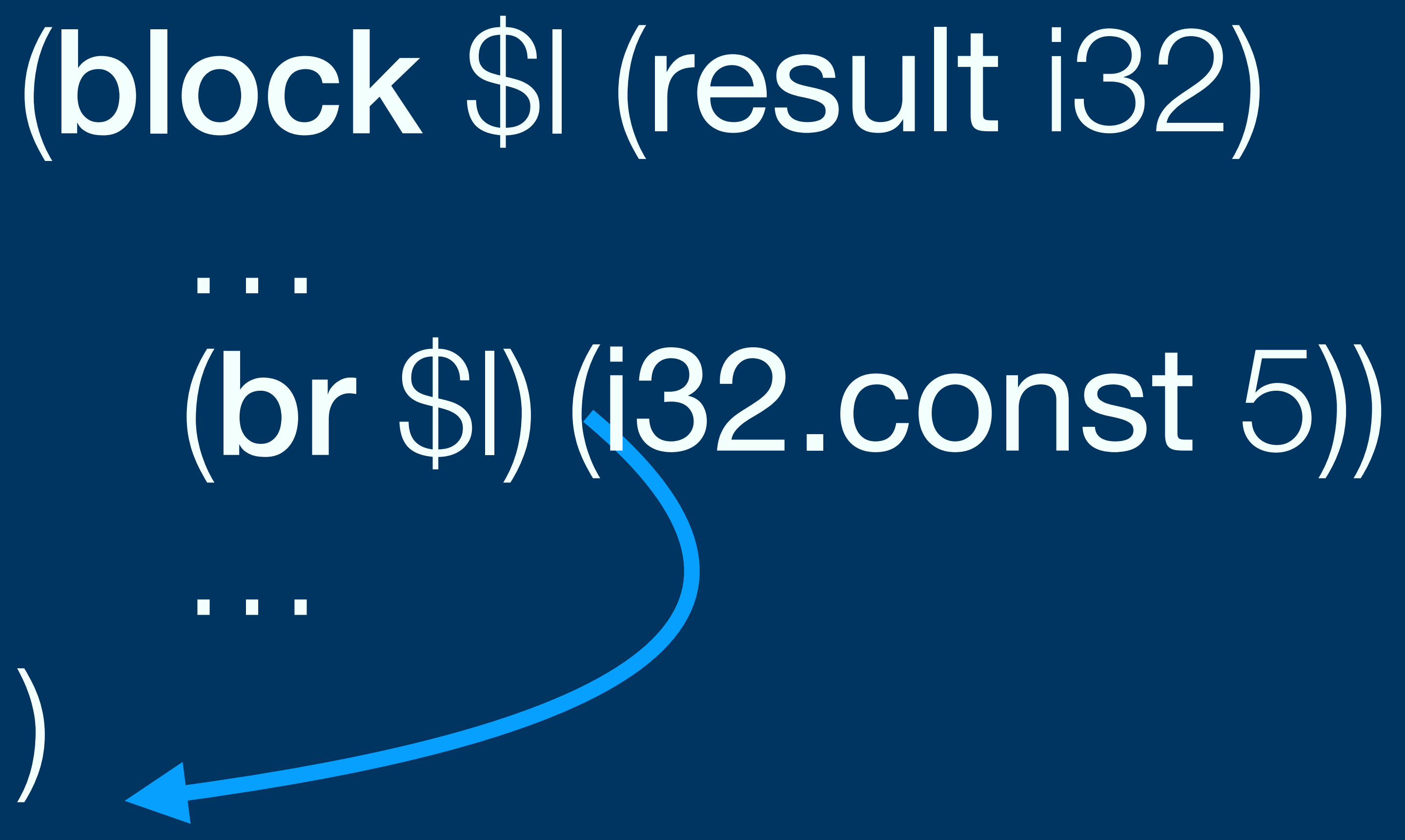
POPL'07 January 17–19, 2007, Nice, France.

outtakes

```
(module
  (import "env" "mem" (memory 10))
  (export "sum" (func $sum))
  (func $sum (param $ptr i32) (param $end i32) (result f64)
    (local $r f64)          ;; double r = 0.0
    (f64.const 0)
    (set_local $r)
    (loop $continue
      (get_local $ptr)      ;; ptr < end
      (get_local $end)
      (i32.lt)
      (if
        (get_local $r)      ;; r += *ptr
        (get_local $ptr)
        (f64.load)
        (f64.add)
        (set_local $r)
        (get_local $ptr)    ;; ptr++
        (i32.const 8)
        (i32.add)
        (set_local $ptr)
        (br $continue)
      )
    )
    (get_local $r)
  )
)
```

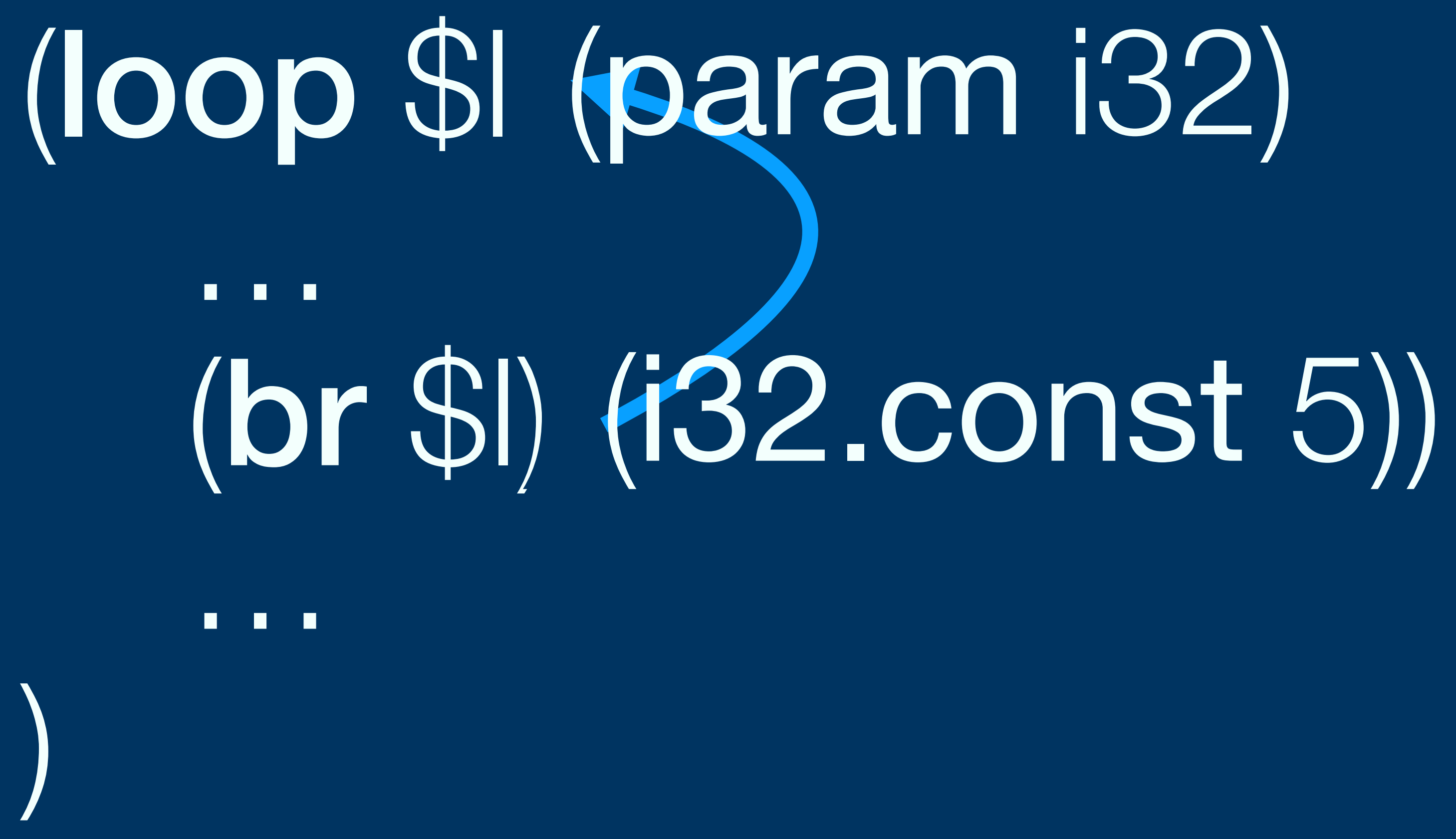
```
double sum(double* ptr, double* end) {
    double r = 0.0;
    while (ptr < end) {
        r += *ptr++;
    }
    return r;
}
```

control flow



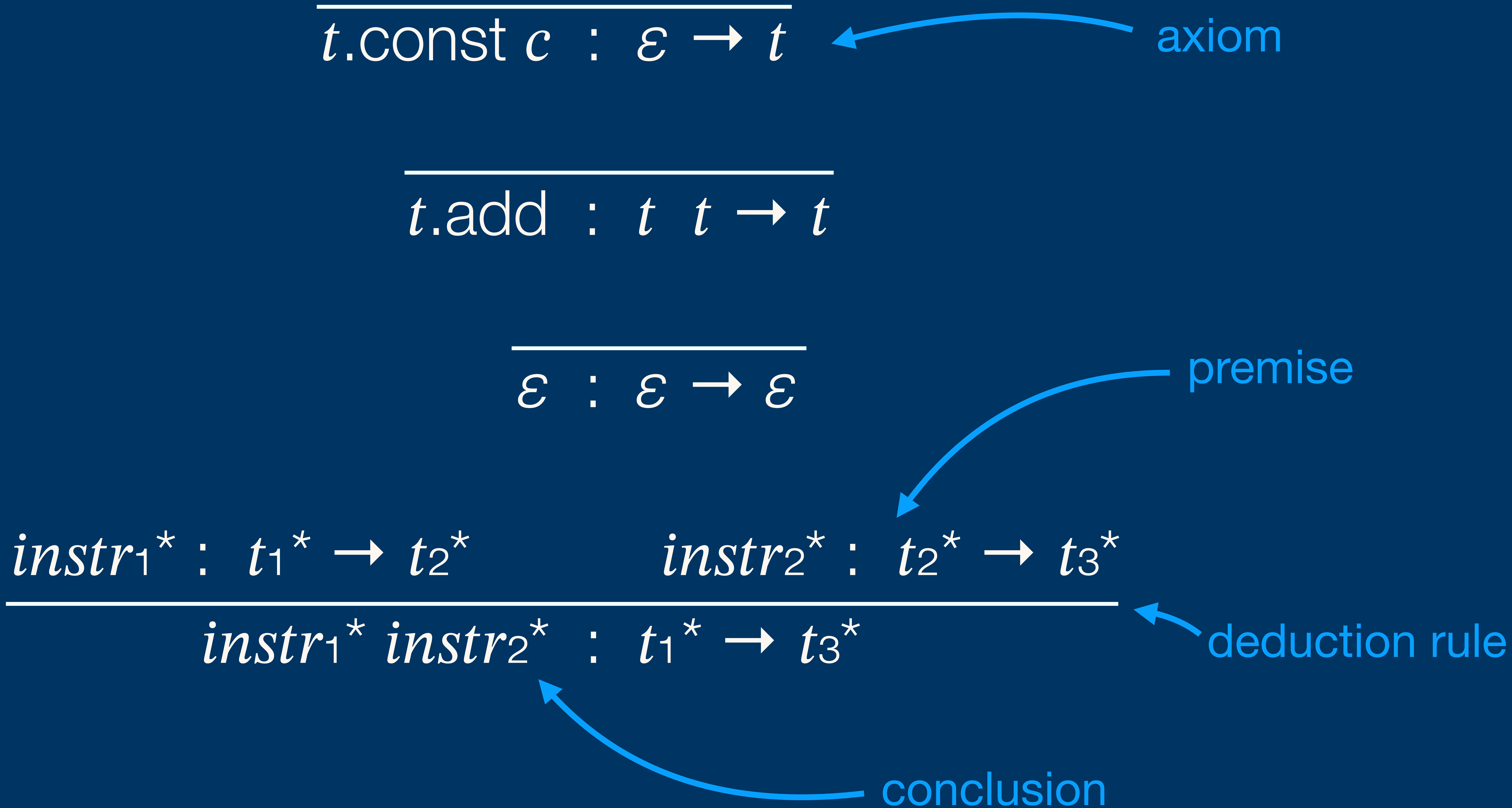
```
(block $l (result i32)
  …
  (br $l) (i32.const 5))
  …
)
```

break



```
(loop $l (param i32)
  …
  (br $l) (i32.const 5))
  …
)
```

continue



$$\frac{}{t.\mathrm{const}\ c \ :\ \varepsilon \to t}$$

axiom

$$\frac{}{t.\mathrm{add} \ :\ t \ \ t \to t}$$

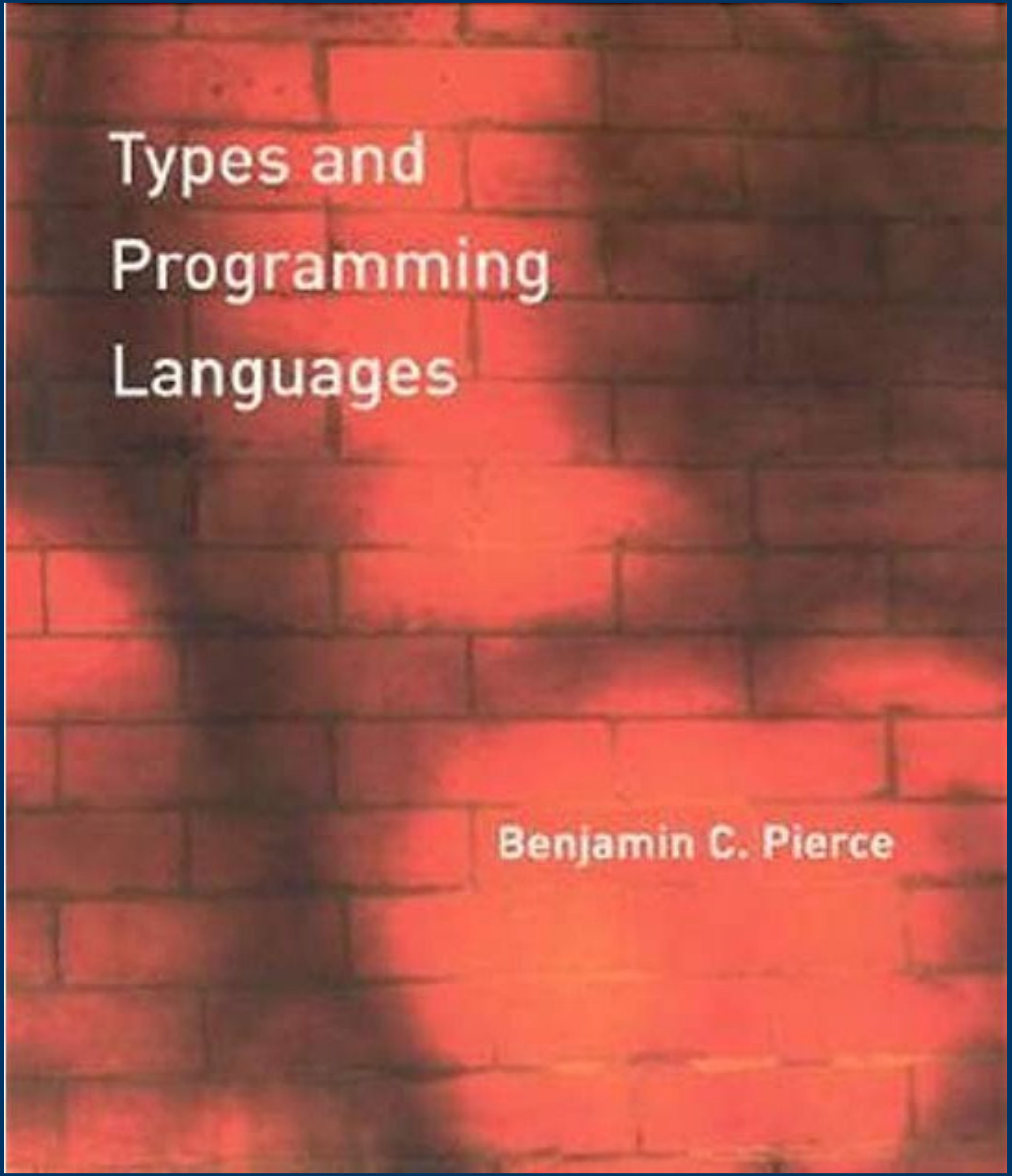
$$\frac{}{\varepsilon \ :\ \varepsilon \to \varepsilon}$$

premise

$$\frac{instr_1{}^* : \ t_1{}^* \to t_2{}^* \qquad\qquad instr_2{}^* : \ t_2{}^* \to t_3{}^*}{instr_1{}^*\ instr_2{}^* \ :\ t_1{}^* \to t_3{}^*}$$

deduction rule

conclusion



Types and Programming Languages

Benjamin C. Pierce