Mini-Wasm Tutorial

1 Introduction

In this tutorial, we will practice how to write a spec in Wasm-DSL. Here, instead of a full Wasm, a very simplified version of Wasm (which we call Mini-Wasm) is used as our goal.

1.1 Working Directory

The directory `<root>/spectec/tutorial` is where we will work on.

1.2 How To Run

Running `make` in `<root>/spectec` will yield an executable file `watsup` in same directory. You can use this to generate various types of spec. For example, in directory `<root>/spectec`:

```
./watsup ./tutorial/*.watsup --prose
```

will generate prose, using every `./watsup` file in directory `./tutorial` as input. There are various options including `--interpreter`, `--latex`, or `--print-il`. You can see all possible options with command `./watsup -help`.

2 Building Blocks of Wasm-DSL

First, we'll study building blocks of Wasm-DSL, and how to use them.
(For now, You don't have to understand this part completely. You may start from Section 3, and refer to this section when you want to.)

2.1 Syntax Definitions

Syntax definitions describe the grammar of the input language or auxiliary constructs. These are essentially type definitions for Wasm-DSL.

```
syntax <name_of_syntax> = <case> | <case> | <case>
```

Defining syntax in Wasm-DSL is basically done as above. Use keyword `syntax`, write the name of syntax in lowercase, and simply list the possible cases with the separator `|`. Here, each of the case can be a nonterminal node (usually written in lowercases) which refers to another syntax, or a terminal node (usually written in uppercases).

2.2 Variable Declarations

Variable declarations ascribe the syntactic class (i.e., type) that meta variables used in rules range over.

```
var <name_of_var> : <type>
```

We can explicitly declare a metavariable as above. Use keyword `var`, and give its name and type. Here, `<type>` may be a complex form, which contains iteration, parametric syntax, etc.

Also, every syntax name is implicitly usable as a variable of the respective type. For example, instead of below code:

```
var t : valtype

rule Step/example:
  z; (CONST t c_1) (CONST t c_2)  ~>  z; (CONST t c_1)
```

we can write like this:

```
rule Step/example:
  z; (CONST valtype c_1) (CONST valtype c_2)  ~>  z; (CONST valtype c_1)
```

Also, we can use a metavariable without declaration in suitable situations. In this case, its type will be inferred. Now, we may rewrite the above rule as:

```
rule Step/example:
  z; (CONST anyname c_1) (CONST anyname c_2)  ~>  z; (CONST anyname c_1)
```

2.3 Relation Declarations

Relation declarations, defining the shape of judgement forms, such as typing or reduction relations. These are essentially type declarations for the meta language.
Below is a general form of relation, which is quite brief:

```
relation <name_of_relation>: <content_of_relation>
```

Content of relation may vary. For example, general form of typing relation will be:

```
relation <name_of_relation>: <type_of_context> |- <type> : <type>
```

or, general form of reduction relation will be:

```
relation <name_of_relation>: <type> ~> <type>
```

2.4 Rule Definitions

```
rule <name_of_relation>/<name_of_rule>:
  <content_of_rule>
  -- <name_of_other_relation>: <premise>
  -- if <condition>
```

We can define the individual rules for each relation as above. `<content_of_rule>` has different form, depending on its relation. Every rule is named, so that it can be referenced.
You can add premises by including the name of referenced relation, and conditions using keyword `if`.

2.5 Auxiliary Functions

```
def $<func_name>(<type_of_arg1>, <type_of_arg2>, ... , <type_of_argn>) : <result_type>
def $<func_name>(<arg1>, <arg2>, ... , <argn>) = <result>
```

We can declare auxiliary functions as above. The first line indicates the type of arguments and result of the function. The second line is function body, which describes how the actual result comes out.
Function body can be multiple cases, which defines a function by pattern matching.
For example, we can define a function `size`, which returns the size of a `valtype` as:

```
def $size(valtype) : nat
def $size(I32) = 32
def $size(I64) = 64
```

Now, the function `size` will return `32` for input `I32`, and `64` for input `I64`.

We can also add condition to a function body.
For example, we can define a function `min`, which returns a smaller integer between two:

```
def $min(nat, nat) : nat
def $min(i, j) = i
  -- if i < j
def $min(i, j) = j
```

Now, the function `min` will return `i` if `i < j`, else `j`.
Functions can be used when defining other functions or reduction rules. When using function, write as below form:

```
$<func_name>(<arg1>, <arg2>, ... , <argn>)
```

For example, you can use the function `size` to define the syntax `num_` as below:

```
syntax num_(valtype) = iN($size(valtype))
```

3 Basic Syntax

Now, we will start from writing the basic syntax of Mini-Wasm. Refer to Appendix A.1 for a full version.
Make a new file `1-syntax.wastup` and write on it.
We'll declare each of the syntax one by one.

3.1 N, n

$$\begin{array}{rcl} N & ::= & \mathbb{N} \\ n & ::= & \mathbb{N} \end{array}$$

Use a pre-defined syntax `nat`, to indicate any natural number.
Hint: Refer to Subsection 2.1.
Answer:

```
syntax N = nat
syntax n = nat
```

This means the syntax `N` and `n` is a natural number.

3.2 iN

$$\text{(integer)} \quad iN(N) \quad ::= \quad 0 \mid \ldots \mid 2^N - 1$$

You can declare a parametric syntax as below:

```
syntax <name_of_syntax>(<parameter>) = <case> | <case> | <case>
```

where <case>s contain <parameter>.
Use `...` to indicate a range.
Answer:

```
syntax iN(N) = 0 | ... | 2^N-1
```

Now, the syntax `iN` is declared in regard with parameter `N`.

3.3 $char$, $name$

$$\begin{array}{lrcl} \text{(character)} & char & ::= & \mathrm{U{+}00} \mid \ldots \mid \mathrm{U{+}D7FF} \mid \mathrm{U{+}E000} \mid \ldots \mid \mathrm{U{+}10FFFF} \\ \text{(name)} & name & ::= & char^* \end{array}$$

Use a built-in syntax like below:

```
U+0000 | ... | U+1111
```

to indicate a range of Unicode code points.
Use `*` to represent a sequence.
Answer:

```
syntax char = U+0000 | ... | U+D7FF | U+E000 | ... | U+10FFFF
syntax name = char*
```

3.4 idx, $typeidx$, $funcidx$, $labelidx$, $localidx$

$$\begin{array}{lrcl} \text{(index)} & idx & ::= & iN(32) \\ \text{(type index)} & typeidx & ::= & idx \\ \text{(function index)} & funcidx & ::= & idx \\ \text{(label index)} & labelidx & ::= & idx \\ \text{(local index)} & localidx & ::= & idx \end{array}$$

Answer:

```
syntax idx = iN(32)
syntax typeidx = idx
syntax funcidx = idx
syntax labelidx = idx
syntax localidx = idx
```

3.5 $valtype$

$$\text{(number type)} \quad valtype \quad ::= \quad \mathsf{i32} \mid \mathsf{i64}$$

Answer:

```
syntax valtype = I32 | I64
```

3.6 *functype*

$$\text{(function type)} \quad \mathit{functype} \quad ::= \quad \mathit{valtype}^* \rightarrow \mathit{valtype}^*$$

Use `->` to indicate a function:
Answer:

```
syntax functype = valtype* -> valtype*
```

3.7 *externtype*

$$\text{(external type)} \quad \mathit{externtype} \quad ::= \quad \mathsf{func}\ \mathit{functype}$$

Answer:

```
syntax externtype = FUNC functype
```

3.8 size, $num_{valtype}$

$$\begin{aligned} \mathrm{size}(\mathsf{i32}) &= 32 \\ \mathrm{size}(\mathsf{i64}) &= 64 \end{aligned}$$

$$\mathit{num}_{\mathit{valtype}} \quad ::= \quad \mathit{iN}(\mathrm{size}(\mathit{valtype}))$$

You need to declare an auxiliary function named **`size`** here. About auxiliary function, refer to Subsection 2.5.
Hint: The type declaration of function **`size`** is as follows:

```
def $size(valtype) : nat
```

Answer:

```
def $size(valtype) : nat
def $size(I32) = 32
def $size(I64) = 64

syntax num_(valtype) = iN($size(valtype))
```

The first line declares the type of function **`size`**, and the next two lines give the actual definition of it. Then, you can use **`size`** to declare **`num_`**.

3.9 *binop*

$$\mathit{binop} \quad ::= \quad \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}$$

Answer:

```
syntax binop = ADD | SUB | MUL | DIV
```

3.10 *instr*

$$
\begin{array}{llcl}
\text{(instruction)} & \mathit{instr} & ::= & \mathsf{nop} \\
 & & | & \mathsf{drop} \\
 & & | & \mathsf{select} \\
 & & | & \mathsf{block}\ \mathit{functype}\ \mathit{instr}^* \\
 & & | & \mathsf{loop}\ \mathit{functype}\ \mathit{instr}^* \\
 & & | & \mathsf{if}\ \mathit{functype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^* \\
 & & | & \mathsf{br}\ \mathit{labelidx} \\
 & & | & \mathsf{br\_if}\ \mathit{labelidx} \\
 & & | & \mathsf{call}\ \mathit{funcidx} \\
 & & | & \mathsf{return} \\
 & & | & \mathsf{const}\ \mathit{valtype}\ \mathit{num}_{\mathit{valtype}} \\
 & & | & \mathsf{binop}\ \mathit{valtype}\ \mathit{binop} \\
 & & | & \mathsf{local.get}\ \mathit{localidx} \\
 & & | & \mathsf{local.set}\ \mathit{localidx}
\end{array}
$$

Answer:

```
syntax instr =
  | NOP
  | DROP
  | SELECT
  | BLOCK functype instr*
  | LOOP functype instr*
  | IF functype instr* ELSE instr*
  | BR labelidx
  | BR_IF labelidx
  | CALL funcidx
  | RETURN
  | CONST valtype num_(valtype)
  | BINOP valtype binop
  | LOCAL.GET localidx
  | LOCAL.SET localidx
```

3.11 *expr*

$$
\text{(expression)} \quad \mathit{expr} \quad ::= \quad \mathit{instr}^*
$$

Answer:

```
syntax expr = instr*
```

3.12 *module*

$$
\begin{array}{lrcl}
\text{(type)} & \mathit{type} & ::= & \mathsf{type}\ \mathit{functype} \\
\text{(local)} & \mathit{local} & ::= & \mathsf{local}\ \mathit{valtype} \\
\text{(function)} & \mathit{func} & ::= & \mathsf{func}\ \mathit{typeidx}\ \mathit{local}^*\ \mathit{expr} \\
\text{(external index)} & \mathit{externidx} & ::= & \mathsf{func}\ \mathit{funcidx} \\
\text{(export)} & \mathit{export} & ::= & \mathsf{export}\ \mathit{name}\ \mathit{externidx} \\
\text{(module)} & \mathit{module} & ::= & \mathsf{module}\ \mathit{type}^*\ \mathit{func}^*\ \mathit{export}^*
\end{array}
$$

Answer:

```
syntax type = TYPE functype
syntax local = LOCAL valtype
syntax func = FUNC typeidx local* expr
syntax externidx = FUNC funcidx
syntax export = EXPORT name externidx
syntax module = MODULE type* func* export*
```

3.13 Metavariables

Declare metavariables for some basic syntax as below:

```
var i : nat
var x : idx
var l : labelidx
var t : valtype
var ft : functype
var in : instr
var e : expr
```

4 Runtime-Related Syntax

Refer to Appendix A-2 for a full version.
Make a new file `2-runtime.wastup` and write on it.

4.1 *addr*, *funcaddr*

$$\begin{array}{llll} \text{(address)} & \mathit{addr} & ::= & \mathbb{N} \\ \text{(function address)} & \mathit{funcaddr} & ::= & \mathit{addr} \end{array}$$

Answer:

```
syntax addr = nat
syntax funcaddr = addr
```

4.2 *val*

$$\text{(value)} \quad \mathit{val} \quad ::= \quad \mathsf{const}\ \mathit{valtype}\ \mathit{num}_{\mathit{valtype}}$$

Answer:

```
syntax val = CONST valtype num_(valtype)
```

4.3 *externval*

$$\text{(external value)} \quad \mathit{externval} \quad ::= \quad \mathsf{func}\ \mathit{funcaddr}$$

Answer:

```
syntax externval = FUNC funcaddr
```

4.4 $funcinst$, $exportinst$, $moduleinst$

$$
\begin{array}{llcl}
\text{(function instance)} & funcinst & ::= & \{\mathsf{type}\ functype, \\
 & & & \ \mathsf{module}\ moduleinst, \\
 & & & \ \mathsf{code}\ func\} \\
\text{(export instance)} & exportinst & ::= & \{\mathsf{name}\ name, \\
 & & & \ \mathsf{value}\ externval\} \\
\text{(module instance)} & moduleinst & ::= & \{\mathsf{types}\ functype^*, \\
 & & & \ \mathsf{funcs}\ funcaddr^*, \\
 & & & \ \mathsf{exports}\ exportinst^*\}
\end{array}
$$

Use { and } to indicate records.

Answer:

```
syntax funcinst =
  { TYPE functype,
    MODULE moduleinst,
    CODE func }
syntax exportinst =
  { NAME name,
    VALUE externval }
syntax moduleinst =
  { TYPES functype*,
    FUNCS funcaddr*,
    EXPORTS exportinst* }
```

4.5 $store$, $frame$, $state$, $config$

$$
\begin{array}{llcl}
\text{(store)} & store & ::= & \{\mathsf{funcs}\ funcinst^*\} \\
\text{(frame)} & frame & ::= & \{\mathsf{locals}\ val^*, \\
 & & & \ \mathsf{module}\ moduleinst\} \\
\text{(state)} & state & ::= & store; frame \\
\text{(configuration)} & config & ::= & state; admininstr^*
\end{array}
$$

Answer:

```
syntax store = { FUNCS funcinst* }

syntax frame =
  { LOCALS val*,
    MODULE moduleinst }

syntax state = store; frame
syntax config = state; admininstr*
```

4.6 $admininstr$

$$
\begin{array}{llcl}
\text{(administrative instruction)} & admininstr & ::= & instr \\
 & & | & \mathsf{label}_n\ \{instr^*\}\ admininstr^* \\
 & & | & \mathsf{frame}_n\ \{frame\}\ admininstr^* \\
 & & | & \mathsf{trap}
\end{array}
$$

Write like below to indicate ??:

```
‘{<syntax>}
```

to avoid confusion with records.
Answer:

```
syntax admininstr =
  | instr
  | LABEL_ n ‘{instr*} admininstr*
  | FRAME_ n ‘{frame} admininstr*
  | TRAP
```

4.7 funcaddr

$$\text{funcaddr}((s; f)) \quad = \quad f.\mathsf{module.funcs}$$

Hint: The type declaration of function `funcaddr` is as follows:

```
def $funcaddr(state) : funcaddr*
```

Answer:

```
def $funcaddr(state) : funcaddr*
def $funcaddr((s; f)) = f.MODULE.FUNCS
```

4.8 local

$$\text{local}((s; f), x) \quad = \quad f.\mathsf{locals}[x]$$

Use `[` and `]` to indicate an index.
Hint: The type declaration of function `local` is as follows:

```
def $local(state, localidx) : val
```

Answer:

```
def $local(state, localidx) : val
def $local((s; f), x) = f.LOCALS[x]
```

4.9 with$_{local}$

$$\text{with}_{local}((s; f), x, v) \quad = \quad s; f[.\mathsf{locals}[x] = v]$$

Hint: The type declaration of function `with_local` is as follows:

```
def $with_local(state, localidx, val) : state
```

Answer:

```
def $with_local(state, localidx, val) : state
def $with_local((s; f), x, v) = s; f[.LOCALS[x] = v]
```

4.10 funcinst

$$\text{funcinst}((s; f)) \quad = \quad s.\mathsf{funcs}$$

Hint: The type declaration of function `funcinst` is as follows:

```
def $funcinst(state) : funcinst*
```

Answer:

```
def $funcinst(state) : funcinst*
def $funcinst((s; f)) = s.FUNCS
```

4.11 default$_{valtype}$

$$\begin{array}{lcl} \text{default}_{\mathsf{i32}} & = & (\mathsf{const}\ \mathsf{i32}\ 0) \\ \text{default}_{\mathsf{i64}} & = & (\mathsf{const}\ \mathsf{i64}\ 0) \end{array}$$

Hint: The type declaration of function `default_` is as follows:

```
def $default_(valtype) : val
```

Answer:

```
def $default_(valtype) : val
def $default_(I32) = (CONST I32 0)
def $default_(I64) = (CONST I64 0)
```

4.12 Metavariables

Declare metavariables for some runtime-related syntax as below:

```
var a : addr
var fa : funcaddr
var v : val
var xv : externval
var mm : moduleinst
var fi : funcinst
var xi : exportinst
var s : store
var f : frame
var z : state
var ty : type
var loc : local
var ex : export
var xx : externidx
```

5 Validation Rules

We're done with writing syntax of Mini-Wasm. Now, we will declare validation rules of Mini-Wasm.
Refer to Appendix A-3 for a full version.
Make a new file `3-typing.watsup` and write on it.

5.1 *context*

$$(\text{context}) \quad \mathit{context} \; ::= \; \{\mathsf{types}\ \mathit{functype}^*,\ \mathsf{funcs}\ \mathit{functype}^*,\ \mathsf{locals}\ \mathit{valtype}^*,\ \mathsf{labels}\ \mathit{resulttype}^*,\ \mathsf{return}\ \mathit{resulttype}^?\}$$

Answer:

```
syntax context =
{ TYPES functype*, FUNCS functype*,
  LOCALS valtype*, LABELS resulttype*, RETURN resulttype? }
```

Additionally, define its metavariable as below:

```
var C : context
```

5.2 Types

$$\boxed{\vdash \mathit{functype} : \mathsf{ok}}$$

$$\boxed{\vdash \mathit{externtype} : \mathsf{ok}}$$

Name the two relations `Functype_ok`, and `Externtype_ok`.
Use `OK` to indicate the type is valid.
Hint: Refer to Subsection 2.3.
Answer:

```
relation Functype_ok: |- functype : OK
relation Externtype_ok: |- externtype : OK
```

5.2.1 Functype_ok

$$\frac{}{\vdash t_1^* \to t_2^? : \mathsf{ok}} \; [\text{Functype\_ok}]$$

Hint: Refer to Subsection 2.4.
Answer:

```
rule Functype_ok:
  |- t_1* -> t_2? : OK
```

5.2.2 Externtype_ok-func

$$\frac{\vdash \mathit{functype} : \mathsf{ok}}{\vdash \mathsf{func}\ \mathit{functype} : \mathsf{ok}} \; [\text{Externtype\_ok-func}]$$

Premise of [Externtype_ok-func] comes from relation `Functype_ok`.
Answer:

```
rule Externtype_ok/func:
  |- FUNC functype : OK
  -- Functype_ok: |- functype : OK
```

5.3 Subtyping

$$\boxed{\vdash \mathit{functype} \leq \mathit{functype}}$$

$$\boxed{\vdash \mathit{externtype} \leq \mathit{externtype}}$$

Name the two relations `Functype_sub`, and `Externtype_sub`.
Use `<:` to indicate subtype.
Answer:

```
relation Functype_sub: |- functype <: functype
relation Externtype_sub: |- externtype <: externtype
```

5.3.1 Functype_sub

$$\frac{}{\vdash \mathit{ft} \leq \mathit{ft}} \;[\textsc{Functype\_sub}]$$

Answer:

```
rule Functype_sub:
  |- ft <: ft
```

5.3.2 Externtype_sub-func

$$\frac{\vdash \mathit{ft}_1 \leq \mathit{ft}_2}{\vdash \mathsf{func}\ \mathit{ft}_1 \leq \mathsf{func}\ \mathit{ft}_2} \;[\textsc{Externtype\_sub-func}]$$

Premise of [Externtype_sub-func] comes from relation `Functype_sub`.
Answer:

```
rule Externtype_sub/func:
  |- FUNC ft_1 <: FUNC ft_2
  -- Functype_sub: |- ft_1 <: ft_2
```

5.4 Instructions

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

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

$$\boxed{\mathit{context} \vdash \mathit{expr} : \mathit{resulttype}}$$

Name the three relations `Instr_ok`, `Instrs_ok`, and `Expr_ok`.
Answer:

```
relation Instr_ok: context |- instr : functype
relation Instrs_ok: context |- instr* : functype
relation Expr_ok: context |- expr : resulttype
```

5.4.1 Expr_ok

$$\frac{C \vdash instr^* : \epsilon \rightarrow t^?}{C \vdash instr^* : t^?} \; [\textsc{Expr\_ok}]$$

Premise of [Expr_ok] comes from relation `Instrs_ok`.
Answer:

```
rule Expr_ok:
  C |- instr* : t?
  -- Instrs_ok: C |- instr* : eps -> t?
```

5.4.2 Instrs_ok-empty

$$\frac{}{C \vdash \epsilon : \epsilon \rightarrow \epsilon} \; [\textsc{Instrs\_ok-empty}]$$

Answer:

```
rule Instrs_ok/empty:
  C |- eps : eps -> eps
```

5.4.3 Instrs_ok-seq

$$\frac{C \vdash instr_1 : t_1^* \rightarrow t_2^* \qquad C \vdash instr_2 : t_2^* \rightarrow t_3^*}{C \vdash instr_1 \; instr_2^* : t_1^* \rightarrow t_3^*} \; [\textsc{Instrs\_ok-seq}]$$

Premises of [Instrs_ok-seq] come from relation `Instr_ok` and `Instrs_ok`.
Answer:

```
rule Instrs_ok/seq:
  C |- instr_1 instr_2* : t_1* -> t_3*
  -- Instr_ok: C |- instr_1 : t_1* -> t_2*
  -- Instrs_ok: C |- instr_2 : t_2* -> t_3*
```

5.4.4 Instrs_ok-frame

$$\frac{C \vdash instr^* : t_1^* \rightarrow t_2^*}{C \vdash instr^* : t^* \; t_1^* \rightarrow t^* \; t_2^*} \; [\textsc{Instrs\_ok-frame}]$$

Premise of [Instrs_ok-frame] comes from relation `Instrs_ok`.
Answer:

```
rule Instrs_ok/frame:
  C |- instr* : t* t_1* -> t* t_2*
  -- Instrs_ok: C |- instr* : t_1* -> t_2*
```

5.4.5 Instr_ok-nop

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \rightarrow \epsilon} \; [\textsc{Instr\_ok-nop}]$$

Use `eps` to indicate an empty instruction sequence.
Answer:

```
rule Instr_ok/nop:
  C |- NOP : eps -> eps
```

5.4.6 Instr_ok-drop

$$\frac{}{C \vdash \mathsf{drop} : t \to \epsilon} \ [\textsc{Instr\_ok-drop}]$$

Answer:

```
rule Instr_ok/drop:
  C |- DROP : t -> eps
```

5.4.7 Instr_ok-select

$$\frac{}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \to t} \ [\textsc{Instr\_ok-select}]$$

Answer:

```
rule Instr_ok/select:
  C |- SELECT : t t I32 -> t
```

5.4.8 Instr_ok-block

$$\frac{C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}^* : \epsilon \to t^?}{C \vdash \mathsf{block}\ (\epsilon \to t^?)\ \mathit{instr}^* : \epsilon \to t^?} \ [\textsc{Instr\_ok-block}]$$

Premise of [Instr_ok-block] comes from relation `Instrs_ok`.
Answer:

```
rule Instr_ok/block:
  C |- BLOCK (eps -> t?) instr* : eps -> t?
  -- Instrs_ok: C, LABELS (t?) |- instr* : eps -> t?
```

5.4.9 Instr_ok-loop

$$\frac{C, \mathsf{labels}\ (\epsilon) \vdash \mathit{instr}^* : \epsilon \to \epsilon}{C \vdash \mathsf{loop}\ (\epsilon \to t^?)\ \mathit{instr}^* : \epsilon \to t^?} \ [\textsc{Instr\_ok-loop}]$$

Premise of [Instr_ok-loop] comes from relation `Instrs_ok`.
Answer:

```
rule Instr_ok/loop:
  C |- LOOP (eps -> t?) instr* : eps -> t?
  -- Instrs_ok: C, LABELS (eps) |- instr* : eps -> eps
```

5.4.10 Instr_ok-if

$$\frac{C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}_1^* : \epsilon \to t^? \qquad C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}_2^* : \epsilon \to t^?}{C \vdash \mathsf{if}\ (\epsilon \to t^?)\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^* : \mathsf{i32} \to t^?} \ [\textsc{Instr\_ok-if}]$$

Premises of [Instr_ok-block] come from relation `Instrs_ok`.
Answer:

```
rule Instr_ok/if:
  C |- IF (eps -> t?) instr_1* ELSE instr_2* : I32 -> t?
  -- Instrs_ok: C, LABELS (t?) |- instr_1* : eps -> t?
  -- Instrs_ok: C, LABELS (t?) |- instr_2* : eps -> t?
```

5.4.11 Instr_ok-br

$$\frac{C.\mathsf{labels}[l] = t^?}{C \vdash \mathsf{br}\ l : t_1^*\ t^? \to t_2^*}\ [\textsc{Instr\_ok-br}]$$

Write like below to indicate a condition:

```
rule Instr_ok/br:
  <content_of_rule>
  -- if <condition>
```

Answer:

```
rule Instr_ok/br:
  C |- BR l : t_1* t? -> t_2*
  -- if C.LABELS[l] = t?
```

5.4.12 Instr_ok-br_if

$$\frac{C.\mathsf{labels}[l] = t^?}{C \vdash \mathsf{br\_if}\ l : t^?\ \mathsf{i32} \to t^?}\ [\textsc{Instr\_ok-br\_if}]$$

Answer:

```
rule Instr_ok/br_if:
  C |- BR_IF l : t? I32 -> t?
  -- if C.LABELS[l] = t?
```

5.4.13 Instr_ok-call

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

Answer:

```
rule Instr_ok/call:
  C |- CALL x : t_1* -> t_2?
  -- if C.FUNCS[x] = t_1* -> t_2?
```

5.4.14 Instr_ok-return

$$\frac{C.\mathsf{return} = t^?}{C \vdash \mathsf{return} : t_1^*\ t^? \to t_2^*}\ [\textsc{Instr\_ok-return}]$$

Answer:

```
rule Instr_ok/return:
  C |- RETURN : t_1* t? -> t_2*
  -- if C.RETURN = t?
```

5.4.15 Instr_ok-const

$$\frac{}{C \vdash \mathsf{const}\ t\ c_t : \epsilon \to t}\ [\textsc{Instr\_ok-const}]$$

Answer:

```
rule Instr_ok/const:
  C |- CONST t c_t : eps -> t
```

5.4.16 Instr_ok-binop

$$\frac{}{C \vdash \mathsf{binop}\ t\ binop_t : t\ t \to t}\ [\textsc{Instr\_ok-binop}]$$

Answer:

```
rule Instr_ok/binop:
  C |- BINOP t binop_t : t t -> t
```

5.4.17 Instr_ok-local.get

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.get}\ x : \epsilon \to t}\ [\textsc{Instr\_ok-local.get}]$$

Answer:

```
rule Instr_ok/local.get:
  C |- LOCAL.GET x : eps -> t
  -- if C.LOCALS[x] = t
```

5.4.18 Instr_ok-local.set

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.set}\ x : t \to \epsilon}\ [\textsc{Instr\_ok-local.set}]$$

Answer:

```
rule Instr_ok/local.set:
  C |- LOCAL.SET x : t -> eps
  -- if C.LOCALS[x] = t
```

5.5 Constant Expressions

$$\boxed{context \vdash instr\ \mathsf{const}}$$

$$\boxed{context \vdash expr\ \mathsf{const}}$$

$$\boxed{context \vdash expr : valtype^?\ \mathsf{const}}$$

Name the three relations `Instr_const`, `Expr_const`, and `Expr_ok_const`.
Answer:

```
relation Instr_const: context |- instr CONST
relation Expr_const: context |- expr CONST
relation Expr_ok_const: context |- expr : valtype? CONST
```

5.5.1 Instr_const-const

$$\frac{}{C \vdash (\mathsf{const}\ t\ c)\ \mathsf{const}}\ [\textsc{Instr\_const-const}]$$

Answer:

```
rule Instr_const/const:
  C |- (CONST t c) CONST
```

5.5.2 Expr_const

$$\frac{(C \vdash instr\ \mathsf{const})^*}{C \vdash instr^*\ \mathsf{const}}\ [\textsc{Expr\_const}]$$

Premise of [Expr_const] is a sequence of premise, each one of which comes from relation `Instr_const`. To indicate a sequence of premise, group a single premise by `(` and `)`, then use `*` .

Answer:

```
rule Expr_const:
  C |- instr* CONST
  -- (Instr_const: C |- instr CONST)*
```

5.5.3 Expr_ok_const

$$\frac{C \vdash expr : t^? \qquad C \vdash expr\ \mathsf{const}}{C \vdash expr : t^?\ \mathsf{const}}\ [\textsc{Expr\_ok\_const}]$$

Premises of [Expr_ok_const] come from relation `Expr_ok` and `Expr_const`.

Answer:

```
rule Expr_ok_const:
  C |- expr : t? CONST
  -- Expr_ok: C |- expr : t?
  -- Expr_const: C |- expr CONST
```

5.6 Modules

$$\boxed{\vdash type : functype}$$

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

$$\boxed{context \vdash export : externtype}$$

$$\boxed{context \vdash externidx : externtype}$$

$$\boxed{\vdash module : \mathsf{ok}}$$

Name the five relations `Type_ok`, `Func_ok`, `Export_ok`, `Externidx_ok`, and `Module_ok`.

Answer:

```
relation Type_ok: |- type : functype
relation Func_ok: context |- func : functype
relation Export_ok: context |- export : externtype
relation Externidx_ok: context |- externidx : externtype
relation Module_ok: |- module : OK
```

5.6.1 Type_ok

$$\frac{\vdash ft : \mathsf{ok}}{\vdash \mathsf{type}\ ft : ft}\ [\textsc{Type\_ok}]$$

Premise of [Type_ok] comes from relation `Functype_ok`.

Answer:

```
rule Type_ok:
  |- TYPE ft : ft
  -- Functype_ok: |- ft : OK
```

5.6.2 Func_ok

$$\frac{C.\mathsf{types}[x] = t_1^* \to t_2^? \qquad C, \mathsf{locals}\ t_1^*\ t^*, \mathsf{labels}\ (t_2^?), \mathsf{return}\ (t_2^?) \vdash \mathit{expr} : t_2^?}{C \vdash \mathsf{func}\ x\ (\mathsf{local}\ t)^*\ \mathit{expr} : t_1^* \to t_2^?}\ [\text{Func\_ok}]$$

The second premise of [Func_ok] comes from relation `Expr_ok`.
Answer:

```
rule Func_ok:
  C |- FUNC x (LOCAL t)* expr : t_1* -> t_2?
  -- if C.TYPES[x] = t_1* -> t_2?
  -- Expr_ok: C, LOCALS t_1* t*, LABELS (t_2?), RETURN (t_2?) |- expr : t_2?
```

5.6.3 Export_ok

$$\frac{C \vdash \mathit{externidx} : \mathit{xt}}{C \vdash \mathsf{export}\ \mathit{name}\ \mathit{externidx} : \mathit{xt}}\ [\text{Export\_ok}]$$

Premise of [Export_ok] comes from relation `Externidx_ok`.
Answer:

```
rule Export_ok:
  C |- EXPORT name externidx : xt
  -- Externidx_ok: C |- externidx : xt
```

5.6.4 Externidx_ok-func

$$\frac{C.\mathsf{funcs}[x] = \mathit{ft}}{C \vdash \mathsf{func}\ x : \mathsf{func}\ \mathit{ft}}\ [\text{Externidx\_ok-func}]$$

Answer:

```
rule Externidx_ok/func:
  C |- FUNC x : FUNC ft
  -- if C.FUNCS[x] = ft
```

5.6.5 Module_ok

$$\frac{(\vdash \mathit{type} : \mathit{ft}')^* \qquad (C \vdash \mathit{func} : \mathit{ft})^* \qquad (C \vdash \mathit{export} : \mathit{xt})^* \qquad C = \{\mathsf{types}\ \mathit{ft}'^*,\ \mathsf{funcs}\ \mathit{ift}^*\ \mathit{ft}^*\}}{\vdash \mathsf{module}\ \mathit{type}^*\ \mathit{func}^*\ \mathit{export}^* : \mathsf{ok}}\ [\text{Module\_ok}]$$

The first premise of [Module_ok] is a sequence of premise, each one of which comes from relation `Type_ok`. The second premise of [Module_ok] is a sequence of premise, each one of which comes from relation `Func_ok`. The third premise of [Module_ok] is a sequence of premise, each one of which comes from relation `Export_ok`.
Answer:

```
rule Module_ok:
  |- MODULE type* func* export* : OK
  -- (Type_ok: |- type : ft')*
  -- (Func_ok: C |- func : ft)*
  -- (Export_ok: C |- export : xt)*
  -- if C = {TYPES ft'*, FUNCS ift* ft*}
```

6 Reduction Rules

Now, let's write the reduction rules for the instructions. Refer to Appendix A-4 for a full version. Make a new file `4-reduction.wastup` and write on it.

6.1 Relations

$$\boxed{\mathit{config} \hookrightarrow \mathit{config}}$$

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

$$\boxed{\mathit{config} \hookrightarrow \mathit{admininstr}^*}$$

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

$$\begin{array}{llcll}
[\textsc{Step-pure}] & z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \\
[\textsc{Step-read}] & z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } z; \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \\
[\textsc{Steps-refl}] & z; \mathit{admininstr}^* & \hookrightarrow^* & z; \mathit{admininstr}^* & \\
[\textsc{Steps-trans}] & z; \mathit{admininstr}^* & \hookrightarrow^* & z''; \mathit{admininstr}''^* & \text{if } z; \mathit{admininstr}^* \hookrightarrow z'; \mathit{admininstr}'^* \\
 & & & & \wedge\ z'; \mathit{admininstr}' \hookrightarrow^* z''; \mathit{admininstr}''^*
\end{array}$$

Name the four relations `Step`, `Step_pure`, `Step_read`, and `Steps`.
Premise of [Step-pure] comes from relation `Step_pure`.
Premise of [Step-read] comes from relation `Step_read`.
Premise of [Step-trans] comes from relation `Step` and `Steps`.
Use ~> to indicate a reduction.
Answer:

```
relation Step: config ~> config
relation Step_pure: admininstr* ~> admininstr*
relation Step_read: config ~> admininstr*
relation Steps: config ~>* config

rule Step/pure:
  z; instr*  ~>  z; instr'*
  -- Step_pure: instr* ~> instr'*

rule Step/read:
  z; instr*  ~>  z; instr'*
  -- Step_read: z; instr* ~> instr'*

rule Steps/refl:
  z; admininstr* ~>* z; admininstr*

rule Steps/trans:
  z; admininstr*  ~>*  z''; admininstr''*
  -- Step: z; admininstr*  ~>  z'; admininstr'*
  -- Steps: z'; admininstr'  ~>*  z''; admininstr''*
```

6.2 Step_pure-nop

$$[\textsc{Step\_pure-nop}] \quad \mathsf{nop} \quad \hookrightarrow \quad \epsilon$$

Answer:

```
rule Step_pure/nop:
  NOP  ~>  eps
```

6.3 Step_pure-drop

$$[\textsc{Step\_pure-drop}] \quad \mathit{val}\ \mathsf{drop} \quad \hookrightarrow \quad \epsilon$$

as a variable.

Answer:

```
rule Step_pure/drop:
  val DROP  ~>  eps
```

6.4 Step_pure-select

$$\begin{array}{llcll} [\textsc{Step\_pure-select-true}] & \mathit{val}_1\ \mathit{val}_2\ (\mathsf{const}\ \mathsf{i32}\ c)\ \mathsf{select} & \hookrightarrow & \mathit{val}_1 & \text{if } c \neq 0 \\ [\textsc{Step\_pure-select-false}] & \mathit{val}_1\ \mathit{val}_2\ (\mathsf{const}\ \mathsf{i32}\ c)\ \mathsf{select} & \hookrightarrow & \mathit{val}_2 & \text{if } c = 0 \end{array}$$

When we get `SELECT` from stack, we have two cases: condition is true or false.
Define each of the case as seperate rule as follows:

```
rule Step_pure/select-true:
  ...

rule Step_pure/select-false:
  ...
```

Write like `val_1`, `val_2` (and so on) to distinguish multiple `val`s.
Use `=/=` and `=` for integer comparison.

Answer:

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

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

As you can see here, you can distinguish multiple metavariables with same type by adding _1, _2 (and so on) after the metavariable name. Also, the names of two rules should be the same before hyphen (-), so that they can be treated as the same reduction rule.

6.5 Step_read-block

$$[\textsc{Step\_read-block}] \quad z; (\mathsf{block}\ (\epsilon \to t^?)\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_n\ \{\epsilon\}\ instr^*) \qquad \text{if } t^? = \epsilon \land n = 0 \lor t^? \neq \epsilon \land n = 1$$

Use `/\` and `\/` for logical and/or.

Answer:

```
rule Step_pure/block:
  (BLOCK t? instr*)  ~>  (LABEL_ n ‘{eps} instr*)
  -- if t? = eps /\ n = 0 \/ t? =/= eps /\ n = 1
```

6.6 Step_read-loop

$$[\textsc{Step\_read-loop}] \quad z; (\mathsf{loop}\ \mathit{ft}\ instr^*) \quad \hookrightarrow \quad (\mathsf{label}_0\ \{\mathsf{loop}\ \mathit{ft}\ instr^*\}\ instr^*)$$

Answer:

```
rule Step_pure/loop:
  (LOOP t? instr*)  ~>  (LABEL_ 0 ‘{LOOP t? instr*} instr*)
```

6.7 Step_pure-if

$$\begin{array}{llcll} [\textsc{Step\_pure-if-true}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{if}\ \mathit{ft}\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ \mathit{ft}\ instr_1^*) & \text{if } c \neq 0 \\ [\textsc{Step\_pure-if-false}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{if}\ \mathit{ft}\ instr_1^*\ \mathsf{else}\ instr_2^*) & \hookrightarrow & (\mathsf{block}\ \mathit{ft}\ instr_2^*) & \text{if } c = 0 \end{array}$$

Answer:

```
rule Step_pure/if-true:
  (CONST I32 c) (IF t? instr_1* ELSE instr_2*)  ~>  (BLOCK t? instr_1*)
  -- if c =/= 0

rule Step_pure/if-false:
  (CONST I32 c) (IF t? instr_1* ELSE instr_2*)  ~>  (BLOCK t? instr_2*)
  -- if c = 0
```

6.8 Step_pure-br

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

For arithmetic expressions, you should write like `$(l+1)`, instead of `l+1`.
Use `^` for a sequence with given length. e.g. `instr^n`.

Answer:

```
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)
```

6.9 Step_pure-br_if

$$
\begin{array}{llcll}
[\textsc{Step\_pure-br\_if-true}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } c \neq 0 \\
[\textsc{Step\_pure-br\_if-false}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & \text{if } c = 0
\end{array}
$$

Answer:

```
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)  ~>  eps
  -- if c = 0
```

6.10 Step_read-call

$$
\begin{array}{llcll}
[\textsc{Step\_read-call}] & z; val^k\ (\mathsf{call}\ x) & \hookrightarrow & (\mathsf{frame}_n\ \{f\}\ (\mathsf{label}_n\ \{\epsilon\}\ instr^*)) & \text{if } a = \mathrm{funcaddr}(z)[x] \\
 & & & & \wedge\ \mathrm{funcinst}(z)[a] = \{\mathsf{type}\ (t_1^k \to t_2^n),\ \mathsf{module}\ mm,\ \mathsf{co} \\
 & & & & \wedge\ func = \mathsf{func}\ x\ (\mathsf{local}\ t)^*\ instr^* \\
 & & & & \wedge\ f = \{\mathsf{locals}\ val^k\ (\mathrm{default}_t)^*,\ \mathsf{module}\ mm\}
\end{array}
$$

Answer:

```
rule Step_read/call:
  z; val^k (CALL x)  ~>  (FRAME_ n `{f} (LABEL_ n `{eps} instr*))
  -- if a = $funcaddr(z)[x]
  -- if $funcinst(z)[a] = {TYPE (t_1^k -> t_2^n), MODULE mm, CODE func}
  -- if func = FUNC x (LOCAL t)* instr*
  -- if f = {LOCALS val^k ($default_(t))*, MODULE mm}
```

6.11 Step_pure-frame-vals

$$
[\textsc{Step\_pure-frame-vals}] \quad (\mathsf{frame}_n\ \{f\}\ val^n) \quad \hookrightarrow \quad val^n
$$

Answer:

```
rule Step_pure/frame-vals:
  (FRAME_ n `{f} val^n)  ~>  val^n
```

6.12 Step_pure-return

$$
\begin{array}{lrcl}
[\textsc{Step\_pure-return-frame}] & (\mathsf{frame}_n\ \{f\}\ val'^*\ val^n\ \mathsf{return}\ instr^*) & \hookrightarrow & val^n \\
[\textsc{Step\_pure-return-label}] & (\mathsf{label}_n\ \{instr'^*\}\ val^*\ \mathsf{return}\ instr^*) & \hookrightarrow & val^*\ \mathsf{return}
\end{array}
$$

Answer:

```
rule Step_pure/return-frame:
  (FRAME_ n `{f} val'* val^n RETURN instr*)  ~>  val^n

rule Step_pure/return-label:
  (LABEL_ n `{instr'*} val* RETURN instr*)  ~>  val* RETURN
```

6.13 Step_pure-trap

$$
\begin{array}{llcll}
[\textsc{Step\_pure-trap-vals}] & \mathit{val}^*~\mathsf{trap}~\mathit{instr}^* & \hookrightarrow & \mathsf{trap} & \text{if } \mathit{val}^* \neq \epsilon \vee \mathit{instr}^* \neq \epsilon \\
[\textsc{Step\_pure-trap-label}] & (\mathsf{label}_n~\{\mathit{instr'}^*\}~\mathsf{trap}) & \hookrightarrow & \mathsf{trap} & \\
[\textsc{Step\_pure-trap-frame}] & (\mathsf{frame}_n~\{f\}~\mathsf{trap}) & \hookrightarrow & \mathsf{trap} &
\end{array}
$$

Answer:

```
rule Step_pure/trap-vals:
  val* TRAP instr*  ~>  TRAP
  -- if val* =/= eps \/ instr* =/= eps

rule Step_pure/trap-label:
  (LABEL_ n `{instr'*} TRAP)  ~>  TRAP

rule Step_pure/trap-frame:
  (FRAME_ n `{f} TRAP)  ~>  TRAP
```

6.14 Step-ctxt

$$
\begin{array}{llcll}
[\textsc{Step-ctxt-label}] & z; (\mathsf{label}_n~\{\mathit{instr}_0^*\}~\mathit{instr}^*) & \hookrightarrow & z'; (\mathsf{label}_n~\{\mathit{instr}_0^*\}~\mathit{instr'}^*) & \text{if } z; \mathit{instr}^* \hookrightarrow z'; \mathit{instr'}^* \\
[\textsc{Step-ctxt-frame}] & s; f; (\mathsf{frame}_n~\{f'\}~\mathit{instr}^*) & \hookrightarrow & s'; f; (\mathsf{frame}_n~\{f'\}~\mathit{instr'}^*) & \text{if } s; f'; \mathit{instr}^* \hookrightarrow s'; f'; \mathit{instr'}^*
\end{array}
$$

Premise of [Step-ctxt-label] comes from relation `Step`.
Premise of [Step-ctxt-frame] comes from relation `Step`.
Answer:

```
rule Step/ctxt-label:
  z; (LABEL_ n `{instr_0*} instr*)  ~>  z'; (LABEL_ n `{instr_0*} instr'*)
  -- Step: z; instr* ~> z'; instr'*

rule Step/ctxt-frame:
  s; f; (FRAME_ n `{f'} instr*)  ~>  s'; f; (FRAME_ n `{f'} instr'*)
  -- Step: s; f'; instr* ~> s'; f'; instr'*
```

6.15 Step_pure-binop

6.15.1 iadd, isub, imul, idiv

$$
\begin{array}{lcl}
\mathrm{iadd}(N, c_1, c_2) & = & (c_1 + c_2) \bmod 2^N \\
\mathrm{isub}(N, c_1, c_2) & = & (c_1 - c_2 + 2^N) \bmod 2^N \\
\mathrm{imul}(N, c_1, c_2) & = & (c_1 \cdot c_2) \bmod 2^N \\
\mathrm{idiv}(N, c_1, 0) & = & \epsilon \\
\mathrm{idiv}(N, c_1, c_2) & = & c_1 / c_2
\end{array}
$$

Use \ to indicate modular calculation.
Hint: The type declaration of functions `iadd`, `isub`, `imul`, and `idiv` is as follows:

```
def $iadd(N, iN(N), iN(N)) : iN(N)
def $isub(N, iN(N), iN(N)) : iN(N)
def $imul(N, iN(N), iN(N)) : iN(N)
def $idiv(N, iN(N), iN(N)) : iN(N)*
```

Answer:

```
def $iadd(N, iN(N), iN(N)) : iN(N)
def $isub(N, iN(N), iN(N)) : iN(N)
def $imul(N, iN(N), iN(N)) : iN(N)
def $idiv(N, iN(N), iN(N)) : iN(N)*
def $iadd(N, c_1, c_2) = $((c_1 + c_2) \ 2^N)
def $isub(N, c_1, c_2) = $((c_1 - c_2 + 2^N) \ 2^N)
def $imul(N, c_1, c_2) = $((c_1 * c_2) \ 2^N)
def $idiv(N, c_1, 0) = eps
def $idiv(N, c_1, c_2) = $(c_1 / c_2)
```

6.15.2 binop

$$
\begin{array}{rcl}
\mathrm{binop}(t, \mathsf{add}, c_1, c_2) & = & \mathrm{iadd}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{sub}, c_1, c_2) & = & \mathrm{isub}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{mul}, c_1, c_2) & = & \mathrm{imul}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{div}, c_1, c_2) & = & \mathrm{idiv}(\mathrm{size}(t), c_1, c_2)
\end{array}
$$

Hint: The type declaration of function `binop` is as follows:

```
def $binop(valtype, binop, num_(valtype), num_(valtype)) : num_(valtype)*
```

Answer:

```
def $binop(valtype, binop, num_(valtype), num_(valtype)) : num_(valtype)*

def $binop(t, ADD, c_1, c_2) = $iadd($size(t), c_1, c_2)
def $binop(t, SUB, c_1, c_2) = $isub($size(t), c_1, c_2)
def $binop(t, MUL, c_1, c_2) = $imul($size(t), c_1, c_2)
def $binop(t, DIV, c_1, c_2) = $idiv($size(t), c_1, c_2)
```

6.15.3 Step_pure-binop

$$
\begin{array}{llcll}
[\textsc{Step\_pure-binop-val}] & (\mathsf{const}\ t\ c_1)\ (\mathsf{const}\ t\ c_2)\ (\mathsf{binop}\ t\ \mathit{binop}) & \hookrightarrow & (\mathsf{const}\ t\ c) & \text{if } \mathrm{binop}(t, \mathit{binop}, c_1, c_2) = c \\
[\textsc{Step\_pure-binop-trap}] & (\mathsf{const}\ t\ c_1)\ (\mathsf{const}\ t\ c_2)\ (\mathsf{binop}\ t\ \mathit{binop}) & \hookrightarrow & \mathsf{trap} & \text{if } \mathrm{binop}(t, \mathit{binop}, c_1, c_2) = \epsilon
\end{array}
$$

Answer:

```
rule Step_pure/binop-val:
  (CONST t c_1) (CONST t c_2) (BINOP t binop)  ~>  (CONST t c)
  -- if $binop(t, binop, c_1, c_2) = c

rule Step_pure/binop-trap:
  (CONST t c_1) (CONST t c_2) (BINOP t binop)  ~>  TRAP
  -- if $binop(t, binop, c_1, c_2) = eps
```

6.16 Step_read-local.get

$$
[\textsc{Step\_read-local.get}] \quad z; (\mathsf{local.get}\ x) \quad \hookrightarrow \quad \mathrm{local}(z, x)
$$

Answer:

```
rule Step_read/local.get:
  z; (LOCAL.GET x)  ~>  $local(z, x)
```

6.17 Step-local.set

$$[\textsc{Step-local.set}] \quad z; val\ (\mathsf{local.set}\ x) \quad \hookrightarrow \quad \mathrm{with}_{local}(z, x, val); \epsilon$$

Answer:

```
rule Step/local.set:
  z; val (LOCAL.SET x)  ~>  $with_local(z, x, val); eps
```

A Appendix

A.1 Basic Syntax

$$\begin{array}{rcl} N & ::= & \mathbb{N} \\ n & ::= & \mathbb{N} \end{array}$$

$$\text{(integer)} \quad iN(N) \quad ::= \quad 0 \mid \ldots \mid 2^N - 1$$

$$\begin{array}{lrcl} \text{(character)} & char & ::= & \text{U+00} \mid \ldots \mid \text{U+D7FF} \mid \text{U+E000} \mid \ldots \mid \text{U+10FFFF} \\ \text{(name)} & name & ::= & char^* \end{array}$$

$$\begin{array}{lrcl} \text{(index)} & idx & ::= & iN(32) \\ \text{(type index)} & typeidx & ::= & idx \\ \text{(function index)} & funcidx & ::= & idx \\ \text{(label index)} & labelidx & ::= & idx \\ \text{(local index)} & localidx & ::= & idx \end{array}$$

$$\text{(number type)} \quad valtype \quad ::= \quad \mathsf{i32} \mid \mathsf{i64}$$

$$\text{(function type)} \quad functype \quad ::= \quad valtype^* \rightarrow valtype^*$$

$$\text{(external type)} \quad externtype \quad ::= \quad \mathsf{func}\ functype$$

$$\begin{array}{rcl} \mathrm{size}(\mathsf{i32}) & = & 32 \\ \mathrm{size}(\mathsf{i64}) & = & 64 \end{array}$$

$$num_{valtype} \quad ::= \quad iN(\mathrm{size}(valtype))$$

$$binop \quad ::= \quad \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div}$$

$$\begin{array}{lrcl} \text{(instruction)} & instr & ::= & \mathsf{nop} \\ & & \mid & \mathsf{drop} \\ & & \mid & \mathsf{select} \\ & & \mid & \mathsf{block}\ functype\ instr^* \\ & & \mid & \mathsf{loop}\ functype\ instr^* \\ & & \mid & \mathsf{if}\ functype\ instr^*\ \mathsf{else}\ instr^* \\ & & \mid & \mathsf{br}\ labelidx \\ & & \mid & \mathsf{br\_if}\ labelidx \\ & & \mid & \mathsf{call}\ funcidx \\ & & \mid & \mathsf{return} \\ & & \mid & \mathsf{const}\ valtype\ num_{valtype} \\ & & \mid & \mathsf{binop}\ valtype\ binop \\ & & \mid & \mathsf{local.get}\ localidx \\ & & \mid & \mathsf{local.set}\ localidx \end{array}$$

$$\text{(expression)} \quad expr \quad ::= \quad instr^*$$

$$
\begin{array}{lrcl}
\text{(type)} & \mathit{type} & ::= & \mathsf{type}\ \mathit{functype} \\
\text{(local)} & \mathit{local} & ::= & \mathsf{local}\ \mathit{valtype} \\
\text{(function)} & \mathit{func} & ::= & \mathsf{func}\ \mathit{typeidx}\ \mathit{local}^*\ \mathit{expr} \\
\text{(external index)} & \mathit{externidx} & ::= & \mathsf{func}\ \mathit{funcidx} \\
\text{(export)} & \mathit{export} & ::= & \mathsf{export}\ \mathit{name}\ \mathit{externidx} \\
\text{(module)} & \mathit{module} & ::= & \mathsf{module}\ \mathit{type}^*\ \mathit{func}^*\ \mathit{export}^*
\end{array}
$$

A.2 Runtime-Related Syntax

$$
\begin{array}{lrcl}
\text{(address)} & \mathit{addr} & ::= & \mathbb{N} \\
\text{(function address)} & \mathit{funcaddr} & ::= & \mathit{addr}
\end{array}
$$

$$
\text{(value)}\quad \mathit{val} \quad ::= \quad \mathsf{const}\ \mathit{valtype}\ \mathit{num}_{\mathit{valtype}}
$$

$$
\text{(external value)}\quad \mathit{externval} \quad ::= \quad \mathsf{func}\ \mathit{funcaddr}
$$

$$
\begin{array}{lrcl}
\text{(function instance)} & \mathit{funcinst} & ::= & \{\mathsf{type}\ \mathit{functype}, \\
 & & & \ \mathsf{module}\ \mathit{moduleinst}, \\
 & & & \ \mathsf{code}\ \mathit{func}\} \\
\text{(export instance)} & \mathit{exportinst} & ::= & \{\mathsf{name}\ \mathit{name}, \\
 & & & \ \mathsf{value}\ \mathit{externval}\} \\
\text{(module instance)} & \mathit{moduleinst} & ::= & \{\mathsf{types}\ \mathit{functype}^*, \\
 & & & \ \mathsf{funcs}\ \mathit{funcaddr}^*, \\
 & & & \ \mathsf{exports}\ \mathit{exportinst}^*\}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(store)} & \mathit{store} & ::= & \{\mathsf{funcs}\ \mathit{funcinst}^*\} \\
\text{(frame)} & \mathit{frame} & ::= & \{\mathsf{locals}\ \mathit{val}^*, \\
 & & & \ \mathsf{module}\ \mathit{moduleinst}\} \\
\text{(state)} & \mathit{state} & ::= & \mathit{store}; \mathit{frame} \\
\text{(configuration)} & \mathit{config} & ::= & \mathit{state}; \mathit{admininstr}^*
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(administrative instruction)} & \mathit{admininstr} & ::= & \mathit{instr} \\
 & & | & \mathsf{label}_n\ \{\mathit{instr}^*\}\ \mathit{admininstr}^* \\
 & & | & \mathsf{frame}_n\ \{\mathit{frame}\}\ \mathit{admininstr}^* \\
 & & | & \mathsf{trap}
\end{array}
$$

$$
\mathrm{funcaddr}((s; f)) \quad = \quad f.\mathsf{module}.\mathsf{funcs}
$$

$$
\mathrm{local}((s; f), x) \quad = \quad f.\mathsf{locals}[x]
$$

$$
\mathrm{with}_{local}((s; f), x, v) \quad = \quad s; f[.\mathsf{locals}[x] = v]
$$

$$
\mathrm{funcinst}((s; f)) \quad = \quad s.\mathsf{funcs}
$$

$$
\begin{array}{lcl}
\mathrm{default}_{\mathsf{i32}} & = & (\mathsf{const}\ \mathsf{i32}\ 0) \\
\mathrm{default}_{\mathsf{i64}} & = & (\mathsf{const}\ \mathsf{i64}\ 0)
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(context)} & \mathit{context} & ::= & \{\mathsf{types}\ \mathit{functype}^*,\ \mathsf{funcs}\ \mathit{functype}^*, \\
 & & & \ \mathsf{locals}\ \mathit{valtype}^*,\ \mathsf{labels}\ \mathit{resulttype}^*,\ \mathsf{return}\ \mathit{resulttype}^?\}
\end{array}
$$

A.3 Context, Validation Rules

$$\boxed{\vdash \mathit{functype} : \mathsf{ok}}$$

$$\boxed{\vdash \mathit{externtype} : \mathsf{ok}}$$

$$\frac{}{\vdash t_1^* \rightarrow t_2^? : \mathsf{ok}} \ [\textsc{Functype\_ok}]$$

$$\frac{\vdash \mathit{functype} : \mathsf{ok}}{\vdash \mathsf{func}\ \mathit{functype} : \mathsf{ok}} \ [\textsc{Externtype\_ok-func}]$$

$$\boxed{\vdash \mathit{functype} \leq \mathit{functype}}$$

$$\boxed{\vdash \mathit{externtype} \leq \mathit{externtype}}$$

$$\frac{}{\vdash \mathit{ft} \leq \mathit{ft}} \ [\textsc{Functype\_sub}]$$

$$\frac{\vdash \mathit{ft}_1 \leq \mathit{ft}_2}{\vdash \mathsf{func}\ \mathit{ft}_1 \leq \mathsf{func}\ \mathit{ft}_2} \ [\textsc{Externtype\_sub-func}]$$

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

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

$$\boxed{\mathit{context} \vdash \mathit{expr} : \mathit{resulttype}}$$

$$\frac{C \vdash \mathit{instr}^* : \epsilon \rightarrow t^?}{C \vdash \mathit{instr}^* : t^?} \ [\textsc{Expr\_ok}]$$

$$\frac{}{C \vdash \epsilon : \epsilon \rightarrow \epsilon} \ [\textsc{Instrs\_ok-empty}]$$

$$\frac{C \vdash \mathit{instr}_1 : t_1^* \rightarrow t_2^* \qquad C \vdash \mathit{instr}_2 : t_2^* \rightarrow t_3^*}{C \vdash \mathit{instr}_1\ \mathit{instr}_2^* : t_1^* \rightarrow t_3^*} \ [\textsc{Instrs\_ok-seq}]$$

$$\frac{C \vdash \mathit{instr}^* : t_1^* \rightarrow t_2^*}{C \vdash \mathit{instr}^* : t^*\ t_1^* \rightarrow t^*\ t_2^*} \ [\textsc{Instrs\_ok-frame}]$$

$$\frac{}{C \vdash \mathsf{nop} : \epsilon \rightarrow \epsilon} \ [\textsc{Instr\_ok-nop}]$$

$$\frac{}{C \vdash \mathsf{drop} : t \rightarrow \epsilon} \ [\textsc{Instr\_ok-drop}]$$

$$\frac{}{C \vdash \mathsf{select} : t\ t\ \mathsf{i32} \rightarrow t} \ [\textsc{Instr\_ok-select}]$$

$$\frac{C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}^* : \epsilon \rightarrow t^?}{C \vdash \mathsf{block}\ (\epsilon \rightarrow t^?)\ \mathit{instr}^* : \epsilon \rightarrow t^?} \ [\textsc{Instr\_ok-block}]$$

$$\frac{C, \mathsf{labels}\ (\epsilon) \vdash \mathit{instr}^* : \epsilon \rightarrow \epsilon}{C \vdash \mathsf{loop}\ (\epsilon \rightarrow t^?)\ \mathit{instr}^* : \epsilon \rightarrow t^?} \ [\textsc{Instr\_ok-loop}]$$

$$\frac{C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}_1^* : \epsilon \rightarrow t^? \qquad C, \mathsf{labels}\ (t^?) \vdash \mathit{instr}_2^* : \epsilon \rightarrow t^?}{C \vdash \mathsf{if}\ (\epsilon \rightarrow t^?)\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^* : \mathsf{i32} \rightarrow t^?} \ [\textsc{Instr\_ok-if}]$$

$$\frac{C.\mathsf{labels}[l] = t^?}{C \vdash \mathsf{br}\ l : t_1^*\ t^? \rightarrow t_2^*} \ [\textsc{Instr\_ok-br}]$$

$$\frac{C.\mathsf{labels}[l] = t^?}{C \vdash \mathsf{br\_if}\ l : t^?\ \mathsf{i32} \rightarrow t^?} \ [\textsc{Instr\_ok-br\_if}]$$

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

$$\frac{C.\mathsf{return} = t^?}{C \vdash \mathsf{return} : t_1^*\ t^? \to t_2^*}\ [\textsc{Instr\_ok-return}]$$

$$\frac{}{C \vdash \mathsf{const}\ t\ c_t : \epsilon \to t}\ [\textsc{Instr\_ok-const}]$$

$$\frac{}{C \vdash \mathsf{binop}\ t\ binop_t : t\ t \to t}\ [\textsc{Instr\_ok-binop}]$$

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.get}\ x : \epsilon \to t}\ [\textsc{Instr\_ok-local.get}]$$

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.set}\ x : t \to \epsilon}\ [\textsc{Instr\_ok-local.set}]$$

$$\boxed{context \vdash instr\ \mathsf{const}}$$

$$\boxed{context \vdash expr\ \mathsf{const}}$$

$$\boxed{context \vdash expr : valtype^?\ \mathsf{const}}$$

$$\frac{}{C \vdash (\mathsf{const}\ t\ c)\ \mathsf{const}}\ [\textsc{Instr\_const-const}]$$

$$\frac{(C \vdash instr\ \mathsf{const})^*}{C \vdash instr^*\ \mathsf{const}}\ [\textsc{Expr\_const}]$$

$$\frac{C \vdash expr : t^? \qquad C \vdash expr\ \mathsf{const}}{C \vdash expr : t^?\ \mathsf{const}}\ [\textsc{Expr\_ok\_const}]$$

$$\boxed{\vdash type : functype}$$

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

$$\boxed{context \vdash export : externtype}$$

$$\boxed{context \vdash externidx : externtype}$$

$$\boxed{\vdash module : \mathsf{ok}}$$

$$\frac{\vdash ft : \mathsf{ok}}{\vdash \mathsf{type}\ ft : ft}\ [\textsc{Type\_ok}]$$

$$\frac{C.\mathsf{types}[x] = t_1^* \to t_2^? \qquad C, \mathsf{locals}\ t_1^*\ t^*, \mathsf{labels}\ (t_2^?), \mathsf{return}\ (t_2^?) \vdash expr : t_2^?}{C \vdash \mathsf{func}\ x\ (\mathsf{local}\ t)^*\ expr : t_1^* \to t_2^?}\ [\textsc{Func\_ok}]$$

$$\frac{C \vdash externidx : xt}{C \vdash \mathsf{export}\ name\ externidx : xt}\ [\textsc{Export\_ok}]$$

$$\frac{C.\mathsf{funcs}[x] = ft}{C \vdash \mathsf{func}\ x : \mathsf{func}\ ft}\ [\textsc{Externidx\_ok-func}]$$

$$\frac{(\vdash type : ft')^* \qquad (C \vdash func : ft)^* \qquad (C \vdash export : xt)^* \qquad C = \{\mathsf{types}\ ft'^*,\ \mathsf{funcs}\ ift^*\ ft^*\}}{\vdash \mathsf{module}\ type^*\ func^*\ export^* : \mathsf{ok}}\ [\textsc{Module\_ok}]$$

A.4 Reduction Rules

$$\boxed{\mathit{config} \hookrightarrow \mathit{config}}$$

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

$$\boxed{\mathit{config} \hookrightarrow \mathit{admininstr}^*}$$

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

$$
\begin{array}{llcll}
[\textsc{Step-pure}] & z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \\
[\textsc{Step-read}] & z; \mathit{instr}^* & \hookrightarrow & z; \mathit{instr}'^* & \text{if } z; \mathit{instr}^* \hookrightarrow \mathit{instr}'^* \\
[\textsc{Steps-refl}] & z; \mathit{admininstr}^* & \hookrightarrow^* & z; \mathit{admininstr}^* & \\
[\textsc{Steps-trans}] & z; \mathit{admininstr}^* & \hookrightarrow^* & z''; \mathit{admininstr}''^* & \text{if } z; \mathit{admininstr}^* \hookrightarrow z'; \mathit{admininstr}'^* \\
 & & & & \wedge\ z'; \mathit{admininstr}' \hookrightarrow^* z''; \mathit{admininstr}''^*
\end{array}
$$

$$[\textsc{Step\_pure-nop}] \quad \mathsf{nop} \quad \hookrightarrow \quad \epsilon$$

$$[\textsc{Step\_pure-drop}] \quad \mathit{val}\ \mathsf{drop} \quad \hookrightarrow \quad \epsilon$$

$$
\begin{array}{llcll}
[\textsc{Step\_pure-select-true}] & \mathit{val}_1\ \mathit{val}_2\ (\mathsf{const}\ \mathsf{i32}\ c)\ \mathsf{select} & \hookrightarrow & \mathit{val}_1 & \text{if } c \neq 0 \\
[\textsc{Step\_pure-select-false}] & \mathit{val}_1\ \mathit{val}_2\ (\mathsf{const}\ \mathsf{i32}\ c)\ \mathsf{select} & \hookrightarrow & \mathit{val}_2 & \text{if } c = 0
\end{array}
$$

$$[\textsc{Step\_read-block}] \quad z; (\mathsf{block}\ (\epsilon \rightarrow t^?)\ \mathit{instr}^*) \quad \hookrightarrow \quad (\mathsf{label}_n\,\{\epsilon\}\ \mathit{instr}^*) \qquad \text{if } t^? = \epsilon \wedge n = 0 \vee t^? \neq \epsilon \wedge n = 1$$

$$[\textsc{Step\_read-loop}] \quad z; (\mathsf{loop}\ \mathit{ft}\ \mathit{instr}^*) \quad \hookrightarrow \quad (\mathsf{label}_0\,\{\mathsf{loop}\ \mathit{ft}\ \mathit{instr}^*\}\ \mathit{instr}^*)$$

$$
\begin{array}{llcll}
[\textsc{Step\_pure-if-true}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{if}\ \mathit{ft}\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*) & \hookrightarrow & (\mathsf{block}\ \mathit{ft}\ \mathit{instr}_1^*) & \text{if } c \neq 0 \\
[\textsc{Step\_pure-if-false}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{if}\ \mathit{ft}\ \mathit{instr}_1^*\ \mathsf{else}\ \mathit{instr}_2^*) & \hookrightarrow & (\mathsf{block}\ \mathit{ft}\ \mathit{instr}_2^*) & \text{if } c = 0
\end{array}
$$

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

$$
\begin{array}{llcll}
[\textsc{Step\_pure-br\_if-true}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & (\mathsf{br}\ l) & \text{if } c \neq 0 \\
[\textsc{Step\_pure-br\_if-false}] & (\mathsf{const}\ \mathsf{i32}\ c)\ (\mathsf{br\_if}\ l) & \hookrightarrow & \epsilon & \text{if } c = 0
\end{array}
$$

$$
\begin{array}{llcll}
[\textsc{Step\_read-call}] & z; \mathit{val}^k\ (\mathsf{call}\ x) & \hookrightarrow & (\mathsf{frame}_n\,\{f\}\ (\mathsf{label}_n\,\{\epsilon\}\ \mathit{instr}^*)) & \text{if } a = \mathrm{funcaddr}(z)[x] \\
 & & & & \wedge\ \mathrm{funcinst}(z)[a] = \{\mathsf{type}\ (t_1^k \rightarrow t_2^n),\ \mathsf{module}\ \mathit{mm},\ \mathsf{co} \\
 & & & & \wedge\ \mathit{func} = \mathsf{func}\ x\ (\mathsf{local}\ t)^*\ \mathit{instr}^* \\
 & & & & \wedge\ f = \{\mathsf{locals}\ \mathit{val}^k\ (\mathrm{default}_t)^*,\ \mathsf{module}\ \mathit{mm}\}
\end{array}
$$

$$[\textsc{Step\_pure-frame-vals}] \quad (\mathsf{frame}_n\,\{f\}\ \mathit{val}^n) \quad \hookrightarrow \quad \mathit{val}^n$$

$$
\begin{array}{llcl}
[\textsc{Step\_pure-return-frame}] & (\mathsf{frame}_n\,\{f\}\ \mathit{val}'^*\ \mathit{val}^n\ \mathsf{return}\ \mathit{instr}^*) & \hookrightarrow & \mathit{val}^n \\
[\textsc{Step\_pure-return-label}] & (\mathsf{label}_n\,\{\mathit{instr}'^*\}\ \mathit{val}^*\ \mathsf{return}\ \mathit{instr}^*) & \hookrightarrow & \mathit{val}^*\ \mathsf{return}
\end{array}
$$

$$
\begin{array}{llcll}
[\textsc{Step\_pure-trap-vals}] & \mathit{val}^*\ \mathsf{trap}\ \mathit{instr}^* & \hookrightarrow & \mathsf{trap} & \text{if } \mathit{val}^* \neq \epsilon \vee \mathit{instr}^* \neq \epsilon \\
[\textsc{Step\_pure-trap-label}] & (\mathsf{label}_n\,\{\mathit{instr}'^*\}\ \mathsf{trap}) & \hookrightarrow & \mathsf{trap} & \\
[\textsc{Step\_pure-trap-frame}] & (\mathsf{frame}_n\,\{f\}\ \mathsf{trap}) & \hookrightarrow & \mathsf{trap} &
\end{array}
$$

$$
\begin{array}{llcll}
[\textsc{Step-ctxt-label}] & z; (\mathsf{label}_n\,\{\mathit{instr}_0^*\}\ \mathit{instr}^*) & \hookrightarrow & z'; (\mathsf{label}_n\,\{\mathit{instr}_0^*\}\ \mathit{instr}'^*) & \text{if } z; \mathit{instr}^* \hookrightarrow z'; \mathit{instr}'^* \\
[\textsc{Step-ctxt-frame}] & s; f; (\mathsf{frame}_n\,\{f'\}\ \mathit{instr}^*) & \hookrightarrow & s'; f; (\mathsf{frame}_n\,\{f'\}\ \mathit{instr}'^*) & \text{if } s; f'; \mathit{instr}^* \hookrightarrow s'; f'; \mathit{instr}'^*
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{iadd}(N, c_1, c_2) & = & (c_1 + c_2) \bmod 2^N \\
\mathrm{isub}(N, c_1, c_2) & = & (c_1 - c_2 + 2^N) \bmod 2^N \\
\mathrm{imul}(N, c_1, c_2) & = & (c_1 \cdot c_2) \bmod 2^N \\
\mathrm{idiv}(N, c_1, 0) & = & \epsilon \\
\mathrm{idiv}(N, c_1, c_2) & = & c_1 / c_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{binop}(t, \mathsf{add}, c_1, c_2) & = & \mathrm{iadd}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{sub}, c_1, c_2) & = & \mathrm{isub}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{mul}, c_1, c_2) & = & \mathrm{imul}(\mathrm{size}(t), c_1, c_2) \\
\mathrm{binop}(t, \mathsf{div}, c_1, c_2) & = & \mathrm{idiv}(\mathrm{size}(t), c_1, c_2)
\end{array}
$$

$$
\begin{array}{llcll}
[\textsc{Step\_pure-binop-val}] & (\mathsf{const}\ t\ c_1)\ (\mathsf{const}\ t\ c_2)\ (\mathsf{binop}\ t\ \mathit{binop}) & \hookrightarrow & (\mathsf{const}\ t\ c) & \text{if } \mathrm{binop}(t, \mathit{binop}, c_1, c_2) = c \\
[\textsc{Step\_pure-binop-trap}] & (\mathsf{const}\ t\ c_1)\ (\mathsf{const}\ t\ c_2)\ (\mathsf{binop}\ t\ \mathit{binop}) & \hookrightarrow & \mathsf{trap} & \text{if } \mathrm{binop}(t, \mathit{binop}, c_1, c_2) = \epsilon
\end{array}
$$

$$
[\textsc{Step\_read-local.get}] \quad z; (\mathsf{local.get}\ x) \quad \hookrightarrow \quad \mathrm{local}(z, x)
$$

$$
[\textsc{Step-local.set}] \quad z; \mathit{val}\ (\mathsf{local.set}\ x) \quad \hookrightarrow \quad \mathrm{with}_{\mathit{local}}(z, x, \mathit{val}); \epsilon
$$