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>
```

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. Basic syntax of Mini-Wasm is as follows:

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

$$
\begin{array}{lrcl}
\text{(integer)} & iN(N) & ::= & 0 \mid \ldots \mid 2^N - 1
\end{array}
$$

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

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

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

$$\begin{array}{lrcl}
\text{(function type)} & functype & ::= & valtype^* \rightarrow valtype^* \\
\text{(result type)} & resulttype & ::= & valtype^? \\
\text{(external type)} & externtype & ::= & \mathsf{func}\ functype
\end{array}$$

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

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

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

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

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

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

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

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

This syntax is simply written in Wasm-DSL like this:

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

Here, `nat` is a pre-defined syntax, which indicates any natural number. This means the syntax `N` and `n` is a natural number.

3.2 iN

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

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

Now, the syntax `iN` is declared in regard with parameter `N`. Also, you can use `...` to indicate a range.

3.3 $char$, $name$

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

We need a declaration of *name*, which indicates for a general string. We can declare a syntax for a character like this:

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

This syntax is built-in, and denotes Unicode code points. Now, `name` is simply an iteration of `char`s. Use `*` to represent a sequence.

```
syntax name = char*
```

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

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

```
syntax idx = I32
```

Since we have four types of index (which are semantically same, but syntactically different), write like this:

```
syntax typeidx = idx
syntax funcidx = idx
syntax labelidx = idx
syntax localidx = idx
```

3.5 $valtype$

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

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

3.6 $functype$

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

We can use `->` to indicate a function:

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

3.7 *externtype*

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

```
syntax externtype = FUNC functype
```

3.8 size, $num_{valtype}$

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

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

```
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*

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

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

3.10 *instr*

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

```
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*

$$\begin{array}{lrcl} \text{(expression)} & \mathit{expr} & ::= & \mathit{instr}^* \end{array}$$

```
syntax expr = instr*
```

3.12 *module*

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

Declare *module* and its subcomponents as follows:

```
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*
```

4 Runtime-Related Syntax

Runtime-related syntax of Mini-Wasm is as follows:

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

$$\begin{array}{lrcl} \text{(value)} & \mathit{val} & ::= & \mathsf{const}\ \mathit{valtype}\ \mathit{num}_{\mathit{valtype}} \end{array}$$

$$\begin{array}{lrcl} \text{(external value)} & \mathit{externval} & ::= & \mathsf{func}\ \mathit{funcaddr} \end{array}$$

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

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

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

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

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

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

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

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

Make a new file `2-runtime.wastup` and write on it.

4.1 *addr*, *funcaddr*

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

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

4.2 *val*

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

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

4.3 *externval*

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

```
syntax externval = FUNC funcaddr
```

4.4 *funcinst*, *exportinst*, *moduleinst*

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

```
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*

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

```
syntax store = { FUNCS funcinst* }

syntax frame =
  { LOCALS val*,
    MODULE moduleinst }

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

4.6 *admininstr*

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

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

4.7 funcaddr

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

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

4.8 local

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

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

4.9 with$_{local}$

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

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

4.10 funcinst

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

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

4.11 default$_{valtype}$

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

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

5 Validation Rules

We're done with writing syntax of Mini-Wasm. Now, we will declare validation rules of Mini-Wasm.
Make a new file `3-typing.watsup` and write on it.
First, declare some variables to use in validation rules as below:

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

5.1 Context

Define `context` and its metavariable as below:

```
syntax context hint(desc "context") =
{ TYPES functype*, FUNCS functype*,
  LOCALS valtype*, LABELS resulttype*, RETURN resulttype? }

var C : context
```

5.2 Types

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

5.2.1 `Functype_ok`

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

5.2.2 `Externtype_ok/func`

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

5.3 Subtyping

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

5.3.1 Functype_sub

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

5.3.2 Externtype_sub/func

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

5.4 Instructions

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

5.4.1 Expr_ok

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

5.4.2 Instrs_ok/seq

```
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*
```

5.4.3 Instrs_ok/frame

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

5.4.4 Instr_ok/nop

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

5.4.5 Instr_ok/drop

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

5.4.6 Instr_ok/select

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

5.4.7 Instr_ok/block

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

5.4.8 Instr_ok/loop

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

5.4.9 Instr_ok/if

```
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?
```

5.4.10 Instr_ok/br

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

5.4.11 Instr_ok/br_if

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

5.4.12 Instr_ok/call

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

5.4.13 Instr_ok/return

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

5.4.14 Instr_ok/const

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

5.4.15 Instr_ok/binop

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

5.4.16 Instr_ok/local.get

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

5.4.17 Instr_ok/local.set

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

5.5 Constant Expressions

```
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

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

5.5.2 Expr_const

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

5.5.3 Expr_ok_const

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

5.6 Modules

```
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

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

5.6.2 Func_ok

```
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

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

5.6.4 Externidx_ok/func

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

5.6.5 Module_ok

```
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. First, make a new file `4-reduction.wastup` and write as follows:

```
relation Step: config ~> config
```

It means that `rules` of `relation Step` receives `config` as its input and then yields new `config` as its output. Here, `config` is defined in file `4-runtime.watsup`:

```
syntax config = state; admininstr*
syntax admininstr =
  | instr
  | CALL_ADDR funcaddr
  | LABEL_ n `{instr*} admininstr*
  | FRAME_ n `{frame} admininstr*
  | TRAP
```

As you can see, `config` is a pair of `state` and sequence of `admininstr`. `admininstr` is a superset of `instr`, which contains some additional administrative instructions.
Declaring reduction rule in Wasm-DSL is basically done like this:

```
rule Step/<rule name>:
  <input state>; <input instructions>  ~>  <output state>; <output instructions>
  -- if <condition>
  ...
  -- if <condition>
```

This means that if every `condition` is satisfied, then the state and instructions from stack is reduced to the right hand side. There may be no conditions.
Now we'll declare each of the reduction rules one by one.

6.1 NOP

Use `eps` to indicate an empty instruction sequence.
Use metavariable `z` for a `state`.

Answer:

```
rule Step/nop:
  z; NOP  ~>  z; eps
```

6.2 DROP

Use following syatax `val`, which is defined in `4-runtime.watsup`:

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

as a variable.

Answer:

```
rule Step/drop:
  z; val DROP  ~>  z; eps
```

6.3 SELECT

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/select-true:
  ...

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

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

Answer:

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

rule Step/select-false:
  z; val_1 val_2 (CONST I32 c) SELECT  ~>  z; 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 prosed as the same reduction rule.

6.4 BINOP

When we get BINOP from stack, there are three cases for binop_(valtype): ADD, SUB, MUL.
Define each of the case as seperate rule, and use built-in functions iadd, isub, imul 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 $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)
def $imul(N, c_1, c_2) = $((c_1 * c_2) \ 2^N)

rule Step/binop-add:
  ...

rule Step/binop-sub:
  ...

rule Step/binop-mul:
  ...
```

Use function size, which is defined from subsection 4.1.

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)

rule Step/binop-add:
  z; (CONST t c_1) (CONST t c_2) (BINOP t ADD)  ~>  z; (CONST t c)
-- if $iadd($size(t), c_1, c_2) = c

rule Step/binop-sub:
z; (CONST t c_1) (CONST t c_2) (BINOP t SUB)  ~>  z; (CONST t c)
-- if $isub($size(t), c_1, c_2) = c

rule Step/binop-mul:
z; (CONST t c_1) (CONST t c_2) (BINOP t MUL)  ~>  z; (CONST t c)
-- if $imul($size(t), c_1, c_2) = c
```

Now, we may combine them by declaring a new function `binop` 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 $binop(valtype, binop_(valtype), val_(valtype), val_(valtype)) : val_(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)

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

6.5 BLOCK

To indicate a label, refer to following syntax:

```
LABEL_ n `{instr*} admininstr*
```

Answer:

```
rule Step/block-eps:
  z; (BLOCK eps instr*)  ~>  z; (LABEL_ 0 `{eps} instr*)
rule Step/block-val:
  z; (BLOCK t instr*)  ~>  z; (LABEL_ 1 `{eps} instr*)
```

Here, you may combine them by using `/\` for 'and', `\/` for 'or', and `?` for an optional argument:

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

6.6 LOOP

Answer:

```
rule Step/loop:
  z; (LOOP t? instr*)  ~>  z; (LABEL_ 0 `{LOOP t? instr*} instr*)
```

6.7 IF

Answer:

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

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

6.8 BR

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/br-zero:
z; (LABEL_ n `{instr'*} val'* val^n (BR 0) instr*)  ~>  z; val^n instr'*

rule Step/br-succ:
z; (LABEL_ n `{instr'*} val* (BR $(l+1)) instr*)  ~>  z; val* (BR l)
```

6.9 BR_IF

Answer:

```
rule Step/br_if-true:
  z; (CONST I32 c) (BR_IF l)  ~>  z; (BR l)
  -- if c =/= 0

rule Step/br_if-false:
  z; (CONST I32 c) (BR_IF l)  ~>  z; eps
  -- if c = 0
```

6.10 CALL

To get a sequence of `funcaddr` from `state z`, write `$funcaddr(z)`.
To get `n`th element of sequence `seq`, write `seq[n]`.

Answer:

```
rule Step/call:
  z; (CALL x)  ~>  z; (CALL_ADDR $funcaddr(z)[x])
```

6.11 FRAME

To indicate a frame, refer to following syntax:

```
FRAME_ n ‘{frame} admininstr*
```

Answer:

```
rule Step/frame-vals:
  z; (FRAME_ n ‘{f} val^n)  ~>  z; val^n
```

6.12 RETURN

Answer:

```
rule Step/return-frame:
  z; (FRAME_ n ‘{f} val’* val^n RETURN instr*)  ~>  z; val^n

rule Step/return-label:
  z; (LABEL_ n ‘{instr’*} val* RETURN instr*)  ~>  z; val* RETURN
```

6.13 LOCAL.GET

To get a local value from `state z` and `idx x`, write `$local(z, x)`.

Answer:

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

6.14 LOCAL.SET

To get a new state which is exactly same with `state z` except that its local value with `idx x` is `val v`, write `$with_local(z, x, v)`.

Answer:

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