Minimini-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 Minimini-Wasm) is used as our goal. Abstract syntax of Minimini-Wasm is as follows:

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{type}^*\ \mathit{func}^*\ \mathit{start}^*\ \mathit{export}^* \\
\mathit{type} & ::= & \texttt{type}\ \mathit{type}_{\mathit{func}} \\
\mathit{code} & ::= & \mathit{local}^*\ \mathit{instr}^* \\
\mathit{local} & ::= & \texttt{local}\ \mathit{type}_{\mathit{val}} \\
\mathit{func} & ::= & \texttt{func}\ \mathit{idx}_{\mathit{type}}\ \mathit{code} \\
\mathit{start} & ::= & \texttt{start}\ \mathit{idx}_{\mathit{func}} \\
\mathit{export} & ::= & \texttt{export}\ \text{``}\texttt{name}\text{''}\ (\texttt{func}\ \mathit{idx}_{\mathit{func}}) \\
\mathit{instr} & ::= & \texttt{nop}\ \mid\ \texttt{drop}\ \mid\ \texttt{select}\ \mid\ \mathit{type}_{\mathit{val}}\texttt{.const}\ \mathit{value}\_(\mathit{type}_{\mathit{val}})\ \mid\ \mathit{binop} \\
 & \mid & (\texttt{get} \mid \texttt{set} \mid \texttt{tee})\_\texttt{local}\ \mathit{idx}_{\mathit{local}}\ \mid\ \texttt{call}\ \mathit{idx}_{\mathit{func}}\ \mid\ \texttt{return} \\
 & \mid & \texttt{block}\ \mathit{valtype}?\ \mathit{instr}^*\ \mid\ \texttt{loop}\ \mathit{valtype}?\ \mathit{instr}^* \\
 & \mid & \texttt{if}\ \mathit{valtype}?\ \mathit{instr}^*\ \texttt{else}\ \mathit{instr}^*\ \mid\ \texttt{br}\ \mathit{idx}_{\mathit{label}}\ \mid\ \texttt{br\_if}\ \mathit{idx}_{\mathit{label}} \\
\mathit{binop} & ::= & \texttt{i32.add} \mid \texttt{i32.sub} \mid \texttt{i32.mul} \mid \texttt{i64.add} \mid \texttt{i64.sub} \mid \texttt{i64.mul} \\
\mathit{type}_{\mathit{val}} & ::= & \texttt{i32} \mid \texttt{i64} \\
\mathit{type}_{\mathit{func}} & ::= & \mathit{type}^*_{\mathit{val}} \rightarrow \mathit{type}^*_{\mathit{val}} \\
\mathit{idx}_{\mathit{type}|\mathit{func}|\mathit{local}|\mathit{label}} & \in & [0,\ 2^{32}-1] \\
\mathit{value}\_(\mathit{i32}) & \in & [0,\ 2^{32}-1] \\
\mathit{value}\_(\mathit{i64}) & \in & [0,\ 2^{64}-1]
\end{array}
$$

1.1 Working Directory

From root, the directory `spectec/tutorial` is where we will work on.

2 Syntax

Now, we will start from writing the syntax of Minimini-Wasm. Make a new file `1-syntax.wastup`. Declaring syntax in Wasm-DSL is basically done like this:

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

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).

We'll declare each of the syntax one by one.

2.1 $type_{val}$

$$type_{val} ::= \texttt{i32} \mid \texttt{i64}$$

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

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

This means the syntax `valtype` is either `I32` or `I64`. Here, write terminal nodes `I32` and `I64` instead of `i32` and `i64`, and nonterminal node `valtype` instead of `VALTYPE`.

2.2 idx

$$idx_{func|local|label} \in [0,\ 2^{32}-1]$$

You can use `...` to describe a range. The Wasm-DSL version of upper syntax will be:

```
syntax idx = 0 | ... | 2^32-1
```

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

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

2.3 $value$

$$\begin{array}{lcl} value\_(i32) & \in & [0,\ 2^{32}-1] \\ value\_(i64) & \in & [0,\ 2^{64}-1] \end{array}$$

Here, the syntax $value\_$ is declared in regard with parameter `valtype`. This can be done like this:

```
syntax val_(valtype)
syntax val_(I32) = 0 | ... | 2^32-1
syntax val_(I64) = 0 | ... | 2^64-1
```

Here we can declare a general range, by using parameter again.

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

Here, `nat` is a pre-defined syntax, which indicates any natural number. Now, we can write `iN(32)` and `iN(64)` instead of `0 | ... | 2^32-1` and `0 | ... | 2^64-1`. New declaration of `idx` and `val` is as follows:

```
syntax idx = iN(32)
syntax val(valtype)
syntax val(I32) = iN(32)
syntax val(I64) = iN(64)
```

2.4 *binop*

$$binop \quad ::= \quad \texttt{i32.add} \mid \texttt{i32.sub} \mid \texttt{i32.mul} \mid \texttt{i64.add} \mid \texttt{i64.sub} \mid \texttt{i64.mul}$$

We can declare two groups of `binop`, classifying them by one's parameter type:

```
syntax binop_(I32) = | ADD | SUB | MUL
syntax binop_(I64) = | ADD | SUB | MUL
```

Now, using `valtype`, it can be combined as:

```
syntax binop_(valtype) = | ADD | SUB | MUL
```

2.5 *instr*

$$\begin{array}{lcl} instr & ::= & \texttt{nop} \mid \texttt{drop} \mid \texttt{select} \mid type_{val}\texttt{.const}\ value\_(type_{val}) \mid binop \\ & \mid & (\texttt{get} \mid \texttt{set} \mid \texttt{tee})\_\texttt{local}\ idx_{local} \mid \texttt{call}\ idx_{func} \mid \texttt{return} \mid \texttt{block}\ valtype?\ instr^* \\ & \mid & \texttt{loop}\ valtype?\ instr^* \mid \texttt{if}\ valtype?\ instr^*\ \texttt{else}\ instr^* \mid \texttt{br}\ idx_{label} \mid \texttt{br\_if}\ idx_{label} \end{array}$$

Use `*` to represent a sequence. Now we can fully write `instr` as follows:

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

2.6 *name*

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
```

Now, `name` is simply an iteration of `char`s.

```
syntax name = char*
```

2.7 $type_{func}$

$$type_{func} \quad ::= \quad type^*_{val} \rightarrow type^*_{val}$$

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

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

2.8 *module*

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{type}^*\ \mathit{func}^*\ \mathit{start}^*\ \mathit{export}^* \\
\mathit{type} & ::= & \texttt{type}\ \mathit{type}_{\mathit{func}} \\
\mathit{code} & ::= & \mathit{local}^*\ \mathit{instr}^* \\
\mathit{local} & ::= & \texttt{local}\ \mathit{type}_{\mathit{val}} \\
\mathit{func} & ::= & \texttt{func}\ \mathit{idx}_{\mathit{type}}\ \mathit{code} \\
\mathit{start} & ::= & \texttt{start}\ \mathit{idx}_{\mathit{func}} \\
\mathit{export} & ::= & \texttt{export}\ \text{``}\texttt{name}\text{''}\ (\texttt{func}\ \mathit{idx}_{\mathit{func}})
\end{array}
$$

Declare *module* and its subcomponents as follows:

```
syntax module = MODULE type* func* start* export*
syntax type = TYPE functype
syntax code = local* instr*
syntax local = LOCAL valtype
syntax func = FUNC typeidx code
syntax start = START funcidx
syntax export = EXPORT name (FUNC funcidx)
```

3 Metavariables

We're done with writing syntax of Minimini-Wasm. Now, we will declare metavariables, which is used in reduction rules.
There are three ways to make use of metavariables.

3.1 Explicit Declarartion

```
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.
Declare variables in file `1-syntax.wastup` as follows:

```
var x : idx
var l : labelidx
var t : valtype
var ft : functype
var in : instr
var e : instr*
var ty : type
var loc : local
var ex : export
var st : start
```

3.2 Using Syntax Name

Also, we can use the syntax name directly as a variable of same type. For example, instead of below code:

```
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)
```

3.3 Using Without Declaration

Finally, 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)
```

4 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 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.

4.1 Declaring Function Body

Function body can be multiple lines to describe case-in-case.
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 `64`.

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`.

4.2 Built-in Functions

Here are some built-in functions that we can use:

```
$ibytes
$inverse_of_ibytes
$nbytes
$vbytes
...
```

You can find their definition from the file `<root>/spectec/src/backend-interpreter/numerics.ml`. Also, even though they are built-in, you should declare their type first, and use them in Wasm-DSL.

4.3 Using Functions

Functions can be used in the definition of other functions or reduction rules like this:

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

5 Reduction Rules

Now, let's write the reduction rules for the instructions. First, make a new file `8-reduction.wastup` and write as follows:

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

It means that `rule`s 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`s are satisfied, then the state and instructions from stack is reduced to the right hand side.
Now we'll declare each of the reduction rules one by one.

5.1 `NOP`

`NOP` means 'no operation', so if we get `NOP` from stack, we can just remove it from stack. Use `eps` to indicate an empty stack.
We'll use metavariable `z` for a `state`. Since `NOP` makes no change on current `state`, just pass the input `state` to the output `state`.

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

5.2 DROP

If we get DROP from stack, then we should remove a value from stack.
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
```

5.3 SELECT

If we get BINOP from stack, then we have two cases: condition is true or false.
Define each of the case as seperate rule as following:

```
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, we can distinguish multiple variables with same type by adding _1, _2 (and so on) after the variable name.
Also, the names of two rules should be the same before hyphen (-), so that they can be prosed as the same reduction rule.

5.4 BINOP

When we get BINOP from stack, there are six cases. Two cases for valtype (I32, I64), and three cases for binop_(valtype) (ADD, SUB, MUL).
First, in the case that valtype is I32 and binop_(valtype) is ADD:

```
def $iadd(N, iN(N), iN(N)) : iN(N)
rule Step/binop-add-i32:
  z; (CONST I32 c_1) (CONST I32 c_2) (BINOP I32 ADD)  ~>  z; (CONST I32 c)
  -- if $iadd(32, c_1, c_2) = c
```

Here, iadd is a built-in function, so only its type is declared.
We can do similarly with the case where valtype is I64:

```
def $iadd(N, iN(N), iN(N)) : iN(N)
rule Step/binop-add-i32:
  z; (CONST I32 c_1) (CONST I32 c_2) (BINOP I32 ADD)  ~>  z; (CONST I32 c)
  -- if $iadd(32, c_1, c_2) = c
rule Step/binop-add-i64:
  z; (CONST I64 c_1) (CONST I64 c_2) (BINOP I64 ADD)  ~>  z; (CONST I64 c)
  -- if $iadd(64, c_1, c_2) = c
```

Now, we can combine them using the function `size` and metavariable `t`:

```
def $iadd(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
```

We can do similarly with the case where `binop_(valtype)` is `SUB` and `MUL`:

```
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 can 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
```

```
;; Block instructions

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

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

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


;; Branch instructions

;; TODO: may want a label context instead of bubbling up
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)
```

```
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


;; Function instructions

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

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

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


;; Numeric instructions


;; Local instructions

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

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

rule Step/local.tee:
z; val (LOCAL.TEE x)  ~>  z; val val (LOCAL.SET x)
```