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 Abstract Syntax of Mini-Wasm

Abstract syntax of Mini-Wasm is as follows:

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

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

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

$$\begin{array}{lrcl} \text{(index)} & \mathit{idx} & ::= & i32 \\ \text{(type index)} & \mathit{typeidx} & ::= & \mathit{idx} \\ \text{(function index)} & \mathit{funcidx} & ::= & \mathit{idx} \\ \text{(label index)} & \mathit{labelidx} & ::= & \mathit{idx} \\ \text{(local index)} & \mathit{localidx} & ::= & \mathit{idx} \end{array}$$

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

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

$$\mathit{num}_{\mathit{valtype}} \quad ::= \quad i|\mathit{valtype}|$$

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

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

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

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

1.2 Working Directory

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

1.3 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 Syntax

Now, we will start from writing the syntax of Mini-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 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.

2.2 iN

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

Write as following:

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

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

2.3 $char$, $name$

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

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.

```
syntax name = char*
```

2.4 idx

$$\begin{array}{lrcl} \text{(index)} & idx & ::= & i32 \\ \text{(type index)} & typeidx & ::= & idx \\ \text{(function index)} & funcidx & ::= & idx \\ \text{(label index)} & labelidx & ::= & idx \\ \text{(local index)} & localidx & ::= & idx \end{array}$$

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
```

2.5 *valtype*

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

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`.

2.6 *value*

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

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
```

The type is defined differently for different parameters, by pattern-matching on the parameter. Here we can declare a general range, by using parameter again.

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

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)
```

2.7 *binop*

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

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

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

2.8 *instr*

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

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
  | CALL funcidx
  | RETURN
  | BLOCK valtype? instr*
  | LOOP valtype? instr*
  | IF valtype? instr* ELSE instr*
  | BR labelidx
  | BR_IF labelidx
```

2.9 $type_{func}$

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

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

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

2.10 *module*

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

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 Mini-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 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`.

4.2 Using Functions

Functions can be used in definition of other functions or reduction rules. When using function, write as below form:

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

For example,

```
$iadd(c_1, c_2)
```

equals to the return value of function `iadd` with arguments `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 `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.

5.1 NOP

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

Answer:

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

5.2 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
```

5.3 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 following:

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

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

Write like `val_1`, `val_2` (and so on) to distinguish multiple `val`s.
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.

5.4 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 following:

```
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
```

5.5 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
```

5.6 LOOP

Answer:

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

5.7 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
```

5.8 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)
```

5.9 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
```

5.10 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])
```

5.11 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
```

5.12 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
```

5.13 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)
```

5.14 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
```