Quite formal documentation of usuba syntax

Usuba team

June 17, 2022

Contents

1 Notes

- Symbols in `< >` are nonterminal. Other symbols are terminal
- When the construction `< <nt_symbol> list <t_symbol> >` appears it means a list of `<nt_symbol>` separated by `<t_symbol>` (if no `<t_symbol>` is provided, the list is separated by spaces)
  - example: `<int list ,>` is a list of `<int>` separated by `,` or `<int>, <int>, ..., <int>`
  - this list can be empty. A non-empty list is marked as `< <nt_symbol> ne_list <t_symbol> >`
- Symbols preceded by `\` are escaped (i.e. `\|` or `\<`)
- `(...)?` means that the symbol inside the parentheses is optional

2 Utility

2.1 Identifier

An ASCII word starting with `_` or a letter (majuscule or minuscule) followed by `_`, `'`, letters or digits (i.e.: `_b1'A`). Keywords, types etc can obviously not be identifiers.

```
<ident>: [a-zA-Z_][[a-zA-Z0-9_']*
```

2.2 Integer

A decimal or hexadecimal integer (i.e. `12`, `0xAF3`)

```
<int>:
| [0-9]+
| 0x[0-9a-fA-F]+
```

2.3 Simple type

A simple type is a symbol represented by:

- `nat` or
- `u` followed optionally by a `<dir>`, an integer or `'`identifier and optionally `x`integer or
- `v|b` followed optionally by a `<dir>` and an integer
  - a `<dir>` is `U`, `V`, `B`, an integer or an identifier delimited by chevrons

2.4 Binary operators

2.4.1 Arithmetic operators

```
<arith_op>: + | * | / | - | %
```

2.4.2 Logical operators

Notice the absence of `not` as this is not a binary operator.

```
<log_op>: & | \| | ^
```

2.4.3 Shifting and rotation operators

```
// < and > are terminal symbols here
<shift_op>: << | <<< | >> | >>> | >>!
```

2.5 Variable

An identifier or a variable followed by:

- an array access: [arithmetic expression],
- a range: [arithmetic expression..arithmetic expression],
- a slice: [arithmetic expression, ..., arithmetic expression]

(i.e., `arr[2..3][3,2]`)

```
<var>:
| <ident>
| <var> [<arith_exp>]
| <var> [<arith_exp>..<arith_exp>]
| <var> [<arith_exp list ,>]
```

2.6 Parameter

A comma separated list of identifiers, a colon, nothing, `const` and/or `lazyLift` and a type (i.e. `a1,a2,b1,b2:  const u32`)

```
<param>: <ident list ,> : <opt_param list> <type>

<opt_param>: const | lazyLift
```

3 Type

- a type is composed of a `<simple_type>` and a (possibly empty) list of `<size>`,
- a size is a bracketed arithmetic expression

3.1 Example

```
u32x2[17][5]
u<U>32
```

3.2 Grammar

```
<type>: <simple_type><size list>

<simple_type>: nat | u(<dir>)?(<int>|'<ident>)(x<int>)? | (v|b)(<dir>)?<int>

<dir>: \< U|V|B|<int>|'<ident> \>

<size>: [<arith_exp>]
```

4 Arithmetic expression

A parenthesized `<arith_exp>`, an integer, an identifier or an arithmetic operation over two arithmetic expressions (i.e. `(3+foo)`).

```
<arith_exp>:
| (<arith_exp>)
| <int>
| <ident>
| <arith_exp> <arith_op> <arith_exp>
```

5 Usuba program

An usuba program is a list of:

- includes: statements of the form `include "<string>"`,
- declarations

```
<usuba_program>: <decl_or_include list>

<decl_or_include>:
| include "<string>"
| <declaration>

<string>: ( [^\"] | \. )+
```

5.1 Declaration

A declaration is build according to the following sequence:

- a (possibly empty) list of options (`_no_inline`, `_inline`, `_no_opt`, `_interleave (<int>)`),
- the type of declaration (`node`, `perm` or `table`),
- the identifier of the node
- `[]` if the body is an array, nothing otherwise
- a parenthesized list of parameters separated by commas
- the `returns` keyword
- a parenthesized output (same syntax as the parameters)
- if the type of the declaration is `node` then a node_body
- else if the type of the declaration is `perm/table` then a perm_or_table_body

5.1.1 Example

```
_inline node f(x:u32) returns (y:u32)
let
    y = ((x <<< 5) & refresh(x)) ^ (x <<< 1)
tel

perm sbox_perm(a:b8) returns (b:b8) {
     6, 7, 1, 2, 4, 8, 5, 3
}

table[] sbox (input:v4) returns (out:v4) [
    { 3, 8,15, 1,10, 6, 5,11,14,13, 4, 2, 7, 0, 9,12 } ;
    {15,12, 2, 7, 9, 0, 5,10, 1,11,14, 8, 6,13, 3, 4 }
]
```

5.1.2 Grammar

```
<node>:
| <opt list> node <ident> (<param list ,>) returns (<output>) <node_body>
| <opt list> (perm|table) <ident> (<param list ,>) returns (<output>)
     <perm_or_table>

<opt>: _no_inline | _inline | _no_opt | _interleave (<int>)
```

5.2 Node body

A node body is a `<deq>` or a bracketed semi-colon separated list (an array) of `<deq>`

A `<deq>` consists of:

- Optional variable declarations ~vars <parameter list ,>
- `let` assignment `tel`

```
<node_body>:
| <deq>
| [ <deq list ;> ]

<deq>: (vars <param list ,>)? let <assignments> tel
```

5.2.1 Assignment

An assignment is a list of forall assignments and simple assignments.

The list has semicolons separators after simple assignments but they're optional if for forall assignments.

```
assignment:
  | <forall_assignment> <; list> (<assignment>)?
  | <simple_assignment> <; ne_list> (<assignment>)?
```

5.2.2 Forall assignment

A forall assignment is a way of enclosing assignment in a for loop. The forall loop can be parameterized by one or more options then is built in the following way:

- `forall`
- identifier
- `in`
- `[` arithmetic expression `,` arithmetic expression `]`
- `{` assignment `}`

1. Example

   ```
   forall i in [0, 15] {
       state[i+1] = ACE_step(state[i], RC[0,1,2][i],SC[0,1,2][i]);
   }
   ```

2. Grammar

   ```
   <forall_assignment>: <opt list> forall <ident> in [ <arith_exp>, <arith_exp>] { <a

   <opt>: _unroll | _no_unroll | _pipelined
   ```

5.2.3 Simple assignment

A simple assignment (simple because it can not contain assignments and thus no `forall` constructions) is either an imperative assignment or an "equational" assignment of an expression to a single variable or a parenthesized comma separated of variables.

```
<simple_assignment>: <var_pattern> (<op>)?(:)?= <expr>

<var_pattern>: <var> | (<var list ,>)

<op>: <log_op> | <arith_op>
```

5.2.4 Expression

Expressions are a bit long to describe with words but simple to understand so let's jump directly to their grammar (expressions use integers, identifiers, variables, types, arithmetic expressions, binary operators. Patterns marked with a number are explained here:

1. Tuple of `<exp>`
2. Shuffle bits in a variable according to a list of integers
3. Extracts a mask of `0xfff` or `0` from a single bit
4. Merge an uk and ul variable into an u(k+l) variable
5. Function call
6. Function call in an array of function (node[], table[] or perm[])

```
<exp>:
| (<exp>)
| <int>
| <int> : <type>
| <var>
| (<exp ne_list ,>)                          1
| Shuffle (<var, [<int ne_list ,>])          2
| <var> { <int ne_list ,> }
| <exp> <log_op> <exp>
| <exp> <arith_op> <exp>
| <exp> <shift_op> <exp>
```

```
| ~<exp>
| Bitmask (<exp>, <arith_exp>)               3
| Pack (<exp>, <exp>)(: <type>)?             4
| <ident>(<exp ne_list ,>)                   5
| <ident>\< <arith_exp> \>(<exp ne_list ,>)  6
```

5.3 Permutation/Table body

A permutation or a table are simply an `<int_list>` (a comma separated list of integers) or an array of `<int_list>`

5.3.1 Examples

```
6, 7, 1, 2, 4, 8, 5, 3

[ { 3, 8,15, 1,10, 6, 5,11,14,13, 4, 2, 7, 0, 9,12 } ;
  {15,12, 2, 7, 9, 0, 5,10, 1,11,14, 8, 6,13, 3, 4 } ;
  { 1,13,15, 0,14, 8, 2,11, 7, 4,12,10, 9, 3, 5, 6 }
]
```

5.3.2 Grammar

```
<perm_or_table>:
| <int_list>
| [ <int_list list ;> ]

<int_list>: <int list ,>
```