Macrolop Specification

Temirkhan Myrzamadi (a.k.a. Hirrolot)

November 17, 2020

Contents

1 EBNF Grammar

```
<eval> ::= "MACROLOP_EVAL(" { <term> }* ")" ;

<term> ::= "call(" <op> "," { <term> }* ")"
         | "v(" <preprocessor-token-list> ")" ;

<op>   ::= <ident> | { <term> }+ ;
```

Figure 1: Grammar rules

A metaprogram in Macrolop consists of a (possibly empty) sequence of terms, each of which is either a macro call or just a value.

Notes:

- The grammar above describes metaprograms already expanded by the C preprocessor, except for `MACROLOP_EVAL`, `call`, and `v`.
- `call` accepts `op` either as an identifier or as a non-empty sequence of terms that reduces to an identifier.
- `call` accepts arguments without a separator.

2 Notations

Notation 1 (Sequences)

1. *A sequence has the form* $(x_1, \dots, x_n)$*.*
2. $()$ *denotes the empty sequence.*
3. *An element can be appended by comma: if* $a = (1, 2, 3)$ *and* $b = 4$*, then* $a, b = (1, 2, 3, 4)$*.*
4. *`seq-extract` extracts elements from a sequence without a separator: `seq-extract((a, b, c)) = a b c`.*
5. *`seq-comma-sep` extracts elements from a sequence separated by comma: `seq-comma-sep((a, b, c)) = a, b, c`.*

3 Reduction Semantics

We define reduction semantics for Macrolop. The abstract machine executes configurations of the form $\langle k; acc; control\rangle$:

- k is a continuation of the form $\langle k; acc; control\rangle$, where *control* include the ? sign, which will be substituted with a result after a continuation is called. For example: let $k = \langle k'; (1, 2, 3); v(abc)\ ?\rangle$, then $k(v(ghi))$ is $\langle k'; (1, 2, 3); v(abc)\ v(ghi)\rangle$. A special continuation *halt* terminates the abstract machine with provided result.
- *acc* is an accumulator, a sequence of already computed results.
- *control* is a concrete sequence of terms upon which the abstract machine is operating right now. For example: `call(FOO, v(123) v(456)) v(w 8) v(blah)`.

And here are the computational rules:

$$\begin{aligned}
(v) &: \langle k; acc; v(\overline{tok})\ term\ \overline{term'}\rangle & &\to_1 \langle k; acc,\ \overline{tok}; term\ \overline{term'}\rangle \\
(v\text{-}end) &: \langle k; acc; v(\overline{tok})\rangle & &\to_1 k(seq\text{-}extract(acc, \overline{tok})) \\
(op) &: \langle k; acc; call(\overline{term}, \overline{a})\ \overline{term'}\rangle & &\to_1 \langle\langle k; acc; call(?, \overline{a})\ \overline{term'}\rangle; (); \overline{term}\rangle \\
(args) &: \langle k; acc; call(ident, \overline{a})\ \overline{term}\rangle & &\to_1 \langle\langle k; acc; ident(seq\text{-}comma\text{-}sep(?))\ \overline{term}\rangle; (); \overline{a}\rangle \\
(start) &: MACROLOP\_EVAL(\overline{term}) & &\to_1 \langle halt; (); \overline{term}\rangle
\end{aligned}$$

Figure 2: Computational rules

Notation 2 (Reduction step; concrete sequence; meta-variables)

1. *$\to_1$ denotes a single step of reduction (computation).*

2. *$\overline{x}$ denotes a concrete sequence $x_1 \dots x_n$. For example:* `v(abc) call(FOO, v(123)) v(u 8 9)`.

3. `tok` *denotes a single C preprocessor token,* `term` *is a term defined by the grammar,* `a` *is a term used as an argument.*

Notes:

- A body of a macro called using `call` must follow the grammar of Macrolop, otherwise it might result in a compilation error.

- With the current implementation, at most 2^{14} reduction steps is possible. After exceeding this limit, compilation will likely fail.