Macrolop Specification

Temirkhan Myrzamadi
e-mail: hirrolot@gmail.com

December 11, 2020

Abstract

This paper formally describes the form and execution of metaprograms written in Macrolop, an embedded metalanguage aimed at language-oriented programming in C. See also the official repository [1] for the user-friendly overview and the accompanied standard library [2].

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. This is intentional: suppose you need to generate arguments for some macro and then call it. Without separators, all arguments can be generated uniformly, unlike separation by commas where the last argument need no comma after itself.

However, the `call` syntax hurts IDE support: bad code formatting, no parameters documentation highlighting, et cetera. The workaround is to define a wrapper around an implementation macro like this:

```
/// A documentation string here.
#define FOO(a, b, c) FOO_REAL(a b c)
#define FOO_REAL(a, b, c) // The actual implementation here.
```

Then `FOO` can be called as `FOO(v(1), v(2), v(3))`.

All the public std's macros follow this convention, and moreover, std's public higher-order macros require so for supplied user macros.

2 Reduction Semantics

We define reduction semantics for Macrolop. The abstract machine executes configurations of the form $\langle K; A; C \rangle$:

- K is a continuation of the form $\langle K; A; C \rangle$, where C includes the ? sign denoting a result passed into a continuation. For example, let K be $\langle K'; (1, 2, 3); v(abc)\ ? \rangle$, 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.
- A is an accumulator, a sequence 2 of already computed results.

- C (control) is a concrete sequence 3 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:

$$
\begin{array}{rl}
(v) : \langle K; A; v(\overline{tok})\ t\ \overline{t'}\rangle & \rightarrow \langle K; A,\ \overline{tok}; t\ \overline{t'}\rangle \\
(v\text{-}end) : \langle K; A; v(\overline{tok})\rangle & \rightarrow K(unseq(A, \overline{tok})) \\
(op) : \langle K; A; call(\overline{t}, \overline{a})\ \overline{t'}\rangle & \rightarrow \langle\langle K; A; call(?, \overline{a})\ \overline{t'}\rangle; (); \overline{t}\rangle \\
(args) : \langle K; A; call(ident, \overline{a})\ \overline{t}\rangle & \rightarrow \langle\langle K; A; ident(?)\ \overline{t}\rangle; (); comma\text{-}sep(\overline{a})\rangle \\
(start) : MACROLOP\_EVAL(\overline{t'}) & \rightarrow \langle halt; (); \overline{t'}\rangle
\end{array}
$$

Figure 2: Computational rules

The following notations are used:

Notation 1 (Reduction step)
$\rightarrow$ *denotes a single step of reduction (computation).*

Notation 2 (Sequence)

1. *A sequence has the form* $(x_1, \ldots, x_n)$.
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. `unseq` *extracts elements from a sequence without a separator:* `unseq((a, b, c)) = a b c`.
5. `comma-sep` *places* `v(,)` *between terms in a concrete sequence of terms:* `comma-sep(v(123) call(FOO, v(a)) call(BAR, v(b)))` *results in* `v(123) v(,) call(FOO, v(a)) v(,) call(BAR, v(b))`.

Notation 3 (Concrete sequence)
$\overline{x}$ *denotes a concrete sequence* $x_1 \ldots x_n$. *For example:* `v(abc) call(FOO, v(123)) v(u 8 9)`.

Notation 4 (Meta-variables)

`tok`	*C preprocessor token*
`ident`	*C preprocessor identifier*
`t`	*Macrolop term*
`a`	*Macrolop term used as an argument*

The rules are fairly simple: a concrete sequence of terms provided to `MACROLOP_EVAL` is evaluated sequentially till the end. A function's arguments are evaluated before the function is applied, e.g. Macrolop follows applicative evaluation strategy [3]. When there's no more terms to evaluate, the result is pasted where `MACROLOP_EVAL` has been invoked.

Notes:

- Look at (*args*). Macrolop generates a usual C-style macro invocation with fully evaluated arguments, which will be then expanded by the C preprocessor, resulting in yet another concrete sequence of Macrolop terms to be evaluated by the computational rules.
- With the current implementation, at most 2^{14} reduction steps are possible. After exceeding this limit, compilation will likely fail.

The essence of the Macrolop metalanguage is that it allows recursive macro calls. But to be precise, it allows only indirect recursion. First define the $\twoheadrightarrow$ meta-operator (resembles that in lambda calculi):

Notation 5 (Multiple reduction steps)
$\twoheadrightarrow$ denotes one or more single evaluation steps, e.g. $\overline{t} \twoheadrightarrow \overline{t'}$ is the same as $\overline{t} \to \ldots \to \overline{t'}$.

Then consider these two cases:

- Direct recursion: $call(X, \overline{tok}) \to \overline{t}$, where $\overline{t}$ contains X. Then this X will be blocked forever due to the rules of the C preprocessor (an expansion of `X(...)` containing `X`).
- Indirect recursion: $call(X, \overline{a}) \twoheadrightarrow \overline{t}\ call(Y, \overline{a'})\ \overline{t'}$ and $call(Y, \overline{a'}) \twoheadrightarrow \overline{t''}$, where $\overline{t''}$ contains X. Then this X will **not** be blocked by the C preprocessor, e.g. can be invoked again.

Now let's move on to the examples of reduction. Take the following code:

```
#define X(op)         call(op, v(123))
#define CALL_X(_123) call(X, v(ID))
#define ID(x)         v(x)
```

See how `call(X, v(CALL_X))` is evaluated:

Example 1 (Evaluation of terms)

$$
\begin{gathered}
MACROLOP\_EVAL(call(X, v(CALL\_X))) \\
\downarrow (start) \\
\langle halt; (); call(X, v(CALL\_X))\rangle \\
\downarrow (args) \\
\langle\langle halt; (); X(?)\rangle; (); v(CALL\_X)\rangle \\
\downarrow (v\text{-}end) \\
\langle halt; (); call(CALL\_X, v(123))\rangle \\
\downarrow (args) \\
\langle\langle halt; (); CALL\_X(?)\rangle; (); v(123)\rangle \\
\downarrow (v\text{-}end) \\
\langle halt; (); call(X, v(ID))\rangle \\
\downarrow (args) \\
\langle\langle halt; (); X(?)\rangle; (); v(ID)\rangle \\
\downarrow (v\text{-}end) \\
\langle halt; (); call(ID, v(123))\rangle \\
\downarrow (args) \\
\langle\langle halt; (); ID(?)\rangle; (); v(123)\rangle \\
\downarrow (v\text{-}end) \\
\langle halt; (); v(123)\rangle \\
\downarrow (v\text{-}end) \\
halt(123)
\end{gathered}
$$

The analogous version written in ordinary C looks like this:

```
#define X(op)         op(123)
#define CALL_X(_123) X(ID)
#define ID(x)         x
```

However, unlike the Macrolop version above, it gets blocked due to the second call to `X`:

$$X(CALL\_X) \to CALL\_X(123) \to X(ID)$$

3 Caveats

- Consider this scenario:
  - You call `FOO(1, 2, 3)`
  - It gets expanded by the C preprocessor (not by Macrolop)
  - Its expansion contains `FOO`

  Then `FOO` gets blocked by the C preprocessor, e.g. Macrolop cannot handle ordinary macro recursion; you must use `call` to be sure that recursive calls

will behave as expected. I therefore recommend to use only primitive C-style macros, e.g. for performance reasons or because of you cannot express them in terms of Macrolop.

References

[1] Temirkhan Myrzamadi. *Language-oriented programming in C*. URL: `https://github.com/Hirrolot/macrolop`.

[2] Temirkhan Myrzamadi. *The Macrolop standard library documentation*. URL: `https://hirrolot.github.io/macrolop/`.

[3] Wikipedia. *Applicative order*. URL: `https://en.wikipedia.org/wiki/Evaluation_strategy#Applicative_order`.