

Macrolop Specification

Temirkhan Myrzamadi
e-mail: hirrolot@gmail.com

December 14, 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. It is **not** designed as a user-friendly overview – see the official repository [1].

Contents

1	EBNF Grammar	2
2	Notations	3
3	Reduction Semantics	3
3.1	Examples	5
4	Caveats	6

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 Notations

Notation 1 (Concrete sequence)

- \bar{x} denotes a concrete sequence $x_1 \dots x_n$ (either of terms or preprocessor tokens). For example:
 - $v(abc) \text{ call}(FOO, v(123)) \ v(u \ 8 \ 9)$ (of Macrolop terms)
 - $abc \ 13 \ \text{"hello"} \ + \ -$ (of preprocessor tokens)
- $S \ x$ is appending an element or sequence x to a sequence S . For example:
 - let S be $v(1) \ v(2)$ and x is $v(3)$, then $S \ x$ is $v(1) \ v(2) \ v(3)$ (appending an element)
 - let S be $v(1) \ v(2)$ and x is $v(3) \ v(abc)$, then $S \ x$ is $v(1) \ v(2) \ v(3) \ v(abc)$ (appending a sequence)
- comma-sep places $v(,)$ between terms in a concrete sequence of terms: $\text{comma-sep}(v(1) \ v(2) \ \text{call}(FOO, v(a)))$ results in $v(1) \ v(,) \ v(2) \ v(,) \ \text{call}(FOO, v(a))$.

Notation 2 (Reduction step)

\rightarrow denotes a single step of reduction (computation).

Notation 3 (Multiple reduction steps)

\rightarrow denotes one or more single evaluation steps, e.g. $\bar{t} \rightarrow \bar{t}'$ is the same as $\bar{t} \rightarrow \dots \rightarrow \bar{t}'$.

Notation 4 (Meta-variables)

<i>tok</i>	<i>C</i> preprocessor token
<i>ident</i>	<i>C</i> preprocessor identifier
<i>t</i>	Macrolop term
<i>a</i>	Macrolop term used as an argument

3 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 $\langle K'; (1, 2, 3); v(x) ? \rangle$, then $K(v(y))$ is $\langle K'; (1, 2, 3); v(x) v(y) \rangle$.
- A special continuation *halt* terminates the abstract machine and substitutes itself with a provided result.
- A is an accumulator, a concrete sequence of already computed results.
- C (control) is a concrete sequence of terms upon which the abstract machine is operating right now.

And here are the computational rules:

$(v) : \langle K; A; v(\overline{tok}) \bar{t} \rangle$	$\rightarrow \langle K; A \overline{tok}; \bar{t} \rangle$
$(op) : \langle K; A; call(\bar{t}, \bar{a}) \bar{t}' \rangle$	$\rightarrow \langle \langle K; A; call(?, \bar{a}) \bar{t}' \rangle; () ; \bar{t} \rangle$
$(args) : \langle K; A; call(ident, \bar{a}) \bar{t} \rangle$	$\rightarrow \langle \langle K; A; ident(?) \bar{t} \rangle; () ; comma-sep(\bar{a}) \rangle$
$(end) : \langle K; A; \rangle$	$\rightarrow K(A)$
$(start) : MACROLOP_EVAL(\bar{t})$	$\rightarrow \langle halt; () ; \bar{t} \rangle$

Figure 2: Computational rules

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 [2]. When there's no more terms to evaluate, a 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.

3.1 Examples

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)

$$\begin{aligned} & \text{MACROLOP_EVAL}(\text{call}(X, v(\text{CALL_X}))) \\ & \quad \downarrow (\text{start}) \\ & \langle \text{halt}; (); \text{call}(X, v(\text{CALL_X})) \rangle \\ & \quad \downarrow (\text{args}) \\ & \langle \langle \text{halt}; (); X(?) \rangle; (); v(\text{CALL_X}) \rangle \\ & \quad \downarrow (v) \\ & \langle \langle \text{halt}; (); X(?) \rangle; \text{CALL_X}; \rangle \\ & \quad \downarrow (\text{end}) \\ & \langle \text{halt}; (); \text{call}(\text{CALL_X}, v(123)) \rangle \\ & \quad \downarrow (\text{args}) \\ & \langle \langle \text{halt}; (); \text{CALL_X}(?) \rangle; (); v(123) \rangle \\ & \quad \downarrow (v) \\ & \langle \langle \text{halt}; (); \text{CALL_X}(?) \rangle; 123; \rangle \\ & \quad \downarrow (\text{end}) \\ & \langle \text{halt}; (); \text{call}(X, v(\text{ID})) \rangle \\ & \quad \downarrow (\text{args}) \\ & \langle \langle \text{halt}; (); X(?) \rangle; (); v(\text{ID}) \rangle \\ & \quad \downarrow (v) \\ & \langle \langle \text{halt}; (); X(?) \rangle; \text{ID}; \rangle \\ & \quad \downarrow (\text{end}) \\ & \langle \text{halt}; (); \text{call}(\text{ID}, v(123)) \rangle \\ & \quad \downarrow (\text{args}) \\ & \langle \langle \text{halt}; (); \text{ID}(?) \rangle; (); v(123) \rangle \\ & \quad \downarrow (v) \end{aligned}$$

$$\begin{aligned}
&\langle\langle halt; (); ID(?) \rangle; 123; \rangle \\
&\quad \downarrow (end) \\
&\langle halt; (); v(123) \rangle \\
&\quad \downarrow (v) \\
&\langle halt; 123; \rangle \\
&\quad \downarrow (end) \\
&halt(123)
\end{aligned}$$

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:

$$X(CALL_X) \rightarrow CALL_X(123) \rightarrow X(ID)$$

4 Caveats

- Consider this scenario:
 - You call F00(1, 2, 3)
 - It gets expanded by the C preprocessor (not by Macrolop)
 - Its expansion contains F00

Then F00 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] Wikipedia. *Applicative order*. URL: https://en.wikipedia.org/wiki/Evaluation_strategy#Applicative_order.