Macrolop Specification

Temirkhan Myrzamadi
e-mail: hirrolot@gmail.com

December 10, 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 [**Macrolop**] for the user-friendly overview and the accompanied standard library [**MacrolopDocs**].

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 non-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: it lets arguments be generated programmatically without worrying about separators (for instance, ensuring that there's no separator after the last argument).

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

```
#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 $K = \langle K'; (1, 2, 3); v(abc)\ ? \rangle$, 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 **??** of already computed results.
- C (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:

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

Figure 2: Computational rules

Notation 1 (Sequences)

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`.

Notation 2 (Reduction step)
$\to$ *denotes a single step of reduction (computation).*

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*

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.

  Therefore, an expansion of $call(\overline{t}, \overline{a})$ must match the Macrolop grammar, otherwise it might result in a compilation error.

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

The rules are fairly simple: a concrete sequence of terms provided into `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. When there's no more terms to evaluate, the result is pasted where `MACROLOP_EVAL` has been invoked.

The essence of the Macrolop metalanguage is that it allows recursive macro calls. But to be precise, it allows only indirect recursion. 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.

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

Now let's move 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)

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

The analogous version written in ordinary C looks like this:

```c
#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 Macrolop).