Epilepsy Specification

Temirkhan Myrzamadi
e-mail: hirrolot@gmail.com

December 23, 2020

Abstract

This paper formally describes the form and execution of metaprograms written Epilepsy, a metalanguage aimed at full-blown C/C++ preprocessor metaprogramming. This paper is **not** designed as a user-friendly overview – see the official repository [1].

Contents

1 EBNF Grammar

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

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

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

Figure 1: Grammar rules

A metaprogram in Epilepsy 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 preprocessor, except for `EPILEPSY_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_IMPL(a b c)
#define FOO_IMPL(a, b, c) // The actual implementation here.
```

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

Notice that variadic macros are a bit special here. Their calls should be desugared as follows:

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

Then `FOO` can be called as `FOO(v(1), v(2), v(3), v(5) v(6) v(7))`. `v(5) v(6) v(7)` are **not** separated by commas.

2 Notations

Notation 1 (Sequence)

- $\overline{x} \coloneqq x_1 \ldots x_n$. *Examples:*
  - *Epilepsy terms:* `v(abc) call(FOO, v(123)) v(u 8 9)`
  - *Preprocessor tokens:* `abc 13 "hello" + -`
- $()$ *denotes the empty sequence.*
- *Appending to a sequence:*
  - *Appending an element:* $S\ y \coloneqq x_1 \ldots x_n\ y$, *where* $S = x_1 \ldots x_n$
  - *Appending a sequence:* $S_1\ S_2 \coloneqq x_1 \ldots x_n\ y_1 \ldots y_m$, *where* $S_1 = x_1 \ldots x_n$ *and* $S_2 = y_1 \ldots y_m$

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

Notation 3 (Meta-variables)

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

3 Reduction Semantics

We define a reduction semantics for Epilepsy 2. The abstract machine executes configurations of the form $\langle K; F; A; C \rangle$:

$$
\begin{array}{rll}
(v) : & \langle K; F; A; v(\overline{tok})\ \overline{t}\rangle & \rightarrow \langle K; F; F(A, \overline{tok}); \overline{t}\rangle \\
(op) : & \langle K; F; A; call(\overline{t}, \overline{a})\ \overline{t'}\rangle & \rightarrow \langle\langle K; F; A; call(?, \overline{a})\ \overline{t'}\rangle; fappend; (); \overline{t}\rangle \\
(args) : & \langle K; F; A; call(ident, \overline{a})\ \overline{t}\rangle & \rightarrow \langle\langle K; F; A; ident(?)\ \overline{t}\rangle; fcomma; (); \overline{a}\rangle \\
(end) : & \langle K; F; A; ()\rangle & \rightarrow K(A) \\
(start) : & EPILEPSY\_EVAL(\overline{t}) & \rightarrow \langle halt; fappend; (); \overline{t}\rangle
\end{array}
$$

Figure 2: Reduction Semantics

- K is a continuation of the form $\langle K; F; A; C\rangle$, 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. For example, when the abstract machine encounters $halt(1 + 2)$, it will just stop and paste $1 + 2$.
- F is a left folder of the form $(acc, \overline{tok}) \rightarrow acc$. It is used to flexibly append a newly evaluated term to an accumulator without extra reduction steps. There are the only two folders:
  - $fappend(acc, \overline{tok}) := acc\ \overline{tok}$
  - $fcomma(acc, \overline{tok}) := if(acc\ is\ ())\ then\ \overline{tok}\ else\ acc\ \text{","}\ \overline{tok}$
- A (accumulator) is a sequence of already computed results.
- C (control) is a sequence of terms upon which the abstract machine is operating right now.

Notes:

- Epilepsy follows applicative evaluation strategy [2].
- Look at $(args)$. Epilepsy generates a usual C-style macro invocation with fully evaluated arguments, which will be then expanded by the preprocessor, resulting in yet another concrete sequence of Epilepsy terms to be evaluated by the computational rules.
- With the current implementation, at most 2^{16} 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)

$$\begin{aligned}
EPILEPSY\_EVAL(call(X, v(CALL\_X))) &\rightarrow (start) \\
\langle halt; fappend; (); call(X, v(CALL\_X))\rangle &\rightarrow (args) \\
\langle\langle halt; fappend; (); X(?)\rangle; fcomma; (); v(CALL\_X)\rangle &\rightarrow (v) \\
\langle\langle halt; fappend; (); X(?)\rangle; fcomma; CALL\_X; ()\rangle &\rightarrow (end) \\
\langle halt; fappend; (); call(CALL\_X, v(123))\rangle &\rightarrow (args) \\
\langle\langle halt; fappend; (); CALL\_X(?)\rangle; fcomma; (); v(123)\rangle &\rightarrow (v) \\
\langle\langle halt; fappend; (); CALL\_X(?)\rangle; fcomma; 123; ()\rangle &\rightarrow (end) \\
\langle halt; fappend; (); call(X, v(ID))\rangle &\rightarrow (args) \\
\langle\langle halt; fappend; (); X(?)\rangle; fcomma; (); v(ID)\rangle &\rightarrow (v) \\
\langle\langle halt; fappend; (); X(?)\rangle; fcomma; ID; \rangle &\rightarrow (end) \\
\langle halt; fappend; (); call(ID, v(123))\rangle &\rightarrow (args) \\
\langle\langle halt; fappend; (); ID(?)\rangle; fcomma; (); v(123)\rangle &\rightarrow (v) \\
\langle\langle halt; fappend; (); ID(?)\rangle; fcomma; 123; ()\rangle &\rightarrow (end) \\
\langle halt; fappend; (); v(123)\rangle &\rightarrow (v) \\
\langle halt; fappend; 123; ()\rangle &\rightarrow (end) \\
halt(123)
\end{aligned}$$

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 Epilepsy version above, `X(CALL_X)` gets blocked due to the second call to `X`. The trick is that Epilepsy performs evaluation step-by-step, unlike the preprocessor:

- The Epilepsy version: `X(CALL_X)` expands to `call(CALL_X, v(123))`. This expansion does not contain `X`, and therefore `X` is **not** blocked by the preprocessor.
- The ordinary version: `X(CALL_X)` expands to `X(ID)`. This expansion does contains `X`, and therefore `X` is blocked by the preprocessor.

4 Properties

4.1 Progress

TODO: prove the progress theorem.

5 Caveats

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

  Then `FOO` gets blocked by the preprocessor, e.g. Epilepsy 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 Epilepsy.

References

[1] Temirkhan Myrzamadi. *Full-blown preprocessor metaprogramming in C.* URL: `https://github.com/Hirrolot/epilepsy`.

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