Metalang99 Specification

Hirrolot
e-mail: hirrolot@gmail.com

February 8, 2021

Abstract

This paper formally describes the form and execution of metaprograms written in Metalang99, a metalanguage aimed at full-blown C/C++ preprocessor metaprogramming. This paper is **not** designed as a user-friendly overview – see the official repository [2].

Contents

1 EBNF Grammar

```
<eval> ::= "METALANG99_eval(" { <term> }* ")" ;

<term> ::=
      "METALANG99_call(" <op> "," { <term> }* ")"
    | "METALANG99_callTrivial(" <ident> "," <pp-token-list> ")"
    | "METALANG99_abort(" { <term> }* ")"
    | "METALANG99_fatal(" <ident> "," <pp-token-list> ")"
    | "v(" <pp-token-list> ")" ;

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

Figure 1: Grammar rules

A metaprogram in Metalang99 consists of a possibly empty sequence of terms, each of which is either a macro call or just a value.

Notes:

- `<pp-token-list>` stands for a list of preprocessor tokens (e.g., `a 123, hello!`).
- The grammar above describes metaprograms already expanded by the preprocessor, except for `METALANG99_eval`, `METALANG99_call`, `METALANG99_callTrivial`, `METALANG99_abort`, `METALANG99_fatal`, and `v`.
- `METALANG99_call` accepts `op` either as an identifier or as a non-empty sequence of terms that reduces to an identifier.
- `METALANG99_callTrivial` accepts an operation strictly as an identifier.
- Both `METALANG99_call` and `METALANG99_callTrivial` accept arguments of `op` without a comma separator (e.g., `v(1) v(2) v(3)` and not `v(1), v(2), v(3)`).

The `METALANG99_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:

```
/// The documentation string.
#define FOO(a, b, c) METALANG99_call(FOO, a b c)
#define FOO_IMPL(a, b, c) // The implementation.
```

Then `FOO` can be conveniently 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:

```
/// The documentation string.
#define FOO(a, b, c, ...) \
    METALANG99_call(FOO, a b c __VA_ARGS__)
#define FOO_IMPL(a, b, c, ...) // The implementation.
```

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)

- $\overline{x} \coloneqq x_1 \dots x_n$. *Examples:*
  - *Metalang99 terms:* `v(abc) METALANG99_call(FOO, v(123)) v(u 8 9)`
  - *Preprocessor tokens:* `abc 13 "hello" + -`
- () *denotes the empty sequence.*
- *Appending to a sequence:*
  - *Appending an element:* $S\ y \coloneqq x_1 \dots x_n\ y$, *where* $S = x_1 \dots x_n$
  - *Appending a sequence:* $S_1\ S_2 \coloneqq x_1 \dots x_n\ y_1 \dots y_m$, *where* $S_1 = x_1 \dots x_n$ *and* $S_2 = y_1 \dots y_m$

Notation 2 (Reduction step)

$\to$ *denotes a single step of reduction (computation, evaluation).*

Notation 3 (Metavariables)

tok	*preprocessor token*
ident	*preprocessor identifier*
t	*Metalang99 term*
a	*Metalang99 term used as an argument*

3 Reduction Semantics

We define a reduction semantics for Metalang99 2. The abstract machine executes configurations of the form $\langle K; F; A; C\rangle$:

- 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 $(acc, \overline{tok}) \to acc$. It is used to flexibly append a newly evaluated term to an accumulator without extra reduction steps. There are the only two folders:
  - $fappend(acc, \overline{tok}) \coloneqq acc\ \overline{tok}$
  - $fcomma(acc, \overline{tok}) \coloneqq if(acc\ is\ ())\ then\ \overline{tok}\ else\ acc\ ",\ \overline{tok}$
- 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:

- Metalang99 follows applicative evaluation strategy [3].
- ($args$) Metalang99 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 Metalang99 terms to be evaluated by the computational rules.
- ($args$) Metalang99 appends `_IMPL` to every macro identifier called using `METALANG99_call` – it makes easier to follow the convention that all implementations of metafunctions shall have the postfix `_IMPL`.
- ($callTrivial$) `METALANG99_callTrivial` is used when an operation and all arguments are already evaluated. It is semantically the same as `METALANG99_call(ident, v(...))` but performs one less reduction steps to benefit in performance.

$$
\begin{aligned}
(v) &: \langle K; F; A; \texttt{v}(\overline{tok})\ \bar{t}\rangle \to \langle K; F; F(A, \overline{tok}); \bar{t}\rangle \\
(op) &: \langle K; F; A; \texttt{METALANG99\_call}(\bar{t}, \bar{a})\ \bar{t'}\rangle \to \langle \\
&\quad \langle K; F; A; \texttt{METALANG99\_call}(?, \bar{a})\ \bar{t'}\rangle; \\
&\quad fappend; \\
&\quad (); \\
&\quad \bar{t}\rangle \\
(args) &: \langle K; F; A; \texttt{METALANG99\_call}(ident, \bar{a})\ \bar{t}\rangle \to \langle \\
&\quad \langle\langle K; F; F(A, ?); \bar{t}\rangle; fappend; (); ident\texttt{\_IMPL}(?)\rangle; \\
&\quad fcomma; \\
&\quad (); \\
&\quad \bar{a}\rangle \\
(callTrivial) &: \langle K; F; A; \texttt{METALANG99\_callTrivial}(ident, \overline{tok})\ \bar{t}\rangle \to \langle \\
&\quad \langle K; F; F(A, ?); \bar{t}\rangle; \\
&\quad fappend; \\
&\quad (); \\
&\quad ident\texttt{\_IMPL}(\overline{tok})\rangle \\
(abort) &: \langle K; F; A; \texttt{METALANG99\_abort}(\bar{t})\ \bar{t'}\rangle \to \langle halt; fappend; (); \bar{t}\rangle \\
(fatal) &: \langle K; F; A; \texttt{METALANG99\_fatal}(ident, \overline{tok})\ \bar{t}\rangle \to halt(\ldots) \\
(end) &: \langle K; F; A; ()\rangle \to K(A) \\
(start) &: \texttt{METALANG99\_call}(\bar{t}) \to \langle halt; fappend; (); \bar{t}\rangle
\end{aligned}
$$

Figure 2: Reduction Semantics

- ($fatal$) The ellipsis means that an implementation is free to provide diagnostics in any format.
- ($fatal$) interprets its variadic arguments without preprocessor expansion – i.e., they are pasted as-is. This is intended because otherwise identifiers located in an error message may stand for other macros that will be unintentionally expanded.
- With the current implementation, at most 2^{16} reduction steps are possible. After exceeding this limit, Metalang99 will not be able to perform reduction of a given metaprogram anymore.

3.1 Examples

Take the following code:

```
#define X_IMPL(op)          METALANG99_call(op, v(123))
#define CALL_X_IMPL(_123) METALANG99_call(X, v(ID))
#define ID_IMPL(x)          v(x)
```

See how `METALANG99_call(X, v(CALL_X))` is evaluated:

Example 1 (Evaluation of terms)

$$\begin{aligned}
\texttt{METALANG99_eval}(\texttt{METALANG99_call}(X, \texttt{v}(\texttt{CALL_X}))) &\to (start)\\
\langle halt; fappend; (); \texttt{METALANG99_call}(X, \texttt{v}(\texttt{CALL_X}))\rangle &\to (args)\\
\langle\langle halt; fappend; (); \texttt{X}(?)\rangle; fcomma; (); \texttt{v}(\texttt{CALL_X})\rangle &\to (v)\\
\langle\langle halt; fappend; (); \texttt{X}(?)\rangle; fcomma; \texttt{CALL_X}; ()\rangle &\to (end)\\
\langle halt; fappend; (); \texttt{METALANG99_call}(\texttt{CALL_X}, \texttt{v}(123))\rangle &\to (args)\\
\langle\langle halt; fappend; (); \texttt{CALL_X}(?)\rangle; fcomma; (); \texttt{v}(123)\rangle &\to (v)\\
\langle\langle halt; fappend; (); \texttt{CALL_X}(?)\rangle; fcomma; 123; ()\rangle &\to (end)\\
\langle halt; fappend; (); \texttt{METALANG99_call}(X, \texttt{v}(\texttt{ID}))\rangle &\to (args)\\
\langle\langle halt; fappend; (); \texttt{X}(?)\rangle; fcomma; (); \texttt{v}(\texttt{ID})\rangle &\to (v)\\
\langle\langle halt; fappend; (); \texttt{X}(?)\rangle; fcomma; \texttt{ID}; \rangle &\to (end)\\
\langle halt; fappend; (); \texttt{METALANG99_call}(\texttt{ID}, \texttt{v}(123))\rangle &\to (args)\\
\langle\langle halt; fappend; (); \texttt{ID}(?)\rangle; fcomma; (); \texttt{v}(123)\rangle &\to (v)\\
\langle\langle halt; fappend; (); \texttt{ID}(?)\rangle; fcomma; 123; ()\rangle &\to (end)
\end{aligned}$$

$$\langle halt; fappend; (); \texttt{v}(123)\rangle \to (v)$$
$$\langle halt; fappend; 123; ()\rangle \to (end)$$
$$halt(123)$$

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 Metalang99 version above, `X(CALL_X)` gets blocked [1] due to the second call to `X`. The trick is that Metalang99 performs evaluation step-by-step, unlike the preprocessor:

- The Metalang99 version: `X(CALL_X)` expands to `METALANG99_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

Proposition 1 (Progress) *Either* $\langle K; F; A; \overline{t}\rangle \to \langle K; F; A; \overline{t'}\rangle$ *or* $\langle K; F; A; \overline{t}\rangle \to halt(\overline{x})$. □

Proof By inspection of 2. ■

5 Caveats

- Consider this scenario:
  - You call `FOO(1, 2, 3)`
  - It gets expanded by the preprocessor (not by Metalang99)
  - Its expansion contains `FOO`

Then `FOO` gets blocked [1] by the preprocessor, i.e. Metalang99 cannot handle ordinary macro recursion; you must use `METALANG99_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 Metalang99.

References

[1] *C99 draft, section 6.10.3.4, paragraph 2 – Rescanning and further replacement*. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[2] Hirrolot. *A functional language for C/C++ preprocessor metaprogramming*. URL: `https://github.com/Hirrolot/metalang99`.

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