Metalang99 Specification (v1.4.1)

Hirrolot
e-mail: hirrolot@gmail.com

August 5, 2021

Abstract

This paper formally describes the syntax and semantics of metaprograms written in Metalang99, a metalanguage aimed at full-blown C99 preprocessor metaprogramming. For the motivation and a user-friendly overview, please see the official repository [**Metalang99**].

Contents

1 EBNF Grammar

```
<eval> ::= "ML99_EVAL(" <term-seq> ")" ;

<term-seq> ::= <term> { "," <term> }* ;

<term> ::=
      "ML99_call(" <op> "," <term-seq> ")"
    | "ML99_callUneval(" <ident> "," <pp-token-list> ")"
    | "ML99_abort(" <term-seq> ")"
    | "ML99_fatal(" <ident> "," <pp-token-list> ")"
    | "v(" <pp-token-list> ")" ;

<op> ::= <ident> | <term> ;
```

Figure 1: Grammar rules

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 `ML99_EVAL`, `ML99_call`, `ML99_callUneval`, `ML99_abort`, `ML99_fatal`, and `v`.
- `ML99_call` accepts `op` either as an identifier or as a term that reduces to an identifier.
- `ML99_callUneval` accepts an operation strictly as an identifier.

The `ML99_call` syntax hurts IDE support (no parameters documentation highlighting) and is also less natural. The workaround is to define a wrapper around an implementation macro like this:

```
/// The documentation string.
#define FOO(a, b, c) ML99_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))`.

2 Notations

Notation 1 (Sequence)

- $\overline{x} \coloneqq x_1 \ldots x_n$. *Examples:*
  - *Metalang99 terms:* `v(abc), ML99_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 (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 **??**. 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 the 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:
  - $fspace(acc, \overline{tok}) \coloneqq acc\ \overline{tok}$
  - $fcomma(acc, \overline{tok}) \coloneqq if(acc\ is\ ())\ then\ \overline{tok}\ else\ acc\ ","\ \overline{tok}$
- 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 [**ApplicativeEvaluationStrategy**].
- ($args$) 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 `ML99_call` – it makes easier to follow the convention that all implementations of metafunctions must have the postfix `_IMPL`.
- ($callUneval$) `ML99_callUneval` is used when an operation and all arguments are already evaluated. It is semantically the same as `ML99_call(ident, v(...))` but performs one less reduction steps to benefit in performance.
- ($fatal$) 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.
- The number of reduction steps is finite and may vary from version to version. If the limit is exceeded, Metalang99 will not be able to perform reduction of a given metaprogram anymore.

$$
\begin{aligned}
(v) &: \langle K; F; A; \texttt{v}(\overline{tok}), \overline{t}\rangle \rightarrow \langle K; F; F(A, \overline{tok}); \overline{t}\rangle \\
(op) &: \langle K; F; A; \texttt{ML99\_call}(t, \overline{a}), \overline{t'}\rangle \rightarrow \langle \\
&\quad \langle K; F; A; \texttt{ML99\_call}(?), \overline{t'}\rangle; \\
&\quad fcomma; \\
&\quad (); \\
&\quad t, \overline{a}\rangle \\
(args) &: \langle K; F; A; \texttt{ML99\_call}(ident, \overline{a}), \overline{t}\rangle \rightarrow \langle \\
&\quad \langle\langle K; F; F(A, ?); \overline{t}\rangle; fspace; (); ident\_\texttt{IMPL}(?)\rangle; \\
&\quad fcomma; \\
&\quad (); \\
&\quad \overline{a}\rangle \\
(callUneval) &: \langle K; F; A; \texttt{ML99\_callUneval}(ident, \overline{tok}), \overline{t}\rangle \rightarrow \langle \\
&\quad \langle K; F; F(A, ?); \overline{t}\rangle; \\
&\quad fspace; \\
&\quad (); \\
&\quad ident\_\texttt{IMPL}(\overline{tok})\rangle \\
(abort) &: \langle K; F; A; \texttt{ML99\_abort}(\overline{t}), \overline{t'}\rangle \rightarrow \langle halt; fspace; (); \overline{t}\rangle \\
(fatal) &: \langle K; F; A; \texttt{ML99\_fatal}(ident, \overline{tok}), \overline{t}\rangle \rightarrow halt(\ldots) \\
(end) &: \langle K; F; A; ()\rangle \rightarrow K(A) \\
(start) &: \texttt{ML99\_call}(\overline{t}) \rightarrow \langle halt; fspace; (); \overline{t}\rangle
\end{aligned}
$$

Figure 2: Reduction Semantics

3.1 Examples

Take the following code:

```
#define X_IMPL(op)          ML99_call(op, v(123))
#define CALL_X_IMPL(_123)   ML99_call(X, v(ID))
#define ID_IMPL(x)          v(x)
```

See how `ML99_call(X, v(CALL_X))` is evaluated:

Example 1 (Evaluation of terms)

$$
\begin{aligned}
&\texttt{ML99\_EVAL}(\texttt{ML99\_call}(X, \texttt{v}(\texttt{CALL\_X}))) \xrightarrow{(start)} \\
&\langle halt; fspace; (); \texttt{ML99\_call}(X, \texttt{v}(\texttt{CALL\_X}))\rangle \xrightarrow{(args)} \\
&\langle\langle halt; fspace; (); \texttt{X}(?)\rangle; fcomma; (); \texttt{v}(\texttt{CALL\_X})\rangle \xrightarrow{(v)} \\
&\langle\langle halt; fspace; (); \texttt{X}(?)\rangle; fcomma; \texttt{CALL\_X}; ()\rangle \xrightarrow{(end)} \\
&\langle halt; fspace; (); \texttt{ML99\_call}(\texttt{CALL\_X}, \texttt{v}(123))\rangle \xrightarrow{(args)} \\
&\langle\langle halt; fspace; (); \texttt{CALL\_X}(?)\rangle; fcomma; (); \texttt{v}(123)\rangle \xrightarrow{(v)} \\
&\langle\langle halt; fspace; (); \texttt{CALL\_X}(?)\rangle; fcomma; 123; ()\rangle \xrightarrow{(end)} \\
&\langle halt; fspace; (); \texttt{ML99\_call}(X, \texttt{v}(\texttt{ID}))\rangle \xrightarrow{(args)} \\
&\langle\langle halt; fspace; (); \texttt{X}(?)\rangle; fcomma; (); \texttt{v}(\texttt{ID})\rangle \xrightarrow{(v)} \\
&\langle\langle halt; fspace; (); \texttt{X}(?)\rangle; fcomma; \texttt{ID}; \rangle \xrightarrow{(end)} \\
&\langle halt; fspace; (); \texttt{ML99\_call}(\texttt{ID}, \texttt{v}(123))\rangle \xrightarrow{(args)} \\
&\langle\langle halt; fspace; (); \texttt{ID}(?)\rangle; fcomma; (); \texttt{v}(123)\rangle \xrightarrow{(v)} \\
&\langle\langle halt; fspace; (); \texttt{ID}(?)\rangle; fcomma; 123; ()\rangle \xrightarrow{(end)} \\
&\langle halt; fspace; (); \texttt{v}(123)\rangle \xrightarrow{(v)} \\
&\langle halt; fspace; 123; ()\rangle \xrightarrow{(end)} \\
&halt(123)
\end{aligned}
$$

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 Metalang99 version above, `X(CALL_X)` gets blocked [**Bluepainting**] 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 `ML99_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* $\langle K; F; A; \overline{t}\rangle \to \mathit{halt}(\overline{x})$. □

Proof By inspection of **??**. ■

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 [**Bluepainting**] by the preprocessor, i.e. Metalang99 cannot handle ordinary macro recursion; you must use `ML99_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.