Hygienic Macros via Staged Environment Machines (Position Paper)

Yuito Murase
murase@fos.kuis.kyoto-u.ac.jp
Kyoto University
Kyoto, Japan

Abstract

The relationship between staged computation and procedural macros is often mentioned in the literature. However, this relationship is not as straightforward as it may appear. Existing approaches tend to compromise the role of macros as syntactic extensions, focusing primarily on staged type systems to enforce the static safety of macros.

In this position paper, we propose a different approach to connecting procedural macros and staged computation: to understand the semantic aspect of procedural macros through the lens of staged computation. We observe that the notion of a syntactic environment in hygienic macros has a natural counterpart in a staged extension of environment machines. Building on this observation, we sketch our draft design of an environment machine for a Lisp-like language with an explicit-renaming macro facility à la Clinger.

***CCS Concepts:* • Software and its engineering → Semantics.**

Keywords: Multi-Stage Programming, Hygienic Macros, Abstract Machine

ACM Reference Format:
Yuito Murase. 2025. Hygienic Macros via Staged Environment Machines (Position Paper). In *Proceedings of the 26th ACM SIGPLAN International Workshop on Scheme and Functional Programming (Scheme '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759537.3762695

1 Introduction

The similarities between staged computation [5, 6, 22, 23] and Lisp-style procedural macros [4, 7, 9, 10] have long been noted and discussed from multiple perspectives. Both treat code fragments as first-class data and provide syntactic mechanisms such as quasiquotation to construct and manipulate code.

In both settings, hygiene—the preservation of lexical scoping—has been recognized as a major concern. However, the way hygiene is handled differs significantly between staged computation and macros.

```
let min = fun exp1 exp2 ->
.< let t1 = .~exp1 in
   let t2 = .~exp2 in
   if (t1 <= t2) then t1 else t2 >.
```

Figure 1. `min` function in MetaOCaml

Hygiene in Staged Computation. In staged computation, hygiene is relatively easy to ensure. The program in Figure 1 defines a function in MetaOCaml [17] that generates a code fragment computing the minimum of two expressions. In MetaOCaml, a *quotation* .<···>. produces a representation of the program inside the quotation instead of evaluating it. A *splice* .~··· within a quotation embeds the given code fragment into the surrounding program. For example, `min .< foo 10 >. .< t1 >.` produces the following code fragment:

```
.< let t1_1 = foo 10 in
   let t2 = t1 in
   if t1_1 <= t2 then t1_1 else t2 >.
```

The outermost .<···>. denotes a code value. The occurrences of `t1` introduced by `min` are renamed to `t1_1` to avoid conflicts with the `t1` in the second argument. In formal semantics, such renaming is typically achieved via α-renaming, as in standard λ-calculi. Thus, α-renaming plays a central role in ensuring hygienic code generation.

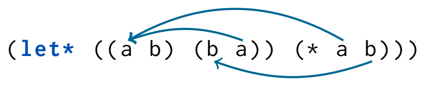
Hygiene in Macros. By contrast, achieving hygiene in the context of Lisp-style procedural macros is more subtle. For example, the following program with the `let*` macro:

```
(let* ((a b) (b a)) (* a b))
```

expands to a chain of `let` as below.

```
(let ((a b)) (let ((b a)) (* a b)))
```

Hence, the first program should have a non-trivial binding structure as displayed below:



```
(let* ((a b) (b a)) (* a b)))
```

```
(define-syntax min
  (er-transformer (lambda (exp rename compare)
    (let ((let/r (rename 'let))
          (if/r (rename 'if))
          (<=/r (rename '<=))
          (tmp1/r (rename 'tmp1))
          (tmp2/r (rename 'tmp2)))
      `(,let/r ((,tmp1/r (cadr ,exp)))
          (,let/r (,tmp2/r (caddr ,exp))
             (,if/r (,<=/r ,tmp1/r ,tmp2/r)
                ,tmp1/r ,tmp2/r)))))))
```

Figure 2. `min` macro wtih explicit-renaming

The issue here is that we do not know the binding structures in S-expressions with this kind of user-defined macros before macro expansion. Hence, we cannot apply α-renaming to achieve hygiene for macros.

Instead, hygienic macros often rely on the notion of a *syntactic environment*, which resolves raw symbols to unique names that refer to specific bindings. *Explicit renaming* by Clinger [3] offers one of the most primitive interfaces among hygienic macros using syntactic environment. Figure 2 shows a Scheme macro that implements `min` using explicit renaming. The macro expansion function takes three arguments. The first argument, `exp`, is the S-expression of the macro invocation. The other two are specific to explicit renaming: `rename` resolves a given symbol to its corresponding name in the syntactic environment of the macro definition, and `compare` compares two symbols under the syntactic environment of the macro definition. One can observe that this macro definition renames **`let`**, **`if`**, and <= so that they refer to the symbols in the macro-definition environment, and renames `tmp1` and `tmp2` to generate fresh variables.

In the rest of this paper, we focus on the renaming aspects of the explicit-renaming macro facility due to space limitations. Comparison can be regarded as equality up to renaming, and is therefore a less important topic than renaming itself.

2 Our Position

We are interested in formal semantics for programming languages with hygienic macros. Such a semantics should account for both hygiene and staging, while preserving the full expressive power of procedural macros.

However, existing formal models of hygienic macros [1, 9, 10, 15, 18] have primarily focused on formalizing how hygiene is ensured. They often do not define the semantics of the entire language. In particular, they do not describe semantics for staging — that is, how and when each expression is evaluated— which is essential to understanding the full behavior of procedural macros. See [4] for an overview of these approaches.

A natural idea to address this gap is to apply the insights from staged computation to hygienic macros. Indeed, several proposals [11, 19–21, 24] have explored this direction. These approaches leverage the well-established theory of staged computation to control evaluation stages and to reason about macros formally.

However, because these approaches emphasize static type safety, they impose strong restrictions inherited from staged computation. In particular, they cannot express macros with non-trivial binding structures such as **`let*`**. Consequently, they fall short of capturing the full expressive power of hygienic macros as used in practice.

Despite these challenges, we believe that insights from staged computation remain crucial to achieving a formal semantics for languages with hygienic macros. In this position paper, we propose an potential approach that explains hygienic macros from the perspective of staged computation using abstract machine semantics.

Specifically, we argue that environment machines provide a key to reconciling the tension between staged computation and hygienic macros. In a staged variant of the CEK machine, α-renaming is realized as deterministic renaming steps. This allows hygiene in staged computation to be described symbolically, making it easier to transfer these ideas to hygienic macros. We observe that syntactic environments in hygienic macros correspond to *renaming environments* in staged environment machines, and we leverage this correspondence to reinterpret the hygienic behavior of macros in terms of the renaming process in such machines.

In the rest of this paper, we first introduce a staged variant of the CEK machine and explain the notion of renaming environments (Section 3). We then sketch a potential design of an environment machine for a Lisp-like language with explicit renaming, where renamers are realized as a reflective interface to renaming environments (Section 4). Finally, we discuss the major remaining challenges toward achieving our goal in Section 5.

This perspective provides a symbolic account of hygiene that bridges staged computation and hygienic macros. By grounding both in staged environment machines, we expect it to enable a more precise comparison between the two, clarifying their similarities and differences.

3 Staged CEK Machine

In this section, we present a variant of the CEK machine for a simplified version of MetaML [22, 23]. This machine introduces explicit evaluation steps for renaming variables in code fragments. Such semantics provides a fine-grained operational account of hygienic code generation, which cannot be captured by standard α-renaming.

There are several variants of the CEK machine that support staging [12, 16]. Our staged CEK machine presented in Figure 3 is similar to both of these, but explicitly introduces the renaming environment as a dedicated object. Our

$$
\begin{array}{rlll}
\textbf{Variables} & x, y & \dots & \\
\textbf{Terms} & t^0 & ::= & x \mid \lambda x.\ t^0 \mid {t_1}^0\ {t_2}^0 \mid \langle t^1 \rangle \\
 & t^{n+} & ::= & x \mid \lambda x.\ t^{n+} \mid {t_1}^{n+}\ {t_2}^{n+} \mid \langle t^{n++} \rangle \mid {\sim} t^n \\
\textbf{Values} & v^0 & ::= & \textbf{clos}\{E, R, \lambda x.\ t^0\} \mid \langle v^1 \rangle \\
 & v^{n+} & ::= & t^n \\
\textbf{Value Envs.} & E & ::= & \epsilon \mid E, x \coloneqq v \\
\textbf{Renaming Envs.} & R & ::= & \epsilon \mid R, x_1 \coloneqq x_2 \\
\textbf{Conts.} & \kappa^0 & ::= & \square \mid \{E, R / \square\, t^0\} \triangleright \kappa^0 \mid \{v^0\, \square\} \triangleright \kappa^0 \\
 & & \mid & \{{\sim} \square\} \triangleright \kappa^1 \\
 & \kappa^{n+} & ::= & \{\lambda x.\, \square\} \triangleright \kappa^{n+} \\
 & & \mid & \{E, R / \square\, t^{n+}\} \triangleright \kappa^{n+} \mid \{v^{n+}\, \square\} \triangleright \kappa^{n+} \\
 & & \mid & \{\langle \square \rangle\} \triangleright \kappa^n \mid \{{\sim} \square\} \triangleright \kappa^{n++}
\end{array}
$$

$$(x, E, R, \kappa^0)^0_{ev} \rightarrow (E(x), \kappa^0)^0_{ret} \qquad \text{(ev-var-0)}$$

$$(\lambda x.\ t^0, E, R, \kappa^0)^0_{ev} \rightarrow (\textbf{clos}\{E, R, \lambda x.\ t^0\}, \kappa)^0_{ret} \qquad \text{(ev-}\lambda\text{-0)}$$

$$(v^0, \{\textbf{clos}\{E, R, \lambda x.\ {t_1}^0\}\, \square\} \triangleright \kappa^0)^0_{ret} \rightarrow (t, (E, x \coloneqq v^0), R, \kappa)^0_{ev} \qquad \text{(ret-app-r-0)}$$

$$(\langle v^1 \rangle, \{{\sim} \square\} \triangleright \kappa^1)^0_{ret} \rightarrow (v^1, \kappa^1)^1_{ret} \qquad \text{(ret-~-0)}$$

$$(x, E, R, \kappa^{n+})^{n+}_{ev} \rightarrow (R(x), \kappa^{n+})^{n+}_{ret} \qquad \text{(ev-var-n+)}$$

$$(\lambda x.\ t^{n+}, E, R, \kappa^{n+})^{n+}_{ev} \rightarrow (t^{n+}, E, (R, x \coloneqq y), \{\lambda y.\, \square\} \triangleright \kappa^{n+})^{n+}_{ev} \qquad \text{(ev-}\lambda\text{-n+)}$$

where y is fresh

$\dots$

$$
\begin{array}{ll}
R(x) = y & \text{if } R = (R_1, x \coloneqq y, R_2) \text{ and } x \notin \textbf{Dom}(R_2) \\
R(x) = x & \text{otherwise} \\
E(x) = v & \text{if } E = E_1, x \coloneqq v, E_2 \text{ and } x \notin \textbf{Dom}(E_2)
\end{array}
$$

Figure 3. Our Staged CEK Machine. n+ is written as a shorthand for $n + 1$. The full rules can be found in Appendix.

machine extends the original CEK machine [8] with the following points:

- Terms, values, continuations, and machine states are stratified by evaluation stages. 0 represents the runtime stage, and n+ represents a future stage inside code fragments.
- Rules for quotations ($\langle\rangle$) and splices (~) move up and down between stages.
- Rules for n+-stages are responsible for constructing code fragments (i.e., values for n+-stages).
- We introduce *renaming environments* R in addition to value environments E. Renaming environments map variable names to other variable names and are used for renaming variables to ensure hygienic code generation.

$(t^n, E, R, \kappa^n)^n_{ev}$ denotes an evaluation state at stage n, where t^n is the term being evaluated, E and R are the value and renaming environments, and κ^n is the continuation. $(v^n, \kappa^n)^n_{ret}$ denotes a returning state at stage n, where the value v^n is returned to the continuation κ^n.

Here we focus on the rules (ev-λ-n+) and (ev-var-n+). These rules use renaming environments to ensure hygiene. When constructing a code fragment for a lambda abstraction, (ev-λ-n+) generates a fresh variable y and adds the mapping $x \coloneqq y$ to the renaming environment. When it reaches a variable inside a code fragment, (ev-var-n+) renames it according to the renaming environment. For example, consider the evaluation of $(\lambda y.\ \langle \lambda x.\ x \,{\sim}(y) \rangle)\ \langle x \rangle$, where we expect the x in the lambda to be renamed to avoid a name conflict.

$$
\begin{array}{lll}
 & ((\lambda y.\ \langle \lambda x.\ x\,{\sim}(y) \rangle)\ \langle x \rangle, \epsilon, \epsilon, \square)^0_{ev} & \\
\rightarrow & \cdots & \\
\rightarrow & (\lambda x.\ x\,{\sim}(y), E, \epsilon, \{\langle \square \rangle\} \triangleright \square)^1_{ev} & \\
\rightarrow & (x\,{\sim}(y), E, R, \{\lambda x_0.\, \square\} \triangleright \{\langle \square \rangle\} \triangleright \square)^1_{ev} \quad (\text{where } R = (x \coloneqq x_0)) & (\bigstar 1) \\
\rightarrow & (x, E, R, \{E, R / \square\,{\sim}(y)\} \triangleright \{\lambda x_0.\, \square\} \triangleright \{\langle \square \rangle\} \triangleright \square)^1_{ev} & \\
\rightarrow & (x_0, \{E, R / \square\,{\sim}(y)\} \triangleright \{\lambda x_0.\, \square\} \triangleright \{\langle \square \rangle\} \triangleright \square)^1_{ret} & (\bigstar 2) \\
\rightarrow & \cdots & \\
\rightarrow & (\langle \lambda x_0.\ x\, x_0 \rangle, \square)^0_{ret} &
\end{array}
$$

In the final state, we observe that x in the lambda is renamed to x_0, avoiding name conflicts as expected. The actual renaming operations occur at (★1) and (★2), where R holds the mapping from x to x_0. In this way, our staged CEK machine offers a finer-grained perspective on hygienic code generation than substitution-based semantics.

Note that it is a common technique in partial evaluation [14] and multi-stage programming [2] to generate fresh names for binding forms in order to ensure hygiene. Compared to these existing approaches, our staged CEK machine introduces renaming environments, which provide explicit information about the renaming process at each step of evaluation. This mechanism allows us to establish a clearer correspondence between multi-stage programming and hygienic macros, as we will discuss in the next section.

4 Abstract Machines with Hygienic Macros

Looking at the staged CEK machine in the previous section, we observe that the notion of renaming environments is quite similar to that of syntactic environments in hygienic macros. Building on this intuition, we sketch the design of an environment machine for programming languages with an explicit-renaming macro facility. We begin by defining the syntax constructs.

$$
\begin{array}{rlll}
\textbf{Symbols} & x, y & \dots & \\
\textbf{S-Exps.} & s & ::= & x \mid () \mid \texttt{\#t} \mid \texttt{\#f} \mid (s_1\ .\ s_2)
\end{array}
$$

Here we define S-expressions instead of terms. Syntax constructs such as lambda abstractions and applications are encoded as S-expressions. Unlike in the staged CEK machine, S-expressions are not stratified, because their structure is unknown at this point.

$$
\begin{array}{rlll}
\textbf{Values} & v & ::= & x \mid () \mid \texttt{\#t} \mid \texttt{\#f} \mid (v_1\ .\ v_2) \\
 & & \mid & s \mid \#\textbf{prim}(x) \mid \#\textbf{clos}(E, (\lambda\ (\overrightarrow{x})\ s)) \\
 & & \mid & \#\textbf{ren}(R) \mid \#\textbf{cmp}(R) \\
\textbf{Value Envs.} & E & ::= & \epsilon \mid E, x \coloneqq v \\
\textbf{Name Envs.} & R & ::= & \epsilon \mid R, x_1 \coloneqq x_2
\end{array}
$$

We define values by extending S-expressions. In addition to closures $\#\textbf{clos}(E, R, (\lambda\ (\overrightarrow{x})\ s))$ and primitive functions $\#\textbf{prim}(x)$, we introduce renamers $\#\textbf{ren}(R)$ and comparators

#**cmp**(R). These serve as interfaces to renaming environments, and are passed to macro-expanding functions. One can assume that the states and continuations are similar to those in staged CEK machines. For simplicity, we consider only stages 0 and 1. At stage 1, the evaluation steps rename the bound variables of lambda abstractions while expanding macros.

The renaming steps are similar to those in the staged CEK machine. When processing lambda abstractions, (ev-λ-1) ensures that λ is not a bound variable by checking the renaming environment.

$$((\lambda\ (\overrightarrow{x})\ s), E, R, \kappa)^1_{ev} \rightarrow (s, E, (R, \overrightarrow{x} \coloneqq \overrightarrow{y}), \{(\lambda\ (\overrightarrow{y})\ \square)\} \triangleright \kappa)^1_{ev} \qquad \text{(ev-}\lambda\text{-1)}$$
$$\text{if } R(\lambda) = \lambda^0 \text{ where } \overrightarrow{y} \text{ are fresh}$$
$$(x, E, R, \kappa)^1_{ev} \rightarrow (R(x), \kappa)^1_{ret} \qquad \text{(ev-var-1)}$$

We consider a simplified model of an explicit-renaming macro system, where a macro call is written in the form $(\mathtt{macro}\, x\ .\ s)$. (ev-macro-1) evaluates x at stage 0 to obtain the macro-expanding function. (ret-macro-1) then applies the function to the given S-expression, generating renamers and comparators from the renaming environment of the function. In this way, we allow macro-expanding functions to access the renaming environments at their definition sites. After the macro transformer returns an expanded S-expression, (ret-embed-1) continues processing it as a stage-1 S-expression.

$$((\mathtt{macro}\, x\ .\ s), E, R, \kappa)^1_{ev} \rightarrow (x, E, R, \{E, R/(\mathtt{macro}\, \square\ .\ s)\} \triangleright \kappa)^0_{ev} \qquad \text{(ev-macro-1)}$$
$$\text{if } R(\mathtt{macro}) = \mathtt{macro}^0$$

$$(\#\mathbf{clos}(E_1, R_1, (\lambda\ (x^{\mathsf{exp}}\, x^{\mathsf{ren}}\, x^{\mathsf{cmp}})\ s_1)), \{E_2, R_2/(\mathtt{macro}\, \square\ .\ s_2)\} \triangleright \kappa)^0_{ret}$$
$$\rightarrow (s_1, E', R, \{E_2, R_2/\mathbf{embed}\, \square\} \triangleright \kappa)^0_{ev} \qquad \text{(ret-macro-1)}$$
$$\text{where } E' = (E_1, x^{\mathsf{exp}} \coloneqq s_2, x^{\mathsf{ren}} \coloneqq \#\mathbf{ren}(R_1), x^{\mathsf{cmp}} \coloneqq \#\mathbf{cmp}(R_{13}))$$

$$(s, \{E, R/\mathbf{embed}\, \square\} \triangleright \kappa)^0_{ret} \rightarrow (s, E, R, \kappa)^1_{ev} \qquad \text{(ret-embed-1)}$$

If a symbol x is passed to a renamer with R, it returns either (a) $R(x)$ if x is in the domain of R (ret-ren-n+-a), or (b) a fresh variable otherwise [1] (ret-ren-n+-b). Comparators work in a similar way.

$$(x, \{E, R_1/(\#\mathbf{ren}(R_2)\ \square)\} \triangleright \kappa)^0_{ret} \rightarrow (R_2(x), \kappa)^0_{ret} \quad \text{if } x \in \mathbf{Dom}(R_2) \qquad \text{(ret-ren-n+-a)}$$
$$(x, \{E, R_1/(\#\mathbf{ren}(R_2)\ \square)\} \triangleright \kappa)^0_{ret} \rightarrow (y, \kappa)^0_{ret} \qquad \text{(ret-ren-n+-b)}$$
$$\text{if } x \notin \mathbf{Dom}(R_2) \text{ where } y \text{ is fresh}$$

That is our preliminary idea of how explicit renaming can be formalized using staged environment machines. The core idea is to (a) use S-expressions instead of structured terms, and (b) provide renamers and comparators as a reflective interface to the renaming environments.

[1]More precisely, a renamer should always return the same name for the same input. Hence, it needs to remember the fresh names it has generated in some way.

5 Discussion and Future Directions

In the previous section, we outlined the basic idea of providing formal models for hygienic macro expansion. However, several challenges remain in defining a full semantics for a programming language with hygienic macros.

The most significant challenge is defining multi-level semantics. Scheme programs with multi-level macros are common — for example, a macro whose definition uses other macros, or a macro that generates the definition of another macro. Thus, handling such cases is crucial for practical applications.

The core difficulty lies in deciding how to treat macros at levels deeper than level 1. Should these macros be expanded, or should we perform renaming without expansion? Expanding them raises the issue of staging discipline, because a macro at a deeper level may be invoked before its definition becomes available. On the other hand, skipping macro expansion requires us to ensure that renaming on unstructured S-expressions can be performed safely.

Possible approaches include employing more sophisticated staging models. For example, MacoCaml [24] adopts a staging model in which each module is compiled separately, and compiled modules can be used by other modules both at run time and at compile time. This model can represent macro-using-macros, like the earlier example, while keeping the language essentially two-level. Such an approach could also be well suited to Lisp-style hygienic macros.

If we instead wish to maintain multi-level language, we may want to elaborate models for renaming S-expressions. This includes the approach by Dybvig et al. [7], which annotates S-expressions with syntactic information, and the approach by Adams [1], which elaborates symbols into pairs of binder and reference parts. We would also need to infer level information from expressions, where multi-level binding-time analysis [13, 14] appears to be a promising approach. However, note that such analysis would need to be performed dynamically during macro expansion, because we cannot analyze unstructured S-expressions before macro expansion.

In any case, addressing these questions is a prerequisite for developing a full semantics of hygienic macros that reflects their practical usage. In this sense, this position paper represents only a starting point. We expect that further research will enable more fine-grained comparisons between multi-stage programming and hygienic macros, which we expect will benefit both areas.

Acknowledgements

We are grateful to Atsushi Igarashi for many fruitful discussions and to the anonymous reviewers for their constructive and helpful feedback. This work is supported by JSPS KAKENHI Grant Number JP24KJ1363.

A Full Definition of Staged CEK Machine

$$
\begin{array}{rlcl}
\textbf{Variables} & x, y & \dots & \\
\textbf{Terms} & t^0 & ::= & x \mid \lambda x.\ t^0 \mid {t_1}^0\ {t_2}^0 \mid \langle t^1 \rangle \\
 & t^{n+} & ::= & x \mid \lambda x.\ t^{n+} \mid {t_1}^{n+}\ {t_2}^{n+} \mid \langle t^{n++} \rangle \mid \mathord{\sim} t^n \\
\textbf{Values} & v^0 & ::= & \mathbf{clos}\{E, R, \lambda x.\ t^0\} \mid \langle v^1 \rangle \\
 & v^{n+} & ::= & t^n \\
\textbf{Value Envs.} & E & ::= & \epsilon \mid E, x \coloneqq v \\
\textbf{Renaming Envs.} & R & ::= & \epsilon \mid R, x_1 \coloneqq x_2 \\
\textbf{Conts.} & \kappa^0 & ::= & \square \mid \{E, R/ \square\, t^0\} \triangleright \kappa^0 \mid \{v^0\, \square\} \triangleright \kappa^0 \\
 & & \mid & \{\mathord{\sim} \square\} \triangleright \kappa^1 \\
 & \kappa^{n+} & ::= & \{\lambda x.\, \square\} \triangleright \kappa^{n+} \\
 & & \mid & \{E, R/ \square\, t^{n+}\} \triangleright \kappa^{n+} \mid \{v^{n+}\, \square\} \triangleright \kappa^{n+} \\
 & & \mid & \{\langle \square \rangle\} \triangleright \kappa^n \mid \{\mathord{\sim} \square\} \triangleright \kappa^{n++}
\end{array}
$$

$$
\begin{aligned}
(x, E, R, \kappa^0)^0_{ev} &\longrightarrow (E(x), \kappa^0)^0_{ret} && \text{(ev-var-0)} \\
(\lambda x.\ t^0, E, R, \kappa^0)^0_{ev} &\longrightarrow (\mathbf{clos}\{E, R, \lambda x.\ t^0\}, \kappa)^0_{ret} && \text{(ev-}\lambda\text{-0)} \\
({t_1}^0\ {t_2}^0, E, R, \kappa^0)^0_{ev} &\longrightarrow ({t_1}^0, E, R, \{E, R/ \square\, t_2\} \triangleright \kappa)^0_{ev} && \text{(ev-app-0)} \\
(\langle t^1 \rangle, E, R, \kappa^0)^0_{ev} &\longrightarrow (t^1, E, R, \{\langle \square \rangle\} \triangleright \kappa^0)^1_{ev} && \text{(ev-}\langle\rangle\text{-0)}
\end{aligned}
$$

$$
\begin{aligned}
(v^0, \{E, R/ \square\, t^0\} \triangleright \kappa^0)^0_{ret} &\longrightarrow (t^0, E, R, \{v\, \square\} \triangleright \kappa^0)^0_{ev} && \text{(ret-app-l-0)} \\
(v^0, \{\mathbf{clos}\{E, R, \lambda x.\ {t_1}^0\}\, \square\} \triangleright \kappa^0)^0_{ret} &\longrightarrow (t, (E, x \coloneqq v^0), R, \kappa)^0_{ev} && \text{(ret-app-r-0)} \\
(\langle v^1 \rangle, \{\mathord{\sim} \square\} \triangleright \kappa^1)^0_{ret} &\longrightarrow (v^1, \kappa^1)^1_{ret} && \text{(ret-}\mathord{\sim}\text{-0)}
\end{aligned}
$$

$$
\begin{aligned}
(x, E, R, \kappa^{n+})^{n+}_{ev} &\longrightarrow (R(x), \kappa^{n+})^{n+}_{ret} && \text{(ev-var-n+)} \\
(\lambda x.\ t^{n+}, E, R, \kappa^{n+})^{n+}_{ev} &\longrightarrow (t^{n+}, E, (R, x \coloneqq y), \{\lambda y.\, \square\} \triangleright \kappa^{n+})^{n+}_{ev} && \text{(ev-}\lambda\text{-n+)} \\
&\qquad \text{where } y \text{ is fresh} \\
({t_1}^{n+}\ {t_2}^{n+}, E, R, \kappa^{n+})^{n+}_{ev} &\longrightarrow ({t_1}^{n+}, E, R, \{E, R/ \square\, {t_2}^{n+}\} \triangleright \kappa^{n+})^{n+}_{ev} && \text{(ev-app-n+)} \\
(\langle t^{n++} \rangle, E, R, \kappa^{n+})^{n+}_{ev} &\longrightarrow (t^{n++}, E, R, \{\langle \square \rangle\} \triangleright \kappa^{n+})^{n++}_{ev} && \text{(ev-}\langle\rangle\text{-n+)} \\
(\mathord{\sim} t^n, E, R, \kappa^{n+})^{n+}_{ev} &\longrightarrow (t^n, E, R, \{\mathord{\sim} \square\} \triangleright \kappa^{n+})^{n}_{ev} && \text{(ev-}\mathord{\sim}\text{-n+)}
\end{aligned}
$$

$$
\begin{aligned}
(v^{n+}, \{\lambda x.\, \square\} \triangleright \kappa^{n+})^{n+}_{ret} &\longrightarrow (\lambda x.\ v^{n+}, \kappa^{n+})^{n+}_{ret} && \text{(ret-}\lambda\text{-n+)} \\
(v^{n+}, \{E, R/ \square\, t^{n+}\} \triangleright \kappa^{n+})^{n+}_{ret} &\longrightarrow (t^{n+}, E, R, \{v^{n+}\, \square\} \triangleright \kappa^{n+})^{n+}_{ev} && \text{(ret-app-r-n+)} \\
({v_1}^{n+}, \{{v_2}^{n+}\, \square\} \triangleright \kappa^{n+})^{n+}_{ret} &\longrightarrow ({v_1}^{n+}\ {v_2}^{n+}, \kappa^{n+})^{n+}_{ret} && \text{(ret-app-l-n+)} \\
(v^{n+}, \{\langle \square \rangle\} \triangleright \kappa^{n})^{n+}_{ret} &\longrightarrow (\langle v^{n+} \rangle, \kappa^{n})^{n}_{ret} && \text{(ret-}\langle\rangle\text{-n+)} \\
(v^{n+}, \{\mathord{\sim} \square\} \triangleright \kappa^{n++})^{n+}_{ret} &\longrightarrow (\langle v^{n+} \rangle, \kappa^{n++})^{n++}_{ret} && \text{(ret-}\mathord{\sim}\text{-n+)}
\end{aligned}
$$

$$
\begin{aligned}
R(x) &= y && \text{if } R = (R_1, x \coloneqq y, R_2) \text{ and } x \notin \mathbf{Dom}(R_2) \\
R(x) &= x && \text{otherwise} \\
E(x) &= v && \text{if } E = E_1, x \coloneqq v, E_2 \text{ and } x \notin \mathbf{Dom}(E_2)
\end{aligned}
$$

B Full Definition of Abstract Machines with Hygienic Macros

$$
\begin{aligned}
(\mathtt{\#t}, E, R, \kappa)^0_{ev} &\longrightarrow (\mathtt{\#t}, \kappa)^0_{ret} \\
(\mathtt{\#f}, E, R, \kappa)^0_{ev} &\longrightarrow (\mathtt{\#f}, \kappa)^0_{ret} \\
((), E, R, \kappa)^0_{ev} &\longrightarrow ((), \kappa)^0_{ret} \\
(x, E, R, \kappa)^0_{ev} &\longrightarrow (\#\mathbf{prim}(x), \kappa)^0_{ret} \text{ if } x \text{ is primitive} \\
(x, E, R, \kappa)^0_{ev} &\longrightarrow (E(x), \kappa)^0_{ret} \\
((\lambda\ (\overrightarrow{x})\ s), E, R, \kappa)^0_{ev} &\longrightarrow (\#\mathbf{clos}(E, R, (\lambda\ (\overrightarrow{x})\ s)), \kappa)^0_{ret} \quad \text{if } R(\lambda) = \lambda^0 \\
((\mathtt{if}\ s_1\ s_2\ s_3), E, R, \kappa)^0_{ev} &\longrightarrow (s_1, E, R, \{E/(\mathtt{if}\ \square\ s_2\ s_3)\} \triangleright \kappa)^0_{ev} \quad \text{if } R(\mathtt{if}) = \mathtt{if}^0 \\
((\mathtt{quote}\ s), E, R, \kappa)^0_{ev} &\longrightarrow (s, \kappa)^0_{ret} \quad \text{if } R(\mathtt{quote}) = \mathtt{quote}^0 \\
((\mathtt{qquote}\ s), E, R, \kappa)^0_{ev} &\longrightarrow (s, E, R, \{(\mathtt{qquote}\ \square)\} \triangleright \kappa)^1_{ev} \quad \text{if } R(\mathtt{qquote}) = \mathtt{qquote}^0 \\
((s_1\ \overrightarrow{s_2}), E, R, \kappa)^0_{ev} &\longrightarrow (s_1, E, R, \{E/(\square\ \overrightarrow{s_2})\} \triangleright \kappa)^0_{ev} \quad \text{otherwise}
\end{aligned}
$$

$$
(v_1, \{E, R/(\overrightarrow{v_2}\ \square\ s_1\ \overrightarrow{s_2})\} \triangleright \kappa)^0_{ret} \longrightarrow (s_1, E, R, \{E, R/(\overrightarrow{v_2}\ v_1\ \square\ \overrightarrow{s_2})\} \triangleright \kappa)^0_{ev}
$$

$$
\begin{aligned}
(\#\mathbf{clos}(E, R, (\lambda\ ()\ s)), \{E, R/(\square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (s, E, R, \kappa)^0_{ev} \\
(v_1, \{E, R/(\#\mathbf{clos}(E, (\lambda\ (\overrightarrow{x})\ s))\ \overrightarrow{v_2}\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (s, (E, \overrightarrow{x} \coloneqq (\overrightarrow{v_2}, v_1)), R, \kappa)^0_{ev} \\
(v_1, \{E, R/(\#\mathbf{prim}(x)\ \overrightarrow{v_2}\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (\delta(x, (\overrightarrow{v_2}, v_1)), \kappa)^0_{ret} \\
(\mathtt{\#t}, \{E, R/(\mathtt{if}\ \square\ s_1\ s_2)\} \triangleright \kappa)^0_{ret} &\longrightarrow (s_1, E, R, \kappa)^0_{ev} \\
(\mathtt{\#f}, \{E, R/(\mathtt{if}\ \square\ s_1\ s_2)\} \triangleright \kappa)^0_{ret} &\longrightarrow (s_2, E, R, \kappa)^0_{ev} \\
(x, \{E, R_1/(\#\mathbf{ren}(R_2)\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (R_2(x), \kappa)^0_{ret} \\
&\qquad \text{if } x \in \mathbf{Dom}(R_2) \\
(x, \{E, R_1/(\#\mathbf{ren}(R_2)\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (y, \kappa)^0_{ret} \\
&\qquad \text{if } x \notin \mathbf{Dom}(R_2) \text{ where } y \text{ is fresh} \\
(x, \{E, R_1/(\#\mathbf{cmp}(R_2)\ y\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (\mathtt{\#t}, \kappa)^0_{ret} \text{ if } R_2(x) = R_2(y) \\
(x, \{E, R_1/(\#\mathbf{cmp}(R_2)\ y\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (\mathtt{\#f}, \kappa)^0_{ret} \text{ otherwise} \\
(s, \{E, R/\mathbf{embed}\ \square\} \triangleright \kappa)^0_{ret} &\longrightarrow (s, E, R, \kappa)^1_{ev} \\
(s, \{(\mathtt{qquote}\ \square)\} \triangleright \kappa)^1_{ret} &\longrightarrow ((\mathtt{qquote}\ s), \kappa)^0_{ret} \\
((\mathtt{qquote}\ s), \{(\mathtt{unquote}\ \square)\} \triangleright \kappa)^0_{ret} &\longrightarrow (s, \kappa)^1_{ret}
\end{aligned}
$$

$$
\begin{aligned}
&(\#\mathbf{clos}(E_1, R_1, (\lambda\ (x^{\mathtt{exp}}\ x^{\mathtt{ren}}\ x^{\mathtt{cmp}})\ s_1)), \{E_2, R_2/(\mathtt{macro}\ \square\ .\ s_2)\} \triangleright \kappa)^0_{ret} \\
&\longrightarrow (s_1, E', R, \{E_2, R_2/\mathbf{embed}\ \square\} \triangleright \kappa)^0_{ev} \\
&\text{where } E' = (E_1, x^{\mathtt{exp}} \coloneqq s_2, x^{\mathtt{ren}} \coloneqq \#\mathbf{ren}(R_1), x^{\mathtt{cmp}} \coloneqq \#\mathbf{cmp}(R_{13}))
\end{aligned}
$$

$$
\begin{aligned}
(\mathtt{\#t}, E, R, \kappa)^1_{ev} &\longrightarrow (\mathtt{\#t}, \kappa)^1_{ret} \\
(\mathtt{\#f}, E, R, \kappa)^1_{ev} &\longrightarrow (\mathtt{\#f}, \kappa)^1_{ret} \\
((), E, R, \kappa)^1_{ev} &\longrightarrow ((), \kappa)^1_{ret} \\
(x, E, R, \kappa)^1_{ev} &\longrightarrow (R(x), \kappa)^1_{ret} \\
((\lambda\ (\overrightarrow{x})\ s), E, R, \kappa)^1_{ev} &\longrightarrow (s, E, (R, \overrightarrow{x} \coloneqq \overrightarrow{y}), \{(\lambda\ (\overrightarrow{y})\ \square)\} \triangleright \kappa)^1_{ev} \\
&\qquad \text{where } \overrightarrow{y} \text{ are fresh} \\
((\mathtt{unquote}\ s), E, R, \kappa)^1_{ev} &\longrightarrow (s, E, R, \{(\mathtt{unquote}\ \square)\} \triangleright \kappa)^0_{ev} \\
&\qquad \text{if } R(\mathtt{unquote}) = \mathtt{unquote}^0 \\
((\mathtt{macro}\ x\ .\ s), E, R, \kappa)^1_{ev} &\longrightarrow (x, E, R, \{E, R/(\mathtt{macro}\ \square\ .\ s)\} \triangleright \kappa)^0_{ev} \\
&\qquad \text{if } R(\mathtt{macro}) = \mathtt{macro}^0 \\
((s_1\ \overrightarrow{s_2}), E, R, \kappa)^{n+}_{ev} &\longrightarrow (s_1, E, R, \{E/(\square\ \overrightarrow{s_2})\} \triangleright \kappa)^{n+}_{ev} \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
R_{\text{init}} &= \lambda \coloneqq \lambda^0, \mathtt{if} \coloneqq \mathtt{if}^0, \dots \\
R(x) &= y && \text{if } R = (R_1, x \coloneqq y, R_2) \text{ and } x \notin \mathbf{Dom}(R_2) \\
R(x) &= x && \text{otherwise} \\
E(x) &= v && \text{if } E = E_1, x \coloneqq v, E_2 \text{ and } x \notin \mathbf{Dom}(E_2)
\end{aligned}
$$

References

[1] Michael D. Adams. 2015. Towards the Essence of Hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’15)*. Association for Computing Machinery, New York, NY, USA, 457–469. doi:10.1145/2676726.2677013

[2] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*. Springer, Berlin, Heidelberg, 57–76. doi:10.1007/978-3-540-39815-8_4

[3] William Clinger. 1991. Hygienic Macros through Explicit Renaming. *SIGPLAN Lisp Pointers* IV, 4 (Oct. 1991), 25–28. doi:10.1145/1317265.1317269

[4] William D. Clinger and Mitchell Wand. 2020. Hygienic Macro Technology. *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 80:1–80:110. doi:10.1145/3386330

[5] Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1 (March 2017), 1:1–1:45. doi:10.1145/3011069

[6] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. doi:10.1145/382780.382785

[7] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic Abstraction in Scheme. *LISP and Symbolic Computation* 5, 4 (Dec. 1993), 295–326. doi:10.1007/BF01806308

[8] Matthias Felleisen and Daniel P Friedman. 1986. Control operators, the SECD-machine, and the λ-calculus. *Formal Description of Programming Concepts III* (1986).

[9] Matthew Flatt. 2016. Binding as Sets of Scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 705–717. doi:10.1145/2837614.2837620

[10] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros That Work Together: Compile-time Bindings, Partial Expansion, and Definition Contexts. *Journal of Functional Programming* 22, 2 (March 2012), 181–216. doi:10.1017/S0956796812000093

[11] Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 74–85. doi:10.1145/507635.507646

[12] Rui Ge and Ronald Garcia. 2019. Refining Semantics for Multi-Stage Programming. *Journal of Computer Languages* 51 (April 2019), 222–240. doi:10.1016/j.jvlc.2018.10.006

[13] Robert Glück and Jesper Jørgensen. 1996. Fast Binding-Time Analysis for Multi-Level Specialization. In *Perspectives of System Informatics*. Springer, Berlin, Heidelberg, 261–272. doi:10.1007/3-540-62064-8_22

[14] Robert Glück and Jesper Jørgensen. 1997. An Automatic Program Generator for Multi-Level Specialization. *LISP and Symbolic Computation* 10, 2 (July 1997), 113–158. doi:10.1023/A:1007763000430

[15] David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Springer, Berlin, Heidelberg, 48–62. doi:10.1007/978-3-540-78739-6_4

[16] Atsushi Igarashi and Masashi Iwaki. 2007. Deriving Compilers and Virtual Machines for a Multi-level Language. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 206–221. doi:10.1007/978-3-540-76637-7_14

[17] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102. doi:10.1007/978-3-319-07151-0_6

[18] Paul Stansifer and Mitchell Wand. 2014. Romeo: A System for More Flexible Binding-Safe Programming. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/2628136.2628162

[19] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-Stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. doi:10.1145/3278122.3278139

[20] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-Stage Programming with Generative and Analytical Macros. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2021)*. Association for Computing Machinery, New York, NY, USA, 110–122. doi:10.1145/3486609.3487203

[21] Walid Taha and Patricia Johann. 2003. Staged Notational Definitions. In *Generative Programming and Component Engineering*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer, Berlin, Heidelberg, 97–116. doi:10.1007/978-3-540-39815-8_6

[22] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 26–37. doi:10.1145/604131.604134

[23] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 203–217. doi:10.1145/258993.259019

[24] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 209:604–209:648. doi:10.1145/3607851

Received 2025-07-19; accepted 2025-08-14