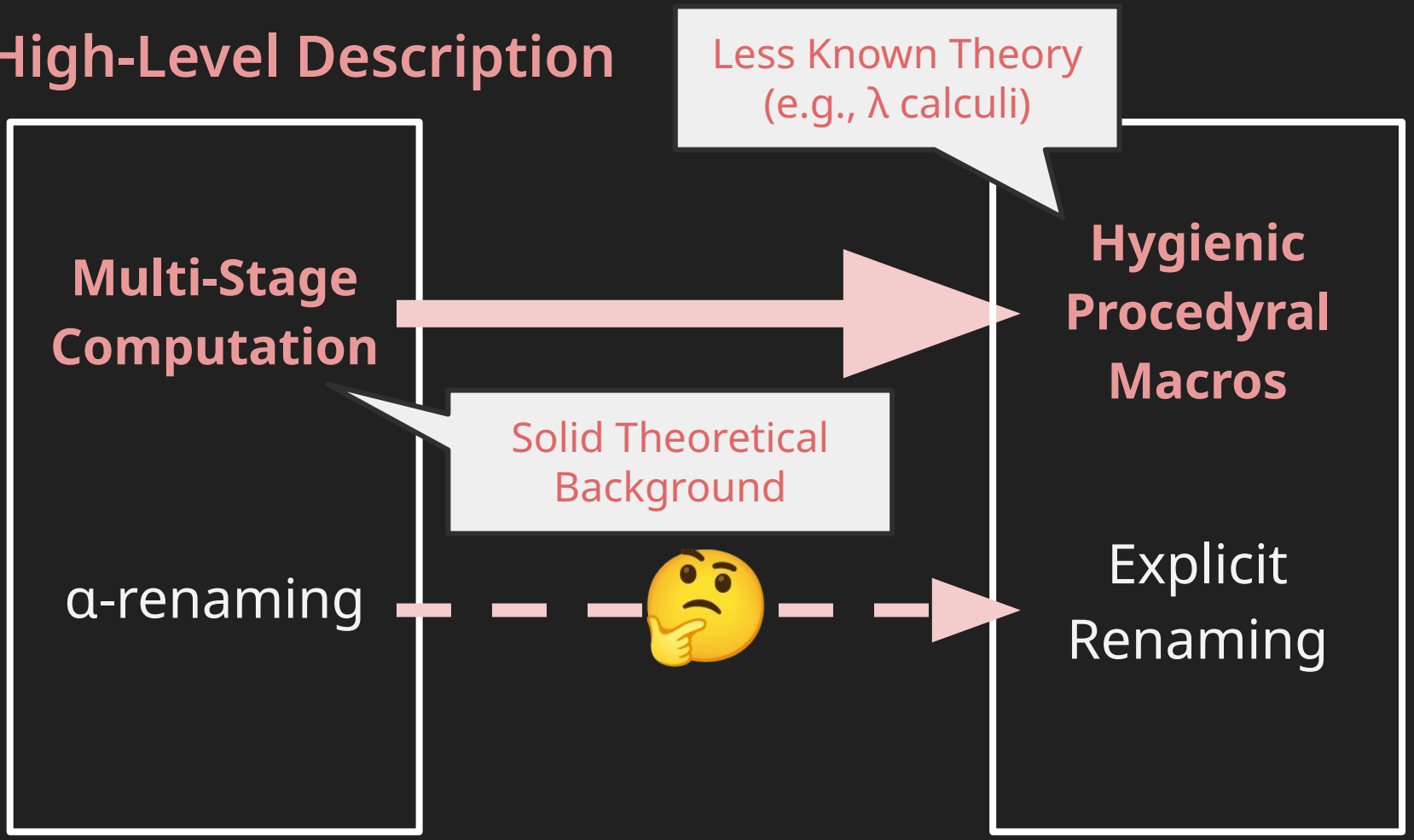
# Hygienic Macros via Staged Environment Machines

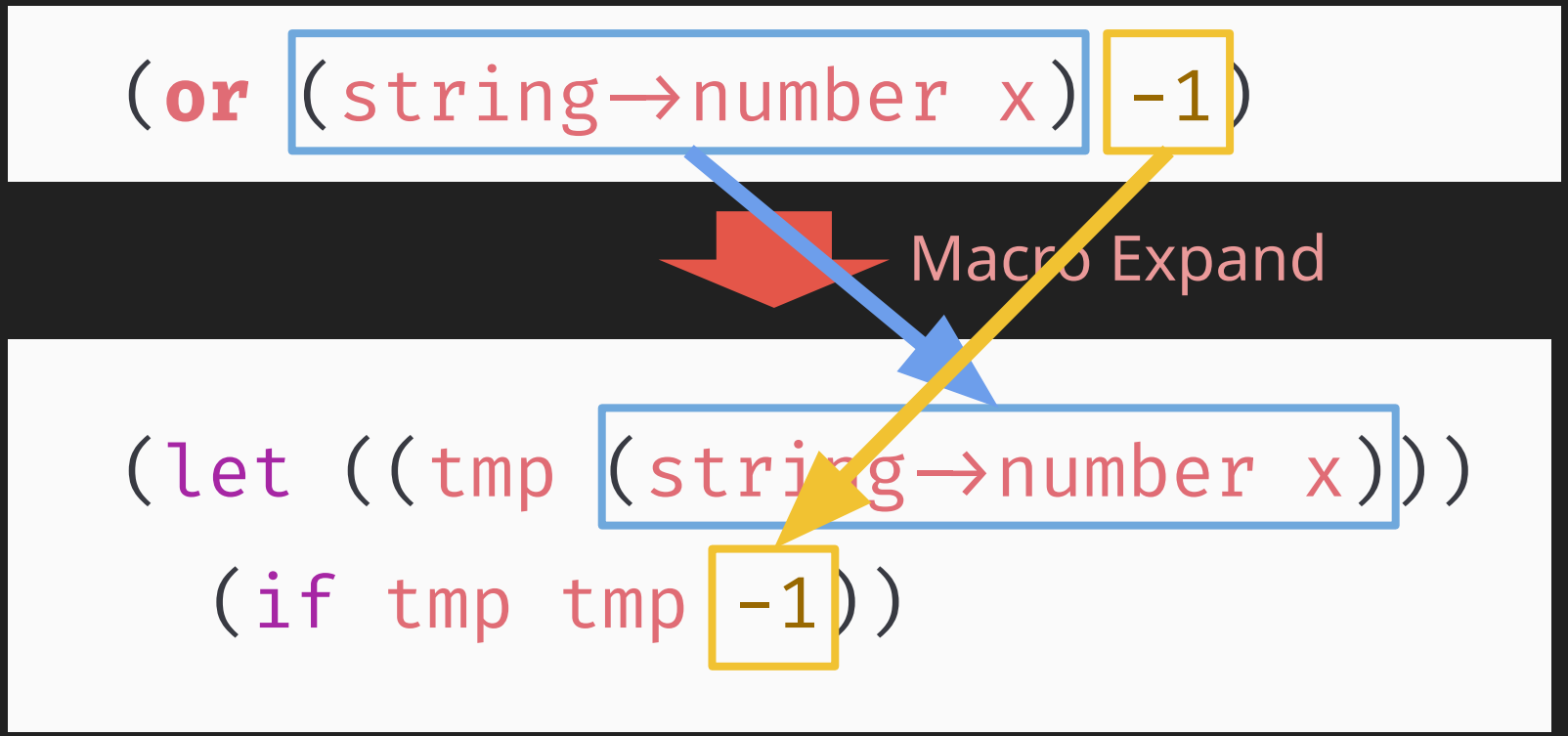

Yuito Murase (Kyoto University)

2025-Oct-16 @ Scheme Workshop 2025, Singapore

# High-Level Description

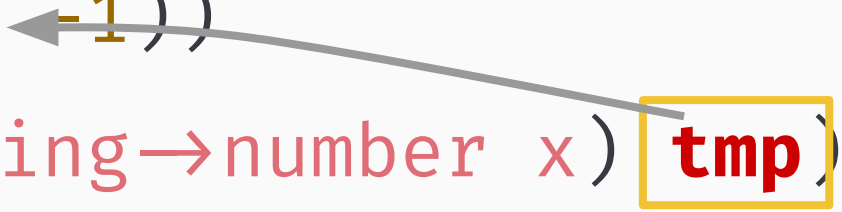


# Macros in Lisp



# Name Conflict in Macros

```
(let ((tmp -1))
  (or (string→number x) tmp))
```



# Name Conflict in Macros

```
(let ((tmp -1))
  (or (string→number x) tmp))
```

Macro Expand

```
(let ((tmp -1))
  (let ((tmp (string→number x)))
    (if tmp tmp tmp)))
```

Variable name conflict

# *Hygienic* Macros [Kohlbecker 1986]

```
(let ((tmp 1))
  (or (string→number x) tmp))
```

Macro Expand

```
(let_0 ((tmp_1 1))
  (let_0 ((tmp_2 (string→number_3 x)))
    (if_4 tmp_2 tmp_2 tmp_1)))
```

# Explicit Renaming Macros [Clinger 1991]

```
(define-syntax or (er-macro-transformer
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if) ,(rename 'tmp)
           ,(rename 'tmp)
           ,b)))))))
```

# Hygienic Proc. Macros w/ *Explicit Renaming*

```
(define-syntax or
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if) ,(rename 'tmp)
           ,(rename 'tmp)
           ,b))))))
```

A ***renamer*** resolves symbols to their unique names according to ***the syntactic environment*** of the macro definition

# Hygienic Proc. Macros w/ *Explicit Renaming*

```
(define-syntax or
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if) ,(rename 'tmp)
           ,(rename 'tmp)
           ,b))))))
```

A ***renamer*** resolves symbols to their unique names according to ***the syntactic environment*** of the macro definition

Resolved to `let_0`, referring to `let` in the standard lib

# Hygienic Proc. Macros w/ *Explicit Renaming*

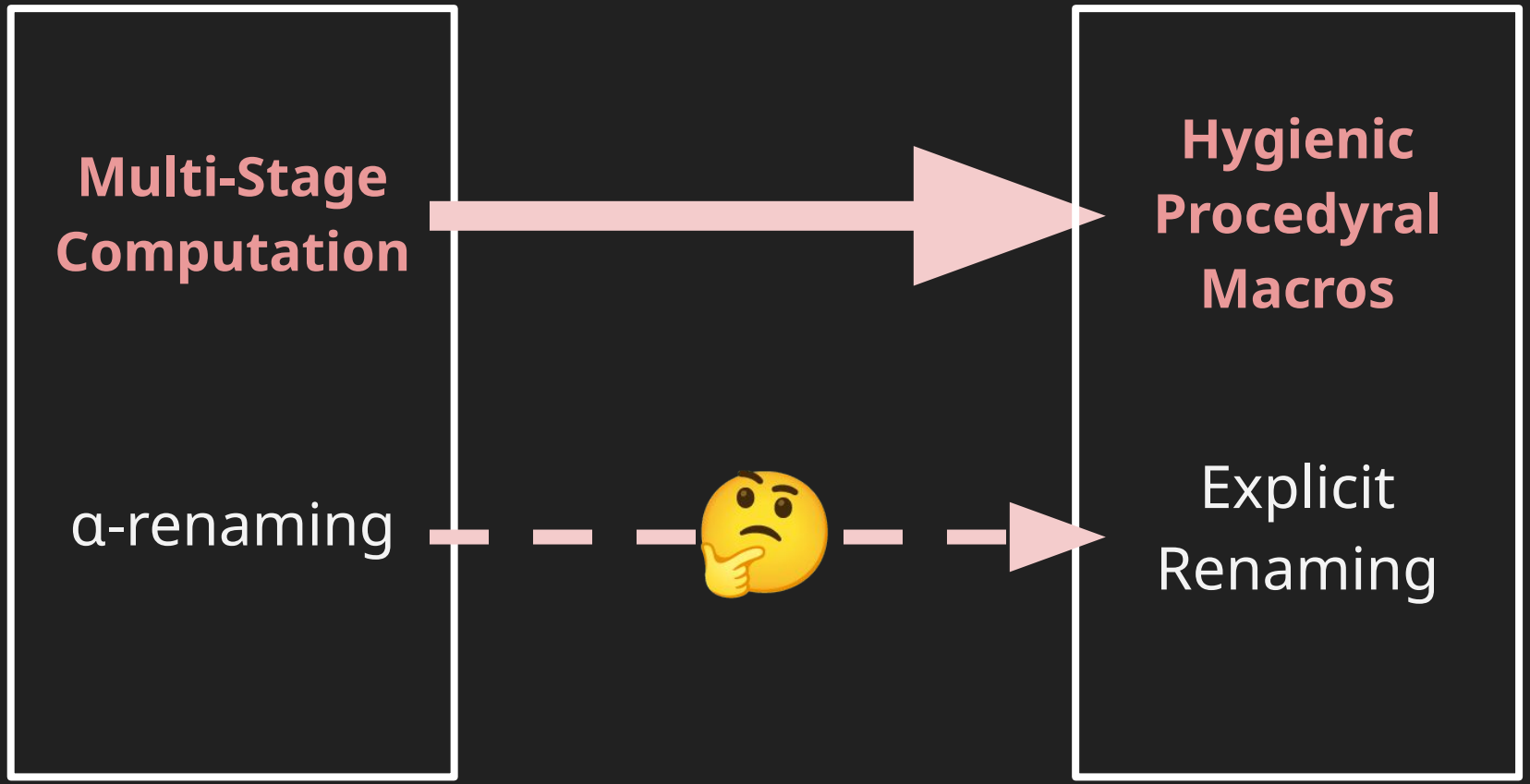
```
(define-syntax or
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if) ,(rena     'tmp)
           ,(rename 'tmp)
           ,b))))))
```

A ***renamer*** resolves symbols to their unique names according to ***the syntactic environment*** of the macro definition

Resolved to `let_0`, referring to `let` in the standard lib

Resolved to `tmp_1`, that doesn't conflict with other `tmp`

# High-Level Description



# Multi-Stage Programming

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in
   if tmp then tmp else .~b >.
```

# Multi-Stage Programming *and α-renaming*

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in
   if tmp then tmp else .~b >.
```

# Multi-Stage Programming *and α-renaming*

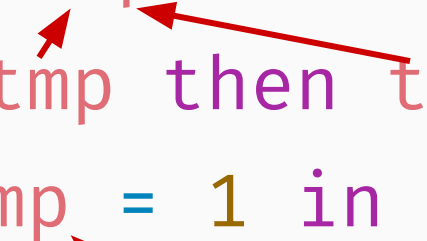
```
(* MetaOCaml style *)
let genOr a b =
.< let abc = .~a in
   if abc then abc else .~b >.
```

# Multi-Stage Programming *and α-renaming*

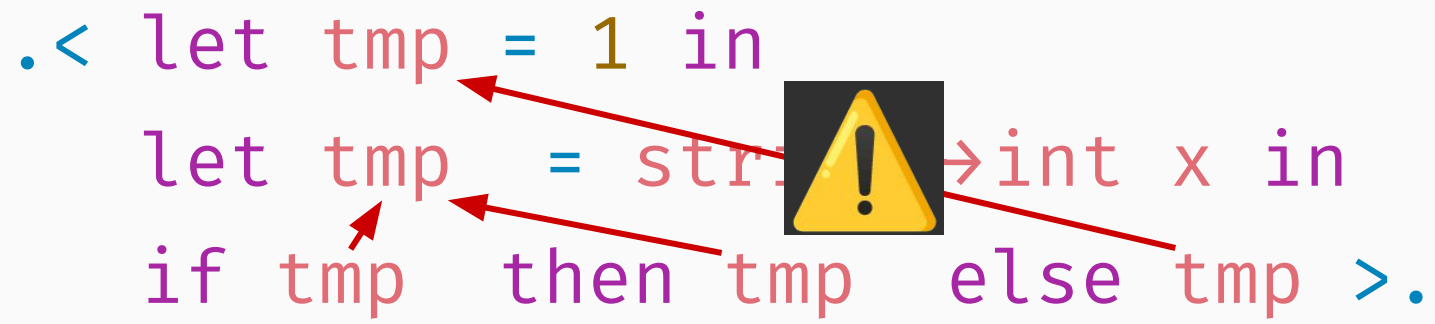
```
(* MetaOCaml style *)
let genOr a b =
.< let ppp = .~a in
   if ppp then ppp else .~b >.
```

# Evaluating a Staged Program

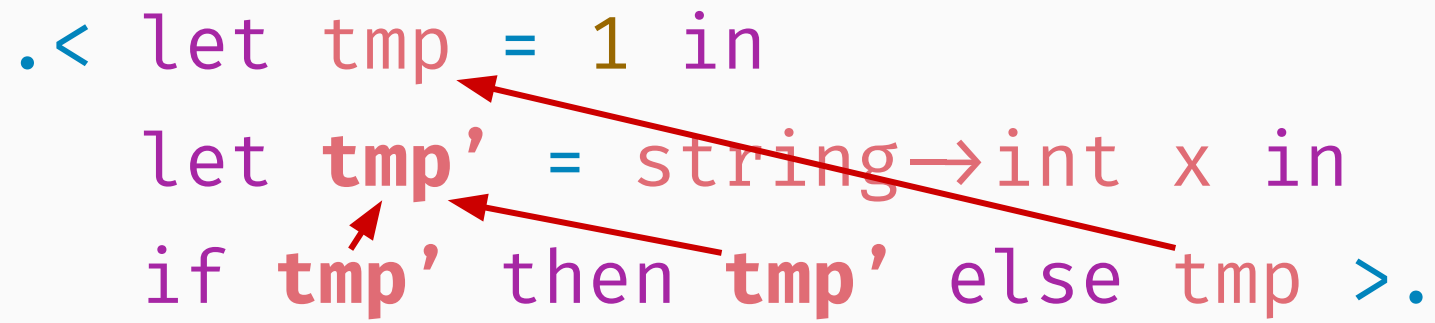
```
let genOr a b =
  .< let tmp = .~a in
     if tmp then tmp else .~b >. in
.< let tmp = 1 in
  .~(genOr .< string→int x >.
           .< tmp >.)
```



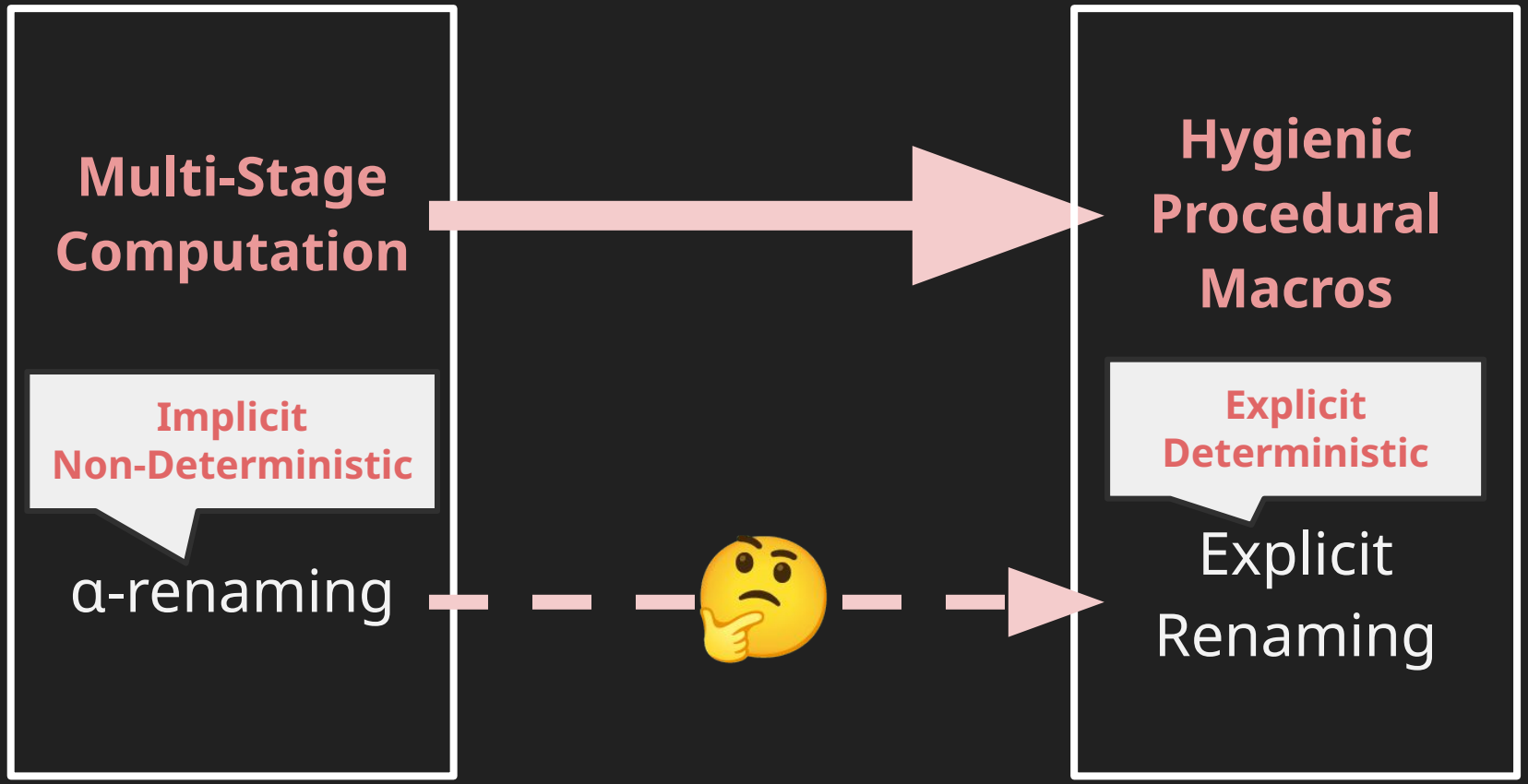
# α-renaming avoids name conflicts



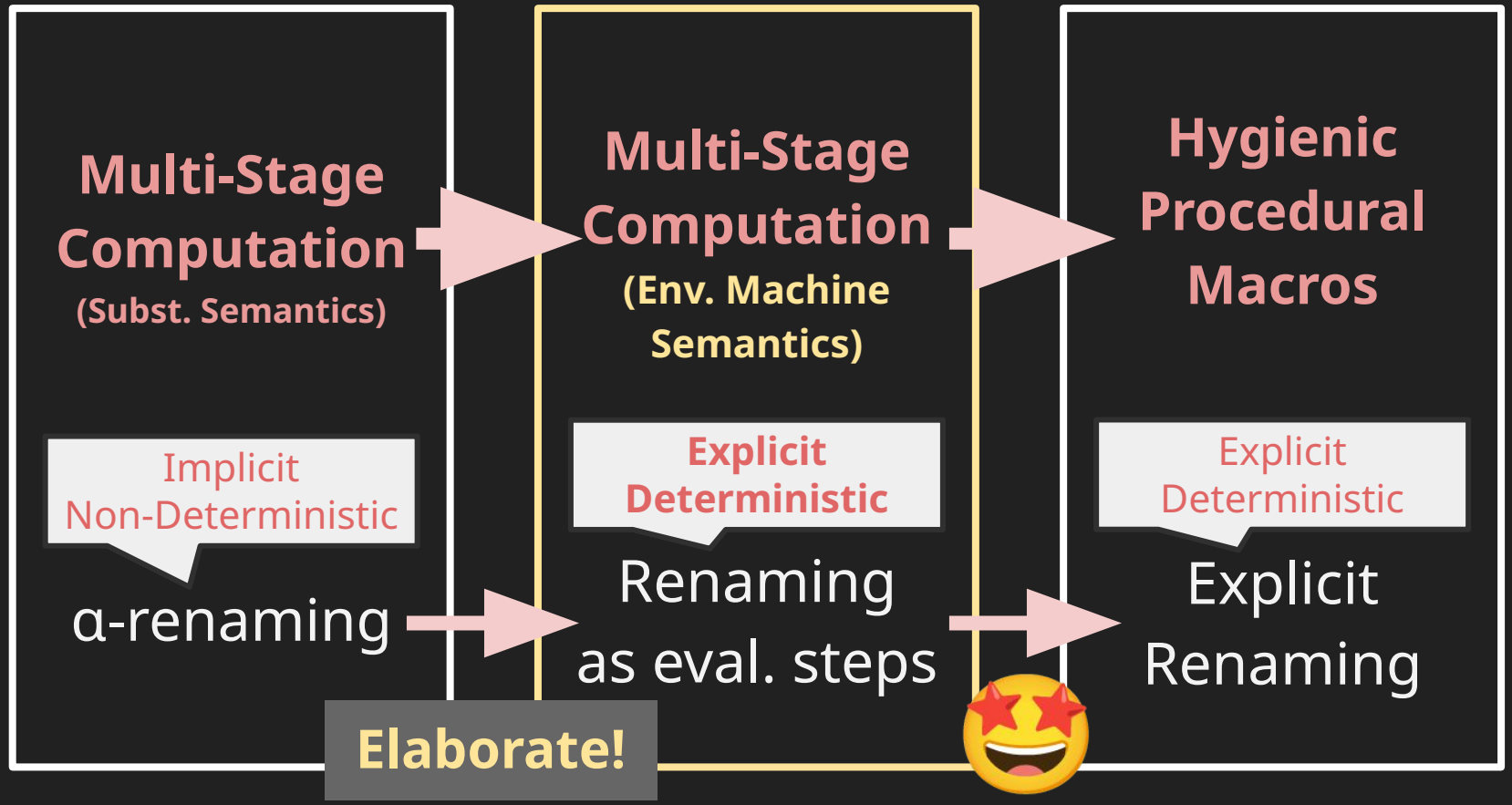
# α-renaming avoids name conflicts



# High-Level Description



# High-Level Description



# Evaluating a Staged Program

**Renaming Env.**

```
tmp → tmp0
```

**Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

```
.< let tmp0 = 1 in
   .~(.< let tmp = .~a in
         if tmp then tmp else .~b >.)
>.
```

# Evaluating a Staged Program (= Env. Machine)

**E. Renaming Env.**

```
tmp → tmp0
```

**D. Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

**C. Evaluation Context (or Continuation)**

```
.< let tmp0 = 1 in
  .~(.< let tmp = .~a in
        if tmp then tmp else .~b >.)
>.
```

**A. Expression under evaluation (or Commands)**
**B. Current level = 0 (= runtime level)**

# Evaluating a Staged Program

**Renaming Env.**

```
tmp → tmp0
```

**Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

```
.< let tmp0 = 1 in
   .~(.< let tmp = s→i x in
          if tmp then tmp else .~b >.)
>.
```

# Evaluating a Staged Program

**Renaming Env.**

~~tmp → tmp0~~
**tmp → tmp1**

**Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp then tmp else .~b >.)
>.
```

# Evaluating a Staged Program

**Renaming Env.**

~~tmp → tmp0~~
**tmp → tmp1**

**Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp then tmp else .~b >.)
>.
```

# Evaluating a Staged Program

**Renaming Env.**

~~tmp → tmp0~~
**tmp → tmp1**

**Value Env.**

```
gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.
```

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp1 then tmp else .~b >.)
>.
```

# Evaluating a Staged Program

**Renaming Env.**

~~tmp → tmp0~~
tmp → tmp1

**Value Env.**

gen → clos( ... )
a → .< s→i x >.
**b → .< tmp0 >.**

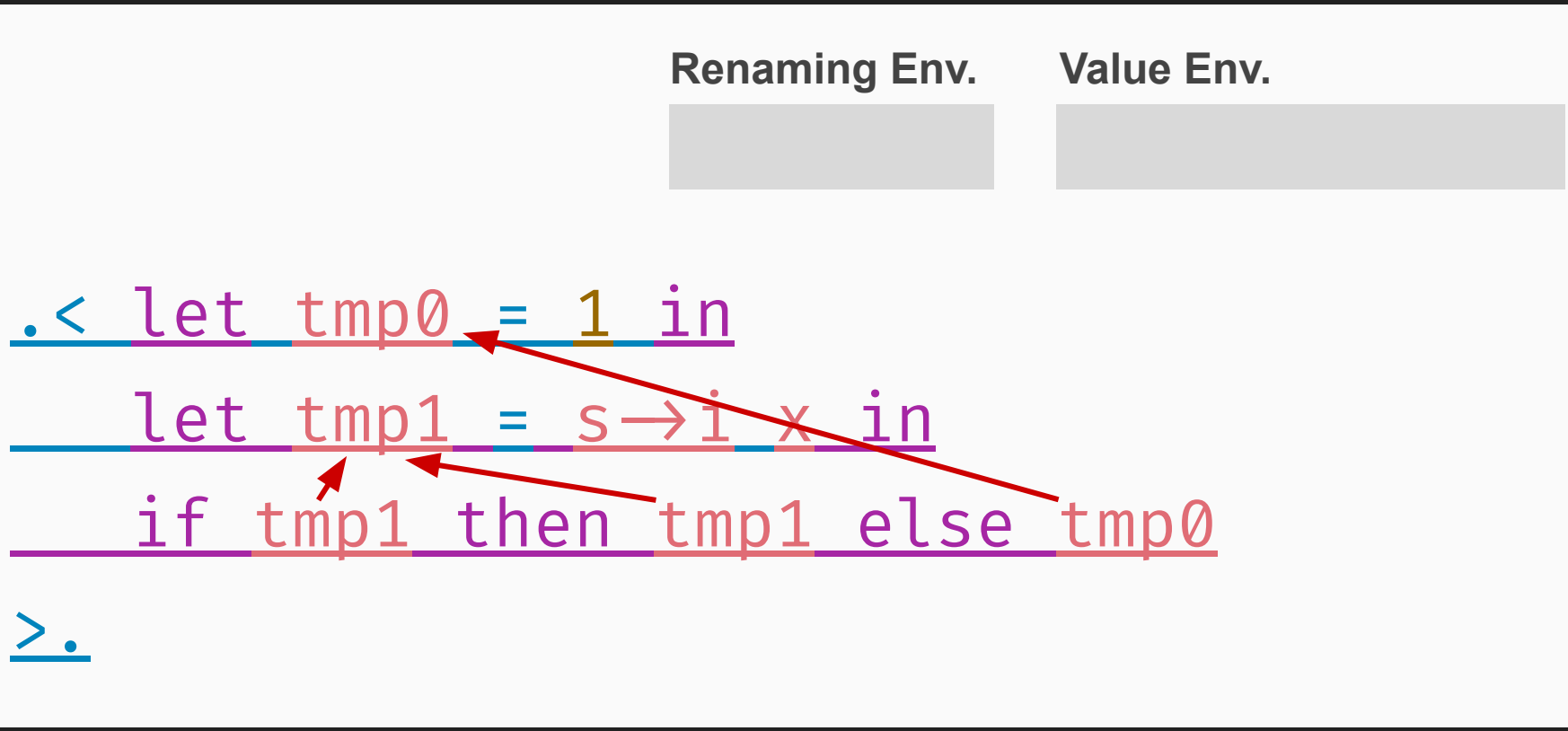
```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp1 then tmp1 else .~b >.)
>.
```

# Evaluating a Staged Program

| Renaming Env. | Value Env. |
| --- | --- |
| | |

```
.< let tmp0 = 1 in
   let tmp1 = s->i x in
   if tmp1 then tmp1 else tmp0
>.
```

# Evaluating a Staged Program



# ER as a Reflective Interface to Renaming Env.

**Renaming Env.**

~~tmp → tmp0~~
tmp → tmp1

**Value Env.**

gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp
         then tmp else .~b >.)
>.
```

# ER as a Reflective Interface to Renaming Env.

**Renaming Env.**

~~tmp → tmp0~~
tmp → tmp1

**Value Env.**

gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp1
         then tmp else .~b >.)
>.
```

# ER as a Reflective Interface to Renaming Env.

**Renaming Env.**

~~tmp → tmp0~~
tmp → tmp1

**Value Env.**

gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if .~(rename 'tmp)
         then tmp else .~b >.)
>.
```

# ER as a Reflective Interface to Renaming Env.

**Renaming Env.**

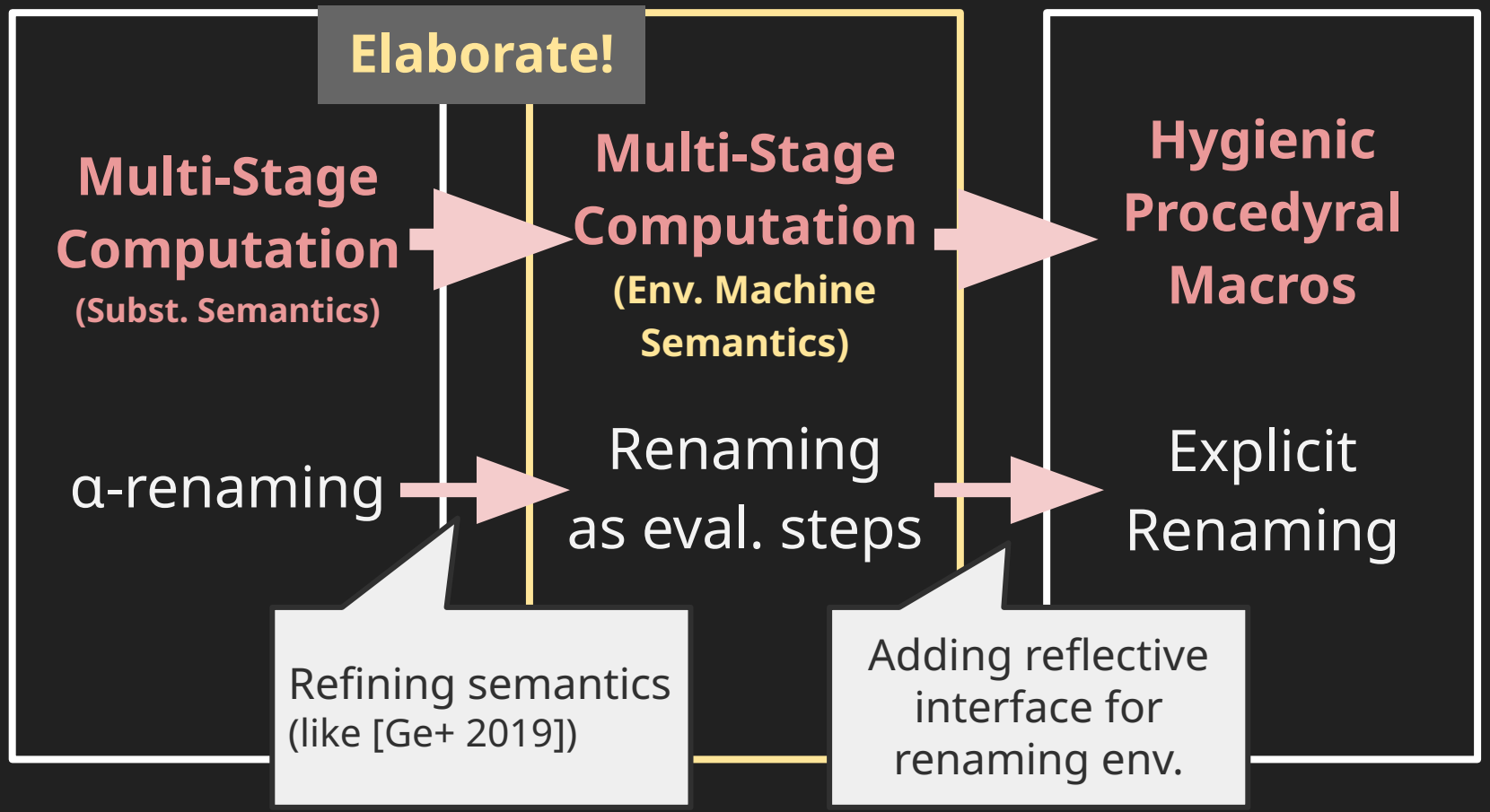
~~tmp → tmp0~~
tmp → tmp1

**Value Env.**

gen → clos( ... )
a → .< s→i x >.
b → .< tmp0 >.

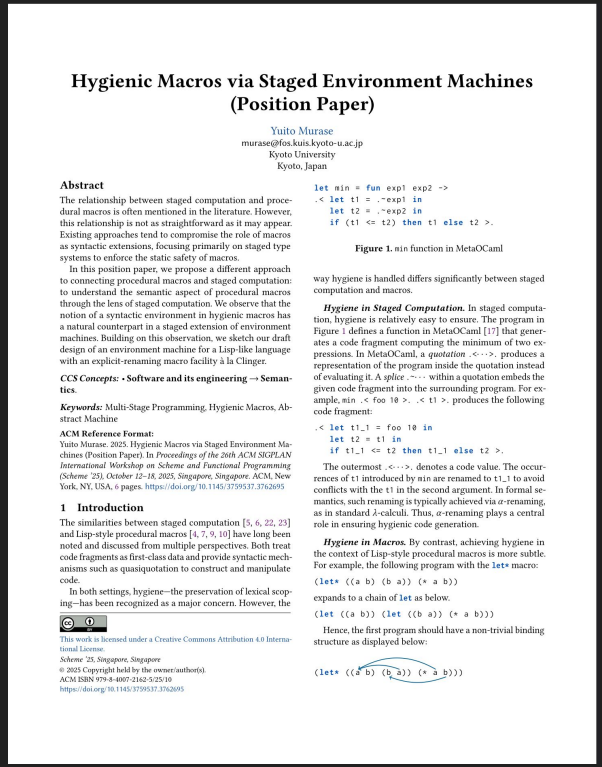
```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp1
         then tmp else .~b >.)
>.
```

# High-Level Description



# See My Position Paper for …

- Discussion on α-renaming vs ER
- Formal definition of the staged CEK machine
- Draft design of an abstract machine for ER
  - Lots of things need to be fixed



**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.


*Scheme '25, Singapore, Singapore*

ACM ISBN 979-8-4007-2162-5/25/10
https://doi.org/10.1145/3759537.3762695

```
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 $\alpha$-renaming, as in standard $\lambda$-calculi. Thus, $\alpha$-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)))
```

# Key takeaway

- We can derive Explicit Renaming from α-renaming
  - It is likely applicable to other hygienic macro facilities like syntactic closures and syntax-case
- Staged Environment Machines provide a novel viewpoint of hygienic code generation
  - … but pretty little work on them

# Supplementary Materials

# What needs to be done

- Correspondence between substitution semantics and Staged CEK machine
- Design full semantics for ER macros inspired by staged CEK machine
  - Challenge 1: Staging semantics in traditional MSP is not sufficient
  - Challenge 2: Gap between S-expresisons in Lisp and quasiquotes in MSP

# Relation to Existing Approaches

- Existing work that apply MSP to macros
  [Ganz+ 2001][Taha+ 2003][Stucki+ 2021][Xie+ 2023]
  1. Focus on type-safety
  2. Is incompatible with ER macros
- Extend α-renaming to S-Expression [Herman+ 2008][Stansifer+ 2014]
  1. Provide explicit binding specification for macros
  2. Is the other direction than ours: provide high-level macro facility that is compatible with α-renaming
- Implementation for MSP or Partial Evaluation
  [Glück+ 1997][Calcagno+ 2003]
  - Provide operational accounts of hygienic code generation
  - Lacks the idea of renaming environment

# Theory of Hyginic Macros is Hard

*… the subject of macro hygiene is not at all decided, and more research is needed to precisely state what hygiene formally means and which precisely assurances it provides.*

from [Kiselyov Scheme'02]

# Summary of Our Position

*Multi-Stage Programming* is considered to provide theoretical foundation for hygienic proc. macros
[Ganz+ 2001][Taha+ 2003][Stucki+ 2021][Xie+ 2023]

**Gap** MSP achieves hygiene via *α-renaming*, which is quite different from ER

**Our Answer** We can fill this gap by considering *an environment machine for MSP*

# What about this case?

Renaming Env.

Value Env.

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp1 then tmp1 else tmp0 >.)
>.
```

# Code Construction Steps in MSP

```
.< let tmp = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

```
.< let tmp = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

```
.< let tmp = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

```
.< let tmp = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

**Renaming Env.**

`tmp → tmp0`

```
.< let tmp0 = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

**Renaming Env.**

`tmp → tmp0`

```
.< let tmp0 = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

**Renaming Env.**

`tmp → tmp0`

```
.< let tmp0 = 1 in tmp + 1 >.
```

# Code Construction Steps in MSP

**Renaming Env.**

`tmp → tmp0`

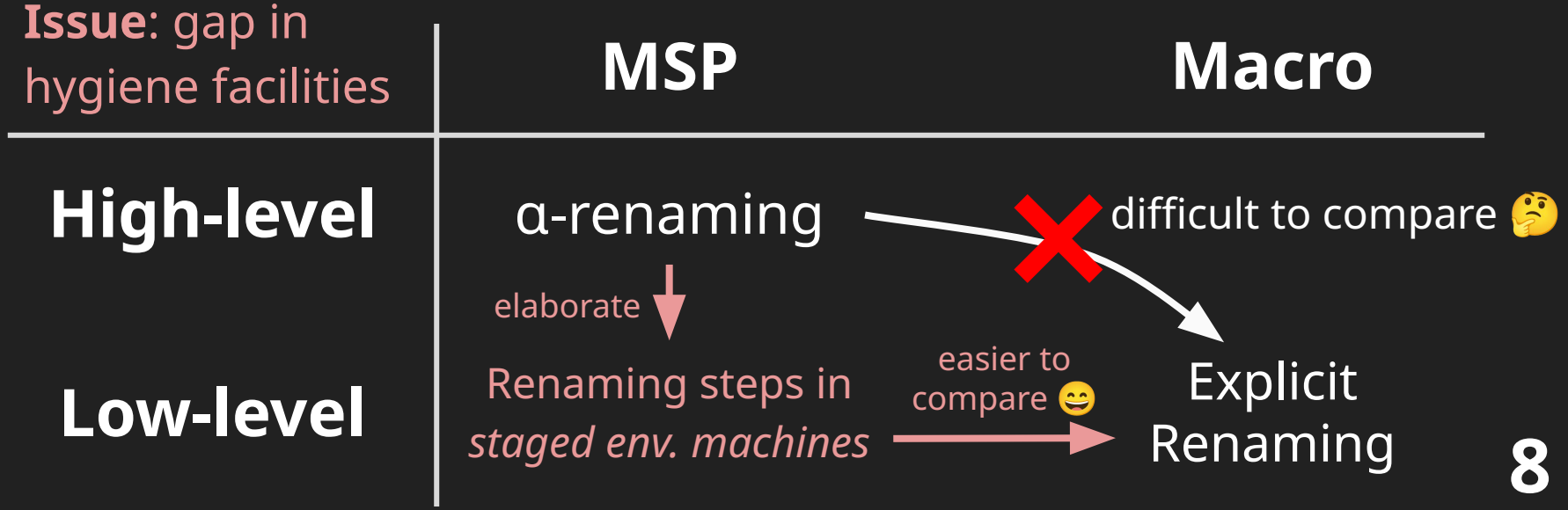
```
.< let tmp0 = 1 in tmp0 + 1 >.
```

# Code Construction Steps in MSP

```
.< let tmp0 = 1 in tmp0 + 1 >.
```

# Staged Env. Machines to Reason about ER macros

*Multi-Stage Programming* is considered to provide theoretical foundation for hygienic proc. macros
[Ganz+ 2001][Taha+ 2003][Stucki+ 2021][Xie+ 2023]



# Theory behind Explicit Renaming?

- The original paper of ER does not provide formal model [Clinger 1991]
- Some proposals provide formal models for hygienic macros in general [Flatt+ 2012][Adams 2015][Flatt 2016]
  - ... but, they are detached from the semantics of the host languages

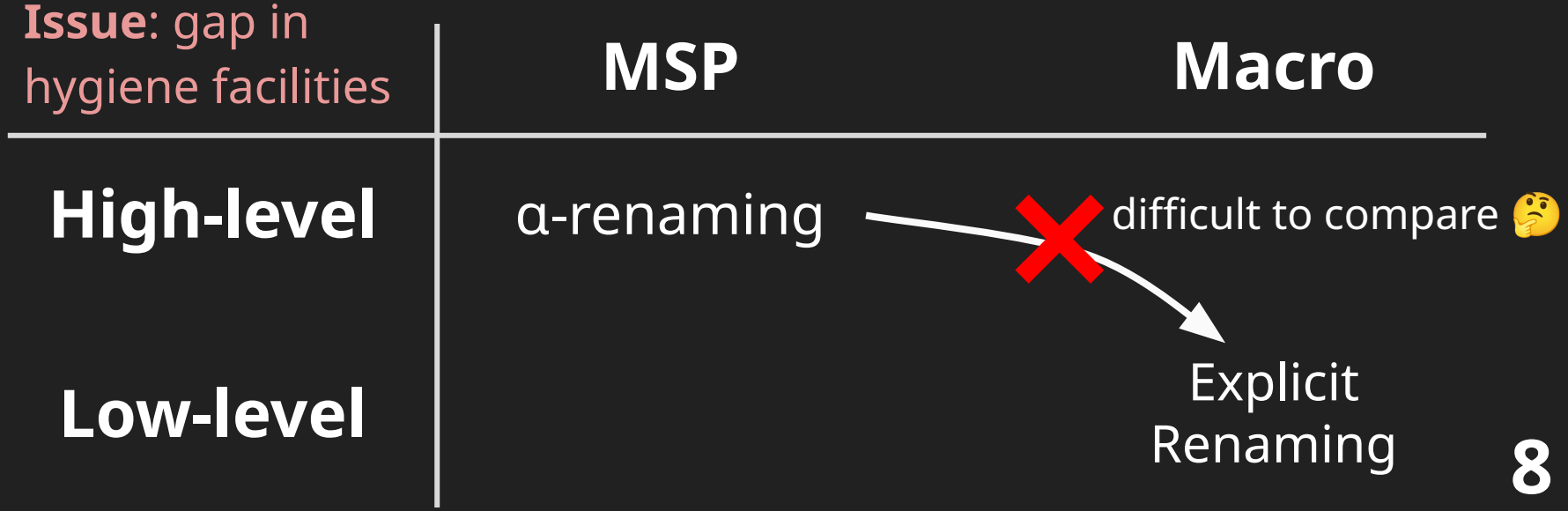
Theory of hygienic procedural macros should provide semantics for both side

# Staged Env. Machines to Reason about ER macros

*Multi-Stage Programming* is considered to provide theoretical foundation for hygienic proc. macros
[Ganz+ 2001][Taha+ 2003][Stucki+ 2021][Xie+ 2023]

# Staged Env. Machines to Reason about ER macros

*Multi-Stage Programming* is considered to provide theoretical foundation for hygienic proc. macros
[Ganz+ 2001][Taha+ 2003][Stucki+ 2021][Xie+ 2023]



# α-renaming vs Explicit Renaming

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in

  if tmp then tmp else .~b >.
```

```
(define-syntax or (er-macro-transformer
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if)
           ,(rename 'tmp)
           ,(rename 'tmp)
           ,b)))))))
```

| | MSP | Macro |
|---|---|---|
| **High-level** | α-renaming | difficult to compare 🤔 |
| | elaborate ↓ | |
| **Low-level** | Renaming steps in *staged env. machines* | easier to compare 😄 → Explicit Renaming |

# α-renaming vs Explicit Renaming

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in

  if tmp then tmp else .~b >.
```

```
(define-syntax or (er-macro-transformer
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if)
           ,(rename 'tmp)
           ,(rename 'tmp)
           ,b)))))))
```

Implicit
Non-Deterministic

**MSP** **Macro**

**High-level**

α-renaming

difficult to compare 🤔

elaborate

Explicit
Deterministic

**Low-level**

Renaming steps in
*staged env. machines*

easier to
compare 😄

Explicit
Renaming

# α-renaming vs Explicit Renaming

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in

  if tmp then tmp else .~b >.
```

```
(define-syntax or (er-macro-transformer
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if)
           ,(rename 'tmp)
           ,(rename 'tmp)
           ,b)))))))
```

Implicit
Non-Deterministic

**MSP**

**Macro**

**High-level**

Explicit
Deterministic

**Low-level**

α-renaming

elaborate

Renaming steps in
*staged env. machines*

difficult to compare 🤔

Explicit
Deterministic

easier to
compare 😄

Explicit
Renaming

# Overview Again

```
(* MetaOCaml style *)
let genOr a b =
.< let tmp = .~a in

  if tmp then tmp else .~b >.
```

```
(define-syntax or (er-macro-transformer
  (lambda (expr rename compare)
    (match expr
      ((_ a b)
       `(,(rename 'let) ((,(rename 'tmp) ,a))
          (,(rename 'if)
           ,(rename 'tmp)
           ,(rename 'tmp)
           ,b)))))))
```

Refining semantics
(like [Biernacka+ 2007]
[Ge+ 2019])

Substitution
→ Explicit Substitution
→ Environment Machine

| Renaming Env. | Value Env. |
|---|---|
| ~~tmp → tmp0~~<br>tmp → tmp1 | gen → clos( ... )<br>a → .< s→i x >.<br>b → .< tmp0 >. |

```
.< let tmp0 = 1 in
   .~(.< let tmp1 = s→i x in
         if tmp then tmp else .~b >.)
>.
```

Adding reflective interface

# Evaluating a Staged Program

```
let genOr a b =
  .< let tmp = .~a in
     if tmp then tmp else .~b >. in
.< let tmp = 1 in
  .~(genOr .< string→int x >.
           .< tmp >.)
```

# Evaluating a Staged Program

```
let genOr a b =
  .< let tmp = .~a in
     if tmp then tmp else .~b >. in
.< let tmp = 1 in
  .~(genOr .< string→int x >.
           .< tmp >.)
```