Macro Hygiene

Michael Ballantyne

0. The problem: generated names and unintended bindings.

1. Hygienic macro expansion: macros that "Do what I mean!"

2. Extending hygiene to fancy macros.

3. An efficient implementation.

4. Scope Sets: intuitive and debuggable hygiene... almost.

0. The Problem

Unintended Binding

```
(or a b) → (let ([v a])
             (if v
                 v
                 b))
```

```
(let ([default 0])
  (or (hash-ref h k)
      default))
```

⇒ 0

```
(let ([v 0])
  (or (hash-ref h k)
      v))
```

⇒ #false

```
(let ([if 0])
  (or (hash-ref h k)
      if))
```

⇒ Error: 0 is not a procedure

```
(let ([v 0])
  (or (hash-ref h k)
      v))
```

```
(let ([v 0])
  (let ([v (hash-ref h k)])
    (if v
        v
        v)))
```

An insidious problem: it only occurs when the macro user choses the one bad variable name.

Prehistoric Workarounds

- gensym

```
(or a b) → (let ([v# a])
             (if v#
                 v#
                 b))
```

v# is consistently replaced with a fresh name.

  - Error-prone.
  - Verbose.

- Reserved names

```
(let ([if 1]) )
(let ([first 1]) )
```

are illegal.

  - Macros can only safely expand to built-in procedures, syntax.
  - Many useful names are reserved.

Macros hold much more potential if they can get names right automatically.

Consider the key motivations for macros:

1. "Lambda the Ultimate"

   Big idea of Scheme: only need to specify, optimize, compile a tiny core. Most features can be macros on top.

   - Macros are pervasive
   - Modern languages: Julia, Elixir, Clojure

2. Domain- and program-specific language extensions

   - Programmers will be writing little compilers <u>all the time</u>!
   - Modern languages: Rust, Scala, Template Haskell

"Hygienic macro expansion"

Step 1

Kohlbecker, Friedman, Felleisen, Duba '86 – "kFFD"

In λ-calculus when we write

$$(\lambda v.b)\ e \to_n b[e/v]$$

we implicitly assume a "hygiene condition":

No bound variable of b is a free variable of e.

So $(\lambda x.\lambda y.x)\ y \not\to_n \lambda y.y$

To reduce, we need to α-rename:

$$(\lambda x.\lambda y.x)\ y$$
$$=_\alpha$$
$$(\lambda x.\lambda z.x)\ y \longrightarrow_n \lambda z.y.$$

In λ-calculus, we think "modulo α-equivalence"

How about applying α-renaming to macros?

'or' looks straightforward:

Before expanding

```
(let ([v 0])
  (or (hash-ref h k)
      v))
```

α-rename the macro template:

```
(let ([v a])
  (if v
      v
      b))
```

$=_\alpha$

```
(let ([go a])
  (if go
      go
      b))
```

expansion produces:

```
(let ([v 0])
  (let ([go (hash-ref h k)])
    (if go
        go
        v)))
```

In essence: automatically 'gensym' bound identifiers in the template.

But... what if 'let' itself is a macro?

(let ([x e]) b) → ((λ (x) b) e)

- Can't understand unexpanded 'or' template to α-rename.

What if 'or' is a procedural macro?

- Arbitrary computation, conditional behaviour.
- All we can do is run it.

Missing information to α-rename before.

Lost information after.

Key idea

1. During expansion, annotate syntax with information about its origin.
2. α-rename <u>retroactively</u> using scoping structure revealed by expansion.

```
(or (hash-ref h k)        ⟶   ((λ (v1)
    v)                            (if v1
                                      v1
                                      v0))
                                  (hash-ref0 h0 k0))
```

Projection of names generated by 'or':

```
((λ (v)                    =α    ((λ (y)
   (if v                              (if y
       v                                  y
       —))                                —))
 (—— — —))                          (—— — —))
```

kFFD algorithm

1. Initial names get timestamp 0

```
(let ([V0 0])
  (or (hash-ref0 h0 k0)
      V0 ))
```

2. After each expansion step...

```
(let ([V0 0])
  (let ([V (hash-ref0 h0 k0)])
    (if V
        V
        V0 )))
```

Give unstamped names a new timestamp

```
(let ([V0 0])
  (let ([V1 (hash-ref0 h0 k0)])
    (if V1
        V1
        V0 )))
```

3. After expansion is over, α-rename bound names.

```
(let [[x 0]])
  (let ([y (hash-ref0 h0 k0)])
    (if y
        y
        x)))
```

4. Drop timestamps from remaining names.

$$\text{hash-ref}_0 \longrightarrow \text{hash-ref}$$

These are free references and refer to top-level or built-in bindings.

Notice names can't be represented as symbols.

Can we say what we mean by hygiene?

"Hygiene Condition for Macro Expansion:

Generated identifiers that become binding instances in the completely expanded program must only bind variables generated in the same transcription step."

But what about generated references?

- Algorithm gets it right.
- Missing from condition.

Clinger and Rees '91

Step 1.5

"It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro"

1. $(m) \rightarrow x$

$$(\text{let } ([x \text{ 'bad}]) \ (m)) \longrightarrow (\text{let } ([x_0 \text{ 'bad}]) \ x_1)$$

2. $(m\ v) \rightarrow (\text{let } ([v \text{ 'bad}]) \ x)$

$$(m\ x) \longrightarrow (\text{let } ([x_0 \text{ 'bad}]) \ x_1)$$

Intuition

```
(for/set [i l]
  b)
```

→

```
(foldl (λ (acc i)
         (set-add
           acc
           b ))
       (set) l )
```

```
(let [nums (list 1 2 1)]
  (for/set [n nums]
    (+ n 1)))
```

↓

```
(let [nums (list 1 2 1)]
  ((_ (λ (_ n)
        (_
          _
          (+ n 1))
        (_) nums))))
```

Top

```
(foldl (λ (acc _)
         (set-add
           acc
           (_ _ _)))
       (set) _ )
```

Extending hygiene to local and macro-defining macros.

Step 2

— Clinger and Rees '91

— Dybvig, Hieb, Bruggeman '43

```
(define find-address
  (lazy-let [table (read "address-book.txt")]
    (λ (person)
      (hash-ref table person))))
```

Implement by expansion to:

```
(define find-address
  (let [table (delay (λ () (read "address-book.txt")))]
    (λ (person)
      (hash-ref (force table) person))))
```

Defining lazy-let:

```
(define-syntax
  (lazy-let [x e] b)
  →
  (let [tmp (delay (λ () e))]
    (let-syntax [x → (force tmp)]
      b)))
```

The intermediate expansion step:

```
(define find-address
  (let [tmp (delay (λ () (read "address-book.txt")))]
    (let-syntax [table → (force tmp)]
      (λ (person)
        (hash-ref table person)))))
```

Changes from KFFD assumptions:

- Free references in macros resolve in context of definition, not top level
- Macro names need hygiene just like variables.
- Syntax may be generated by a macro, and then generated <u>again</u> by a macro-defined macro
  - 'tmp' in 'lazy-let'

Key idea

Interleave expansion and renaming

- Expanded portion uses unique names for every binding.

```
⌜(let [x "good"]
   (let-syntax [(m v) → (let [v "bad"]
                          x)]
     (m x)))⌟
```

↳

```
(let [a "good"]
  ⌜(let-syntax [(m v) → (let [v "bad"]
                          a)]
     (m a))⌟)
```

α-renaming treats unexpanded portions conservatively.

After expanding `m`:

```
(let [a "good"]
  (let-syntax [(m v) → (let [v "bad"]
                         a)]
    ⌜(let [a "bad"]
       a_m1)⌝))
```

Generated syntax gets <u>mark</u> `m1` — like a timestamp.

Marks distinguish use-site vs generated names. Renamings don't apply if marks differ.

Renaming outside-in ensures shared bindings rename <u>before</u> generated syntax is marked

So, rename the use-site binder but not the generated references:

```
(let [a "good"]
  (let [b "bad"]
    ⌜a_m1⌟))
```

Finally, drop marks:

```
(let [a "good"]
  (let [b "bad"]
    a))
```

What about macro names?

Outside-in renaming ensures we understand their scope.

```
(let-syntax [(or a b) → __]
  (let [or __]
    (or 1 #false)))
```

```
(let-syntax [(x a b) → __]
  (let [y __]
    (y 1 #false)))
```

What about syntax generated by macro-defined macros?

- Use multiple marks.

$$x \neq x_{m_1} \neq x_{m_1 m_2}$$

- Can't fit an example.

How do we apply marks to generated syntax?

$\text{timestamp}(x_0, 1) = x_0$

$\text{mark}(x, 0) = x_{m_0}$

$\text{mark}(x_{m_0}, 1) = x_{m_0 m_1}$

Marking twice cancels

$$\text{mark}(x_{m_0}, 0) \equiv x$$

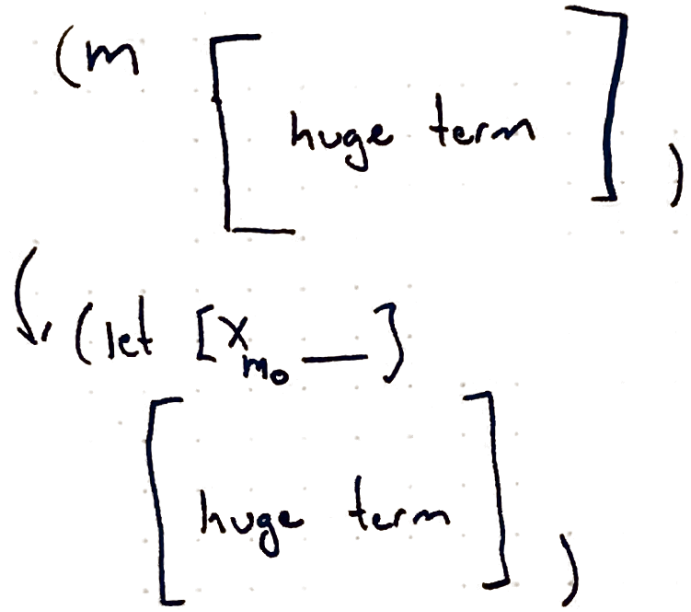
Mark macro input and output.

- Marks on input cancel
- Marks on output remain

Step 3

Performance, especially for procedural macros

Syntax traversal for every mark and every renaming!



Solution: make it lazy (Dybvig et al '93)

- (struct syntax-wrap [content operations])
- syntax-case propagates operations to unwrap, lazily.
- repeated marks cancel before propagation

"Binding as Sets of Scopes"

Step 4

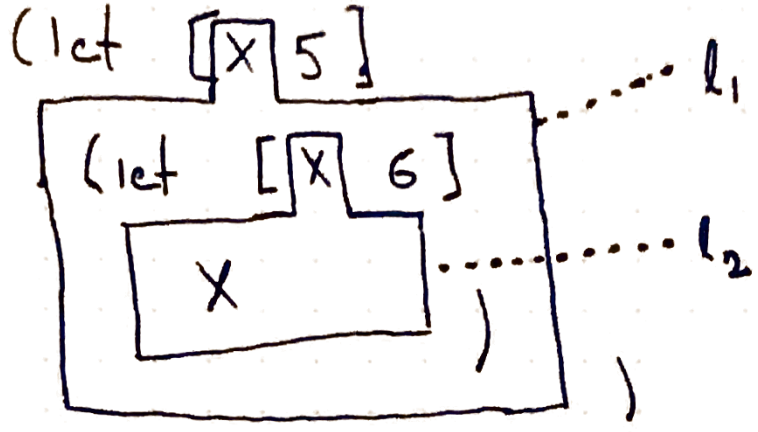
Flatt '16

State of the art was marks and renamings++

- Implements hygiene by a process of renaming intertwined with expansion.
- Hard to visualize
- Hard to debug
- Lazy data structure gets complicated for 'define': mutable, cyclic.

Could we annotate syntax enough to delay renaming to resolution?

Could these annotations simply represent scopes?



Binders are visible to references that are in all the same regions as the binder.

We can annotate names with sets of regions:

$$(\text{let } [X_{\{l_1\}}\ 5]\ (\text{let } [X_{\{l_1, l_2\}}\ 6]\ X_{\{l_1, l_2\}}))$$

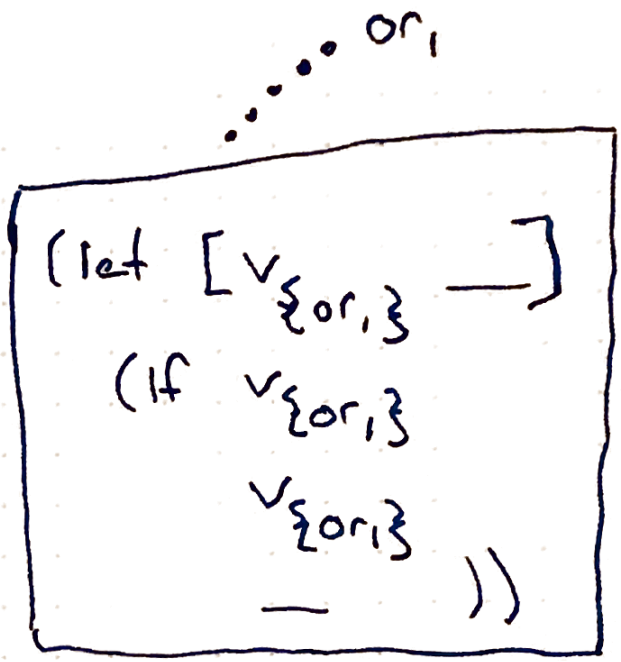
A reference can see bindings with a subset of its scopes.

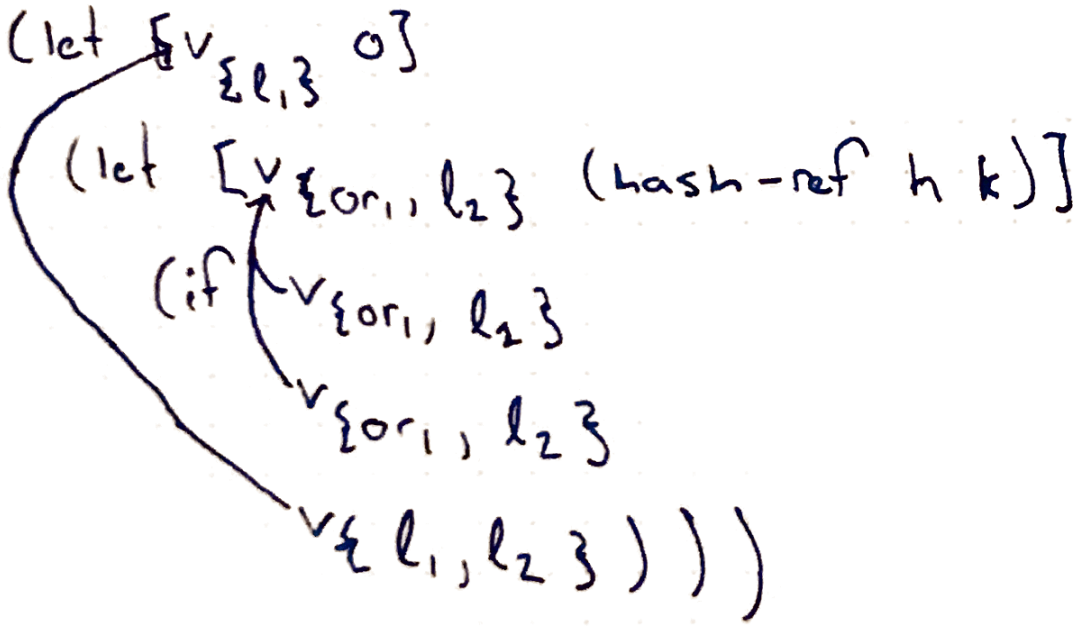
More scopes = narrower region of program

Macros

```
(let [v{l1} 0]
  (or (hash-ref h k)
      v{l1}))
```

Think of macro-generated names belonging together in a "region".





Looks great so far!

What about:

```
(define-syntax
  (m v)
  →
  (let [v "bad"]
    x))
```

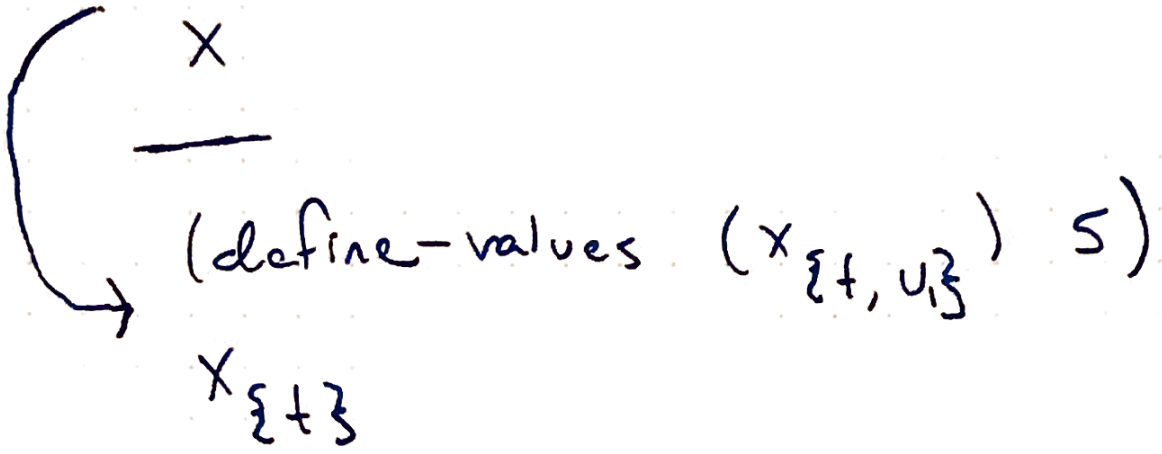
```
(m x)
↳ (let [x {t, l} "bad"]
    x {t, m1, l} )
```

It seems we need a "scope" for the use-site projection of the expanded syntax.

```
(m x)
↳ (let [x {t, u2, l} "bad"]
    x {t, m1, l} )
```

One more:

(define x 5)

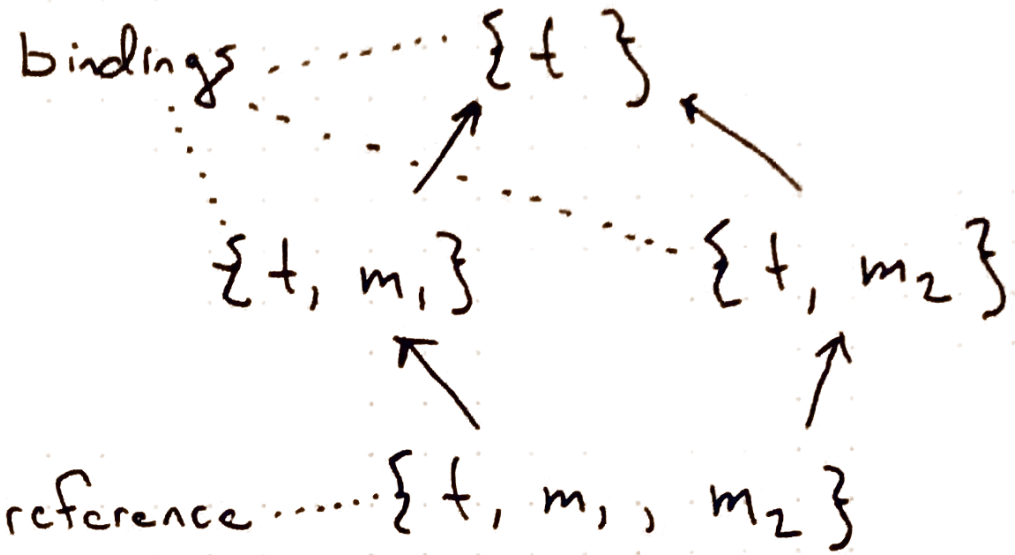


Add a rule that removes use-site scopes from 'define' bindings...

- Are use-site "scopes" sufficiently scope-like?

- Epicycles?

Ambiguity



Usually I like to think of lexical scope as a tree!

Here we can get a lattice.

Reflecting on scope sets

+ Simple intuition
+ Simple resolution rule
+ Macro debugger can show scopes

– Use-site scopes don't seem like scopes. And sometimes we remove them.

– Ambiguity

+ `local-expand`

Reflecting on macro hygiene

- We have algorithms that seem to work, and some vague english hinting at why.
- Theorists have tried to more deeply understand hygiene, but only manage simple cases.

  Herman 2010; Adams 2015;
  Bove, Arbilla 1992

- General hygienic macros haven't escaped Scheme / Racket
  - Hygiene in Rust, Julia, Clojure, Elixir limited.
  - Why not?
    - Difficult to understand
    - Full implementations are big.