Towards the Essence of Hygiene

Michael D. Adams

University of Utah
University of Illinois at Urbana/Champaign
http://michaeldadams.org/

Abstract

Hygiene is an essential aspect of Scheme's macro system that prevents unintended variable capture. However, previous work on hygiene has focused on algorithmic implementation rather than precise, mathematical definition of what constitutes hygiene. This is in stark contrast with lexical scope, alpha-equivalence and capture-avoiding substitution, which also deal with preventing unintended variable capture but have widely applicable and well-understood mathematical definitions.

This paper presents such a precise, mathematical definition of hygiene. It reviews various kinds of hygiene violation and presents examples of how they occur. From these examples, we develop a practical algorithm for hygienic macro expansion. We then present algorithm-independent, mathematical criteria for whether a macro expansion algorithm is hygienic. This characterization corresponds closely to existing hygiene algorithms and sheds light on aspects of hygiene that are usually overlooked in informal definitions.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Algorithms; Languages

Keywords Hygiene; Macros; Nominal logic

1. Introduction

Hygiene is an essential aspect of the Scheme macro system that prevents unintended variable capture. For example, suppose we have a short-circuiting `or` macro that expands as follows.

```
(let ([x #t])
  (or (begin (write "!") #f) x)) ⇝
(let ([x #t])
  (let ([tmp (begin (write "!") #f)])
    (if tmp tmp x)))
```

This expansion introduces a `let` binding for `(begin (write "!") #f)` in order to avoid duplicating its side effect. However, this `let` binding can cause trouble. Suppose the original expression used `tmp` instead of `x`. In a naive macro system, the result would be the following expansion.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2677013

```
(let ([tmp #t])
  (or (begin (write "!") #f) tmp)) ⇝
(let ([tmp #t])
  (let ([tmp (begin (write "!") #f)])
    (if tmp tmp tmp)))
```

In the result of this expansion, `tmp` is captured and no longer bound to what it was at the call to `or`. Changing the binding name from `x` to `tmp` has changed the result from `#f` to `#t`.

Hygiene prevents these problems. In a hygienic macro expansion system, macros are automatically as well-behaved with regard to variable names as the core forms of the language, and users can rely on the expected alpha equivalences. Essentially, hygiene is the moral equivalent of lexical scoping at the macro level.

Hygiene has a long history in the literature (Kohlbecker et al. 1986; Kohlbecker and Wand 1987; Bawden and Rees 1988; Clinger and Rees 1991; Clinger 1991; Dybvig et al. 1993; Herman and Wand 2008; Herman 2010). However, these works leave hygiene defined informally and focus on the algorithmic aspects or apply to only a restricted set of macros that follow certain typing disciplines and do not handle the general case. Furthermore, while they all purport to implement similar concepts of hygiene, they use divergent implementation approaches that are difficult to formally compare without a precise, formal definition of hygiene.

This paper proposes such a definition of hygiene. This definition is formally precise and connects hygiene to concepts from nominal logic (Gabbay and Pitts 2002). In particular, it shows that hygiene is a combination of alpha equivalence and nominal logic's notion of equivariance. This definition is applicable to a wide variety of macro systems, and deriving a practical algorithm that satisfies it leads to connections with existing hygiene algorithms such as the `syntax-case` expansion algorithm used in Scheme (Dybvig et al. 1993). Finally, we use this definition to highlight important aspects of hygiene that are omitted from traditional, informal descriptions of hygiene.

The organization of this paper is as follows. In Section 2, we give an overview of the basic ideas behind our characterization of hygiene. In Section 3, we review the fundamentals of macro expansion in Scheme. In Section 4, we examine various sorts of hygiene violations and how they occur. We show why traditional notions of alpha equivalence and other naive methods for ensuring hygiene are not sufficient. We show that identifiers in a general hygienic system must be diatomic rather than monatomic, and we present a simple algorithmic method for ensuring hygiene. In Section 5, we develop a formal, mathematical characterization of hygiene in terms of nominal logic. In the process, we demonstrate a new sort of hygiene violation that, while implicitly handled by traditional hygiene algorithms, is not widely discussed. In Section 6, we show how this characterization corresponds to the `syntax-case` algorithm. Finally, in Section 7 we discuss related work, and in Section 8 we conclude.

2. Overview of the Main Ideas

While we give a more complete treatment of the mathematical concepts of hygiene in Section 5, the main ideas are as follows.

We show that a syntactic representation that treats identifiers as simple atomic values with no auxiliary information does not contain enough information to express the structures needed to implement a hygienic macro system as powerful as Scheme's. We can, however, express those structures by splitting identifiers into a pair of atoms. Identifiers are thus diatomic instead of monatomic, and we write them as <r b> where r is the *reference part* of the identifier and is inhabited by a *reference atom* while b is the *binder part* and is inhabited by a *binder atom*.

Next, we formally define hygiene in terms of two criteria. The first, the Reference Hygiene Criterion, requires that each macro expansion step respect alpha equivalence with respect to reference atoms in the fully expanded, outer parts of the syntax that have known binding structures.

Unfortunately, we cannot use alpha equivalence in the partially expanded, inner parts of the syntax as their binding structures are unknown. To handle this, the second criterion, the Binder Hygiene Criterion, is expressed in terms of an equivariance condition with respect to binder atoms that must hold on all macro transformation functions.

In essence, the Reference Hygiene Criterion requires that the expansion process respect alpha equivalence in the parts of the syntax with known binding structures. In parts of the syntax where the binding structure is unknown, this will not work, so the Binder Hygiene Criterion requires that macro transformers respect any possible binding structure that that part of the syntax may have.

Together, these two criteria formally define hygiene, and we specify these using nominal logic in Section 5. These criteria are consistent with traditional, informal definitions of hygiene (Bawden and Rees 1988; Clinger and Rees 1991). The Reference Hygiene Criterion ensures that references introduced by a macro are not captured by bindings other than those introduced by the macro. The Binder Hygiene Criterion ensures that binders introduced by a macro are always freshly generated and thus cannot capture reference other than those introduced by the macro.

Finally, though introduced-reference and introduced-binder hygiene are usually described as the two main aspects of hygiene, the equivariance used to define the Binder Hygiene Criterion requires a third aspect. Namely, macro transformers must not observe the values of binders in their inputs other than to compare them with other binders in their inputs. This third form of hygiene is actually enforced by traditional hygiene algorithms even though informal descriptions of those algorithms do not discuss it.

3. Basics of Scheme Macros

Before considering hygiene, we briefly review the basics of Scheme macros and how they operate. As shown in Figure 1, macro expansion occurs between the reader, which parses a string into an s-expression, and the interpreter or compiler, which evaluates or compiles the fully expanded term.

At the start of this process, an s-expression is taken from the reader and injected into the domain of syntax objects. For our purposes, a syntax object is an s-expression with identifiers instead of symbols. In a naive expander these identifiers are plain symbols, but in a hygienic expander they carry extra scoping and binding information. In some presentations, syntax objects additionally contain other information such as source locations or data structures that assist the algorithmic performance of macro expansion. We omit these as they are not relevant to the essential theory of hygiene. In the remainder of this paper, we deal only with syntax objects and do not use s-expressions.

$$\text{String} \xrightarrow{\text{Read}} \text{S-expression} \xrightarrow{\text{Inject}} \text{Syntax}_0 \rightsquigarrow \text{Syntax}_1 \rightsquigarrow \ldots \rightsquigarrow \text{Syntax}_n \xrightarrow{\text{Compile}} \text{Executable}$$

Figure 1. Scheme's compilation pipeline

Next comes the core expansion process. We model this as a sequence of small-step rewrites using $\rightsquigarrow$. Each step corresponds to one particular macro invocation or expansion of a core form. The expansion process always proceeds with the outermost expression that can be expanded. We thus divide the program into two parts. We call the inner part that is waiting to be further expanded *u-syntax* as its binding structure is *unknown*, and we call the outer part that is already fully expanded *k-syntax* as its binding structure is *known*.

There is no more expansion to be done in k-syntax, and we know the form of each component of k-syntax. Thus we can represent it using the abstract-syntax representations used internally by the compiler. Conversely, the structure of the u-syntax waiting to expand is not yet manifest. In order to distinguish u-syntax from k-syntax, we adopt the notation of underlining the former. Thus, for example, we may say that the u-syntax `(lambda (x) (+ 1 x))` expands after a number of steps to the k-syntax `(lambda (x) (+ 1 x))`. Note that u-syntax may be modified by macro expansion though k-syntax cannot. Thus, while we might expect the expression `(f x y)` to be a function call, it may instead be a macro call if `f` is bound to a macro definition. Likewise, even though `(lambda (x) y)` is usually a lambda expression, it may be a macro or function call due to a local definition of `lambda`. This is resolved only once this u-syntax is fully expanded into k-syntax. We define both u-syntax and k-syntax in Figure 2 where we use vector notation for repeated elements. For simplicity of presentation, we gloss over which repetitions are zero-or-more versus one-or-more, and we include `let`-style forms as core forms instead of encoding them in terms of `lambda`. Using these forms, the `or` macro can be written as in Figure 3. The full expansion sequence for `(let ([x #t]) (or #f x))` is then as shown in Figure 4.

To define a macro, we use `let-syntax` and `letrec-syntax`. These are similar to `let` and `letrec` except that the left-hand sides of their bindings are the names of macros and the right-hand sides are macro transformers. For example, we use the following to define the `or` macro where f is the macro transformer for `or`.

```
(letrec-syntax ([or f]) ...)
```

Macros are generally invoked using a function-call-like syntax.[1] Within the body of this `letrec-syntax`, the `or` macro can thus be invoked using `(or #f x)`.

A *macro transformer* is a function that defines how a macro expands. It takes as argument the syntax object representing the macro invocation and returns the syntax object to which the macro call should expand. For example, the macro invocation `(or #f x)` causes `(or #f x)` to be passed as input to the transformer for `or`. The transformer then returns a syntax object such as `(let ([tmp #f]) (if tmp tmp (or x)))`.

Syntax objects are similar to s-expressions, but they have separate methods for construction and deconstruction. Unlike s-expressions, which use functions like `cons`, `pair?`, `car` and `cdr` to create and examine objects, syntax objects use the pattern matching form `syntax-case` and the quotation form `syntax`.

The first argument of `syntax-case` is the scrutinee and is matched against the patterns in each clause. The second argument

[1] Scheme's `make-variable-transformer` and `identifier-syntax` allow macros to be called in other places. As these are orthogonal to hygiene, we omit them from this paper for the sake of simplicity.

Identifiers $i ::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots$

Constants $c ::= \texttt{\#f} \mid \texttt{\#t} \mid \texttt{1} \mid \texttt{2} \mid \cdots$

Pattern Literals $lit ::= i$

U-Syntax $stx ::= \underline{i} \mid \underline{c} \mid \underline{\texttt{()}} \mid \underline{(stx\ .\ stx)}$

K-Syntax

$ast ::= c$	Constants
$\mid i$	Variables
$\mid$ `(lambda (` $\overrightarrow{i}$ `)` $\overrightarrow{ast}$`)`	Functions
$\mid$ `(`ast $\overrightarrow{ast}$`)`	Function App.
$\mid$ `(if` ast ast ast`)`	Conditionals
$\mid$ `(let ([`$\overrightarrow{i\ ast}$`]) `$\overrightarrow{ast}$`)`	Local Variables
$\mid$ `(letrec ([`$\overrightarrow{i\ ast}$`]) `$\overrightarrow{ast}$`)`	
$\mid$ `(let-syntax ([`$\overrightarrow{i\ ast}$`]) `$\overrightarrow{ast}$`)`	Local Macros
$\mid$ `(letrec-syntax ([`$\overrightarrow{i\ ast}$`]) `$\overrightarrow{ast}$`)`	
$\mid$ `(syntax` stx`)`	Syntax Constr.
$\mid$ `(syntax-case` ast `(`$\overrightarrow{lit}$`) [`$\overrightarrow{pat\ ast}$`])`	Syntax Deconstr.
$\mid stx$	Unexpanded Code
$\mid \cdots$	

Patterns

$pat ::= \_$	Wild Card
$\mid$ `()`	Nil Constant
$\mid i$	Pattern Variables
$\mid$ `(`pat `.` pat`)`	Pairs
$\mid \cdots$	

Figure 2. Syntactic forms

is a list of identifiers that are to be treated as literal constants in the patterns of the `syntax-case`. In this paper, pattern literals are not used, and we use only a subset of the pattern language. However, the techniques we describe trivially generalize to pattern literals and the full pattern language. The subset that we use allows a pattern to be either a wild card that matches anything, the nil constant, a pattern variable or a pair of patterns. As with s-expressions, nested pairs can be represented using list notation.

The `syntax` form is a quotation form for syntax objects. As a notational short-hand, `(syntax` e`)` can be written as `#'`e for any e just as `(quote` e`)` can be written as `'`e. In addition, any pattern variables that are bound by `syntax-case` are automatically substituted for the corresponding identifiers in an enclosed `syntax`. Any identifiers that are not bound by a pattern are simply constants. For example, the following evaluates to the syntax object `(a 1 z)`.

```
(syntax-case #'(1 a) () [(x y) #'(y x z)])
```

Finally, for simplicity of presentation we do not discuss library forms, module forms, `define` or `define-syntax`. In addition, we only briefly discuss, in Section 6, how to handle the `quote` form and the hygiene "bending" operators `datum->syntax` and `syntax->datum`.

```
(letrec-syntax ([or
  (lambda (stx)
    (syntax-case stx ()
      [(_)        #'#f]
      [(_ x)      #'x]
      [(_ x . xs) #'(let ([tmp x])
                      (if tmp tmp (or . xs)))]))])
  ...)
```

Figure 3. Definition of the `or` macro

```
(let ([x #t]) (or #f x)) ⇝
(let ([x #t]) (or #f x)) ⇝
(let ([x #t]) (or #f x)) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp (or x)))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp (or x)))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp (or x)))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp (or x)))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp (or x)))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp x))) ⇝
(let ([x #t])
  (let ([tmp #f]) (if tmp tmp x)))
```

Figure 4. Example naive expansion of the `or` macro

4. Hygiene

Before considering how to mathematically define hygiene in Section 5, we examine the different sorts of hygiene, how they are violated and how to handle them algorithmically. To do this we start with a naive algorithm that uses monatomic identifiers and work our way up to a fully hygienic algorithm. The resulting algorithmic techniques are relatively simple, but they provide useful intuitions about how a hygienic expansion system ought to behave. Later in Section 5, we use these intuitions to develop mathematically precise properties that characterize hygiene.

4.1 Types of Hygiene Violations

Hygiene is fundamentally about avoiding unintended variable capture, and there are two primary ways that can happen. The first is by a macro introducing binders, and the second is by a macro introducing references. For example, as seen in the following quote, according to Clinger and Rees (1991) a hygienic macro system is one in which:

> *1. It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.*
>
> *2. 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.*

Interestingly, once we mathematically characterize hygiene, we will discover that there is a third kind of hygiene violation. Specifically, we must ensure that identifiers that act as binders in the input of a macro transformer are not observed by the macro transformer other than to compare them to other binders from the same

input. Though this third kind is not often discussed, it is implicitly averted by traditional hygiene algorithms, and we explore this in Section 5.6.

4.1.1 Introduced-Binder Hygiene

As an example of introduced-binder hygiene, consider the following code.

```
(let ([tmp #t]) (or #f tmp))
```

This should evaluate to `#t` since the first argument to `or` is `#f` and the second argument, `tmp`, is bound to `#t`. However, a naive, non-hygienic expander eventually expands this to the following.

```
(let ([tmp #t])
  (let ([tmp #f]) (if tmp tmp tmp)))
```

The `or` macro introduces an inner binding of `tmp` to `#f` that shadows the binding of `tmp` to `#t` and results in this code evaluating to `#f` instead of the expected `#t`.

4.1.2 Introduced-Reference Hygiene

Hygiene violations due to introduced binders are certainly the most easily recognized sort of violation. However, violations due to introduced references can also occur. Consider, for example, the following variation of the code we had before.

```
(let ([if #t]) (or #f if))
```

All we have done is rename `tmp` to `if`, so this code should still evaluate to `#t`. However, a naive, non-hygienic expander eventually expands this to the following.

```
(let ([if #t])
  (let ([tmp #f]) (if tmp tmp if)))
```

Note that `if` is bound to `#t`, so `(if tmp tmp if)` is not the core form for `if` as it was before, but rather a function application of the variable `if` to the variable references `tmp`, `tmp` and `if`. This is thus attempting to call `#t` as a function. Aside from being a runtime error, it is unlikely to be what the programmer intended.

This time the binding of `tmp` to `#f` introduced by the `or` macro is not the cause of the problem. Rather, it is the fact that the macro introduces a reference to `if` that it expects to be bound to the core form for `if`. Since the macro is called in a context in which `if` is not bound to the core form, we get an incorrect result.

This problem occurs any time a macro introduces a reference that it expects to be bound in a particular way. Though this particular violation occurred due to the rebinding of a core form, this may happen even in a system with multiple namespaces. For example, a macro that introduces references to non-core forms such as `read` or `eval` has the same problem if it is called in a context where those identifiers are rebound.

4.2 Enforcing Hygiene with Binder Renaming

As a first attempt at moving from a naive, non-hygienic macro expander to a hygienic macro expander, let us consider the introduced reference problem more carefully. As before, we start with the following code.

```
(let ([if #t]) (or #f if))
```

As shown before, in a naive expander, this code eventually expands to the following.

```
(let ([if #t])
  (let ([tmp #f]) (if tmp tmp if)))
```

To avoid this problem, we can rename bound variables to freshly generated identifiers as soon as they are exposed. Thus for example, after expanding the outer `let` and before expanding the `or` macro, we can generate a fresh identifier such as `if1` and replace all occurrences of `if` with `if1`. This results in the following code.

```
(let ([if1 #t]) (or #f if1))
```

When this fully expands, we get the following, which has avoided capturing the `if` used by the `or` macro.

```
(let ([if1 #t])
  (let ([tmp #f]) (if tmp tmp if1)))
```

Note that we do not rename bindings until they are fully expanded into k-syntax and thus have a known binding structure that cannot be modified by other macro calls.

4.3 Binder Renaming is Insufficient

Since binder renaming is sufficient to ensure introduced-reference hygiene, a natural question is whether it is sufficient to ensure introduced-binder hygiene. Unfortunately, it is not. To see this, consider the following example where we have placed the definition of the `or` macro inside the binding of `tmp`.

```
(let ([tmp #t])
  (letrec-syntax ([or
      ... #'(let ([tmp ...]) ...) ...])
    (or #f tmp)))
```

For the sake of brevity, we elide most of the `or` macro except for the `tmp` binding introduced by it.

After the outer `let` form expands and before the body of the `let` expands, we generate a fresh identifier for it to bind, and we rename all `tmp` references. This results in the following.

```
(let ([tmp1 #t])
  (letrec-syntax ([or
      ... #'(let ([tmp1 ...]) ...) ...])
    (or #f tmp1)))
```

Note that when `tmp` is renamed to `tmp1`, the `tmp` identifier introduced by the `or` macro is also renamed. If we continue expansion, we get the following.

```
(let ([tmp1 #t])
  (letrec-syntax ([or
      ... #'(let ([tmp1 ...]) ...) ...])
    (or #f tmp1)))
```

If we then expand the call to the `or` macro, we get the following.

```
(let ([tmp1 #t])
  (letrec-syntax ([or
      ... #'(let ([tmp1 ...]) ...) ...])
    (let ([tmp1 #f])
      (if tmp1 tmp1 (or tmp1)))))
```

At this point, we can see that even though we have been careful to rename any bindings discovered during the expansion process, introduced-binder hygiene has been violated. The `or` macro has introduced a binding for `tmp1` that shadows the outer binding to `tmp1` and incorrectly captures the reference to `tmp1` in `(or tmp1)`. The renaming of `tmp1` when the inner `let` is expanded will not help us as it will rename all occurrences of `tmp1` and not just those introduced by the `or` macro.

While we will come back to the use of binder renaming for ensuring hygiene, this example demonstrates that it does not ensure introduced-binder hygiene. To find a complete solution for hygiene we must seek something more.

4.4 Enforcing Hygiene with Gensym

For the moment, we set aside binder renaming and introduced-reference hygiene to consider hygiene violations due to introduced binders. This happens for example in Section 4.1.1, where the binding for `tmp` that is introduced by the `or` macro shadows the locally defined `tmp`.

The problem is that the introduced `tmp` happens to coincide with the `tmp` being used locally. We can avoid this by using a freshly generated identifier instead of `tmp`. Indeed, this is a widely used technique in systems with non-hygienic macro expanders. For example, if we have a `gensym-identifier` operator that generates fresh identifiers, we could define the `or` macro as follows.

```
(letrec-syntax ([or (lambda (stx)
  (syntax-case stx ()
    [(_)   #'#f]
    [(_ x) #'x]
    [(_ x . xs)
      (syntax-case (gensym-identifier #'tmp) ()
        [g #'(let ([g x])
               (if g g (or . xs)))])]))])
  ...)
```

Here we use `syntax-case` to introduce `g` as a pattern variable that gets replaced with the result of `gensym-identifier`. This ensures that the binder we introduce is unique and thus never conflicts with one from the input to the macro.

If we consider the example from Section 4.1.1 with this definition of `or`, the expansion proceeds thusly. The code immediately before expanding the `or` macro is as follows.

```
(let ([tmp #t]) (or #f tmp))
```

When `or` expands, `gensym-identifier` creates a fresh identifier, say `tmp1`, that is globally unique. The identifier `tmp1` does not conflict with `tmp`, and thus we avoid the hygiene violation due to an introduced binder. The fully expanded expression is then the following.

```
(let ([tmp #t])
  (let ([tmp1 #f]) (if tmp1 tmp1 tmp)))
```

Unfortunately, while using `gensym-identifier` allows us to avoid hygiene violations caused by introduced binders, it provides no help with introduced references. Consider again the following example from Section 4.1.2.

```
(let ([if #t]) (or #f if))
```

If we use `gensym-identifier` but not binder renaming, we get the following when we fully expand the `or` macro.

```
(let ([if #t])
  (let ([tmp1 #f]) (if tmp1 tmp1 if)))
```

The binder introduced by the macro uses the freshly generated identifier `tmp1`, but that is not the identifier causing the problem. It is the binding of `if` that is the problem. That binding is outside the reach of the `or` macro, and thus there is nothing the macro can do to avoid the captured variable. This example demonstrates that, on its own, using freshly generated identifiers for introduced binders is not a complete solution to hygiene either.

4.5 Binder Renaming with Gensym is Also Insufficient

Neither the solution in Section 4.2 nor the solution in Section 4.4 is sufficient on its own to ensure hygiene, but they handle complementary aspects. On the one hand, the solution in Section 4.2 handles introduced-reference hygiene. On the other hand, the solution in Section 4.4 handles introduced-binder hygiene. Thus we might consider combining the two. First, we rename identifiers bound by core forms that are discovered during macro expansion. Then, we require that any binders introduced by macro transformers be freshly generated.

Surprisingly, this does not work either. To see why, consider the question of whether `x` should be freshly generated by the `m` macro in the following code.

```
(let ([x 3])
  (let-syntax ([let-inc ...])
    (let-syntax ([m (lambda (stx)
      (syntax-case stx ()
        [(_ y) #'(let-inc x (* x y))]))])
      (m x))))
```

Suppose `let-inc` is defined as follows.

```
(let-syntax ([let-inc (lambda (stx)
  (syntax-case stx ()
    [(_ u v) #'(let ([u 2]) v)]))])
  ...)
```

Whatever is passed to `let-inc` as `u` becomes bound. Thus, the `x` introduced by `m` needs to be freshly generated because it is introducing a new binder. If we do not freshly generate `x`, then the expansion sequence would be the following, which results in the unintended capture of the `x` originally passed to `m` by the binding of `x` introduced by `m`.

```
(m x) ⇝ (let-inc x (* x x)) ⇝
(let ([x 2]) (* x x))
```

On the other hand, freshly generating the `x` introduced by `m` avoids this problem and results in the following expansion sequence where no unintended capture occurs.

```
(m x) ⇝ (let-inc x1 (* x1 x)) ⇝
(let ([x1 2]) (* x1 x))
```

While freshly generating `x` avoids unintended capture in this case, consider if `let-inc` were defined as the following.

```
(let-syntax ([let-inc (lambda (stx)
  (syntax-case stx ()
    [(_ u v) #'(+ 1 u)]))])
  ...)
```

With this definition, `let-inc` ignores its second argument and is not a binding form. If we freshly generate `x` like before, then we get the following expansion sequence.

```
(m x) ⇝ (let-inc x1 (* x1 x)) ⇝ (+ 1 x1)
```

This results in an error since `x1` is now a variable reference and there is no `x1` in scope. If, on the other hand, we do not freshly generate `x`, then we get the following expansion sequence, which behaves correctly.

```
(m x) ⇝ (let-inc x (* x x)) ⇝ (+ 1 x)
```

We might still hold out hope that a sufficiently careful programmer with sufficient information about the binding structure of `let-inc` could write macros like `m` that freshly generate identifiers exactly in those places where they should. However, that hope is dashed once we consider a definition of `let-inc` like the following.

```
(let-syntax ([let-inc (lambda (stx)
  (syntax-case stx ()
    [(_ u v) #'(let ([u (+ 1 u)]) v)]))])
  ...)
```

Here `let-inc` expands to both the increment and `let` binding from our previous definitions of `let-inc`. If `m` does not freshly generate `x`, then the expansion sequence is as follows.

```
(m x) ⇝ (let-inc x (* x x)) ⇝
(let ([x (+ 1 x)]) (* x x))
```

This is wrong as the binding of `x` in the resulting `let` improperly captures the `x` from the original call to `m`.

On the other hand, if `m` does freshly generate `x`, then the expansion sequence is as follows.

```
(m x) ⇝ (let-inc x1 (* x1 x)) ⇝
(let ([x1 (+ 1 x1)]) (* x1 x))
```

This is also wrong as there is a reference to the unbound variable `x1` in the right-hand side of the `let` binding.

With this definition of `let-inc`, we are stuck. The `m` macro *must* freshly generate the `x` that it introduces because of the `let` binding, but the `m` macro also *must not* freshly generate the `x` that it introduces because of the reference in the right-hand side of the `let` binding.

4.6 Enforcing Hygiene with Diatomic Identifiers

The problem in Section 4.5 is because some uses of `x` should use the freshly generated version while others should not. As long as identifiers are atomic symbols, we cannot have both. In order to resolve this, we split identifiers into a reference part and a binder part so they are diatomic instead of monatomic. We thus write identifiers as <*r* *b*> where *r* is the reference part and *b* is the binder part. The reference part is inhabited by a *reference atom* and is used if the identifier eventually ends up in a reference position while the binder part is inhabited by a *binder atom* and is used to determine what identifiers are captured when the identifier ends up in a binding position. These two sorts of atoms are from mutually disjoint domains, and a binder atom is never used in a position that expects a reference atom or vice versa. As a simplifying notation, we write `x` for `<x.r x.b>` when the diatomic nature of an identifier is not significant.

With these diatomic identifiers, `m` can expand with both non-freshly generated and freshly generated atoms in the reference and binder parts of the identifier, respectively. So for example, we could have the following expansion sequence.

```
(m <x.r x.b>) ⇝
(let-inc <x.r x1.b> (* <x.r x1.b> <x.r x.b>)) ⇝
(let ([<x.r x1.b> (+ 1 <x.r x1.b>)])
  (* <x.r x1.b> <x.r x.b>))
```

The binder part of the introduced identifier `<x.r x1.b>` is freshly generated but the reference part is not. In the right-hand side of the `let` binding, we use the reference part of the identifier, `x.r`, which was not freshly generated. On the other hand, in the left-hand side of the `let` binding, we use the binder part of the identifier, `x1.b`. Thus, the `let` does not capture the identifier in the body of the `let` that has `x.b` as a binder part.

To account for this change, we define expansion to operate with diatomic identifiers using the small-step relation in Figure 5. This relation is closed under congruence for k-syntax but not u-syntax. Thus expansion always operates at an outermost u-syntax. Also, though it may appear that the third clause (i.e., macro application) overlaps the other clauses, it does not. Binding forms always use fresh reference atoms, so *r* in the third clause never overlaps with reference-atom constants such as `lambda.r` in the other clauses.

Under this definition, variable references merely expand to their reference parts. For example, we have the following expansion.

```
<x.r x.b> ⇝ x.r
```

In effect, the reference part of an identifier represents what the identifier would refer to if there were no more binding forms discovered during expansion.

Binding forms are a bit more complicated. When expanding a binding form, we use the binder part of the identifier to determine what identifiers it captures. Once that is determined, those identifiers must be modified to refer to the newly expanded binding form. Since variable references expand to a reference atom, the binding forms in k-syntax also use reference atoms for their binding positions. Since the target of a variable reference is determined by the reference part of an identifier, we replace the reference parts of the captured identifiers with freshly generated reference atoms that we then also use in the k-syntax for the binding form. This replacement is implemented using the `subst` helper function defined in Figure 6. Its first clause replaces the reference part of identifiers that have a particular binder part, and the other clauses are merely a standard traversal over u-syntax.

Considering `lambda` again, we have the following expansion sequence.

```
(lambda (<x.r x.b>) (<y.r x.b> <x.r z.b>)) ⇝
(lambda (w.r) (<w.r x.b> <x.r z.b>)) ⇝ ... ⇝
(lambda (w.r) (w.r x.r))
```

As with binder renaming, we use `w.r`, a freshly generated reference atom for the binding form. The `y.r` in `<y.r x.b>` is changed to `w.r` since the binder part of that identifier, `x.b`, matches the binder part of the identifier, `<x.r x.b>`, that is in the binding position for the `lambda` form. In other words, that identifier should be captured by the `lambda`. On the other hand, the identifier `<x.r z.b>` is left alone as its binder part does not match `x.b`.

4.7 Diatomic Identifiers are Sufficient

Returning to the final version of the `let-inc` macro from Section 4.5, we can now see how all of this fits together. First, we start with the following expression.

```
(m <x.r x.b>)
```

When `m` expands, it freshly generates the binder portion of any identifiers that it introduces, and we thus get the following.

```
(let-inc <x.r x1.b> (* <x.r x1.b> <x.r x.b>))
```

When `let-inc` expands, the `<x.r x1.b>` identifier is placed in both the left-hand and right-hand sides of the `let` binding, which results in the following.

```
(let ([<x.r x1.b> (+ 1 <x.r x1.b>)])
  (* <x.r x1.b> <x.r x.b>))
```

At this point, we expand the `let` form. This generates a fresh reference atom `x2.r`, and we use `subst` to replace the reference part of any identifiers in the body that have the same binder part as the bound identifier. This binder part is `x1.b` and was freshly generated by the `m` macro, so the only identifier captured is the first argument to `*`, and we get the following.

```
(let ([x2.r (+ 1 <x.r x1.b>)])
  (* <x2.r x1.b> <x.r x.b>))
```

We then expand the right-hand side of the `let` binding. Since the identifier in it is a variable reference, it expands to its reference part to produce the following.

```
(let ([x2.r (+ 1 x.r)])
  (* <x2.r x1.b> <x.r x.b>))
```

$$
\begin{aligned}
\underline{c} &\rightsquigarrow c \\
\underline{\texttt{<}r\ b\texttt{>}} &\rightsquigarrow r \\
\underline{(\texttt{<}r\ b\texttt{>}\ \overrightarrow{args})} &\rightsquigarrow f\ \underline{(\texttt{<}r\ b\texttt{>}\ \overrightarrow{args})} \text{ where } f \text{ is the currently in-scope transformer bound to } r \\
&\qquad \text{and all identifiers introduced by } f \text{ have freshly generated binder parts} \\
\underline{(\texttt{<lambda.r}\ b_0\texttt{>}\ (\texttt{<}r\ b\texttt{>})\ \overrightarrow{body})} &\rightsquigarrow (\texttt{lambda}\ (\overrightarrow{r'})\ \overrightarrow{\left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{body}\right)}) \\
\underline{(\texttt{<if.r}\ b_0\texttt{>}\ test\ true\ false)} &\rightsquigarrow (\texttt{if}\ \underline{test}\ \underline{true}\ \underline{false}) \\
\underline{(\texttt{<let.r}\ b_0\texttt{>}\ \overrightarrow{[\texttt{<}r\ b\texttt{>}\ rhs]})\ \overrightarrow{body})} &\rightsquigarrow (\texttt{let}\ (\overrightarrow{[r'\ \underline{rhs}]})\ \overrightarrow{\left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{body}\right)}) \\
\underline{(\texttt{<letrec.r}\ b_0\texttt{>}\ \overrightarrow{[\texttt{<}r\ b\texttt{>}\ rhs]})\ \overrightarrow{body})} &\rightsquigarrow (\texttt{letrec}\ (\overrightarrow{[r'\ \left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{rhs}\right)]})\ \overrightarrow{\left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{body}\right)}) \\
\underline{(\texttt{<let-syntax.r}\ b_0\texttt{>}\ \overrightarrow{[\texttt{<}r\ b\texttt{>}\ rhs]})\ \overrightarrow{body})} &\rightsquigarrow (\texttt{let-syntax}\ (\overrightarrow{[r'\ \underline{rhs}]})\ \overrightarrow{\left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{body}\right)}) \\
\underline{(\texttt{<letrec-syntax.r}\ b_0\texttt{>}\ \overrightarrow{[\texttt{<}r\ b\texttt{>}\ rhs]})\ \overrightarrow{body})} &\rightsquigarrow (\texttt{letrec-syntax}\ (\overrightarrow{[r'\ \left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{rhs}\right)]})\ \overrightarrow{\left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{body}\right)}) \\
\underline{(\texttt{<syntax.r}\ b_0\texttt{>}\ stx)} &\rightsquigarrow (\texttt{syntax}\ \underline{stx}) \\
\underline{(\texttt{<syntax-case.r}\ b_0\texttt{>}\ stx\ lits\ \overrightarrow{[pat\ rhs]})} &\rightsquigarrow (\texttt{syntax-case}\ stx\ lits\ \overrightarrow{[pat'\ \left(\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ \underline{rhs}\right)]}) \\
&\qquad \text{where } \overrightarrow{b} \text{ is the binder parts of the non-literal identifiers in the} \\
&\qquad \text{corresponding } \overrightarrow{pat}, \text{ and } pat' \text{ is } pat \text{ with each identifier } \texttt{<}r\ b\texttt{>} \\
&\qquad \text{replaced by a corresponding element of } \overrightarrow{r'} \\
\underline{(fun\ \overrightarrow{args})} &\rightsquigarrow (\underline{fun}\ \underline{\overrightarrow{args}}) \text{ if none of the above cases apply}
\end{aligned}
$$

Figure 5. Macro expansion with diatomic identifiers. In this figure, we let $\overrightarrow{r'}$ be fresh reference atoms and $\mathsf{subst}\ \overrightarrow{b}\ \overrightarrow{r'}\ e$ be the application of subst to e for each pair of corresponding elements from $\overrightarrow{b}$ and $\overrightarrow{r'}$. Closed under congruence for k-syntax.

$$
\begin{aligned}
\mathsf{subst}\ b\ r\ \underline{\texttt{<}r'\ b'\texttt{>}} &= \underline{\texttt{<}r\ b'\texttt{>}} \quad \text{if } b = b' \\
\mathsf{subst}\ b\ r\ \underline{\texttt{<}r'\ b'\texttt{>}} &= \underline{\texttt{<}r'\ b'\texttt{>}} \quad \text{if } b \neq b' \\
\mathsf{subst}\ b\ r\ \underline{c} &= \underline{c} \quad \text{if } c \text{ is a constant} \\
\mathsf{subst}\ b\ r\ \underline{()} &= \underline{()} \\
\mathsf{subst}\ b\ r\ \underline{(stx_1\ .\ stx_2)} &= \underline{(\ \mathsf{subst}\ b\ r\ stx_1\ .\ \mathsf{subst}\ b\ r\ stx_2)}
\end{aligned}
$$

Figure 6. Definition of subst

All the remaining identifiers are references, so the last bit of expansion results in the following.

```
(let ([x2.r (+ 1 x.r)])
  (* x2.r x.r))
```

Note that identifiers for `x` that are introduced by `m` can all capture each other because they have the same freshly generated binder parts, but those introduced identifiers cannot capture anything else as their binder parts are distinct from the binder parts of any other identifiers.

4.8 Summary

At this point, we have an algorithm that completely handles hygiene. We summarize the major points as follows.

First, identifiers are diatomic and contain both a reference part and a binder part. The reference part represents what the identifier refers to if it ends up in a reference position. The binder part represents which identifiers can capture each other.

Second, when expanding a binding form, we use the binder part of an identifier to determine what identifiers are captured by it. A freshly generated reference atom is created for the binding form, and the reference parts of the captured identifiers are then replaced with this reference atom so that those identifiers now refer to that binding form.

Finally, we require that the binder parts of any identifiers introduced by a macro call be freshly generated. This ensures that the only identifiers they can capture are ones that were introduced in that macro call.

If we go back to the examples of hygiene violation in the `or` macro with this algorithm, we get the expansion sequence in Figure 7, which shows the enforcement of both introduced-reference hygiene (with the `if` binding) and introduced-binder hygiene (with the `tmp` binding).

5. The Mathematics of Hygiene

The algorithm in Section 4 provides useful intuitions about hygiene but does not define what hygiene actually is. We now turn to this question and consider the mathematical properties that characterize hygiene. We do this in terms of permutations, support and equivariance from nominal logic (Gabbay and Pitts 2002), so we review these before turning to the main question of characterizing hygiene. In the process, we expose observed-binder hygiene as a third way hygiene can be violated that is not often discussed but is implicitly averted by traditional hygiene algorithms.

5.1 Nominal Logic

5.1.1 Permutations

In nominal logic, permutations are invertible, total mappings over atoms. We write $(\alpha \leftrightarrow \beta)$ for the permutation that maps the atoms

```
(let ([<tmp.r tmp.b> #t])
  (let ([<if.r if.b> 2]) (or #f <tmp.r tmp.b>))) ⇝
(let ([p.r #t])
  (let ([<if.r if.b> 2]) (or #f <p.r tmp.b>))) ⇝
(let ([p.r #t])
  (let ([<if.r if.b> 2]) (or #f <p.r tmp.b>))) ⇝
(let ([p.r #t])
  (let ([q.r 2]) (or #f <p.r tmp.b>))) ⇝
(let ([p.r #t])
  (let ([q.r 2]) (or #f <p.r tmp.b>))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([<tmp.r s.b> #f])
      (<if.r if.b> <tmp.r s.b>
        <tmp.r s.b> (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f])
      (<if.r if.b> <t.r s.b>
        <t.r s.b> (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f])
      (<if.r if.b> <t.r s.b>
        <t.r s.b> (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f])
      (if <t.r s.b>
        <t.r s.b> (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f])
      (if t.r <t.r s.b> (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f])
      (if t.r t.r (or <p.r tmp.b>))))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f]) (if t.r t.r <p.r tmp.b>)))) ⇝
(let ([p.r #t])
  (let ([q.r 2])
    (let ([t.r #f]) (if t.r t.r p.r))))
```

Figure 7. Example hygienic expansion of the `or` macro

Identifiers and Constants

$\pi \bullet \texttt{<}r\ b\texttt{>} = \texttt{<}\pi \bullet r\ \ \pi \bullet b\texttt{>}$

$\pi \bullet c = c$

U-Syntax

$\pi \bullet \underline{i} = \underline{\pi \bullet i}$

$\pi \bullet \underline{c} = \underline{\pi \bullet c}$

$\pi \bullet \underline{()} = \underline{()}$

$\pi \bullet \underline{(stx_1\ .\ stx_2)} = \underline{(\pi \bullet stx_1\ .\ \pi \bullet stx_2)}$

K-Syntax

$\pi \bullet \texttt{(lambda (}\overrightarrow{i}\texttt{)}\ \overrightarrow{body}\texttt{)} = \texttt{(lambda (}\overrightarrow{\pi \bullet i}\texttt{)}\ \overrightarrow{\pi \bullet body}\texttt{)}$

$\pi \bullet \texttt{(}fun\ \overrightarrow{args}\texttt{)} = \texttt{(}\pi \bullet fun\ \overrightarrow{\pi \bullet args}\texttt{)}$

$\pi \bullet \texttt{(if}\ test\ true\ false\texttt{)}$
$= \texttt{(if}\ \pi \bullet test\ \pi \bullet true\ \pi \bullet false\texttt{)}$

$\pi \bullet \texttt{(let (}\overrightarrow{\texttt{[}i\ rhs\texttt{]}}\texttt{)}\ \overrightarrow{body}\texttt{)}$
$= \texttt{(let (}\overrightarrow{\texttt{[}\pi \bullet i\ \pi \bullet rhs\texttt{]}}\texttt{)}\ \overrightarrow{\pi \bullet body}\texttt{)}$

$\pi \bullet \texttt{(letrec (}\overrightarrow{\texttt{[}i\ rhs\texttt{]}}\texttt{)}\ \overrightarrow{body}\texttt{)}$
$= \texttt{(letrec (}\overrightarrow{\texttt{[}\pi \bullet i\ \pi \bullet rhs\texttt{]}}\texttt{)}\ \overrightarrow{\pi \bullet body}\texttt{)}$

$\pi \bullet \texttt{(let-syntax (}\overrightarrow{\texttt{[}i\ rhs\texttt{]}}\texttt{)}\ \overrightarrow{body}\texttt{)}$
$= \texttt{(let-syntax (}\overrightarrow{\texttt{[}\pi \bullet i\ \pi \bullet rhs\texttt{]}}\texttt{)}\ \overrightarrow{\pi \bullet body}\texttt{)}$

$\pi \bullet \texttt{(letrec-syntax (}\overrightarrow{\texttt{[}i\ rhs\texttt{]}}\texttt{)}\ \overrightarrow{body}\texttt{)}$
$= \texttt{(letrec-syntax (}\overrightarrow{\texttt{[}\pi \bullet i\ \pi \bullet rhs\texttt{]}}\texttt{)}\ \overrightarrow{\pi \bullet body}\texttt{)}$

$\pi \bullet \texttt{(syntax}\ stx\texttt{)} = \texttt{(syntax}\ \pi \bullet stx\texttt{)}$

$\pi \bullet \texttt{(syntax-case}\ ast\ lits\ \overrightarrow{\texttt{[}pat\ rhs\texttt{]}}\texttt{)}$
$= \texttt{(syntax-case}\ \pi \bullet ast\ \pi \bullet lits\ \overrightarrow{\texttt{[}\pi \bullet pat\ \pi \bullet rhs\texttt{]}}\texttt{)}$

Patterns

$\pi \bullet \texttt{\_} = \texttt{\_}$

$\pi \bullet \texttt{()} = \texttt{()}$

$\pi \bullet \texttt{(}pat_1\ .\ pat_2\texttt{)} = \texttt{(}\pi \bullet pat_1\ .\ \pi \bullet pat_2\texttt{)}$

Figure 8. Lifted permutation application

α and β to each other while leaving all other atoms unchanged and $(\vec{\alpha} \leftrightarrow \vec{\beta})$ for the permutation that maps corresponding elements of $\vec{\alpha}$ and $\vec{\beta}$ to each other while leaving all other atoms unchanged. We write the application of a permutation π to an atom α as $\pi \bullet \alpha$. Finally, since every permutation is invertible, we write π^{-1} for the inverse of π. We lift these permutations from applying over individual atoms to applying over both k-syntax and u-syntax by a straightforward homomorphism as shown in Figure 8. Note that we apply the permutation even to bound variables. If we do not, the permutation could result in unintended captures. For example, consider the permutation $(\alpha \leftrightarrow \beta)$ applied to the term `(lambda (`α`) `β`)`. If we do not permute the variable binding, α, then we cannot permute the body, β, without causing unintended variable capture.

Finally, nominal logic lifts permutations to apply to functions. If we let $\circ$ compose permutations with functions, then the application of a permutation π to a function f is defined by the equation $\pi \bullet f = \pi \circ f \circ \pi^{-1}$. To see the intuition behind this, consider if we have the intentional representation of f. Applying π to f should apply π to all the atoms in the implementation of f. Thus, if f always returns some constant c, then $\pi \bullet f$ should return $\pi \bullet c$. We achieve this effect by applying π to the output of f. However, we have to be careful about inputs to f that end up in the output. For example, if f is the identity function, then f contains no atoms and applying π to f should not change it. Applying π to the output of f would incorrectly rename elements of the input that end up in the output. We counteract this by applying π^{-1} to the input. Any inputs that end up in the output will have the π canceled out by the π^{-1} and thus be unchanged.

5.1.2 Support

From this notion of permutation, nominal logic defines the concept of the support of an object. Intuitively, the support of an object is the set of atoms that "occur free" in it. It is usually defined over a quotiented domain that equates alpha equivalent terms, but for the sake of clarity we make uses of alpha equivalence explicit. If $\simeq$ is

$$\texttt{(lambda (}\overrightarrow{i}\texttt{)}\ \overrightarrow{body}\texttt{)} \simeq \texttt{(lambda (}\overrightarrow{i'}\texttt{)}\ \overrightarrow{body'}\texttt{)} \quad \text{if } \forall^\infty \overrightarrow{i''}.\ body \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} body'$$

$$\texttt{(let ([}\overrightarrow{i\ rhs}\texttt{])}\ \overrightarrow{body}\texttt{)} \simeq \texttt{(let ([}\overrightarrow{i'\ rhs'}\texttt{])}\ \overrightarrow{body'}\texttt{)} \quad \text{if } \overrightarrow{rhs} \simeq \overrightarrow{rhs'} \text{ and } \forall^\infty \overrightarrow{i''}.\ body \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} body'$$

$$\texttt{(letrec ([}\overrightarrow{i\ rhs}\texttt{])}\ \overrightarrow{body}\texttt{)} \simeq \texttt{(letrec ([}\overrightarrow{i'\ rhs'}\texttt{])}\ \overrightarrow{body'}\texttt{)} \quad \text{if } \forall^\infty \overrightarrow{i''}.\ rhs \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} rhs' \text{ and } body \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} body'$$

$$\texttt{(let-syntax ([}\overrightarrow{i\ rhs}\texttt{])}\ \overrightarrow{body}\texttt{)} \simeq \texttt{(let-syntax ([}\overrightarrow{i'\ rhs'}\texttt{])}\ \overrightarrow{body'}\texttt{)} \quad \text{if } \overrightarrow{rhs} \simeq \overrightarrow{rhs'} \text{ and } \forall^\infty \overrightarrow{i''}.\ body \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} body'$$

$$\texttt{(letrec-syntax ([}\overrightarrow{i\ rhs}\texttt{])}\ \overrightarrow{body}\texttt{)} \simeq \texttt{(letrec-syntax ([}\overrightarrow{i'\ rhs'}\texttt{])}\ \overrightarrow{body'}\texttt{)} \quad \text{if } \forall^\infty \overrightarrow{i''}.\ rhs \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} rhs' \text{ and } body \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} body'$$

$$\texttt{(syntax-case } s\ \texttt{(}\overrightarrow{l}\texttt{)}\ \texttt{[}\overrightarrow{pat\ rhs}\texttt{])} \simeq \texttt{(syntax-case } s'\ \texttt{(}\overrightarrow{l'}\texttt{)}\ \texttt{[}\overrightarrow{pat'\ rhs'}\texttt{])} \quad \text{if } s \simeq s' \text{ and } \overrightarrow{l} \simeq \overrightarrow{l'} \text{ and } \forall^\infty \overrightarrow{i''}.\ pat \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} pat' \text{ and } rhs \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} rhs'$$

where i and i' are the non-literal identifiers in the corresponding pat and pat', respectively

Figure 9. Alpha equivalence for k-syntax. We write $t_1 \overset{\overrightarrow{i},\overrightarrow{i'},\overrightarrow{i''}}{\simeq} t_2$ as an abbreviation for $\left(\overrightarrow{i} \leftrightarrow \overrightarrow{i''}\right) \bullet t_1 \simeq \left(\overrightarrow{i'} \leftrightarrow \overrightarrow{i''}\right) \bullet t_2$ and use $\forall^\infty$ for "for all but finitely many". Closed under reflexivity, symmetry, transitivity and congruence.

the alpha equivalence relation, the support of a term t is the smallest finite set, $\mathsf{supp}\, t$, such that the following holds.

$$\forall \alpha, \beta \notin \mathsf{supp}\, t.\ (\alpha \leftrightarrow \beta) \bullet t \simeq t$$

As an example of this, consider the term `(lambda (`α`) (`α β`))`. Swapping the atom α with any atom other than β results in an alpha-equivalent term. Thus α is *not* in the support of that term. On the other hand, any atom that we swap with β results in a non-alpha-equivalent term. Thus β *is* in the support of that term.

5.1.3 Equivariance

Finally, nominal logic defines equivariance as holding for a function iff its support is empty. Thus, f is equivariant iff any of the following equivalent equations hold, where alpha equivalence over functions is interpreted extensionally (i.e., they take alpha equivalent terms to alpha equivalent terms).

$$\begin{aligned} & \mathsf{supp}\, f = \emptyset \\ \Leftrightarrow\ & \forall \pi.\pi \bullet f \simeq f \\ \Leftrightarrow\ & \forall \pi.\pi \circ f \circ \pi^{-1} \simeq f \\ \Leftrightarrow\ & \forall \pi.\pi \circ f \simeq f \circ \pi \end{aligned}$$

5.2 Alpha Equivalence

As already mentioned, hygiene is essentially lexical scoping at the macro level. Thus it would be natural to characterize hygiene by requiring alpha-equivalent terms to expand to alpha-equivalent terms. However, since macro transformers can be any Turing computable function, we cannot in general know the binding structure of u-syntax and thus cannot define a non-trivial alpha-equivalence relation over them. For example, we may have a macro that uses a complicated function of its input to determine whether or not to expand to a `let`. As another example, even a simple `let` may not actually be a binding form if it is nested in some other macro call as the outer macro may rearrange the u-syntax so the `let` becomes some other form.

Fortunately, while we cannot know the binding structure of u-syntax, we know the binding structure of the parts of the code already expanded into k-syntax. For example, after one step of expansion, the example in Section 4.2 becomes the following.

```
(let ([if.r #t]) (or #f <if.r if.b>))
```

Now that the `let` form has expanded into k-syntax, it cannot expand further, and its binding structure is fixed. We do not know the binding structure of its body, though, because the body is u-syntax that may be further expanded.

Nevertheless, we can use a weakened notion of alpha equivalence that accounts for the binding structure manifest in k-syntax but assumes no non-trivial equivalences in u-syntax. This weakened notion of alpha equivalence is a conservative approximation of whatever the actual binding structure may be and is sufficient to formally define when hygiene violations occur due to introduced references. In Figure 9 we formally define this alpha equivalence. It is entirely standard and behaves as usual for alpha equivalence except that it has no equivalences for u-syntax other than reflexivity. For example, the above expression is alpha equivalent to the following.

```
(let ([x.r #t]) (or #f <x.r if.b>))
```

If these two expressions do not expand to alpha-equivalent expressions, then introduced-reference hygiene is violated. We formally define this as follows.

Criterion 1 (Reference Hygiene)**.** *A macro expansion step is hygienic with regard to references iff for any partially expanded expressions e_1, e_2 and e_1' where $e_1 \simeq e_2$ and $e_1 \rightsquigarrow e_1'$, there exists an e_2' such that $e_1' \simeq e_2'$ and $e_2 \rightsquigarrow e_2'$. A macro system is hygienic with regard to references iff all its expansion steps are hygienic with regard to references.*

Since this criterion is independent of the details of the expansion algorithm, we are free to choose from any of the well-known methods for manipulating terms while respecting alpha equivalence. In Section 4.2, we ensured this property by always freshly generating bound reference atoms, but we can also use the binding forms provided by nominal logic.

Expansion

$$\underline{\texttt{(<fresh.r } b_0\texttt{> (} \overrightarrow{b}\texttt{) } body\texttt{)}} \rightsquigarrow \texttt{(fresh (} \overrightarrow{b}\texttt{) } \underline{body}\texttt{)}$$

Permutation

$$\pi \bullet \texttt{(fresh (} \overrightarrow{b}\texttt{) } body\texttt{)} = \texttt{(fresh (}\overrightarrow{\pi \bullet b}\texttt{) } \pi \bullet body\texttt{)}$$

Alpha Equivalence

$$\texttt{(fresh (} \overrightarrow{b}\texttt{) } body\texttt{)} \simeq \texttt{(fresh (} \overrightarrow{b'}\texttt{) } body'\texttt{)} \quad \text{if } \forall^{\infty} \overrightarrow{b''}.\, body \overset{\vec{b},\vec{b'},\vec{b''}}{\simeq} body'$$

$$\texttt{(fresh () } body\texttt{)} \simeq body$$

$$\texttt{(fresh (} \overrightarrow{b}\texttt{) } body\texttt{)} \simeq \texttt{(fresh (} \overrightarrow{b} \cup \overrightarrow{b_0}\texttt{) } body\texttt{)} \quad \text{if } b_0 \cap \mathsf{supp}\, body = \emptyset$$

Figure 10. Definitions of expansion, permutation and alpha equivalence for `fresh`

Note that, just like binder renaming, the Reference Hygiene Criterion ensures reference hygiene but not binder hygiene. This is due to the fact that we do not know the binding structure of u-syntax and thus have no non-trivial equivalences for it. Tackling that problem is the subject of the remainder of this section.

5.3 Fresh Identifiers

As a first step toward handling binder hygiene, recall that the algorithm in Section 4.4 requires that macros freshly generate the binder atoms in any identifiers they introduce. In order to mathematically model this, we introduce a scoping construct for binder atoms. We write this as `(fresh (`$\vec{b}$`) `$body$`)` where the binder atoms in $\vec{b}$ are introduced into the scope of $body$. Duplicates are not allowed in $\vec{b}$ and order is irrelevant, so we treat $\vec{b}$ as a set.

In Figure 10 we define expansion, permutation and alpha equivalence for this form. In addition to the usual alpha equivalences, we may omit atoms bound by `fresh` that are not in the support of its body, and if `fresh` does not bind any atoms, we can replace it with just its body. We include these equivalences because the sole purpose of `fresh` is to bring the elements of $\vec{b}$ into the scope of its body. If they are not used, then we do not need `fresh` to bring them into scope. Just as with other binding constructs, operations such as substitution must rename the binders in `fresh` to avoid unintended capture.

Now instead of using `gensym-identifier`, we make macros return a `fresh` wrapped around a syntax object. For example, the `or` macro expands as follows.

```
(or #f <tmp.r tmp.b>) ⇝
(fresh (tmp1.b)
  (let ([<tmp.r tmp1.b> #f])
    (if <tmp.r tmp1.b>
      <tmp.r tmp1.b> <tmp.r tmp.b>)))
```

This is identical to when we used `gensym-identifier` except that `tmp1.b` now comes from a locally declared scope instead of being a global gensym.

5.4 Equivariance

A macro transformer should use `fresh` for any introduced binder atoms, but in order to formally require this, we need to define what counts as an introduced binder atom. For example, if a transformer returns `<x.r y.b>` when passed `<x.r x.b>` as input, clearly `y.b` was introduced by the macro and should be included in the atoms scoped by `fresh`. However, what if a macro returns `<x.r x.b>` when passed `<x.r x.b>` as input? The macro transformer might be the identity function. In that case, `x.b` is not introduced by the macro as it comes from the input. On the other hand, the macro transformer could be the constant function that always returns `<x.r x.b>`. In that case, `x.b` *is* introduced by the macro. We just happen to have chosen an input that coincides with the output.

In order to handle this, note that the binder atoms introduced by a transformer are in its support. Thus we can force macro transformers to use `fresh` for introduced binder atoms by requiring that their supports contain no binder atoms. In other words, we require that macro transformers be equivariant with respect to binder atoms. This allows us to distinguish between transformers that return `<x.r x.b>` when passed `<x.r x.b>` because they are an identity function and those that do so because they are a constant function. In the former case, applying the permutation $(\texttt{x.b} \leftrightarrow \texttt{y.b})$ to the function before applying it to `<x.r x.b>` results in the following.

$$\begin{aligned}
&((\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet f)\ \underline{\texttt{<x.r x.b>}} \\
&= ((\texttt{x.b} \leftrightarrow \texttt{y.b}) \circ f \circ (\texttt{y.b} \leftrightarrow \texttt{x.b}))\ \underline{\texttt{<x.r x.b>}} \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet (f\ ((\texttt{y.b} \leftrightarrow \texttt{x.b}) \bullet \underline{\texttt{<x.r x.b>}})) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \left(f\ \underline{\texttt{<x.r y.b>}}\right) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \underline{\texttt{<x.r y.b>}}) \\
&= \underline{\texttt{<x.r x.b>}}
\end{aligned}$$

On the other hand, doing the same on a constant function results in the following.

$$\begin{aligned}
&((\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet f)\ \underline{\texttt{<x.r x.b>}} \\
&= ((\texttt{x.b} \leftrightarrow \texttt{y.b}) \circ f \circ (\texttt{y.b} \leftrightarrow \texttt{x.b}))\ \underline{\texttt{<x.r x.b>}} \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet (f\ ((\texttt{y.b} \leftrightarrow \texttt{x.b}) \bullet \underline{\texttt{<x.r x.b>}})) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \left(f\ \underline{\texttt{<x.r y.b>}}\right) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \underline{\texttt{<x.r x.b>}} \\
&= \underline{\texttt{<x.r y.b>}}
\end{aligned}$$

In the former case, the transformer function is equivariant with respect to binder atoms, and applying the permutation does not change the output. In the latter case, however, the transformer is not equivariant, and the permutation changes the output.

If a transformer needs to introduce a binder atom, as is the case with `tmp` in the `or` macro, then in order to remain equivariant, it must use `fresh`. For example, instead of returning the constant `<x.r x.b>`, we can have f return `(fresh (x.b) <x.r x.b>)`. In that case, modulo alpha equivalence, applying the permutation to f does not change it. This is seen in the following.

$$\begin{aligned}
&((\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet f)\ \underline{\texttt{<x.r x.b>}} \\
&= ((\texttt{x.b} \leftrightarrow \texttt{y.b}) \circ f \circ (\texttt{y.b} \leftrightarrow \texttt{x.b}))\ \underline{\texttt{<x.r x.b>}} \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet (f\ ((\texttt{y.b} \leftrightarrow \texttt{x.b}) \bullet \underline{\texttt{<x.r x.b>}})) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \left(f\ \underline{\texttt{<x.r y.b>}}\right) \\
&= (\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \texttt{(fresh (x.b) } \underline{\texttt{<x.r x.b>}}\texttt{)} \\
&= \texttt{(fresh (y.b) } \underline{\texttt{<x.r y.b>}}\texttt{)} \\
&\simeq \texttt{(fresh (x.b) } \underline{\texttt{<x.r x.b>}}\texttt{)}
\end{aligned}$$

This then gives rise to the following criterion defining hygiene with respect to binders.

Criterion 2 (Binder Hygiene)**.** *A macro transformer is hygienic with respect to binders iff it is equivariant with respect to binder atoms. A macro system is hygienic with respect to binders if all its transformers are hygienic with respect to binders.*

5.5 Automating Hygiene

While the Binder Hygiene Criterion provides a formal description of hygiene with respect to binders, it still places the burden of ensuring hygiene on the macro author. While a mechanical proof checker or a type system like in Herman and Wand (2008) or Herman (2010) could verify that a particular transformer is equivariant, we can also have the macro system automatically enforce binder hygiene. A simple way to do this is to use `fresh` to capture any binder atoms in the support of a transformer. Thus we have the following definition where $\mathsf{supp_b}\ f$ is the support of f with respect to binder atoms.

Definition 3 (Automated Hygiene)**.** Given a transformer f, let the hygienic version of that transformer be the following.

$$\mathsf{hyg}\ f = \texttt{(fresh (}\ \mathsf{supp_b}\ f\texttt{)}\ f\texttt{)}$$

Effectively this assumes f is supposed to be equivariant and uses `fresh` to clean up any binder atoms that the macro author left in the support of f.

In order to apply the result of hyg to an argument, we lift `fresh` to be a function where application of `fresh` is as follows.

Definition 4 (Fresh Application)**.** If $s \cap \mathsf{supp_b}\ x = \emptyset$, then

$$\texttt{(fresh (}s\texttt{)}\ f\texttt{)}\ x = \texttt{(fresh (}s\texttt{)}\ (f\ x)\texttt{)}$$

When s and the support of x are not disjoint, we use alpha-equivalence to rename s to not conflict with x before applying this definition.

This definition trivially ensures the equivariance of hyg as shown in the following theorem.

Theorem 5 (Equivariance of Automated Hygiene)**.** *For any f with a finite support with respect to binder atoms,* hyg f *is equivariant with respect to binder atoms.*

Proof. Trivial. The hyg operator produces a `fresh` that captures the binder atoms in the support of f, and thus the support of hyg f is empty. □

If the macro system applies hyg to all macro transformers, then all macros are automatically hygienic. For example, consider again the macro that returns the constant `<x.r x.b>`. The support of this with respect to binder atoms is the singleton containing `x.b`. If we apply hyg to this, we get `(fresh (x.b) f)`. Applying this to `<x.r x.b>` then results in the following.

$$\begin{aligned}
&\texttt{(fresh (x.b) } f\texttt{) <x.r x.b>} \simeq \\
&\texttt{(fresh (x1.b) (x.b} \leftrightarrow \texttt{x1.b)} \bullet f\texttt{) <x.r x.b>} = \\
&\texttt{(fresh (x1.b) (((x.b} \leftrightarrow \texttt{x1.b)} \bullet f\texttt{) <x.r x.b>)) } = \\
&\texttt{(fresh (x1.b) (x.b} \leftrightarrow \texttt{x1.b)} \bullet \texttt{<x.r x.b>} = \\
&\texttt{(fresh (x1.b) <x.r x1.b>)}
\end{aligned}$$

Of course, computing the support of a function is undecidable in general. To get a practical implementation, we can instead use the following definition.

Definition 6 (Automated Hygiene, Alternative)**.** Given a transformer f and a syntax object x, where π is some minimal permutation mapping each element of $\mathsf{supp_b}\ x$ to an atom not in the support of f or x, let the application of the hygienic version of that transformer be as follows.

$$(\mathsf{hyg}\ f)\ x = \texttt{(fresh (}\ \mathsf{supp_b}\ ((\pi \bullet f)\ x) - \mathsf{supp_b}\ x\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)}$$

Since we can easily compute the support of a syntax object, this definition is practical to implement but behaves the same as Definition 3 as shown in the following theorem.

Theorem 7 (Equivalance of Automated Hygiene Definitions)**.** *Definition 3 and Definition 6 are equivalent definitions of* hyg *up to alpha equivalence.*

Proof. Let $s_1 = \mathsf{supp_b}\ f - \mathsf{supp_b}\ x$, $s_2 = \mathsf{supp_b}\ f \cap \mathsf{supp_b}\ x$ and $s_3 = \mathsf{supp_b}\ x - \mathsf{supp_b}\ f$. Also let π be defined as in Definition 6. Finally, let $v = \mathsf{supp_b}\ ((\pi \bullet f)\ x)$. Note that $\pi \bullet s_1$, $\pi \bullet s_2$, s_2 and s_3 are all disjoint and $v \subseteq \mathsf{supp_b}\ (\pi \bullet f) \cup \mathsf{supp_b}\ x = \pi \bullet s_1 \cup \pi \bullet s_2 \cup s_2 \cup s_3$. We then have the following equalities.

$$\begin{aligned}
&\texttt{(fresh (}\ \mathsf{supp_b}\ f\texttt{)}\ f\texttt{)}\ x \\
=\ &\texttt{(fresh (}s_1 \cup s_2\texttt{)}\ f\texttt{)}\ x \\
\simeq\ &\texttt{(fresh (}\pi \bullet (s_1 \cup s_2)\texttt{)}\ \pi \bullet f\texttt{)}\ x \\
=\ &\texttt{(fresh (}\pi \bullet (s_1 \cup s_2)\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)} \\
=\ &\texttt{(fresh (}(\pi \bullet s_1) \cup (\pi \bullet s_2)\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)} \\
\simeq\ &\texttt{(fresh (}v \cap (\pi \bullet s_1 \cup \pi \bullet s_2)\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)} \\
=\ &\texttt{(fresh (}v - (s_2 \cup s_3)\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)} \\
=\ &\texttt{(fresh (}\mathsf{supp_b}\ ((\pi \bullet f)\ x) - \mathsf{supp_b}\ x\texttt{)}\ ((\pi \bullet f)\ x)\texttt{)}
\end{aligned}$$ □

Note that different choices of π in Definition 6 may result in different results, but as a corollary of Theorem 7 those results are all alpha equivalent to each other.

5.6 Observed-Binder Hygiene

Alert readers may notice that the Binder Hygiene Criterion affects not just binder atoms introduced by transformers but also binder atoms *observed* by macro transformers. For example, consider the following macro transformer where `bound-identifier=?` extracts the binder atoms from two identifiers and tests if they are equal.

```
(let-syntax ([obs (lambda (stx)
  (if (bound-identifier=? stx #'<x.r x.b>)
    #'1 #'2))])
  ...)
```

In a naive expansion system, if the input to this transformer is `<x.r x.b>`, then the output is `1`. However, if we apply the permutation $(\texttt{x.b} \leftrightarrow \texttt{y.b})$ to the transformer, then the result is the following.

$$\begin{aligned}
&((\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \texttt{obs})\ \texttt{<x.r x.b>} \\
=\ &((\texttt{x.b} \leftrightarrow \texttt{y.b}) \circ \texttt{obs} \circ (\texttt{y.b} \leftrightarrow \texttt{x.b}))\ \texttt{<x.r x.b>} \\
=\ &(\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet (\texttt{obs}\ ((\texttt{y.b} \leftrightarrow \texttt{x.b}) \bullet \texttt{<x.r x.b>})) \\
=\ &(\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \left(\texttt{obs}\ \texttt{<x.r y.b>}\right) \\
=\ &(\texttt{x.b} \leftrightarrow \texttt{y.b}) \bullet \texttt{2} \\
=\ &\texttt{2}
\end{aligned}$$

This function does not *introduce* any atoms. Nevertheless, it does *observe* a binder atom in the input. Thus, permuting it changes its output, and it is not equivariant. This is not just an artifact of how we have mathematically defined hygiene but is a distinction present in many hygiene algorithms. Indeed, if `obs` were to take the identifier constant that it is comparing against the input and introduce it into its output, then respecting *introduced*-binder hygiene requires that it not bind over any identifiers from the input. Thus, those identifiers should not be equal when they are observed by `bound-identifier=?`. To achieve this, we must keep the atoms in the input distinct from any atoms internal to the transformer and thus enforce *observed*-binder hygiene. This is not as well known as introduced-binder and reference hygiene but is just as important.

5.7 Summary

At this point, we now have a mathematical definition of hygiene and can summarize the main points as follows.

First, we assume identifiers are diatomic as in Section 4.6.

Second, in order to enforce reference hygiene, we require that the Reference Hygiene Criterion holds. In other words, we require that expansion steps respect alpha equivalences over already exposed binding forms in the k-syntax portion of the code.

Third, introduced binder atoms are noted by their presence in the support of a transformer.

Fourth, we require that the Binder Hygiene Criterion hold in order to force transformers to be equivariant with respect to binder atoms and thus enforce binder hygiene. When macros introduce a binder atom, they must thus use `fresh` so they remain equivariant.

Finally, we can automatically make transformers be equivariant by using the hyg operator as defined in Definition 3 or Definition 6.

6. Hygiene and `syntax-case`

The definition of hygiene in Section 5 has a close correspondence to the `syntax-case` algorithm described by Dybvig et al. (1993). While a detailed explanation of `syntax-case` is beyond the scope of this paper, we highlight a few of the parallels. In the `syntax-case` algorithm, *wraps* are used to annotate syntax objects with binding information. There are two different algorithms described by Dybvig et al. (1993) for applying these wraps. In one, wraps may be suspended in the middle of applying to a syntax object and are lazily applied to descendant syntax objects. In the other, wraps are eagerly applied and always descend syntax objects until they reach individual identifiers. These two models are semantically equivalent. The former is asymptotically more efficient, but the latter is simpler and more directly corresponds to our definition of hygiene.

In the eager version of the `syntax-case` algorithm, wraps are a sequence of *marks* and *substitutions* surrounding a base symbol for an identifier. Marks are atomic, and substitutions map a set of marks and a particular symbol to a *label*, which is also atomic.

New marks are added to the wrap of an identifier during the macro expansion process. A fresh mark, along with its inverse, called an *anti-mark*, is created each time a macro is called. The anti-mark is placed on the identifiers in the input of the macro transformer while the mark is placed on the output of the macro transformer. When a mark is placed on an identifier that has the corresponding anti-mark, they cancel each other out. Identifiers with different marks are considered to be not `bound-identifer=?`. The behavior of marks in this regard mirrors the behavior of the permutation π in Definition 6. The difference is that a mark is essentially an abstract permutation that we treat opaquely without knowing how it maps atoms.

When a binding form expands, a new label is chosen for the binding, and a substitution is created that maps the marks and symbol of the identifier in the binding position to the new label. This substitution is added to the wraps of any identifiers in the body. When an identifier expands in a reference position, the substitutions are consulted to determine to what label it is bound. This models the behavior of how binding forms are expanded in Section 4.6. The freshly generated reference atom corresponds to the new label. The subst operator corresponds to the creation of a substitution.

Finally, note that the operators `bound-identifier=?` and `free-identifier=?` in `syntax-case` are equivalent to comparing the reference and binder parts, respectively, of the diatomic identifiers described in Section 4.6.

In all these regards, the `syntax-case` algorithm corresponds to an almost direct implementation of the mathematical principles in Section 5. The major difference is that `syntax-case` treats permutations as opaque symbolic objects (i.e., marks) and the operations on identifiers are stored in reified forms (i.e., the marks and substitutions in a wrap). This allows operators such as `quote`, `syntax->datum` and `datum->syntax` that "bend" hygiene and either strip wraps from identifiers or transplant the wrap of an identifier to a new symbol. The mathematics in Section 5 can also support these operations if we generalize it to treat permutations as opaque objects and keep subst reified.

7. Related Work

Kohlbecker et al. (1986) defined a hygienic macro expansion algorithm by marking identifiers with time stamps indicating the macro expansion step in which they were introduced. However, their definition of hygiene encompasses only introduced-binder hygiene and not introduced-reference hygiene.

Bawden and Rees (1988) then proposed using syntactic closures to handle hygiene. They define hygiene to include both introduced-binder and introduced-reference hygiene. With syntactic closures, macros take syntactic environments as arguments and return expressions that have been closed over particular environments. These environments are then used when interpreting identifiers in those expressions. Clinger (1991) takes a different approach and passes functions for explicitly renaming and comparing identifiers to each macro transformer. In both these systems, macro authors must be careful to correctly use the tools provided for ensuring hygiene as they may otherwise end up with non-hygienic macros.

Clinger and Rees (1991) combined the pattern matching language of Kohlbecker and Wand (1987) with the ideas from Bawden and Rees (1988) to create a system that lets users write high-level macros that are automatically hygienic. Finally, the `syntax-case` algorithm (Dybvig et al. 1993; Dybvig 2007) extended this by allowing macro transformers to be arbitrary functions and providing mechanisms for "bending" the rules of hygiene. The system specified by the R^{6}RS Scheme standard (Sperber et al. 2007, 2009) is based on this along with extensions for interlibrary macros.

These systems all focus on the algorithmic implementation of hygiene and define hygiene only informally or by way of a particular algorithm. However, the Reference Hygiene Criterion and the Binder Hygiene Criterion are broadly applicable, and we can consider how they apply to these systems. Kohlbecker et al. (1986) satisfies the Binder Hygiene Criterion but not the Reference Hygiene Criterion, which reflects the fact that they are concerned only with binder hygiene. Bawden and Rees (1988) and Clinger (1991) provide the tools for macro authors to satisfy both the Reference Hygiene Criterion and the Binder Hygiene Criterion but do not enforce them. Nevertheless, whether a macro author has used the tools correctly can be determined by whether these criteria are respected by the transformer. Clinger and Rees (1991) and Dybvig et al. (1993) automatically satisfy the Reference Hygiene Criterion and the Binder Hygiene Criterion except when hygiene "bending" operators are used.

A number of systems (Ganz et al. 2001; Culpepper and Felleisen 2003, 2004; Herman and Wand 2008; Herman 2010) use shape types or some other typing discipline to enforce hygiene. These systems generally annotate macros with types that specify their binding structures and enforce the Reference Hygiene Criterion and the Binder Hygiene Criterion as a consequence of typeability. Several of them directly connect their notions of hygiene to nominal logic, albeit in a typed setting. However, these type systems reject many otherwise well-formed, hygienic macros and do not consider hygiene in a general setting without a type system or when not all binding structures are known in advance.

The definition of hygiene that we propose is not only useful for determining whether a particular macro system is hygienic, but also provides a useful perspective on extensions to hygiene that

have been proposed. For example, van Tonder (2005) gives examples of macros that use local functions to manipulate the syntax. Since the usual hygiene algorithms do not distinguish between identifiers introduced by different functions within the same macro expansion step, these helper functions may unintentionally capture identifiers introduced by each other. This is easily solved by exposing the hyg operator to macro authors so they can annotate these functions as needing to introduce distinct identifiers. Similar techniques can be used with the local transformations in Culpepper and Felleisen (2010) and Culpepper (2012). Another example is anaphoric macros (Barzilay et al. 2011). The classic case of this is a `loop` macro that binds `break` to a continuation that breaks out of the loop. This binding is introduced by the macro but should be visible to the loop body written by the macro user. We conjecture that this can be handled by allowing macros to specify that they have a non-empty support. For example, `loop` would specify that `break` is in its support.

8. Conclusion

Hygiene is an essential aspect of the Scheme macro system. There have been multiple implementations of it over the years, but no general, formal definition of what it is. Existing definitions are either informal, tied to a particular expansion algorithm or consider only a restricted subset of macros that follow a certain typing discipline. In this paper we examine several examples of hygiene and how it is violated in naive expanders. From these examples, we then develop a simple expansion algorithm that preserves hygiene. Based on the intuitions developed with that algorithm, we then construct a mathematically precise, formal definition of hygiene. This definition connects hygiene to nominal logic and shows that hygiene is a combination of alpha equivalence and equivariance. It is algorithm independent and can be used to test whether a proposed algorithm is hygienic. This definition closely corresponds to several existing algorithms, and in particular, the `syntax-case` algorithm closely parallels how one would naturally write an algorithm to satisfy this definition.

Acknowledgments

Thanks go to Mitchell Wand, William Byrd, R. Kent Dybvig, J. Ian Johnson, Michael Ballantyne, Celeste Hollenbeck and the anonymous reviewers for their comments, suggestions and help in improving this paper.

This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

This article reports on work supported by the Defense Advanced Research Projects Agency under agreement no. FA8750-10-2-0233. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

Eli Barzilay, Ryan Culpepper, and Matthew Flatt. Keeping it clean with syntax parameters. In *Proceedings of the Twelfth Workshop on Scheme and Functional Programming*, October 2011.

Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 86–95, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: 10.1145/62678.62687.

William Clinger. Hygienic macros through explicit renaming. *ACM SIGPLAN Lisp Pointers*, IV(4):25–28, October 1991. ISSN 1045-3563. doi: 10.1145/1317265.1317269.

William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99607.

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(Special Issue 4–5):439–476, September 2012. ISSN 1469-7653. doi: 10.1017/S0956796812000275.

Ryan Culpepper and Matthias Felleisen. Well-shaped macros. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 59–68, November 2003.

Ryan Culpepper and Matthias Felleisen. Taming macros. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 225–243. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23580-4. doi: 10.1007/978-3-540-30175-2_12.

Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 235–246, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863577.

R. Kent Dybvig. Syntactic abstraction: The `syntax-case` expander. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, Theory in Practice, chapter 25. O'Reilly Media, July 2007. ISBN 978-0-596-51004-6.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *LISP and Symbolic Computation*, 5(4):295–326, December 1993. ISSN 0892-4635 (Print) 1573-0557 (Online). doi: 10.1007/BF01806308.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5): 341–363, July 2002. ISSN 0934-5043 (Print) 1433-299X (Online). doi: 10.1007/s001650200016.

Steven E. Ganz, Amr Sabry, and Walid Taha. 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, pages 74–85, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507646.

David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, USA, May 2010.

David Herman and Mitchell Wand. A theory of hygienic macros. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78738-9. doi: 10.1007/978-3-540-78739-6_4.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319859.

Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 77–84, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41632.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. URL `http://www.r6rs.org/`.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19 (Supplement S1):1–301, August 2009. ISSN 1469-7653. doi: 10.1017/S0956796809990074.

André van Tonder. *SRFI-72: Hygienic macros*, September 2005. URL `http://srfi.schemers.org/srfi-72/srfi-72.html`.