Binding as Sets of Scopes

Abstract

Our new macro expander for Racket builds on a novel approach to hygiene. Instead of basing macro expansion on variable renamings that are mediated by expansion history, our new expander tracks binding through a *set of scopes* that an identifier acquires from both binding forms and macro expansions. The resulting model of macro expansion is simpler and more uniform than one based on renaming, and it is sufficiently compatible with Racket's current expander to be practical.

1. Introduction: Lexical vs. Macro Scope

Hygienic macro expansion is desirable for the same reason as lexical scope: both enable local reasoning about binding so that program fragments compose reliably. The analogy suggests specifying hygienic macro expansion as a kind of translation into lexical-scope machinery. In particular, variables must be *renamed* to match the mechanisms of lexical scope as variables interact with macros.

A specification of hygiene in terms of renaming accommodates simple binding forms well, but it becomes unwieldy for recursive definition contexts (Flatt et al. 2012, section 3.8), especially for contexts that allow a mixture of macro and non-macro definitions. The renaming approach is also difficult to implement compactly and efficiently in a macro system that supports "hygiene bending" operations, such as `datum->syntax`, because a history of renamings must be recorded for replay on an arbitrary symbol.

In a new macro expander for Racket, we discard the renaming approach and start with a generalized idea of macro scope, where lexical scope is just a special case of macro scope when applied to a language without macros. Roughly, every binding form and macro expansion creates a *scope*, and each fragment of syntax acquires a *set of scopes* that determines binding of identifiers within the fragment. In a language without macros, each scope set is identifiable by a single innermost scope. In a language with macros, identifiers acquire scope sets that overlap in more general ways.

Our experience is that this set-of-scopes model is simpler to use that the current macro expander, especially for macros that work with recursive-definition contexts or create unusual binding patterns. Along similar lines, the expander's implementation is simpler than the current one based on renaming, and the implementation avoids bugs that have proven difficult to repair in the current expander. Finally, the new macro expander is able to provide more helpful debugging information when binding resolution fails.

This change to the expander's underlying model of binding can affect the meaning of existing Racket macros. A small amount of incompatibility seems acceptable and even desirable if it enables easier reasoning overall. Drastic incompatibilities would be suspect, however, both because the current expander has proven effective and because large changes to code base would be impractical. Consistent with those aims, purely pattern-based macros work with the new expander the same as with the old one, except for unusual macro patterns within a recursive definition context. More generally, our experiments indicate that the majority of existing Racket macros work unmodified, and other macros can be adapted with reasonable effort.

2. Background: Scope and Macros

An essential consequence of hygienic macro expansion is to enable macro definitions via *patterns* and *templates*—also known as *macros by example* (Kohlbecker and Wand 1987; Clinger and Rees 1991). Although pattern-based macros are limited in various ways, a treatment of binding that can accommodate patterns and templates is key to the overall expressiveness of a hygienic macro system, even for macros that are implemented with more general constructs.

As an example of a pattern-based macro, suppose that a Racket library implements a Java-like object system and provides a `send` form, where

```
(send a-pt rotate 90)
```

evaluates `a-pt` to an object, locates a function mapped to the symbol `'rotate` within the object, and calls the function as a method by providing the object itself followed by the argument `90`. Assuming a `lookup-method` function that locates a method within an object, the `send` form can be implemented by a pattern-based macro as follows:

```
(define-syntax-rule (send obj-expr method-name arg)
  (let ([obj obj-expr])
    ((lookup-method obj 'method-name) obj arg)))
```

With this definition, the example use of `send` above matches the pattern with `a-pt` as *`obj-expr`*, `rotate` as *`method-name`*, and `90` as *`arg`*, so the `send` use expands to

```
(let ([obj a-pt])
  ((lookup-method obj 'rotate) obj 90))
```

Hygienic macro expansion ensures that the identifier `obj` is not accidentally referenced in an expression that replaces *`arg`* in a use of `send` (Kohlbecker et al. 1986). For example, the body of

```
(lambda (obj)
  (send a-pt same? obj))
```

must call the `same?` method of `a-pt` with the function argument `obj`, and not with `a-pt` itself as bound to `obj` in the macro template for `send`. Along similar lines, a local binding of `lookup-method` at a use site of `send` must not affect the meaning of `lookup-method` in `send`'s template. That is,

, .

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

```
(let ([lookup-method #f])
  (send a-pt rotate 90))
```

should still call the `rotate` method of `a-pt`.

Macros can be bound locally, and macros can even expand to definitions of macros. For example, suppose that the library also provides a `with-method` form that performs a method lookup just once for multiple sends:

```
(with-method ([rot-a-pt (a-pt rotate)]) ; find rotate
  (for ([i 1000000])
    (rot-a-pt 90))) ; send rotate to point many times
```

The implementation of `with-method` can make `rot-a-pt` a local macro binding, where a use of `rot-a-pt` expands to a function call with `a-pt` added as the first argument to the function. That is, the full expansion is

```
(let ([obj a-pt])
  (let ([rot-a-pt-m (lookup-method obj 'rotate)])
    (for ([i 1000000])
      (rot-a-pt-m obj 90))))
```

but the intermediate expansion is

```
(let ([obj a-pt])
  (let ([rot-a-pt-m (lookup-method obj 'rotate)])
    (let-syntax ([rot-a-pt (syntax-rules ()
                           [(rot-a-pt arg)
                            (rot-a-pt-m obj arg)])])
      (for ([i 1000000])
        (rot-a-pt 90)))))
```

where `let-syntax` locally binds the macro `rot-a-pt`. The macro is implemented by a `syntax-rules` form that produces an anonymous pattern-based macro (in the same way that `lambda` produces an anonymous function).

In other words, `with-method` is a binding form, it is a macro-generating macro, it relies on local-macro binding, and the macro that it generates refers to a private binding `obj` that is also macro-introduced. Nevertheless, `with-method` is straightforwardly implemented as a pattern-based macro:

```
(define-syntax-rule
  (with-method ([local-id (obj-expr method-name)])
    body)
  (let ([obj obj-expr])
    (let ([method (lookup-method obj 'method-name)])
      (let-syntax ([local-id (syntax-rules ()
                               [(local-id arg)
                                (method obj arg)])])
        body))))
```

Note that the `obj` binding cannot be given a permanently distinct name within `with-method`. A distinct name must be generated for each use of `with-method`, so that nested uses create local macros that reference the correct `obj`.

In general, the necessary bindings or even the binding structure of a macro's expansion cannot be predicted in advance of expanding the macro. For example, the `let` identifier that starts the `with-method` template could be replaced with a macro argument, so that either `let` or, say, a lazy variant of `let` could be supplied to the macro. The expander must accommodate such macros by delaying binding decisions as long as possible. Meanwhile, the expander must accumulate information about the origin of identifiers to enable correct binding decisions.

Even with additional complexities—where the macro-generated macro is itself a binding form, where uses can be nested so the different uses of the generated macro must have distinct bindings, and so on—pattern-based macros support implementations that are essentially specifications (Kohlbecker and Wand 1987). A naive approach to macros and binding fails to accommodate the specifications (Adams 2015), while existing formalizations of suitable binding rules detour into concepts of marks and renamings that are distant from the programmer's sense of the specification.

The details of a formalization matter more when moving beyond pattern-matching macros to *procedural macros*, where the expansion of a macro can be implemented by an arbitrary compile-time function. The `syntax-case` and `syntax` forms provide the pattern-matching and template-construction facilities, respectively, of `syntax-rules`, but they work as expressions within a compile-time function (Dybvig et al. 1993). This combination allows a smooth transition from pattern-based macros to procedural macros for cases where more flexibility is needed. In fact, `syntax-rules` is itself simply a macro that expands to a procedure:

```
(define-syntax-rule (syntax-rules literals
                     [pattern template] ...)
  (lambda (stx)
    (syntax-case stx literals
      [pattern #'template] ; #'_ is short for (syntax _)
      ...)))
```

Besides allowing arbitrary computation mixed with pattern matching and template construction, the `syntax-case` system provides operations for manipulating program representations as *syntax objects*. Those operations include "bending" hygiene by attaching the binding context of one syntax object to another. For example, a macro might accept an identifier `point` and synthesize the identifier `make-point`, giving the new identifier the same context as `point` so that `make-point` behaves as if it appeared in the same source location with respect to binding.

Racket provides an especially rich set of operations on syntax objects to enable macros that compose and cooperate (Flatt et al. 2012). Racket's macro system also relies on a module layer that prevents interference between run-time and compile-time phases of a program, since interference would make macros compose less reliably (Flatt 2002). Finally, modules can be nested and macro-generated, which enables macros and modules to implement facets of a program that have different instantiation times—such as the program's run-time code, its tests, and its configuration meta-data (Flatt 2013). The module-level facets of Racket's macro system are, at best, awkwardly accommodated by existing models of macro binding; those models are designed for expression-level binding, where α-renaming is straightforward, while modules address a more global space of mutually recursive macro and variable definitions. A goal of our new binding model is to more simply and directly account for such definition contexts.

3. Scope Sets for Pattern-Based Macros

Like previous models of macro expansion, our set-of-scopes expander operates on a program from the outside in. The expander detects bindings, macro uses, and references as part of the outside-to-inside traversal. The difference in our expander is the way that bindings and macro expansions are recorded and attached to syntax fragments during expansion.

3.1 Scope Sets

A *scope* corresponds to a binding context, and every identifier in a program has a set of scopes. For example, if we treat `let` and `lambda` as primitive binding forms, then in the fully expanded expression

```
(let ([x 1])
  (lambda (y)
    z))
```

the `let` form corresponds to a scope a_{let}, and the `lambda` form corresponds to b_{lam}. That is, everything in the `let`'s body is in

a_{let}, and everything in the inner `lambda`'s body is in b_{lam}; the set of scopes associated with `z` is $\{a_{let}, b_{lam}\}$.[1]

In a macro-extensible language, expanding a use of a macro creates a new scope in the same way that a binding form creates a new scope. Starting with

```
(let ([x 1])
  (let-syntax ([m (syntax-rules ()
                    [(m) x])])
    (lambda (x)
      (m))))
```

the right-hand side of the `m` binding has the scope set $\{a_{let}\}$, while the final `m` has scope set $\{a_{let}, b_{ls}, c_{lam}\}$ corresponding to the `let`, `let-syntax`, and `lambda` forms. We can write the scope sets next to each `x` and `m` at the point where macro expansion reaches the `(m)` form:

```
(let ([x^{a_let} 1])
  (let-syntax ([m^{a_let, b_ls} (syntax-rules ()
                                  [(m) #'x^{a_let}])])
    (lambda (x^{a_let, b_ls, c_lam})
      (m^{a_let, b_ls, c_lam}))))
```

The expansion of `(m)` produces `x` with the scope set $\{a_{let}, d_{intro}\}$, where d_{intro} is a new scope for identifiers that are introduced by the macro's expansion:

```
(let ([x^{a_let} 1])
  (let-syntax ([m^{a_let, b_ls} (syntax-rules ()
                                  [(m) #'x^{a_let}])])
    (lambda (x^{a_let, b_ls, c_lam})
      x^{a_let, d_intro})))
```

The absence of c_{lam} on the final `x` explains why it doesn't refer to the inner binding of `x`. At the same time, if a different `m` places a macro-introduced `x` in a binding position around an `x` from a macro use `(m x)`, the `x` from the use is not macro-introduced and doesn't have the scope d_{intro}, so it wouldn't refer to the macro-introduced binding.

Lexical scoping corresponds to sets that are constrained to a particular shape: For any given set, there's a single scope s that implies all the others (i.e., the ones around s in the program). As a result, s by itself is enough information to identify a binding for a given reference. We normally describe lexical scope in terms of the closest such s for some notion of "closest." Given scope sets instead of individual scopes, we can define "closest" as the largest relevant set.

More generally, we can define binding based on *subsets*: A reference's binding is found as one whose set of scopes is a subset of the reference's own scopes (in addition to having the same symbolic name). The advantage of using sets of scopes is that macro expansion creates scope sets that overlap in more general ways; there's not always a s that implies all the others. Absent a determining s, we can't identify a binding by a single scope, but we can identify it by a set of scopes.

If arbitrary sets of scopes are possible, then two different bindings might have overlapping scopes, neither might be a subset of the other, and both might be subsets of a particular reference's scope set. In that case, the reference is ambiguous. Creating an ambiguous with only pattern-based macros is possible, but it requires a definition context that supports mingled macro definitions and uses; we provide an example in section 3.5.

3.2 Bindings

When macro expansion encounters a binding form, it

- creates a new scope;
- adds the scope to every identifier in binding position, as well as to the region where the bindings apply; and
- extends a global table that maps a ⟨symbol, scope set⟩ pair to a representation of a binding.

Each local binding is represented by a unique, opaque value (e.g., a gensym).

For example,

```
(let ([x 1])
  (let-syntax ([m (syntax-rules ()
                    [(m) x])])
    (lambda (x)
      (m))))
```

more precisely expands as

```
(let ([x^{a_let} 1])
  (let-syntax ([m^{a_let, b_ls}
                (syntax-rules ()
                  [(m) #'x^{a_let}])])
    (lambda (x^{a_let, b_ls, c_lam})
      x^{a_let, d_intro})))
```

```
x^{a_let}               → x4
m^{a_let, b_ls}         → m8
x^{a_let, b_ls, c_lam}  → x16
```

where the compile-time environment along the way (not shown) maps `x4` to a variable, `m8` to a macro, and `x16` to another variable. The reference $\texttt{x}^{\{a_{let}, d_{intro}\}}$ has the binding `x4`, because $\texttt{x}^{\{a_{let}\}}$ is the mapping for `x` in the binding table that has the largest subset of $\{a_{let}, d_{intro}\}$.

The distinction between the binding table and the compile-time environment is important for a purely "syntactic" view of binding, where a term can be expanded, manipulated, transferred to a new context, and then expanded again. Some approaches to macros, such as syntactic closures (Bawden and Rees 1988) and explicit renaming (Clinger 1991), tangle the binding and environment facets of expansion so that terms cannot be manipulated with the same flexibility.

The binding table can grow forever, but when a particular scope becomes unreachable (i.e., when no reachable syntax object includes a reference to the scope), then any mapping that includes the scope becomes unreachable. This weak mapping can be approximated by attaching the mapping to the scope, instead of using an actual global table. Any scope in a scope set can house the binding, since the binding can only be referenced using all of the scopes in the set. Attaching to the most recently allocated scope is a good heuristic, because the most recent scope is likely to be maximally distinguishing and have the shortest lifetime.

3.3 Recursive Macros and Use-Site Scopes

So far, our characterization of macro-invocation scopes works only for non-recursive macro definitions. To handle recursive macro definitions, in addition to a fresh scope to distinguish forms that are introduced by a macro, a fresh scope is needed to distinguish forms that are present at the macro use site.

Consider the following `letrec-syntax` expression, whose meaning depends on whether a use-site identifier captures a macro-introduced identifier:

```
(letrec-syntax ([identity (syntax-rules ()
                            [(_ misc-id)
                             (lambda (x)
                               (let ([misc-id 'other])
                                 x))])])
  (identity x))
```

---

[1] We might say that "the environment of `z` is $\{a_{let}, b_{lam}\}$," but for macro expansion, we also use the word "environment" to refer to a mapping from bindings to variables and compile-time values. To avoid confusion, we'll refrain from using "environment" to mean a set of scopes.

Assuming that the `letrec-syntax` form creates a scope a_{ls}, the scope must be added to both the right-hand side and body of the `letrec-syntax` form to create a recursive binding:

```
(letrec-syntax ([identity (syntax-rules ()
                            [(_ misc-id)
                             (lambda (x$^{\{a_{ls}\}}$)
                               (let ([misc-id 'other])
                                 x$^{\{a_{ls}\}}$))])])
   (identity x$^{\{a_{ls}\}}$))
```

If we create a scope only for introduced forms in a macro expansion, then expanding `(identity x`$^{\{a_{ls}\}}$`)` creates the scope set b_{intro} and produces

```
(lambda (x$^{\{a_{ls}, b_{intro}\}}$)
  (let ([x$^{\{a_{ls}\}}$ 'other])
    x$^{\{a_{ls}, b_{intro}\}}$))
```

where b_{intro} is added to each of the two introduced `x`s. The `lambda` introduces a new scope c_{lam}, and `let` introduces d_{let}, producing

```
(lambda (x$^{\{a_{ls}, b_{intro}, c_{lam}\}}$)
  (let ([x$^{\{a_{ls}, c_{lam}, d_{let}\}}$ 'other])
    x$^{\{a_{ls}, b_{intro}, c_{lam}, d_{let}\}}$))
```

At this point, the binding of the innermost `x` is ambiguous: $\{a_{ls}, b_{intro}, c_{lam}, d_{let}\}$ is a superset of both $\{a_{ls}, b_{intro}, c_{lam}\}$ and $\{a_{ls}, c_{lam}, d_{let}\}$, neither of which is a subset of the other. Instead, we want `x` to refer to the `lambda` binding.

Adding a scope for the macro-use site corrects this problem. If we call the use-site scope e_{use}, then we start with

```
(identity x$^{\{a_{ls}, e_{use}\}}$)
```

which expands to

```
(lambda (x$^{\{a_{ls}, b_{intro}\}}$)
  (let ([x$^{\{a_{ls}, e_{use}\}}$ 'other])
    x$^{\{a_{ls}, b_{intro}\}}$))
```

which ends up as

```
(lambda (x$^{\{a_{ls}, b_{intro}, c_{lam}\}}$)
  (let ([x$^{\{a_{ls}, c_{lam}, d_{let}, e_{use}\}}$ 'other])
    x$^{\{a_{ls}, b_{intro}, c_{lam}, d_{let}\}}$))
```

There's no ambiguity, and the final `x` refers to the `lambda` binding as intended. In short, each macro expansion needs a use-site scope as the symmetric counterpart to the macro-induction scope.

3.4 Use-Site Scopes and Macro-Generated Definitions

In a binding form such as `let` or `letrec`, bindings are clearly distinguished from uses by their positions within the syntactic form. In addition to these forms, Racket (like Scheme) supports definition contexts that mingle binding forms and expressions. For example, the body of a module contains a mixture of definitions and expressions, all in a single recursive scope. Definitions can include macro definitions, expressions can include uses of those same macros, and macro uses can even expand to further definitions.

With set-of-scopes macro expansion, macro definitions and uses within a single context interact badly with use-site scopes. For example, consider a `define-identity` macro that is intended to expand to a definition of a given identifier as the identity function:

```
(define-syntax-rule (define-identity id)
  (define id (lambda (x) x)))

(define-identity f)
(f 5)
```

If the expansion of `(define-identity f)` adds a scope to the use-site `f`, the resulting definition does not bind the `f` in `(f 5)`.

The underlying issue is that a definition context must treat use-site and introduced identifiers asymmetically as binding identifiers. In

```
(define-syntax-rule (define-five misc-id)
  (begin
   (define misc-id 5)
   x))

(define-five x)
```

the introduced `x` should refer to an `x` that is defined in the enclosing scope, which turns out to be the same `x` that appears at the use site of `define-five`. But in

```
(define-syntax-rule (define-other-five misc-id)
  (begin
   (define x 5)
   misc-id))

(define-other-five x)
```

the `x` from the use site should not refer to the macro-introduced binding `x`.

To support macros that expand to definitions of given identifiers, a definition context must keep track of scopes created for macro uses, and it must remove those scopes from identifiers that end up in binding positions. In the `define-identity` and `define-five` examples, the use-site scope is removed from the binding identifiers `x` and `f`, so they are treated the same as if their definitions appeared directly in the source.

This special treatment of use-site scopes adds complexity to the macro expander, but it is of the kind of complexity that mutually recursive binding contexts create routinely (e.g., along the same lines as ensuring that variables are defined before they are referenced). Definition contexts in Racket have proven convenient and expressive enough to be worth the extra measure of complexity.

3.5 Ambiguous References

The combination of use-site scopes to solve local-binding problems (as in section 3.3) versus reverting use-site scopes to accommodate macro-generated definitions (as in section 3.4) creates the possibility of generating an identifier whose binding is ambiguous.

The following example defines `m` through a `def-m` macro, and it uses `m` in the same definition context:

```
(define-syntax-rule (def-m m given-x)
  (begin
    (define x 1)
    (define-syntax-rule (m)
      (begin
        (define given-x 2)
        x))))

(def-m m x)
(m)
```

The expansion, after splicing `begin`s, ends with an ambiguous reference:

```
(define-syntax-rule (def-m$^{\{a_{def}\}}$ ....) ....)
(define x$^{\{a_{def}, b_{intro1}\}}$ 1)
(define-syntax-rule (m$^{\{a_{def}\}}$)
  (begin
    (define x$^{\{a_{def}, b_{use1}\}}$ 2)
    x$^{\{a_{def}, b_{intro1}\}}$))
(define x$^{\{a_{def}, c_{intro2}\}}$ 2)
x$^{\{a_{def}, b_{intro1}, c_{intro2}\}}$
```

The scope a_{def} corresponds to the definition context, b_{intro1} and b_{use1} correspond to the expansion of `def-m`, c_{intro2} corresponds to the expansion of `m`. The final reference to `x` is ambiguous, because it was introduced through both macro layers.

Unlike the ambiguity that is resolved by use-site scopes, this ambiguity arguably reflects an inherent ambiguity in the macro. Absent the `(define x 1)` definition generated by `def-m`, the final `x` reference should refer to the definition generated from `(define given-x 2)`; similarly, absent the definition generated from `(define given-x 2)`, the final `x` should refer to the one generated from `(define x 1)`. Neither of those definitions is more specific than the other, since they are generated by different macro invocations, so our new expander rejects the reference as ambiguous.

Our previous model of macro expansion to cover definition contexts (Flatt et al. 2012) would treat the final `x` always as a reference to the definition generated from `(define x 1)` and never to the definition generated from `(define given-x 2)`. So far, we have not encountered a practical example that exposes the difference between the expanders' treatment of pattern-based macros in definition contexts.

4. Procedural Macros and Modules

Although our set-of-scopes expander resolves bindings differently than in previous models, it still works by attaching information to identifiers, and so it can provide a smooth path from pattern-matching macros to procedural macros in the same way as `syntax-case` (Dybvig et al. 1993). Specifically, `(syntax` *`form`*`)` quotes the S-expression *`form`* while preserving its scope-set information, so that *`form`* can be used to construct the result of a macro.

More precisely, the primitive `(quote-syntax` *`form`*`)` quotes *`form`* with its scope sets in Racket. The derived `(syntax` *`form`*`)` detects uses of pattern variables and replaces them with their matches while quoting any non-pattern content in *`form`* with `quote-syntax`. A `(syntax` *`form`*`)` can be abbreviated `#'`*`form`*, and when *`form`* includes no pattern variables, `#'`*`form`* is equivalent to `(quote-syntax` *`form`*`)`.

The result of a `quote-syntax` or `syntax` form is a *syntax object*. When a syntax object's S-expression component is just a symbol, then the syntax object is an *identifier*.

4.1 Identifier Comparisons with Scope Sets

Various compile-time functions work on syntax objects and identifiers. Two of the most commonly used functions are `free-identifier=?` and `bound-identifier=?`, each of which takes two identifiers. The `free-identifier=?` function is used to recognize a reference to a known binding, such as recognizing a use of `else` in a conditional. The `bound-identifier=?` function is used to check whether two identifiers would conflict as bindings in the same context, such as when a macro that expands to a binding form checks that identifiers in the macro use are suitably distinct.

These two functions are straightforward to implement with scope sets. A `free-identifier=?` comparison on identifiers checks whether the two identifiers have the same binding by consulting the global binding table. A `bound-identifier=?` comparison checks that two identifiers have exactly the same scope sets, independent of the binding table.

4.2 Local Bindings and Syntax Quoting

The set-of-scopes approach to binding works the same as previous models for macros that are purely pattern-based, but the set-of-scopes approach makes finer distinctions among identifiers than would be expected by existing procedural Racket macros that use `` #` `` or `quote-syntax`. To be consistent with the way that Racket macros are currently written, `quote-syntax` must discard some scopes.

For example, in the macro

```
(lambda (stx)
  (let ([id #'x])
    #`(let ([#,id 1])
        x)))
```

the `x` that takes the place of `#,id` should bind the `x` that is in the resulting `let`'s body. The `x` that is bound to `id`, however, is not in the scope that is introduced by the compile-time `let`:

$$\begin{array}{l}
\texttt{(lambda (stx}^{\{a_{lam}\}}\texttt{)} \\
\quad\texttt{(let ([id}^{\{a_{lam},\,b_{let}\}}\ \texttt{\#'x}^{\{a_{lam}\}}\texttt{])} \\
\qquad\texttt{\#`(let ([\#,id}^{\{a_{lam},\,b_{let}\}}\ \texttt{1])} \\
\qquad\quad\ \texttt{x}^{\{a_{lam},\,b_{let}\}}\texttt{)))}
\end{array}$$

If `quote-syntax` (implicit in `` #` ``) preserves all scopes on an identifier, then with set-of-scopes binding, the `x` that replaces `#,id` will not capture the `x` in the generated `let` body.

It's tempting to think that the compile-time `let` should introduce a phase-specific scope that applies only for compile-time references, in which case it won't affect `x` as a run-time reference. That adjustment doesn't solve the problem in general, since a macro can generate compile-time bindings and references just as well as run-time bindings and references.

A solution is for the expansion of `quote-syntax` to discard certain scopes on its content. The discarded scopes are those from binding forms that enclosed the `quote-syntax` form up to a phase crossing or module top-level, as well as any use-site scopes recorded for macro invocations within those binding forms. In the case of a `quote-syntax` form within a macro binding's right-hand side, those scopes cover all of the scopes introduced on the right-hand side of the macro binding.

The resulting macro system is different than the current Racket macro system. Experiments suggest that the vast majority of macro implementations work either way, but it's easy to construct an example that behaves differently:

```
(free-identifier=? (let ([x 1]) #'x)
                   #'x)
```

In Racket's current macro system, the result is `#f`. The set-of-scopes system with a scope-pruning `quote-syntax` produces `#t`, instead, because the `let`-generated scope is stripped away from `#'x`.

If `quote-syntax` did not prune scopes, then not only would the result above be `#f`, it would also be `#f` with `(let ([y 1]) #'x)` instead of `(let ([x 1]) #'x)`. That similarity reflects the switch to attaching identifier-independent scopes to identifiers instead of attaching identifier-specific renamings.

Arguably, the issue here is the way that pieces of syntax from different local scopes are placed into the same result syntax object, with the expectation that all the pieces are treated the same way. In other words, Racket programmers have gotten used to an unusual variant of `quote-syntax`, and most macros could be written just as well with a non-pruning variant.

Supplying a second, non-pruning variant of `quote-syntax` poses no problems. Our set-of-scopes implementation for Racket implements the non-pruning variant when a `#:local` keyword is added to a `quote-syntax` form. For example,

```
(free-identifier=? (let ([x 1])
                     (quote-syntax x #:local))
                   (quote-syntax x #:local))
```

produces `#f` instead of `#t`, because the scope introduced by `let` is preserved in the body's syntax object. The non-pruning variant of `quote-syntax` is useful for embedding references in a program's full expansion that are meant to be inspected by tools other than the Racket compiler; Typed Racket's implementation uses the `#:local` variant of `quote-syntax` to embed type declarations (including declarations for local bindings) in a program's expansion for use by its type checker.

4.3 First-Class Definition Contexts

Racket exposes the expander's support for definition contexts (see section 3.4) so that new macros can support definition contexts while potentially changing the meaning of a macro or variable definition. For example, the `class` macro allows local macro definitions in the `class` body while it rewrites specified function definitions to methods and other variable definitions to fields. The `unit` form similarly rewrites variable definitions to a mixture of private and exported definitions with a component.

Implementing a definition context starts with a call to `syntax-local-make-definition-context`, which creates a first-class (at compile time) value that represents the definition context. A macro can force expansion of forms in the definition context, it can add variable bindings to the definition context, and it can add compile-time bindings and values that are referenced by further macro expansion within the definition context. To a first approximation, a first-class definition context corresponds to a scope that is added to any form expanded within the definition context and that houses the definition context's bindings. A definition context also has a compile-time environment frame (extending the context of the macro use) to house the mapping of bindings to variables and compile-time values.

Like other definition contexts (see section 3.4), the compile-time environment must track use-site scopes that are generated for macro expansions within a first-class definition context. If the macro moves any identifier into a binding position in the overall expansion, then the macro normally must remove accumulated use-site scopes (for the current definition context only) by applying `syntax-local-identifier-as-binding` to the identifier. For example, the `unit` form implements a definition context that is similar to the body of a `lambda`, but variables are internally transformed to support mutually recursive references across unit boundaries.

```
(unit (import)
      (export)
 (define x 1)
 x)
```

In this example, `(define x 1)` is expanded to `(define-values (x) 1)` with a use-site scope on `x`, but the intent is for this definition of `x` to capture the reference at the end of the `unit` form. If the `unit` macro simply moved the binding `x` into a `letrec` right-hand side, the `x` would not capture the final `x` as moved into the `letrec` body; the use-site scope on the definition's `x` would prevent it from capturing the use. The solution is for the `unit` macro to apply `syntax-local-identifier-as-binding` to the definition's `x` before using it as a `letrec` binding. Macros that use a definition context and `bound-identifier=?` must similarly apply `syntax-local-identifier-as-binding` to identifiers before comparing them with `bound-identifier=?`.

Even if a macro does not create a first-class definition context, some care is needed if a macro forces the expansion of subforms and moves pieces of the result into binding positions. Such a macro probably should not use `syntax-local-identifier-as-binding`, but it should first ensure that the macro use is in an expression context before forcing any subform expansions. Otherwise, the subform expansions could interact in unexpected ways with the use-site scopes of an enclosing definition context.

Use-site scopes associated with a first-class definition context are not stored directly in the compile-time environment frame for the definition context. Instead, they are stored in the closest frame that is not for a first-class definition context, so that the scopes are still tracked when the definition context is discarded (when the macro returns, typically). The scope for the definition context itself is similarly recorded in the closest such frame, so that `quote-syntax` can remove it, just like other binding scopes.

4.4 Rename Transformers

Racket's macro API includes support for binding aliases through *rename transformers*. A compile-time binding to the result of `make-rename-transformer` is similar to a binding to a macro transformer that replaces the binding's identifier with the aliased identifier. In addition, however, binding to a rename transformer causes `free-identifier=?` to report `#t` for the original identifier and its alias.

With set-of-scopes binding, a binding alias is supported through an extension of the binding table. The mapping from a ⟨symbol, scope set⟩ pair is to a ⟨binding, maybe-aliased⟩ pair, where an maybe-aliased is either empty or another identifier (i.e., a symbol and scope set) to which the mapped identifier should be considered `free-identifier=?`. When a transformer-binding form such as `define-syntax` or `letrec-syntax` detects that the value to be installed for a binding as a rename transformer, it updates the binding table to register the identifier within the transformer as an optional-alias.

The implementation of `free-identifier=?` must follow alias chains. Cycles are possible, and they cause the aliased identifier to be treated as unbound.

4.5 Modules and Phases

The `module` form creates a new scope for its body. More precisely, a `module` form creates two new scopes: one that roughly reflects "outside edge" of the module, covering everything that is originally in the module body, and one for the "inside edge" of the module, covering everything that appears in the module through macro expansion for forms in the module's top level. The "inside edge" scope is the one that's like any definition context, while the "outside edge" scope distinguishes identifiers that had no scopes before being introduced through macro expansion.

A `(module*` *name* `#f ....)` submodule form, where `#f` indicates that the enclosing module's bindings should be visible, creates an additional scope in the obvious way. For other `module*` and `module` submodule forms, the macro expander prevents access to the enclosing module's bindings by removing the two scopes of the enclosing module.

A module distinguishes bindings that have the same name but different phases. For example, `lambda` might have one meaning for run-time code within a module, but a different meaning for compile-time code within the same module. Furthermore, instantiating a module at a particular phase implies a phase shift in its syntax literals. Consider the module

```
(define x 1)
(define-for-syntax x 2)

(define id #'x)
(define-for-syntax id #'x)

(provide id (for-syntax id))
```

and suppose that the module is imported both normally and for compile time, the later with a `s:` prefix. In a compile-time context within the importing module, both `id` and `s:id` will be bound to an identifier `x` that had the same scopes originally, but they should refer to different `x` bindings (in different module instances with different values).

Among the possibilities for distinguishing phases, having per-phase sets of scopes on an identifier makes the phase-shifting operation most natural. A local binding or macro expansion can add scopes at all phases, while `module` adds a distinct inside-edge scope to every phase (and the same outside-edge scope to all phases). Since every binding within a module is forced to have that module's phase-specific inside-edge scopes, bindings at different scopes will be appropriately distinguished.

Having a distinct "root" scope for each phase makes most local bindings phase-specific. That is, in

```
(define-for-syntax x 10)
(let ([x 1])
  (let-syntax ([y x])
    ....))
```

the `x` on the right-hand side of `let-syntax` see the top-level phase-1 `x` binding, not the phase-0 local binding. This is a change from Racket's current approach to binding and phases, but the only programs that are affected are ones that would trigger an out-of-context error in the current system. Meanwhile, macros can construct identifiers that have no module scope, so out-of-context errors are still possible.

4.6 The Top Level

A *namespace* in Racket is a top-level evaluation context. Each call to `eval` uses a particular namespace (either the *current namespace* or one supplied to `eval`), and each `read–eval–print` loop works in a particular namespace. Namespaces are first-class values in Racket. A namespace can be created as fresh (e.g., for a sandbox), or it can be extracted from a module instantiation to simulate further evaluation in the module's body.

As the connection to modules may suggest, a top-level namespace corresponds to a pair of scopes in the same way that a module has a scope. Each top-level namespace has the same outside-edge scope, but a distinct inside-edge scope where bindings reside.

A troublesome aspect of top-level namespaces in Racket is that a form might be captured (via `quote-syntax`), expanded, or compiled in one namespace, and then evaluated in another namespace. Historically, top-level bindings have been equated with "unbound," so that expanded and compiled forms originating in a top-level context could move freely among namespaces. This treatment as "unbound" has been fuzzy, however, and forms that originate from module namespaces have been treated differently from forms that originate in a non-module namespace.

To accommodate top-level namespaces with as much consistency (of binding treatment) and convenience (of moving forms among top-level namespaces) as possible, we introduce one more dimension to syntax objects. Instead of having a single set of scopes per phase, each syntax object has a sequence of scope sets per phase. When a syntax object is introduced to a top-level context that is not already included in its scope set (at a gven phase), the current scope set is cloned as a new first item of the list of sets; all further scope-set manipulations affect that first item. When looking up an identifier's binding, however, the sequence is traversed until a binding is found. In other words, all but the first item in the list act as fallbacks for locating a binding. In practice, this fallback mechanisms is consistent with most existing code without otherwise interfering with scope management (since the fallbacks apply only when an identifier is otherwise unbound).

5. Implementation and Experience

Scope sets have an intuitive appeal as a model of binding, but a true test of the model is whether it can accommodate a Racket-scale use of macros—for constructing everything from simple syntactic abstractions to entirely new languages. Indeed, the set-of-scopes model was motivated in part by a fraying of Racket's current macro expander at the frontiers of its implementation, e.g., for submodules (Flatt 2013).[2]

We have implemented a set-of-scopes expander as a replacement of Racket's existing macro expander. A snapshot of the main Racket distribution with the replacement expander is provided as supplementary material.

Build times, memory use, and bytecode footprint are essentially unchanged compared to the current expander. Getting performance on par with the previous system required about two weeks of effort, which we consider promising in comparison to a system that has been tuned over the past 15 years.

5.1 Initial Compatibility Results

The packages in Racket's main distribution have been adjusted to build without error (including all documentation), and most tests in the corresponding test suite pass; 43 out of 7501 modules currently fail.[3] Correcting the failures will most likely require small changes to accommodate the new macro expander.

Achieving the current level of success required small changes to 15 out of about 200 packages in the distribution, plus several substantial macro rewrites in the core package:

- Changed macros in the core package include the `unit`, `class`, and `define-generics` macros, all of which manipulate scope in unusual ways.
- The Typed Racket implementation, which is generally sensitive to the details of macro expansion, required a handful of adjustments to deal with changed expansions of macros and the new scope-pruning behavior of `quote-syntax`.
- Most other package changes involve languages implementations that generate modules or submodules and rely on a non-composable treatment of module scopes by the current expander (which creates trouble for submodules in other contexts).

In about half of all cases, the adjustments for set-of-scopes expansion are compatible with the existing expander. In the other half, the macro adjustments were incompatible with the previous expander and the two separate implementations seem substantially easier to produce than one unified implementation.

Besides porting the main Racket distribution to a set-of-scopes expander, we tried building and testing all packages registered at `http://pkgs.racket-lang.org/`. The result shows 46 failures out of about 400 packages, as opposed to to 21 failures for the same set of packages with the current Racket release. Many new failures involve packages that implement non-S-expression *readers* and rely on namespace-interaction details (as discussed in section 4.6) that change with scope sets; the language implementations can be adjusted to use a different technique that is compatible with both expanders.[4]

---

[2] For an example of a bug report about submodules, see problem report 14521 at `http://bugs.racket-lang.org/query/?debug=&database=default&cmd=view+audit-trail&cmd=view&pr=14521`. The example program fails with the current expander, due to problems meshing mark-oriented module scope with renaming-oriented local scope, but the example works with the set-of-scopes expander.

[3] Many failures are unrelated to the macro system.

[4] See the discussion on compatibility of a reader implementation on the Racket mailing list at `https://groups.google.com/d/msg/racket-dev/6khgHKygmS4/cDIfw5cimDEJ`.

5.2 Longer-Term Compatibility Considerations

As the initial experiments confirm, most Racket programs expand and run the same with a set-of-scope expander as with the current expander. Pattern-based macros are rarely affected. When changes are needed to accommodate the set-of-scopes expander, those changes often can be made compatible with the existing expander. In a few cases, incompatibilities appear unavoidable.

Macros that manipulate bindings or scope in unusual ways can easily expose the difference between the macro systems. As an example, the following program produces `1` with Racket's current expander, but it provokes an ambiguous-binding error with the set-of-scopes expander:

```
(define-syntax-rule (define1 id)
 (begin
   (define x 1)
   ; stash a reference to the introduced identifier:
   (define-syntax id #'x)))

(define-syntax (use stx)
  (syntax-case stx ()
    [(_ id)
     (with-syntax ([old-id (syntax-local-value #'id)])
       #'(begin
           (define x 2)
           ; reference to old-id ends up ambiguous:
           old-id))]))

(define1 foo)
(use foo)
```

In the set-of-scopes model, `define1` and `use` introduce bindings from two separate macro expansions, and they also arrange for an reference to be introduced by *both* of those macros, hence the ambiguity. Arguably, in this case, the `use` macro is brokenThe `use` macro can be fixed by applying `syntax-local-introduce` to the result of `(syntax-local-value #'id)`, which cancels the macro-introduction scope on the identifier, since the identifier conceptually exists outside of this macro's expansion. Such an application of `syntax-local-introduce` is typically needed and typically present in existing Racket macros that bring stashed identifiers into a new context.

The example above illustrates a typical level of macro complexity needed to expose differences between the existing and set-of-scopes expanders. Other existing Racket macros that may fail with the set-of-scopes expander include macros that expand to nested `module` forms, macros that use explicit internal-definition contexts, and macros that introduce identifiers that originate in different modules but expect capture among the identifiers.

The documentation for Racket's current macro system avoids references to the underlying mark-and-rename model. As a result, the documentation is often too imprecise to expose differences created by a change to set-of-scope binding. One goal of the new model is to allow the specification and documentation of Racket's macro expander to be tightened; scope sets are precise enough for specification, but abstract enough to allow high-level reasoning.

5.3 Benefits for New Macros

Certain existing macros in the Racket distribution had to be reimplemented wholesale for the set-of-scopes expander. A notable example is the `package` macro, which simulates the module system of Chez Scheme (Waddell and Dybvig 1999). The implementation of `package` for the current Racket macro expander uses first-class definition contexts, rename transformers, and a facility for attaching mark changes to a rename transformer (to make an introduced name have marks similar to the reference). The implementation with the set-of-scopes expander is considerably simpler, using only scope-set operations and basic rename transformers. Scope sets more directly implement the idea of packages as nested lexical environments. The new implementation is 345 lines versus 459 lines for the original implementation; both versions share much of the same basic structure, and the extra 100 lines of the old implementation represent especially complex pieces.

A similar example was discussed on the Racket mailing list. The `in-package` form is intended to simulate Common Lisp namespaces, where definitions are implicitly prefixed with a package name, a package can import unprefixed names from a different package with `use-package`, and a package can *stop* using unprefixed names for the remainder its body with `unuse-package`. In this case, an implementation for the current expander uses marks, but the implementation is constrained so that macros exported by one package cannot expand to definitions in another package. Again, the set-of-scopes expander is conceptually simpler, more directly reflects binding regions with scopes, and allows definition-producing macros to be used across package boundaries. The version for the current expander also works with the set-of-scopes expander, although with the same limitations as for the current expander; in fact, debugging output from the set-of-scopes expander was instrumental in making that version of `in-package` work.

These two anecdotes involve similar macros that better fit the set-of-scopes model for essentially the same reason, but out experience with others macros—the `unit` macro, `class` macro, and `define-generics` macro—has been similarly positive. In all cases, the set-of-scopes model has felt easier to reason about, and the expander could more readily provide tooling in support of the coneptual model.

5.4 Debugging Support

Although the macro debugger (Culpepper and Felleisen 2010) has proven to be a crucial tool for macro implementors, binding resolution in Racket's current macro expander is completely opaque to macro implementers. When something goes wrong, the expander or macro debugger can report little more than "unbound identifier" or "out of context", because the process of replaying renamings and the encodings used for the renamings are difficult to unpack and relate to the programmer.

A set-of-scopes expander is more frequently in a position to report "unbound identifier, but here are the identifier's scopes, and here are some bindings that are connected to those scopes." In the case of ambiguous bindings, the expander can report the referencing identifier's scopes and the scopes of the competing bindings. These details are reported in a way similar to stack traces: subject to optimization and representation choices, and underspecified as a result, but invaluable for debugging purposes.

For example, when placed in a module named `m`, the ambigious-reference error from section 5.2 produces an error like this one:

```
x: identifier's binding is ambiguous
  context...:
   #(1772 module) #(1773 module m 0) #(2344 macro)
   #(2358 macro)
  matching binding...:
   #<module-path-index:()>
   #(1772 module) #(1773 module m 0) #(2344 macro)
  matching binding...:
   #<module-path-index:()>
   #(1772 module) #(1773 module m 0) #(2358 macro)
  in: x
```

Each scope is printed as a Racket vector, where the vector starts with a number that is distinct for every scope. A symbol afterward provides a hint at the scope's origin: `'module` for a `module` scope, `'macro` for a macro-introduction scope, `'use-site` for a macro use-site scope, or `'local` for a local binding form. In the case of a `'module` scope that corresponds to the inside edge, the module's name and a phase (since an inside-edge scope is generated for each phase) are shown.

The `#<module-path-index:()>`s in the error correspond to the binding, and they mean "in this module." Overall, the message shows that `x` has scopes corresponding to two different macro expansions, and it's bound by definitions that were produced by the expansions separately.

5.5 Scope Sets for JavaScript

Although the set-of-scopes model of binding was developed with Racket as a target, it is also intended as a more understandable model of macros to facilitate the creation of macro systems for other languages. In fact, the Racket implementation was not the first implementation of the model to become available.

Based on an early draft of this report, Tim Disney revised the Sweet.js macro implementation for JavaScript (Disney et al. 2014; Disney et al. 2015)[5] to use scope sets even before the initial Racket prototype was complete. Disney reports that the implementation of hygiene for the macro expander is now "mostly understandable" and faster.

6. Model

We present a formal model of set-of-scope expansion following the style of Flatt et al. (2012). Complete models, both in typeset form and executable form using PLT Redex, are provided as supplementary material.

As a first step, we present a model where only run-time expressions are expanded, and implementations of macros are simply parsed. As a second step, we generalize the model to include phase-specific scope sets and macro expansion at all phases. The third step adds support for local expansion, and the fourth step adds first-class definition contexts. The model does not cover modules or top-level namespaces.

6.1 Single-Phase Expansion

Our macro-expansion model targets a language that includes with variables, function calls, functions, atomic constants, lists, and syntax objects:

$$
\begin{aligned}
ast &::= var \mid \mathbf{App}(ast, ast, ...) \mid val \\
var &::= \mathbf{Var}(name) \\
val &::= \mathbf{Fun}(var, ast) \mid atom \mid \mathbf{List}(val, ...) \mid stx \\
stx &::= \mathbf{Stx}(atom, ctx) \mid \mathbf{Stx}(\mathbf{List}(stx, ...), ctx) \\
id &::= \mathbf{Stx}(sym, ctx) \\
atom &::= sym \mid prim \mid .... \\
sym &::= \text{'}name \\
name &::= \text{a token such as } \texttt{x}, \texttt{egg}, \text{or } \texttt{lambda}
\end{aligned}
$$

Since the model is concerned with macro expansion and programmatic manipulation of program terms, we carefully distinguish among

- *names*, which are abstract tokens;
- *variables*, which correspond to function arguments and references in an AST and are formed by wrapping a name as $\mathbf{Var}(name)$;
- *symbols*, which are values with respect to the evaluator and are formed by prefixing a name with a quote; and
- *identifiers*, which are also values, are formed by combining a symbol with a set of scopes, and are a subset of *syntax objects*.

For a further explanation of the distinctions among these different uses of names, see Flatt et al. (2012, section 3.2.1).

The model's evaluator is standard and relies on a δ function to implement primitives:

$$
\begin{aligned}
&\underline{\mathsf{eval} : ast \rightarrow val} \\
&\mathsf{eval}[\![\mathbf{App}(\mathbf{Fun}(var, ast_{body}), ast_{arg})]\!] = \mathsf{eval}[\![ast_{body}[var \leftarrow \mathsf{eval}[\![ast_{arg}]\!]]]\!] \\
&\mathsf{eval}[\![\mathbf{App}(prim, ast_{arg}, ...)]\!] = \delta(prim, \mathsf{eval}[\![ast_{arg}]\!], ...) \\
&\mathsf{eval}[\![val]\!] = val
\end{aligned}
$$

Interesting primitives include the ones that manipulate syntax objects,

$$prim ::= \texttt{stx-e} \mid \texttt{mk-stx} \mid ....$$

where **`stx-e`** extracts the content of a syntax object, and **`mk-stx`** creates a new syntax object with a given content and the scopes of a given existing syntax object:

$$
\begin{aligned}
\delta(\texttt{stx-e}, \mathbf{Stx}(val, ctx)) &= val \\
\delta(\texttt{mk-stx}, atom, \mathbf{Stx}(val, ctx)) &= \mathbf{Stx}(atom, ctx) \\
\delta(\texttt{mk-stx}, \mathbf{List}(stx, ...), \mathbf{Stx}(val, ctx)) &= \mathbf{Stx}(\mathbf{List}(stx, ...), ctx)
\end{aligned}
$$

Macro expansion takes a program that is represented as as a syntax object and produces a fully expanded syntax object. To evaluate the program, the syntax object must be parsed into an AST. The parser uses a resolve metafunction that takes an identifier and a binding store, Σ. The names `lambda`, `quote`, and `syntax`, represent the core syntactic forms, along with the implicit forms of function calls and variable reference:

$$
\begin{aligned}
&\underline{\mathsf{parse} : stx\ \Sigma \rightarrow ast} \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(id_{lam}, id_{arg}, stx_{body}), ctx), \Sigma]\!] = \mathbf{Fun}(\mathbf{Var}(\mathsf{resolve}[\![id_{arg}, \Sigma]\!]), \mathsf{parse}[\![stx_{body}, \Sigma]\!]) \\
&\quad \text{subject to } \mathsf{resolve}[\![id_{lam}, \Sigma]\!] = \texttt{lambda} \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(id_{quote}, stx), ctx), \Sigma]\!] = \mathsf{strip}[\![stx]\!] \\
&\quad \text{subject to } \mathsf{resolve}[\![id_{quote}, \Sigma]\!] = \texttt{quote} \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(id_{syntax}, stx), ctx), \Sigma]\!] = stx \\
&\quad \text{subject to } \mathsf{resolve}[\![id_{syntax}, \Sigma]\!] = \texttt{syntax} \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(stx_{rator}, stx_{rand}, ...), ctx), \Sigma]\!] = \mathbf{App}(\mathsf{parse}[\![stx_{rator}, \Sigma]\!], \mathsf{parse}[\![stx_{rand}, \Sigma]\!], ...) \\
&\mathsf{parse}[\![id, \Sigma]\!] = \mathbf{Var}(\mathsf{resolve}[\![id, \Sigma]\!])
\end{aligned}
$$

The resolve metafunction extracts an identifier's name and its binding context. For now, we ignore phases and define a binding context as simply a set of scopes. A binding store maps a name to a mapping from scope sets to bindings, where bindings are represented by fresh names.

$$
\begin{aligned}
ctx &::= \overline{scp} \\
\overline{scp} &::= \{scp, ...\} \\
\Sigma &::= \text{binding store}, name \rightarrow (\overline{scp} \rightarrow name) \\
scp &::= \text{a token that represents a scope}
\end{aligned}
$$

The resolve metafunction uses these pieces along with a biggest-subset helper function to select a binding. If no binding is available in the store, the identifier's symbol's name is returned, which effectively allows access to the four primitive syntactic forms; the macro expander will reject any other unbound reference.

$$
\begin{aligned}
&\underline{\mathsf{resolve} : id\ \Sigma \rightarrow name} \\
&\mathsf{resolve}[\![\mathbf{Stx}(\text{'}name, ctx), \Sigma]\!] = name_{biggest} \\
&\quad \text{subject to } \Sigma(name) = \{\overline{scp}_{bind} \leftarrow name_{bind}, ...\}, \\
&\qquad\qquad \mathsf{biggest\text{-}subset}[\![ctx, \{\overline{scp}_{bind}, ...\}]\!] = \overline{scp}_{biggest}, \\
&\qquad\qquad \{\overline{scp}_{bind} \leftarrow name_{bind}, ...\}(\overline{scp}_{biggest}) = name_{biggest} \\
&\mathsf{resolve}[\![\mathbf{Stx}(\text{'}name, ctx), \Sigma]\!] = name
\end{aligned}
$$

$$
\begin{aligned}
&\underline{\mathsf{biggest\text{-}subset} : \overline{scp}\ \{\overline{scp}, ...\} \rightarrow \overline{scp}} \\
&\mathsf{biggest\text{-}subset}[\![\overline{scp}_{ref}, \{\overline{scp}_{bind}, ...\}]\!] = \overline{scp}_{biggest} \\
&\quad \text{subject to } \overline{scp}_{biggest} \subseteq \overline{scp}_{ref}, \overline{scp}_{biggest} \in \{\overline{scp}_{bind}, ...\}, \\
&\qquad\qquad \overline{scp}_{bind} \subseteq \overline{scp}_{ref} \Rightarrow \overline{scp}_{bind} \subseteq \overline{scp}_{biggest}
\end{aligned}
$$

Finally, we're ready to define the expand metafunction. In addition to a syntax object (for a program to expand) and a bind-

---

[5] See pull request 461 at `https://github.com/mozilla/sweet.js/pull/461`.

ing store, the expander needs an environment, ξ, that maps bindings to compile-time meanings. The possible meanings of a binding are the three primitive syntactic forms recognized by parse, the `let-syntax` primitive form, a reference to a function argument, or a compile-time value—where a compile-time function represents a macro transformer.

$$\begin{aligned} \xi &::= \text{a mapping from } \mathit{name} \text{ to } \mathit{transform} \\ \mathit{transform} &::= \texttt{lambda} \mid \texttt{let-syntax} \mid \texttt{quote} \mid \texttt{syntax} \mid \text{VAR}(\mathit{id}) \mid \mathit{val} \end{aligned}$$

The process of macro expansion creates new bindings, so the expand metafunction produces a tuple containing an updated binding store along with the expanded program. For example, the simplest case is for the `quote` form, which leaves the body of the form and the store as-is:

$$\mathsf{expand} : \mathit{stx}\ \xi\ \Sigma \rightarrow \langle \mathit{stx}, \Sigma \rangle$$

$$\begin{aligned} &\mathsf{expand}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{quote}, \mathit{stx}), \mathit{ctx}), \xi, \Sigma]\!] = \langle \mathbf{Stx}(\mathbf{List}(\mathit{id}_{quote}, \mathit{stx}), \mathit{ctx}), \Sigma \rangle \\ &\quad \text{subject to } \mathsf{resolve}[\![\mathit{id}_{quote}, \Sigma]\!] = \texttt{quote} \end{aligned}$$

Since we are not yet dealing with expansion of compile-time terms, no scope pruning is needed for `syntax`, and it can be essentially the same as `quote`.

$$\begin{aligned} &\mathsf{expand}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{syntax}, \mathit{stx}), \mathit{ctx}), \xi, \Sigma]\!] = \langle \mathbf{Stx}(\mathbf{List}(\mathit{id}_{syntax}, \mathit{stx}), \mathit{ctx}), \Sigma \rangle \\ &\quad \text{subject to } \mathsf{resolve}[\![\mathit{id}_{syntax}, \Sigma]\!] = \texttt{syntax} \end{aligned}$$

Expansion of a `lambda` form creates a fresh name and fresh scope for the argument binding. Adding the new scope to the formal argument (we define the add metafunction later) creates the binding identifier. The new binding is added to the store, Σ, and it is also recorded in the compile-time environment, ξ, as a variable binding. The body of the function is expanded with those extensions after receiving the new scope, and the pieces are reassembled into a `lambda` form.

$$\begin{aligned} &\mathsf{expand}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{lam}, \mathit{id}_{arg}, \mathit{stx}_{body}), \mathit{ctx}), \xi, \Sigma]\!] = \langle \mathbf{Stx}(\mathbf{List}(\mathit{id}_{lam}, \mathit{id}_{new}, \mathit{stx}_{body2}), \mathit{ctx}), \Sigma_4 \rangle \\ &\quad \text{subject to } \mathsf{resolve}[\![\mathit{id}_{lam}, \Sigma]\!] = \texttt{lambda}, \mathsf{alloc\text{-}name}[\![\Sigma]\!] = \langle \mathit{name}_{new}, \Sigma_1 \rangle, \\ &\qquad\qquad \mathsf{alloc\text{-}scope}[\![\Sigma_1]\!] = \langle \mathit{scp}_{new}, \Sigma_2 \rangle, \mathsf{add}[\![\mathit{id}_{arg}, \mathit{scp}_{new}]\!] = \mathit{id}_{new}, \\ &\qquad\qquad \Sigma_2 + \{\mathit{id}_{new} \rightarrow \mathit{name}_{new}\} = \Sigma_3, \xi + \{\mathit{name}_{new} \rightarrow \text{VAR}(\mathit{id}_{new})\} = \xi_{new}, \\ &\qquad\qquad \mathsf{expand}[\![\mathsf{add}[\![\mathit{stx}_{body}, \mathit{scp}_{new}]\!], \xi_{new}, \Sigma_3]\!] = \langle \mathit{stx}_{body2}, \Sigma_4 \rangle \end{aligned}$$

When the generated binding is referenced (i.e., when resolving an identifier produces a binding that is mapped as a variable), then the reference is replaced with the recorded binding, so that the reference is `bound-identifier=?` to the binding in the expansion result.

$$\begin{aligned} &\mathsf{expand}[\![\mathit{id}, \xi, \Sigma]\!] = \langle \mathit{id}_{new}, \Sigma \rangle \\ &\quad \text{subject to } \xi(\mathsf{resolve}[\![\mathit{id}, \Sigma]\!]) = \text{VAR}(\mathit{id}_{new}) \end{aligned}$$

A local macro binding via `let-syntax` is similar to an argument binding, but the compile-time environment records a macro transformer instead of a variable. The transformer is produced by using parse and then eval on the compile-time expression for the transformer. After the body is expanded, the macro binding is no longer needed, so the body expansion is the result.

$$\begin{aligned} &\mathsf{expand}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{ls}, \mathit{id}, \mathit{stx}_{rhs}, \mathit{stx}_{body}), \mathit{ctx}), \xi, \Sigma]\!] = \mathsf{expand}[\![\mathit{stx}_{body2}, \xi_{new}, \Sigma_3]\!] \\ &\quad \text{subject to } \mathsf{resolve}[\![\mathit{id}_{ls}, \Sigma]\!] = \texttt{let-syntax}, \mathsf{alloc\text{-}name}[\![\Sigma]\!] = \langle \mathit{name}_{new}, \Sigma_1 \rangle, \\ &\qquad\qquad \mathsf{alloc\text{-}scope}[\![\Sigma_1]\!] = \langle \mathit{scp}_{new}, \Sigma_2 \rangle, \mathsf{add}[\![\mathit{id}, \mathit{scp}_{new}]\!] = \mathit{id}_{new}, \\ &\qquad\qquad \Sigma_2 + \{\mathit{id}_{new} \rightarrow \mathit{name}_{new}\} = \Sigma_3, \\ &\qquad\qquad \xi + \{\mathit{name}_{new} \rightarrow \mathsf{eval}[\![\mathsf{parse}[\![\mathit{stx}_{rhs}, \Sigma_3]\!]]\!]\} = \xi_{new}, \\ &\qquad\qquad \mathsf{add}[\![\mathit{stx}_{body}, \mathit{scp}_{new}]\!] = \mathit{stx}_{body2} \end{aligned}$$

Finally, when the expander encounters an identifier that resolves to a binding mapped to a macro transformer, the transformer is applied to the macro use. Fresh scopes are generated to represent the use site, scp_u, and introduced syntax, scp_i, where the introduced-syntax scope is applied using flip to both the macro argument and result, where flip corresponds to an exclusive-or operation to leave the scope intact on syntax introduced by the macro (see below).

$$\begin{aligned} &\mathsf{expand}[\![\mathit{stx}_{macapp}, \xi, \Sigma]\!] = \mathsf{expand}[\![\mathsf{flip}[\![\mathit{stx}_{exp}, \mathit{scp}_i]\!], \xi, \Sigma_3]\!] \\ &\quad \text{subject to } \mathit{stx}_{macapp} = \mathbf{Stx}(\mathbf{List}(\mathit{id}_{mac}, \mathit{stx}_{arg}, ...), \mathit{ctx}), \\ &\qquad\qquad \xi(\mathsf{resolve}[\![\mathit{id}_{mac}, \Sigma]\!]) = \mathit{val}, \\ &\qquad\qquad \mathsf{alloc\text{-}scope}[\![\Sigma]\!] = \langle \mathit{scp}_u, \Sigma_2 \rangle, \\ &\qquad\qquad \mathsf{alloc\text{-}scope}[\![\Sigma_2]\!] = \langle \mathit{scp}_i, \Sigma_3 \rangle, \\ &\qquad\qquad \mathsf{eval}[\![\mathbf{App}(\mathit{val}, \mathsf{flip}[\![\mathsf{add}[\![\mathit{stx}_{macapp}, \mathit{scp}_u]\!], \mathit{scp}_i]\!])]\!] = \mathit{stx}_{exp} \end{aligned}$$

The only remaining case of expand is to recur for function-call forms, threading through the binding store using an accumulator-style expand* helper:

$$\begin{aligned} &\mathsf{expand}[\![\mathbf{Stx}(\mathbf{List}(\mathit{stx}_{rtor}, \mathit{stx}_{rnd}, ...), \mathit{ctx}), \xi, \Sigma]\!] = \langle \mathbf{Stx}(\mathbf{List}(\mathit{stx}_{exprtor}, \mathit{stx}_{exprnd}, ...), \mathit{ctx}), \Sigma_1 \rangle \\ &\quad \text{subject to } \mathsf{expand^*}[\![(), (\mathit{stx}_{rtor}\ \mathit{stx}_{rnd}\ ...), \xi, \Sigma]\!] = \langle (\mathit{stx}_{exprtor}\ \mathit{stx}_{exprnd}\ ...), \Sigma_1 \rangle \end{aligned}$$

$$\mathsf{expand^*} : (\mathit{stx}\ ...)\ (\mathit{stx}\ ...)\ \xi\ \Sigma \rightarrow \langle (\mathit{stx}\ ...), \Sigma \rangle$$

$$\begin{aligned} &\mathsf{expand^*}[\![(\mathit{stx}_{done}\ ...), (), \xi, \Sigma]\!] = \langle (\mathit{stx}_{done}\ ...), \Sigma \rangle \\ &\mathsf{expand^*}[\![(\mathit{stx}_{done}\ ...), (\mathit{stx}_0\ \mathit{stx}_1\ ...), \xi, \Sigma]\!] = \mathsf{expand^*}[\![(\mathit{stx}_{done}\ ...\ \mathit{stx}_{done0}), (\mathit{stx}_1\ ...), \xi, \Sigma_1]\!] \\ &\quad \text{subject to } \mathsf{expand}[\![\mathit{stx}_0, \xi, \Sigma]\!] = \langle \mathit{stx}_{done0}, \Sigma_1 \rangle \end{aligned}$$

For completeness, here are the add and flip metafunctions for propagating scopes to all parts of a syntax object, where $\mathit{scp} \oplus \mathit{ctx}$ adds scp to ctx if is not already in ctx or removes it otherwise:

$$\mathsf{add} : \mathit{stx}\ \mathit{scp} \rightarrow \mathit{stx}$$

$$\begin{aligned} &\mathsf{add}[\![\mathbf{Stx}(\mathit{atom}, \mathit{ctx}), \mathit{scp}]\!] = \mathbf{Stx}(\mathit{atom}, \{\mathit{scp}\} \cup \mathit{ctx}) \\ &\mathsf{add}[\![\mathbf{Stx}(\mathbf{List}(\mathit{stx}, ...), \mathit{ctx}), \mathit{scp}]\!] = \mathbf{Stx}(\mathbf{List}(\mathsf{add}[\![\mathit{stx}, \mathit{scp}]\!], ...), \{\mathit{scp}\} \cup \mathit{ctx}) \end{aligned}$$

$$\mathsf{flip} : \mathit{stx}\ \mathit{scp} \rightarrow \mathit{stx}$$

$$\begin{aligned} &\mathsf{flip}[\![\mathbf{Stx}(\mathit{atom}, \mathit{ctx}), \mathit{scp}]\!] = \mathbf{Stx}(\mathit{atom}, \mathit{scp} \oplus \mathit{ctx}) \\ &\mathsf{flip}[\![\mathbf{Stx}(\mathbf{List}(\mathit{stx}, ...), \mathit{ctx}), \mathit{scp}]\!] = \mathbf{Stx}(\mathbf{List}(\mathsf{flip}[\![\mathit{stx}, \mathit{scp}]\!], ...), \mathit{scp} \oplus \mathit{ctx}) \end{aligned}$$

To take a program from source to value, use expand, then parse, then eval.

6.2 Multi-Phase Expansion

To support phase-specific scope sets, we change the definition of ctx so that it is a mapping from phases to scope sets:

$$\begin{aligned} \mathit{ph} &::= \mathit{integer} \\ \mathit{ctx} &::= \text{a mapping from } \mathit{ph} \text{ to } \overline{\mathit{scp}} \end{aligned}$$

With this change, many metafunctions must be indexed by the current phase of expansion. For example, the result of resolve depends on the current phase:

$$\mathsf{resolve} : \mathit{ph}\ \mathit{id}\ \Sigma \rightarrow \mathit{name}$$

$$\begin{aligned} &\mathsf{resolve}_{ph}[\![\mathbf{Stx}(\text{'}\mathit{name}, \mathit{ctx}), \Sigma]\!] = \mathit{name}_{biggest} \\ &\quad \text{subject to } \Sigma(\mathit{name}) = \{\overline{\mathit{scp}}_{bind} \leftarrow \mathit{name}_{bind}, ...\}, \\ &\qquad\qquad \mathsf{biggest\text{-}subset}[\![\mathit{ctx}(\mathit{ph}), \{\overline{\mathit{scp}}_{bind}, ...\}]\!] = \overline{\mathit{scp}}_{biggest}, \\ &\qquad\qquad \{\overline{\mathit{scp}}_{bind} \leftarrow \mathit{name}_{bind}, ...\}(\overline{\mathit{scp}}_{biggest}) = \mathit{name}_{biggest} \\ &\mathsf{resolve}_{ph}[\![\mathbf{Stx}(\text{'}\mathit{name}, \mathit{ctx}), \Sigma]\!] = \mathit{name} \end{aligned}$$

Phase-specific expansion allows `let-syntax` to expand the compile-time expression for a macro implementation, instead of just parsing the expression. Note that the uses of expand and parse on the transformer expression are indexed by $\mathit{ph}+1$:

$$\mathsf{expand} : \mathit{ph}\ \mathit{stx}\ \xi\ \overline{\mathit{scp}}\ \Sigma \rightarrow \langle \mathit{stx}, \Sigma \rangle$$

$$\begin{aligned} &\mathsf{expand}_{ph}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{ls}, \mathit{id}, \mathit{stx}_{rhs}, \mathit{stx}_{body}), \mathit{ctx}), \xi, \overline{\mathit{scp}}_p, \Sigma]\!] = \mathsf{expand}_{ph}[\![\mathit{stx}_{body2}, \xi_2, \overline{\mathit{scp}}_{p2}, \Sigma_4]\!] \\ &\quad \text{subject to } \mathsf{resolve}_{ph}[\![\mathit{id}_{ls}, \Sigma]\!] = \texttt{let-syntax}, \mathsf{alloc\text{-}name}[\![\Sigma]\!] = \langle \mathit{name}_{new}, \Sigma_1 \rangle, \\ &\qquad\qquad \mathsf{alloc\text{-}scope}[\![\Sigma_1]\!] = \langle \mathit{scp}_{new}, \Sigma_2 \rangle, \mathsf{add}_{ph}[\![\mathit{id}, \mathit{scp}_{new}]\!] = \mathit{id}_{new}, \\ &\qquad\qquad \Sigma_2 + \{\mathit{id}_{new} \rightarrow \mathit{name}_{new}\} = \Sigma_3, \mathsf{expand}_{ph+1}[\![\mathit{stx}_{rhs}, \xi_{primitives}, \varnothing, \Sigma_3]\!] = \langle \mathit{stx}_{exp}, \Sigma_4 \rangle, \\ &\qquad\qquad \xi + \{\mathit{name}_{new} \rightarrow \mathsf{eval}[\![\mathsf{parse}_{ph+1}[\![\mathit{stx}_{exp}, \Sigma_4]\!]]\!]\} = \xi_2, \mathsf{add}_{ph}[\![\mathit{stx}_{body}, \mathit{scp}_{new}]\!] = \mathit{stx}_{body2}, \\ &\qquad\qquad \{\mathit{scp}_{new}\} \cup \overline{\mathit{scp}}_p = \overline{\mathit{scp}}_{p2} \end{aligned}$$

In addition to carrying a phase index, the revised expand takes a set of scopes created for bindings. Those scopes are the ones to be pruned from quoted syntax by the revised `syntax` expansion:

$$\begin{aligned}&\mathsf{expand}_{ph}\llbracket \mathbf{STX}(\mathbf{LIST}(id_{syntax}, stx), ctx), \xi, \overline{scp}_p, \Sigma\rrbracket = \langle \mathbf{STX}(\mathbf{LIST}(id_{syntax}, stx_{pruned}), ctx), \Sigma\rangle\\ &\quad\text{subject to } \mathsf{resolve}_{ph}\llbracket id_{syntax}, \Sigma\rrbracket = \mathtt{syntax},\ \mathsf{prune}_{ph}\llbracket stx, \overline{scp}_p\rrbracket = stx_{pruned}\end{aligned}$$

The prune metafunction recurs through a syntax object to remove all of the given scopes at the indicated phase:

$$\underline{\mathsf{prune} : ph\ stx\ \overline{scp} \rightarrow stx}$$

$$\begin{aligned}&\mathsf{prune}_{ph}\llbracket \mathbf{STX}(atom, ctx), \overline{scp}_p\rrbracket &&= \mathbf{STX}(atom, ctx{+}\{ph \rightarrow ctx(ph) \setminus \overline{scp}_p\})\\ &\mathsf{prune}_{ph}\llbracket \mathbf{STX}(\mathbf{LIST}(stx, ...), ctx), \overline{scp}_p\rrbracket &&= \mathbf{STX}(\mathbf{LIST}(stx_{pruned}, ...), ctx{+}\{ph \rightarrow ctx(ph) \setminus \overline{scp}_p\})\\ &\quad\text{subject to } \mathsf{prune}_{ph}\llbracket stx, \overline{scp}_p\rrbracket, ... = stx_{pruned}, ...\end{aligned}$$

6.3 Local Expansion

Environment inspection via `syntax-local-value` and local expansion via `local-expand` are accommodated in the model essentially as in Flatt et al. (2012), but since local expansion can create bindings, the eval metafunction must consume and produce a binding store. The eval metafunction also must be index by the phase used for syntax operations.

Local expansion needs the current macro expansion's introduction scope, if any. In addition, local expansions that move identifiers into binding positions need `syntax-local-identifier-as-binding`, which requires information about scopes in the current expansion context. Local expansion, meanwhile, can create new such scopes. To support those interactions, eval and expand must both consume and produce scope sets for the current use-site scopes, and binding scopes must also be available for local expansion of `syntax` forms. To facilitate threading through all of that information, we define $s\hat{c}p$ as an optional current scope and $\hat{\Sigma}$ as an extended store:

$$\begin{aligned}s\hat{c}p &::= scp \mid \bullet\\ \hat{\Sigma} &::= \langle \Sigma, \overline{scp}, \overline{scp}\rangle\end{aligned}$$

The second part of a $\hat{\Sigma}$ tuple is a set of scopes to be pruned at `syntax` forms. The third part is a subset of those scopes that are the current expansion context's use-site scopes, which are pruned by `syntax-local-identifier-as-binding`. The different parts of a $\hat{\Sigma}$ tuple vary in different ways: the Σ part is consistently threaded through evaluation and expansion, while the scope-set parts are stack-like for expansion and threaded through evaluation. In the case of a macro-application step, the scope-set parts of the tuple are threaded through expansion, too, more like evaluation.

In the model, the **`lvalue`**, **`lexpand`**, and **`lbinder`** primitives represent `syntax-local-value`, `local-expand`, and `syntax-local-identifier-as-binding`, respectively:

$$\underline{\mathsf{eval} : ph\ ast\ s\hat{c}p\ \xi\ \hat{\Sigma} \rightarrow \langle val, \hat{\Sigma}\rangle}$$

$$\begin{aligned}&\mathsf{eval}_{ph}\llbracket \mathbf{APP}(\mathtt{\textbf{lvalue}}, ast_{id}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \xi(\mathsf{resolve}_{ph}\llbracket id_{result}, \Sigma_2\rrbracket), \hat{\Sigma}_2\rangle\\ &\quad\text{subject to } \mathsf{eval}_{ph}\llbracket ast_{id}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle id_{result}, \hat{\Sigma}_2\rangle, \hat{\Sigma}_2 = \langle \Sigma_2, \_, \_\rangle\end{aligned}$$

$$\begin{aligned}&\mathsf{eval}_{ph}\llbracket \mathbf{APP}(\mathtt{\textbf{lexpand}}, ast_{expr}, ast_{stops}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \mathsf{flip}_{ph}\llbracket stx_{exp}, scp_i\rrbracket, \hat{\Sigma}_4\rangle\\ &\quad\text{subject to } \mathsf{eval}_{ph}\llbracket ast_{expr}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle stx, \hat{\Sigma}_2\rangle,\\ &\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{stops}, scp_i, \xi, \hat{\Sigma}_2\rrbracket = \langle \mathbf{LIST}(id_{stop}, ...), \hat{\Sigma}_3\rangle,\\ &\qquad\qquad \{var \rightarrow \mathsf{unstop}\llbracket \xi(var)\rrbracket \mid var \in \mathrm{dom}(\xi)\} = \xi_{unstops},\\ &\qquad\qquad \hat{\Sigma}_3 = \langle \Sigma_3, \_, \_\rangle,\\ &\qquad\qquad \xi_{unstops}{+}\{\mathsf{resolve}_{ph}\llbracket id_{stop}, \Sigma_3\rrbracket \rightarrow \mathrm{STOP}(\xi(\mathsf{resolve}_{ph}\llbracket id_{stop}, \Sigma_3\rrbracket))\}\ ... = \xi_{stops},\\ &\qquad\qquad \mathsf{expand}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket stx, scp_i\rrbracket, \xi_{stops}, \hat{\Sigma}_3\rrbracket = \langle stx_{exp}, \hat{\Sigma}_4\rangle\end{aligned}$$

$$\begin{aligned}&\mathsf{eval}_{ph}\llbracket \mathbf{APP}(\mathtt{\textbf{lbinder}}, ast_{id}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \mathsf{prune}_{ph}\llbracket id_{result}, \overline{scp}_{u2}\rrbracket, \hat{\Sigma}_2\rangle\\ &\quad\text{subject to } \mathsf{eval}_{ph}\llbracket ast_{id}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle id_{result}, \hat{\Sigma}_2\rangle, \hat{\Sigma}_2 = \langle \_, \_, \overline{scp}_{u2}\rangle\end{aligned}$$

The implementation of **`lexpand`** uses a new STOP(*transform*) transformer to make an identifier a stopping point for expansion while remembering the former *transform* mapping of the identifier. The unstop helper function strips away a STOP constructor:

$$\underline{\mathsf{unstop} : transform \rightarrow transform}$$

$$\begin{aligned}&\mathsf{unstop}\llbracket \mathrm{STOP}(transform)\rrbracket &&= transform\\ &\mathsf{unstop}\llbracket transform\rrbracket &&= transform\end{aligned}$$

The expander must recognize STOP transformers to halt expansion at that point:

$$\underline{\mathsf{expand} : ph\ stx\ \xi\ \hat{\Sigma} \rightarrow \langle stx, \hat{\Sigma}\rangle}$$

$$\begin{aligned}&\mathsf{expand}_{ph}\llbracket \mathbf{STX}(\mathbf{LIST}(id_{stop}, stx, ...), ctx), \xi, \hat{\Sigma}\rrbracket = \langle \mathbf{STX}(\mathbf{LIST}(id_{stop}, stx, ...), ctx), \hat{\Sigma}\rangle\\ &\quad\text{subject to } \hat{\Sigma} = \langle \Sigma, \_, \_\rangle, \xi(\mathsf{resolve}_{ph}\llbracket id_{stop}, \Sigma\rrbracket) = \mathrm{STOP}(\_)\end{aligned}$$

The revised macro-application rule for expand shows how the use-site scopes component of $\hat{\Sigma}$ is updated and how the current application's macro-introduction scope is passed to eval:

$$\begin{aligned}&\mathsf{expand}_{ph}\llbracket stx_{macapp}, \xi, \langle \Sigma, \overline{scp}_p, \overline{scp}_u\rangle\rrbracket = \langle stx_{result}, \hat{\Sigma}_5\rangle\\ &\quad\text{subject to } stx_{macapp} = \mathbf{STX}(\mathbf{LIST}(id_{mac}, stx_{arg}, ...), ctx),\\ &\qquad\qquad \xi(\mathsf{resolve}_{ph}\llbracket id_{mac}, \Sigma\rrbracket) = val, \mathsf{alloc\text{-}scope}\llbracket \Sigma\rrbracket = \langle scp_u, \Sigma_2\rangle,\\ &\qquad\qquad \mathsf{alloc\text{-}scope}\llbracket \Sigma_2\rrbracket = \langle scp_i, \Sigma_3\rangle,\\ &\qquad\qquad \langle \Sigma_3, \{scp_u\} \cup \overline{scp}_p, \{scp_u\} \cup \overline{scp}_u\rangle = \hat{\Sigma}_3,\\ &\qquad\qquad \mathsf{eval}_{ph}\llbracket \mathbf{APP}(val, \mathsf{flip}_{ph}\llbracket \mathsf{add}_{ph}\llbracket stx_{macapp}, scp_u\rrbracket, scp_i\rrbracket), scp_i, \xi, \hat{\Sigma}_3\rrbracket = \langle stx_{exp}, \hat{\Sigma}_4\rangle,\\ &\qquad\qquad \mathsf{expand}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket stx_{exp}, scp_i\rrbracket, \xi, \hat{\Sigma}_4\rrbracket = \langle stx_{result}, \hat{\Sigma}_5\rangle\end{aligned}$$

In contrast, the revised `lambda` rule shows how the pruning scope set is extended for expanding the body of the function, the use-site scope set is reset to empty, and all extensions are discarded in the expansion's resulting store tuple.

$$\begin{aligned}&\mathsf{expand}_{ph}\llbracket \mathbf{STX}(\mathbf{LIST}(id_{lam}, id_{arg}, stx_{body}), ctx), \xi, \langle \Sigma, \overline{scp}_p, \overline{scp}_u\rangle\rrbracket\\ &\ = \langle \mathbf{STX}(\mathbf{LIST}(id_{lam}, id_{new}, stx_{body2}), ctx), \langle \Sigma_4, \overline{scp}_p, \overline{scp}_u\rangle\rangle\\ &\quad\text{subject to } \mathsf{resolve}_{ph}\llbracket id_{lam}, \Sigma\rrbracket = \mathtt{lambda}, \mathsf{alloc\text{-}name}\llbracket \Sigma\rrbracket = \langle name_{new}, \Sigma_1\rangle,\\ &\qquad\qquad \mathsf{alloc\text{-}scope}\llbracket \Sigma_1\rrbracket = \langle scp_{new}, \Sigma_2\rangle, \mathsf{add}_{ph}\llbracket id_{arg}, scp_{new}\rrbracket = id_{new},\\ &\qquad\qquad \Sigma_2{+}\{id_{new} \rightarrow name_{new}\} = \Sigma_3, \xi{+}\{name_{new} \rightarrow \mathrm{VAR}(id_{new})\} = \xi_{new},\\ &\qquad\qquad \mathsf{expand}_{ph}\llbracket \mathsf{add}_{ph}\llbracket stx_{body}, scp_{new}\rrbracket, \xi_{new}, \langle \Sigma_3, \{scp_{new}\} \cup \overline{scp}_p, \varnothing\rangle\rrbracket = \langle stx_{body2}, \langle \Sigma_4, \_, \_\rangle\rangle\end{aligned}$$

6.4 First-Class Definition Contexts

Supporting first-class definition contexts requires no further changes to the expand metafunction, but the eval metafunction must be extended to implement the **`new-defs`** and **`def-bind`** primitives, which model the `syntax-local-make-definition-context` and `syntax-local-bind-syntaxes` functions.

The **`new-defs`** primitive allocates a new scope to represent the definition context, and it also allocates a mutable reference to a compile-time environment that initially references the current environment. The two pieces are combined with a **DEFS** value constructor:

$$\begin{aligned}&\mathsf{eval}_{ph}\llbracket \mathbf{APP}(\mathtt{\textbf{new-defs}}), scp_i, \xi, \langle \Sigma, \overline{scp}_p, \overline{scp}_u\rangle\rrbracket = \langle \mathbf{DEFS}(scp_{defs}, addr_{env}), \hat{\Sigma}_3\rangle\\ &\quad\text{subject to } \mathsf{alloc\text{-}scope}\llbracket \Sigma\rrbracket = \langle scp_{defs}, \Sigma_2\rangle, \mathsf{alloc\text{-}def\text{-}env}\llbracket \Sigma_2\rrbracket = \langle addr_{env}, \Sigma_3\rangle,\\ &\qquad\qquad \langle \Sigma_2{+}\{addr_{env} \rightarrow \xi\}, \{scp_{defs}\} \cup \overline{scp}_p, \overline{scp}_u\rangle = \hat{\Sigma}_3\end{aligned}$$

The **`def-bind`** primitive works in two modes. In the first mode, it is given only a definition context and an identifier, and it creates a new binding for the identifier that includes the definition context's scope. The new binding is mapped to variable in an updated environment for definition context:

$$\begin{aligned}&\mathsf{eval}_{ph}\llbracket \mathbf{APP}(\mathtt{\textbf{def-bind}}, ast_{defs}, ast_{id}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle 0, \langle \Sigma_6, \overline{scp}_{p3}, \overline{scp}_{u3}\rangle\rangle\\ &\quad\text{subject to } \mathsf{eval}_{ph}\llbracket ast_{defs}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \mathbf{DEFS}(scp_{defs}, addr_{env}), \hat{\Sigma}_2\rangle,\\ &\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{id}, scp_i, \xi, \hat{\Sigma}_2\rrbracket = \langle id_{arg}, \hat{\Sigma}_3\rangle, \hat{\Sigma}_3 = \langle \Sigma_3, \overline{scp}_{p3}, \overline{scp}_{u3}\rangle,\\ &\qquad\qquad \mathsf{add}_{ph}\llbracket \mathsf{prune}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket id_{arg}, scp_i\rrbracket, \overline{scp}_{u3}\rrbracket, scp_{defs}\rrbracket = id_{defs},\\ &\qquad\qquad \mathsf{alloc\text{-}name}\llbracket \Sigma_3\rrbracket = \langle name_{new}, \Sigma_4\rangle, \Sigma_4{+}\{id_{defs} \rightarrow name_{new}\} = \Sigma_5,\\ &\qquad\qquad \Sigma_5(addr_{env}) = \xi_{defs},\\ &\qquad\qquad \Sigma_5{+}\{addr_{env} \rightarrow \xi_{defs}{+}\{name_{new} \rightarrow \mathrm{VAR}(id_{defs})\}\} = \Sigma_6\end{aligned}$$

When **`def-bind`** is given an additional syntax object, it expands and evaluates the additional syntax object as a compile-time expression, and it adds a macro binding to the definition context's environment:

$$\begin{aligned}
&\mathsf{eval}_{ph}\llbracket \textbf{A}\textsc{pp}(\texttt{def-bind}, ast_{defs}, ast_{id}, ast_{stx}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle 0, \langle \Sigma_9, \overline{scp}_{p4}, \overline{scp}_{u4}\rangle\rangle \\
&\text{ subject to } \mathsf{eval}_{ph}\llbracket ast_{defs}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \textbf{D}\textsc{efs}(scp_{defs}, addr_{env}), \hat{\Sigma}_2\rangle, \\
&\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{id}, scp_i, \xi, \hat{\Sigma}_2\rrbracket = \langle id_{arg}, \hat{\Sigma}_3\rangle, \\
&\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{stx}, scp_i, \xi, \hat{\Sigma}_3\rrbracket = \langle stx_{arg}, \hat{\Sigma}_4\rangle, \hat{\Sigma}_4 = \langle \Sigma_4, \overline{scp}_{p4}, \overline{scp}_{u4}\rangle, \\
&\qquad\qquad \mathsf{expand}_{ph+1}\llbracket \mathsf{add}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket stx_{arg}, scp_i\rrbracket, scp_{defs}\rrbracket, \xi_{primitives}, \langle \Sigma_4, \emptyset, \emptyset\rangle\rrbracket = \langle stx_{exp}, \langle \Sigma_5, \_, \_\rangle\rangle, \\
&\qquad\qquad \mathsf{eval}_{ph}\llbracket \mathsf{parse}_{ph+1}\llbracket stx_{exp}, \Sigma_5\rrbracket, \bullet, \xi, \langle \Sigma_5, \overline{scp}_{p4}, \emptyset\rangle\rrbracket = \langle val_{exp}, \langle \Sigma_6, \_, \_\rangle\rangle, \\
&\qquad\qquad \Sigma_6(addr_{env}) = \xi_{defs}, \mathsf{add}_{ph}\llbracket \mathsf{prune}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket id_{arg}, scp_i\rrbracket, \overline{scp}_{u4}\rrbracket, scp_{defs}\rrbracket = id_{defs}, \\
&\qquad\qquad \mathsf{alloc\text{-}name}\llbracket \Sigma_6\rrbracket = \langle name_{new}, \Sigma_7\rangle, \Sigma_7{+}\{id_{defs} \rightarrow name_{new}\} = \Sigma_8, \\
&\qquad\qquad \Sigma_8{+}\{addr_{env} \rightarrow \xi_{defs}{+}\{name_{new} \rightarrow val_{exp}\}\} = \Sigma_9
\end{aligned}$$

Note that **`def-bind`** in this mode defines a potentially recursive macro, since the definition context's scope is added to compile-time expression before expanding and parsing it.

Finally, a definition context is used to expand an expression by providing the definition context as an extra argument to **`lexpand`**. The implementation of the new case for **`lexpand`** is similar to the old one, but the definition context's scope is applied to the given syntax object before expanding it, and the definition context's environment is used for expansion.

$$\begin{aligned}
&\mathsf{eval}_{ph}\llbracket \textbf{A}\textsc{pp}(\texttt{lexpand}, ast_{expr}, ast_{stops}, ast_{defs}), scp_i, \xi, \hat{\Sigma}\rrbracket = \langle \mathsf{flip}_{ph}\llbracket stx_{exp}, scp_i\rrbracket, \hat{\Sigma}_5\rangle \\
&\text{ subject to } \mathsf{eval}_{ph}\llbracket ast_{expr}, scp_i, \xi, \hat{\Sigma}\rrbracket = \langle stx, \hat{\Sigma}_2\rangle, \\
&\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{stops}, scp_i, \xi, \hat{\Sigma}_2\rrbracket = \langle \textbf{L}\textsc{ist}(id_{stop}, ...), \hat{\Sigma}_3\rangle, \\
&\qquad\qquad \mathsf{eval}_{ph}\llbracket ast_{defs}, scp_i, \xi, \hat{\Sigma}_3\rrbracket = \langle \textbf{D}\textsc{efs}(scp_{defs}, addr_{env}), \hat{\Sigma}_4\rangle, \hat{\Sigma}_4 = \langle \Sigma_4, \_, \_\rangle, \\
&\qquad\qquad \Sigma_4(addr_{env}) = \xi_{defs}, \\
&\qquad\qquad \{var \rightarrow \mathsf{unstop}\llbracket \xi_{defs}(var)\rrbracket \mid var \in \mathrm{dom}(\xi_{defs})\} = \xi_{unstops}, \\
&\qquad\qquad \xi_{unstops}{+}\{\mathsf{resolve}_{ph}\llbracket id_{stop}, \Sigma_4\rrbracket \rightarrow \textsc{Stop}(\xi_{defs}(\mathsf{resolve}_{ph}\llbracket id_{stop}, \Sigma_4\rrbracket))\} \; ... = \xi_{stops}, \\
&\qquad\qquad \mathsf{expand}_{ph}\llbracket \mathsf{add}_{ph}\llbracket \mathsf{flip}_{ph}\llbracket stx, scp_i\rrbracket, scp_{defs}\rrbracket, \xi_{stops}, \hat{\Sigma}_4\rrbracket = \langle stx_{exp}, \hat{\Sigma}_5\rangle
\end{aligned}$$

7. Defining Hygiene

Although most previous work on hygienic macros has focused on expansion algorithms, some work addresses the question of what *hygiene* means independent of a particular expansion algorithm. In his dissertation, Herman (2008) addresses the question through a type system that constrains and exposes the binding structure of macro expansions, so that α-renaming can be applied to unexpanded programs. More recently, Adams (2015) defines hygienic macro-expansion steps as obeying invariants that are expressed in terms of renaming via nominal logic (Pitts 2003), and the concept of equivariance plays an important role in characterizing hygienic macro transformers.

Since our notion of binding is based on scope sets instead of renaming, previous work on defining hygiene via renaming does not map directly into our setting. A related obstacle is that our model transforms a syntax object to a syntax object, instead of directly producing an AST; that difference is necessary to support local and partial expansion, which in turn is needed for definition contexts. A more technical obstacle is that we have specified expansion in terms of a meta-function (i.e., a big-step semantics) instead of as a rewriting system (i.e., a small-step semantics).

Adams's approach to defining hygiene nevertheless seems applicable to our notion of binding. We leave a full exploration for future work, but we can offer an informed guess about how that exploration will turn out.

Although our model of expansion does not incorporate renaming as a core concept, if we make assumptions similar to Adams (including omitting the `quote` form), then a renaming property seems useful and within reach. For a given set of scopes and a point during expansion (exclusive of macro invocations), the symbol can be swapped in every identifier that has a superset of the given set of scopes; such a swap matches the programmer's intuition that any variable can be consistently renamed within a binding region, which corresponds to a set of scopes. Hygienic expansion then means that the parse of the continued expansion after swapping is α-equivalent to what it would be without swapping. An individual transformer could be classified as hygienic based on all introduced identifiers having a fresh scope, so that they cannot bind any non-introduced identifiers; the fresh scope ensures an analog to Adams's equivariance with respect to binders.

Note that swapping `x` with `y` for the scope set $\{a_{def}, b_{intro1}\}$ would *not* produce an equivalent program for the expansion in section 3.5, because it would convert an ambiguous reference $\mathtt{x}^{\{a_{def}, b_{intro1}, c_{intro2}\}}$ to an unambiguous $\mathtt{y}^{\{a_{def}, b_{intro1}, c_{intro2}\}}$. This failure should not suggest that the pattern-matching macros in that example are non-hygienic in themselves, but that the (implicit) definition-context macro is potentially non-hygienic. That is, a macro in a definition context can introduce an identifier that is captured at the macro-use site, since the definition and use sites can be the same. That potential for non-hygienic expansion appears to be one of the trade-offs of providing a context that allows a mixture of mutually recursive macro and variable definitions.

If macro bindings are constrained to `letrec-syntax`, and if macro implementations are constrained use `syntax-case`, `free-identifier=?`, and `syntax->datum` (not `bound-identifier=?` or `datum->syntax`), then we expect that all expansion steps will be provably hygienic and all macro transformers will be provably hygienic by the definitions sketched above.

8. Other Related Work

While our work shares certain goals with techniques for representing resolved bindings, such as de Bruijn indices, higher-order abstract syntax (Pfenning and Elliott 1988), and nominal sets (Pitts 2013), those techniques seem to be missing a dimension that is needed to incrementally resolve bindings as introduced and manipulated by macros. Adams (2015) demonstrates how pairs of conventional identifiers provide enough of an extra dimension for hygienic macro expansion, but supporting `datum->syntax` would require the further extension of reifying operations on identifiers (in the sense of marks and renamings). Scope sets provides the additional needed dimension in a simpler way.

Scope graphs (Neron et al. 2015) abstract over a program's syntax to represent the structure needed to resolve binding relationships—including support for constructs, such as modules and class bodies, that create static scopes different than nested lexical scopes. Binding resolution with macro expansion seems more dynamic, in that a program and its binding structure evolve during expansion, so that up-front scope graphs are not clearly applicable. Scope sets, meanwhile, do not explicitly represent import relationships, relying on macros that implement modular constructs to create scopes and bindings that reflect the import structure. Further work on scope graphs and scope sets seems needed to reveal the connections.

Stansifer and Wand (2014) build on the direction of Herman (2008) with Romeo, which supports program manipulations that respect scope by requiring that every transformer's type exposes its effect on binding. The resulting language is more general than Herman's macros, but transformers are more constrained than hygienic macros in Scheme and Racket.

9. Conclusion

Hygienic macro expansion is a successful, decades-old technology in Racket and the broader Scheme community. Hygienic macros have also found a place in some other languages, but the difficulties of specifying hygiene, understanding macro scope, and implementing a macro expander have surely been an obstacle to the broader adoption and use of hygienic macros. Those obstacles, in turn, suggest that our models of macro expansion have not yet hit the mark. Scope sets are an attempt to move the search for a model of macro expansion to a substantially different space, and initial results with Racket and JavaScript show that this new space is promising.

Bibliography

Michael D. Adams. Towards the Essence of Hygiene. In *Proc. Principles of Programming Languages*, 2015.

Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proc. Lisp and Functional Programming*, 1988.

William Clinger and Jonathan Rees. Macros that Work. In *Proc. Principles of Programming Languages*, 1991.

William D. Clinger. Hygienic Macros Through Explicit Renaming. *Lisp Pointers* 4(4), 1991.

Ryan Culpepper and Matthias Felleisen. Debugging Hygienic Macros. *Science of Computer Programming* 75(7), 2010.

Tim Disney et al. Sweet.js. 2015. `http://sweetjs.org/`

Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic Macros for ES5. In *Proc. Symposium on Dynamic Languages*, 2014.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), 1993.

Matthew Flatt. Compilable and Composable Macros, You Want it *When?* In *Proc. International Conference on Functional Programming*, 2002.

Matthew Flatt. Submodules in Racket: You Want it *When*, Again? In *Proc. Generative Programming: Concepts and Experiences*, 2013.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *Journal of Functional Programming* 22(2), 2012.

David Herman. Dissertation. PhD dissertation, Northeastern University, 2008.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, 1986.

Eugene E. Kohlbecker and Mitchell Wand. Macro-by-Example: Deriving Syntactic Transformations from their Specifications. In *Proc. Principles of Programming Languages*, 1987.

Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In *Proc. European Symposium on Programming*, 2015.

Frank Pfenning and Conal Elliott. Higher-order Abstract Syntax. In *Proc. Programming Language Design and Implementation*, 1988.

Andrew M. Pitts. Nominal Logic, a First Order Theory of Names and Binding. *Information and Computation* 186(2), 2003.

Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

Paul Stansifer and Mitch Wand. Romeo: a System for More Flexible Binding-Safe Programming. In *Proc. International Conference on Functional Programming*, 2014.

Oscar Waddell and R. Kent Dybvig. Extending the Scope of Syntactic Abstraction. In *Proc. Principles of Programming Languages*, 1999.