Keeping it Clean with Syntax Parameters

Eli Barzilay
Northeastern University
eli@barzilay.org

Ryan Culpepper
University of Utah
ryan@cs.utah.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

Racket's syntax parameters support the hygienic implementation of syntactic forms that would otherwise introduce implicit identifiers unhygienically.

1. Introduction

There are two common kinds of unhygienic macros in Scheme, distinguished by whether the bindings they introduce are based on identifiers from their arguments or completely independent. An example of the first kind is the `define-record-type` form of R6RS [Sperber (Ed.) 2007], which constructs identifiers that are synthesized from the names explicitly given to the macro. The unhygienic aspects of these macros do not lead to problems.

In contrast, the other common kind of unhygienic macro always binds the same name or names, and these macros are notoriously difficult to deal with. One example is a `while` loop form that provides an escape procedure as a binding for an auxiliary `abort` identifier. These auxiliary names are part of the macro's interface just like the literals that it recognizes: `while` and `abort` go together like `cond` and `else`. Unhygienic binding introduction, however, is a poor mechanism for implementing these auxiliary bindings. In this paper, we present examples of macros that bind auxiliary names, we show the problems that arise with existing hygienic and unhygienic implementation approaches, and we present an elegant solution, which we call *syntax parameters*.

In Section 2 we demonstrate the problem concretely using examples which motivate looking for a better solution. Section 3 investigates an alternative that frees us from the problems of unhygienic binding entirely, which leads to syntax parameters, which are described in Section 4. We then describe some of the existing uses of this facility in the Racket code base in Section 5, as well as some subtleties that macro writers may need to be aware of in Section 6.

But first, we begin with an introduction to the problem.

1.1 The Problem with Hygienic Macros

Although the benefits of hygienic macros are well established, there are occasions when traditional hygienic bindings are insufficient. Two well-known examples are looping macros that implicitly bind `abort` for use in the loop body to escape the loop [Clinger 1991], and "anaphoric conditionals" where the value of the tested expression is available as an `it` binding.

```
(define-syntax forever
  (syntax-rules ()
    [(forever body ...)
     (call/cc (lambda (abort)
                (let loop () body ... (loop))))]))

(define-syntax aif
  (syntax-rules ()
    [(aif test then else)
     (let ([it test])
       (if it then else))]))
```

In these examples, we wish to introduce the underlined identifiers as-is, unhygienically. Before we do so, we note that another popular design approach is to avoid unhygienic macros at all costs, which in this case dictates that instead of making up a new identifier we should make them part of the input to the macro. As we shall see in Section 2.2, this leads to the same kind of code management problem as the unhygienic solution.

Using a `syntax-case` macro system [Dybvig et al. 1993; Sperber (Ed.) 2007], macros can "break" hygiene by constructing new identifiers from a known name (a symbol) and the lexical scope of an existing identifier. In the `forever` macro example, we introduce `abort` unhygienically by giving it the lexical context of the `forever` input keyword.

```
(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body ...)
     (with-syntax ([abort (datum->syntax #'forever 'abort)])
       #'(call/cc (lambda (abort)
                    (let loop () body ... (loop)))))]))
```

That is, `forever` binds `abort`, and this binding is available in the body because the use of `abort` has the same context as the use of `forever`. Using these solutions can be tempting when `datum->syntax` is readily available and often serves as the classic example for breaking hygiene when needed. Such uses are, however, often severely broken.

1.2 The Problem with Unhygienic Macros

The problem with this approach is that it does not compose well with new macros that expand to uses of `forever`. For example, suppose that a `while` macro is defined as follows, with a goal of having `abort` as well:

```
(define-syntax while
  (syntax-rules ()
    [(while test body ...)
     (forever (unless test (abort)) body ...)]))
```

The use of `abort` that is introduced by `while` works, because it is introduced in the same context as the `forever` reference itself. That context is different, however, from the context of the `body` expressions, so `abort` is not available to the `body` expressions:

```
> (while #t (abort))
reference to undefined identifier: abort
```

The problem is that the `while` macro definition is itself hygienic, and therefore the implicit `abort` binding from `forever` is introduced hygienically with respect to `while`, making `abort` unavailable to the `while` macro's own `body` expressions. In terms of the `syntax-case` hygiene algorithm [Dybvig et al. 1993], the `abort` binding occurrence is created based on `forever`, which has a mark from the expansion of `while`. The marked `abort` binding captures the marked `abort` use that is also introduced by `while`, but not the unmarked reference to `abort` in the `while` macro's `body` expressions.

We can attempt to fix this mismatch by making `while` introduce `forever` itself unhygienically:

```
(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     (with-syntax ([forever (datum->syntax #'while 'forever)])
       #'(forever (unless test (abort)) body ...))]))
```

Now `while` fails in a different way: the `abort` that appears inside the `while` macro implementation is unbound, because it does not have the context of the `while` macro use. Another serious problem with this definition of `while` is that we have no guarantee that `forever` is bound where `while` is used. For example, a module might define `while` in terms of `forever` but only export `while`.

Yet another attempt to fix the problem is to use the lexical context of forms that come from the macro's input:

```
(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body1 body ...)
     (with-syntax ([abort (datum->syntax #'body1 'abort)])
       #'(call/cc (lambda (abort)
                    (let loop () body ... (loop)))))]))
```

This solution is too fragile: how do we know which input will come from the end use? What about macros that generate that first expression? But even if we ignore these questions, the main problem is that it still fails in the same way as the previous version.

The core of the problem lies in the fact that we want `abort` to be available for *both* the `while` macro code *and* its input code. Given that our macro system is hygienic, these will inevitably be two different scopes, and therefore two `abort` bindings are needed for the two scopes.[1]

To make things worse, we run into similar problems in macros that abstract over `abort`, such as an `abort-when` macro that expands to a use of `abort`, intended to be used in `forever` and `while` loops. Such macros must either be defined within the loop body, or they must carefully construct the reference to `abort` unhygienically too.

In both cases the lack of hygiene is infectious. If a new macro builds on an unhygienic macro, then the new macro must contain some unhygienic construction of identifiers as well. The resulting "chain of responsibility" hinders the creation and composition of such macros. We therefore need some new mechanism for modular binding of auxiliaries, one that does not hinder composition.

1.3 The Syntax Parameters Solution

An alternative solution is the subject of this paper: Racket's *syntax parameters*. In this solution, the `abort` and `it` identifiers of the above macros get actual definitions as syntax parameters which are initially unusable (that is, their initial transformers always raise errors),

```
(require racket/stxparam)
(define-syntax-parameter abort (syntax-rules ()))
(define-syntax-parameter it    (syntax-rules ()))
```

and then the corresponding macros "adjust" the meaning of these bindings for expansion of code in their body

```
(define-syntax forever
  (syntax-rules ()
    [(forever body ...)
     (call/cc (lambda (abort-k)
                (syntax-parameterize
                    ([abort (syntax-rules () [(_) (abort-k)])])
                  (let loop () body ... (loop)))))]))

(define-syntax aif
  (syntax-rules ()
    [(aif test then else)
     (let ([t test])
       (syntax-parameterize ([it (syntax-id-rules () [_ t])])
         (if t then else)))]))
```

In other words, instead of breaking hygiene, we create proper bindings of the auxiliary identifiers, which are then referred to like any other bindings.

Before we describe this mechanism, we first motivate it by attempting to "fix" the unhygienic approach in the next section.

2. Writing Correct Macros

In this section, we show how to write working unhygienic macros, by correctly linking the two contexts that are created by the unhygienic macros in Section 1.2. We then automate the linking process via a helper macro. In the end, we find that even this conveniently automated solution creates a chain of responsibility that interferes with modularity. We then consider the typical hygienic solution, and observe that it ends with even worse variation of the same macro modularity problem.

2.1 Correct Unhygienic Macros

As we have seen, the hygienic macro framework means that we have two different lexical scopes in the `while` macro: the first is its implementation body, and the second is the scope of the user's `body` expressions which `while` consumes. Since we want `abort` to be bound in both scopes, we need to introduce two different `abort` identifiers, one for each scope, and somehow link the two identifiers together so they have the same meaning. This is simple to do with a `let`, leading to a correct macro:

```
(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     (with-syntax (; abort* is accessible as 'abort'
                   [abort* (datum->syntax #'while 'abort)])
       #'(forever (let (; link the two bindings
                        [abort* abort])
                    (unless test (abort))
                    body ...)))]))
```

The `abort` binding that is introduced by `forever` covers the use of `abort` in the `unless` expression. The `let`-bound `abort` covers the `body` expressions. We assume that the `body` expressions have the same lexical context as the `while` identifier—or if not, that they also have code linking `while`'s `abort` to their own, just as this macro links `forever`'s `abort` to `while`'s.

Using this approach we can layer an additional macro and verify that the result works as expected.

[1] Faced with such problems, some people conclude that an unhygienic macro system is superior: in such systems there is essentially a single global lexical scope, and `abort` becomes a *symbol* that is bound throughout any parts of any code.

```
(define-syntax (until stx)
  (syntax-case stx ()
    [(until test body ...)
     (with-syntax ([abort* (datum->syntax #'until 'abort)])
       #'(while (not test)
           (let ([abort* abort]) body ...)))]))
```

We now have a working solution that is almost mechanical enough to be abstracted over by a higher-level macro. But there are two technical problems that we need to address. First, using `let` works in this case because `abort` is a variable binding—but this fails if the unhygienic identifier is bound to a macro.

Fortunately, Racket's macro system provides a solution for this problem: `(make-rename-transformer id)` creates a special kind of an identifier indirection macro that expands to `id` [Flatt and PLT 2010]. In fact, the resulting macro cooperates in additional ways with Racket's macro expander: for example, the identifier that is bound to it is considered `free-identifier=?` to `id`. This facility allows us to perform our linking at the syntactic level. The change is simple; instead of linking with `let`, we use `let-syntax` instead:

```
(let-syntax ([abort* (make-rename-transformer #'abort)])
  ...)
```

The second problem is harder to deal with: the sketched solution is not mechanical enough. We still need to know *where* to link the two `abort` identifiers together—we cannot just wrap the whole macro body with the linking `let-syntax`, since `forever`'s `abort` binding is yet to be created. The linking code must go *inside* the scope of the unhygienic binding. In the case of `while`, the linking must be placed inside the `forever` body.

To address this problem, we define our macro so that the linking point is marked explicitly with an `L`. We call the macro `define-syntax-rules/capture`, and `L` serves as an auxiliary binding for use in it. (Note that `L` is itself introduced unhygienically.)

Using this `define-syntax-rules/capture` macro, we can define `while` as follows, resulting in a macro that has the same behavior as the correct version that was written manually in the above:

```
(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
   (forever (L (unless test (abort)) body ...))])
```

The `define-syntax-rules/capture` macro consumes a name to be defined, a parenthesized sequence of unhygienic identifiers to propagate through the new definition, and the usual keywords and rewrite rules of `syntax-rules`. In the result templates, `L` marks the `let-syntax` linking points.

The implementation of `define-syntax-rules/capture` is shown in Figure 1. The definition is a little verbose and has a few subtle points, including the use of another Racket extension, `syntax-local-introduce`. However, the implementation is irrelevant for the purpose of our discussion—it suffices to know that such a macro *can* be defined, resulting in a way to conveniently define composable macros correctly.

Using `define-syntax-rules/capture`, we can even avoid writing the code that creates the initial unhygienic `abort` in the base `forever` macro: we simply let `define-syntax-rules/capture` do the required work for us. Our three looping macros are now succinctly defined as follows:

DEFINITION 1.

```
(define-syntax-rules/capture forever (abort) ()
  [(forever body ...)
   (call/cc (lambda (abort)
              (L (let loop () body ... (loop)))))])

(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
   (forever (L (unless test (abort)) body ...))])

(define-syntax-rules/capture until (abort) ()
  [(until test body ...)
   (while (L (not test)) (L body ...))])
```

Note that the first two links in the macro chain use `abort`, but in the `until` definition it is not used. It would therefore seem that we could replace that definition with a simpler one that uses `syntax-rules`:

```
(define-syntax until
  (syntax-rules ()
    [(until test body ...)
     (while (not test) body ...)]))
```

Doing so will, however, prevent "propagating" the `abort` binding to users of `until`—eliminating the uses of `L` drops the wrong `abort` binding.

Consider the following alternative definition of the three macros, where `while` is the base-level one, then `forever` and `until` are derived from it in sequence.

DEFINITION 2.

```
(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
   (call/cc (lambda (abort)
              (L (let loop ()
                   (when test body ... (loop))))))])

(define-syntax-rules/capture forever (abort) ()
  [(forever body ...)
   (while #t (L body ...))])

(define-syntax-rules/capture until (abort) ()
  [(until test body ...)
   (forever (L (unless test (abort))
               body ...))])
```

In this implementation, `forever` does not need the `abort` binding. If we further assume that it is not intended for public consumption, for example, if it is an internal helper macro for the `until` definition, then it seems that defining it via `syntax-rules` *should* work in this case. Such a definition will, again, break `until` since there must be an explicit link that ties `until` to the `abort` that `while` introduces.

Now that the code is clear of distractions, we can see the infectious nature of these bindings at work: once a macro introduces an identifier unhygienically, any other macro that is derived from it must itself do a similar unhygienic introduction. Any macro that fails to do so is breaking the chain, essentially making the introduced identifier unavailable to it and to any code that uses it. This is analogous to carrying arguments through a chain of function calls: once a function fails to pass on an argument, it is unavailable to other functions down the callee chain.

To conclude, this implementation strategy works, and we can even conveniently automate the plumbing work. However, see that it requires explicit linking, from the first macro that creates the binding, and up to all forms that are derived from it—either directly or indirectly, and whether the derived macros need to use the introduced identifier or not. This requirement is impractical: some macros in the chain might come from libraries that are not under

```
(define-syntax (define-syntax-rules/capture stx0)
  (syntax-case stx0 ()
    [(def name (capture ...) (keyword ...) [patt templ] ...)
     (with-syntax ([L (datum->syntax #'def 'L)])
       #'(define-syntax (name stx)
           (syntax-case stx (keyword ...)
             [patt (with-syntax ([user-ctx stx])
                     ;; pass the original syntax as a context carrier
                     #'(with-links L user-ctx (capture ...) templ))]
             ...)))]))

(define-syntax with-links
  (syntax-rules ()
    [(with-links L user-ctx (capture ...) template)
     (let-syntax
         ([L (lambda (stx)
               (syntax-case stx ()
                 [(L e (... ...))
                  (with-syntax ([(id  (... ...)) (list (datum->syntax #'L 'capture) ...)]
                                [(id* (... ...)) (list (syntax-local-introduce
                                                        (datum->syntax #'user-ctx 'capture))
                                                       ...)])
                    #'(let-syntax ([id* (make-rename-transformer #'id)]
                                   (... ...))
                        e (... ...)))]))])
       template)]))
```

Figure 1. The definition of `define-syntax-rules/capture`

our control, and composing macros with different unhygienic keywords makes for additional explicit linking. If we wish to create a language where a fundamental form like `if` is extended with such a keyword to create an anaphoric conditional, then we would need to link up the introduced `it` in *any* derived macros. This makes the effort of constructing and maintaining such languages prohibitively expensive.

A similar problem occurs in ordinary programming. Programs that perform I/O operate on an input port and an output port. Their behavior may also depend on a character encoding, a locale, a current directory, and many other variables. Passing these values as function arguments, even grouped together, is burdensome—and in cases where a fundamental feature of the language such as the default I/O ports is concerned, such explicit argument passing makes for a prohibitively expensive effort. Instead, such values are implemented as a kind of dynamically scoped values. Functions can access and update them without enumerating them in their interfaces, and consequently they do not hinder functional composition. We therefore consider that such dynamically scoped values can be applied to our problem at the syntax level—with similar benefits.

2.2 Comparison with the Hygienic Solution

At this point it is worth re-considering the strictly hygienic solution, where instead of making up identifiers unhygienically they are passed as inputs to the macros. This is a popular solution to such problems with unhygienic macros, yet it leads to exactly the same issue with respect to layering macros. Specifically, if we define our `forever` macro to take in `abort` as one of its inputs, then the derived `while` will need to do so as well.

To see this identifier cascading in action we translate the code from Definition 1 into this style. To make it more interesting, we add an anaphoric conditional, `aif`, into the mix and use it to implement `while`.

```
(define-syntax forever
  (syntax-rules ()
    [(forever abort body ...)
     (call/cc (lambda (abort)
                (let loop () body ... (loop))))]))

(define-syntax aif
  (syntax-rules ()
    [(aif it test then else)
     (let ([it test]) (if it then else))]))

(define-syntax while
  (syntax-rules ()
    [(while abort it test body ...)
     (forever abort
       (aif it test (begin body ...) (abort)))]))

(define-syntax until
  (syntax-rules ()
    [(until abort it test body ...)
     (while abort it (not test) body ...)]))
```

Note that the auxiliary identifiers—now hygienic—need to be carried through all macros, essentially achieving a similar kind of explicitly specified linking, but with this approach things are more complicated. Unlike the previous solution, however, the complication applies not only to the macro implementor, but to its users. For example, end programmers who wish to use `while` must specify both identifiers as well:

```
(while abort it (memq x l)
  (display (car it))
  (set! l (cdr it)))
```

In other words, the responsibility of maintaining the binding chain exists whether we use unhygienic or hygienic binding.

3. Dynamic Binding

The key to staying clean with `forever` is to think about `abort` differently. As we remarked in Section 1, `forever` and `abort` go together like cond and else. Scheme has a single definition of the else auxiliary keyword. Similarly, instead of having every occurrence of forever introduce a new local abort variable, there should be a single definition of the abort auxiliary syntax, defined at the same level that forever is—usually as a module top-level binding. The forever macro should "adjust" the meaning of abort within the context of the loop body, *without* introducing a new binding. In other words, abort becomes a kind of a meta-binding, dynamically adjustable for macro expansion. Since no *new* binding is introduced, there is no need to break hygiene.

The concept of "adjusting the meaning of a binding" does not exist in all macro systems; it would be a new feature for some. This concept does have a known precedent for run-time bindings, however, in the form of *dynamic bindings*.

Before we proceed to discuss the application of dynamic bindings at the syntax level, we should consider existing mechanisms related to dynamic scoping.

3.1 Dynamic Binding in the Runtime World

There are two common mechanisms to simulate dynamic bindings: one such mechanism is the fluid-let construct; another mechanism is based on parameter objects and the parameterize form.

The fluid-let simulation of dynamic scope mutates a set of bindings on entry to the body, and ensures (using dynamic-wind) that the old bindings are restored on exit from the body. For example, a thunk-based implementation of a loop that uses a dynamically scoped binding to abort the loop might be implemented as follows:

```
(define (abort)
  (error "abort must be used in a loop"))

(define (thunk-forever body-thunk)
  (call/cc
   (lambda (k)
     (fluid-let ([abort k])
       (let loop () (body-thunk) (loop))))))

(thunk-forever
 (lambda ()
   (let ([c (read-char)])
     (if (eof-object? c)
         (abort)
         (display (char-upcase c))))))
```

While fluid-let is properly simulating dynamic scope, it may lead to problems if used indiscriminately. For example, (fluid-let ([cons +]) ...) is unlikely to be a good idea. Indeed, Scheme dialects with a module system might prevent the assignment to cons, on the grounds that a random expression in some library should not be able to make such a global change. Even with such a restriction on changes to module-provided bindings, fluid-let is still too unrestricted in that there are still enough bindings for it to mutate, leading to broken code.

Using fluid-let makes the most sense when it adjusts identifiers that were defined with fluid-let in mind. For example, the above definition of abort is designed as an initially useless function, to be mutated into an abort continuation in the dynamic scope of a thunk-forever loop. If programmers are required to make this intent *explicit*, then dynamic binding can be implemented in a way that does not compromise all other bindings.

Along these lines, the other common approach for implementing dynamic bindings among Scheme systems is to provide a constructor for dynamic values and a way to adjust their value—make-parameter and parameterize [Dybvig 2009b; Feeley 2003; Flatt and PLT 2010] or similar forms. Re-implementing the above thunk-forever using parameters, we get:

```
(define current-abort
  (make-parameter
   (lambda () (error "abort must be used in a loop"))))

(define (abort) ((current-abort)))

(define (thunk-forever body-thunk)
  (call/cc
   (lambda (k)
     (parameterize ([current-abort k])
       (let loop () (body-thunk) (loop))))))
```

The parameter acts as a function that fetches its value when applied. The parameterize form plays the role of fluid-let, but it works only on parameter values, created by make-parameter. In this example, abort retrieves the value of the current-abort parameter,[2] and then applies this value to invoke the continuation it contains (or the default error function).

But abort serves another important role: it separates the right to *adjust* a parameter from the right to *access* its value. In this example, current-abort can be used to do both, but abort can only retrieve the value. We can put the above implementation in a module and provide only thunk-forever and abort out, making it impossible for the value of current-abort to be modified by any unknown code.

3.2 Dynamic Binding at the Syntax Level

Back at the syntax level, we can try the analogy to fluid-let, suggesting a fluid-let-syntax form, as in Chez Scheme [Dybvig 2009a]:

```
(fluid-let-syntax ([id expression] ...)
  body ...)
```

A fluid-let-syntax form is similar to let-syntax, but the transformers associated with existing *id*s are replaced with the new transformers while expanding *body*. That is, fluid-let-syntax does *not* introduce a new binding for each *id*.

The transformer adjustments for the *id*s apply "dynamically" during the expansion of *body*; that is, they apply not only to the *id*s that appear within the original fluid-let-syntax form, but also to any occurrences inserted by macros encountered during the expansion of *body*. Note that this form of "dynamic adjustments" happens at the meta-level of macro expansion—it should not be confused with dynamic scope in code.

Using fluid-let-syntax, our forever macro and its abort auxiliary can be implemented correctly as follows:

```
;; outside of a loop, 'abort' is always a syntax error
(define-syntax abort
  (syntax-rules ()))

(define-syntax forever
  (syntax-rules ()
    [(forever body ...)
     (call/cc (lambda (abort-k)
                (fluid-let-syntax
                    ([abort (syntax-rules () [(_) (abort-k)])])
                  (let loop () body ... (loop)))))]))
```

With this definition, the derived while and until forms can be defined as simple syntax-rules macros, they work as expected since they do not need to deal with propagating the abort binding.

Of course, a binding may be further adjusted by nested instances of fluid-let-syntax forms, so nested forever forms work as expected; and a binding may be shadowed by a local variable or

[2] In Racket, we frequently name parameters with a current- prefix.

syntax binding, so a local `let`-binding of `abort` inside a `forever` is a new binding, not the one that `forever` adjusts.

The problem with `fluid-let-syntax` is the same as the problem with `fluid-let`: indiscriminate use of `fluid-let-syntax` can expose the implementation details of a syntactic form that is defined elsewhere. In particular, imagine trying to predict the effect of using `fluid-let-syntax` on `lambda`; which syntactic forms expand to `lambda`, and which do not? Forms that expand to `lambdas` could get utterly broken, much like the damage that `(fluid-let ([cons +]) ...)` can inflict.

The natural solution to this problem is the same as for dynamic run-time values: introduce a new construct, so that a programmer who writes such macros can control which identifiers can be adjusted dynamically. We therefore continue with a similar analogy that is based on parameters.

4. Syntax Parameters

Adding `parameterize`-like capability to the syntax layer requires two new forms: one for declarations of adjustable bindings, and another to adjust such bindings. In Racket, these two parts are `define-syntax-parameter` and `syntax-parameterize`:

```
(define-syntax-parameter id expression)

(syntax-parameterize ([id expression] ...)
  body ...)
```

In both of these forms, the *`expression`* typically evaluates to a macro transformer, typically using `syntax-case`, but these forms are just as useful when used with simple `syntax-rules` macros.

A `define-syntax-parameter` form defines a macro, just like `define-syntax`. Indeed, if `syntax-parameterize` is never used, there is no difference between the two. Macro names defined using `define-syntax-parameter`, however, can be updated to use new transformers using `syntax-parameterize`.

The `syntax-parameterize` form is similar to `fluid-let-syntax`. Unlike `fluid-let-syntax`, each *`id`* in `syntax-parameterize` must refer to a syntax parameter defined in the environment where the `syntax-parameterize` occurs.

Using these two forms, the `forever` macro can be implemented as follows:

```
(define-syntax-parameter abort
  (syntax-rules ()))

(define-syntax forever
  (syntax-rules ()
    [(forever body ...)
     (call/cc (lambda (abort-k)
                (syntax-parameterize
                    ([abort (syntax-rules () [(_) (abort-k)])])
                  (let loop () body ... (loop)))))]))
```

Again, in Racket's case we can use other macro-producing expressions, such as `(make-rename-transformer #'abort-k)` which we have previously mentioned.

If only `forever` should be allowed to adjust the syntax parameter, then we can proceed in the same way we did with plain parameters: change the name of the above syntax parameter from `abort` to `internal-abort`. Then, `forever` can be exported from a library along with an `abort` macro that accesses the syntax parameter (by expanding to it) but *does not* grant an ability to update it:

```
(define-syntax abort
  (syntax-rules ()
    [(abort) (internal-abort)]))
```

The revised `forever` macro composes correctly with other macros, in the sense that hygienic macros can reliably expand into `forever` expressions. For example, the `while` macro works as expected, allowing both uses of `abort` introduced by the macro and in the original body expressions. Furthermore, a macro that abstracts over uses of `abort` can be defined hygienically and possibly outside of the loop body where it is used.

Besides preserving hygiene, syntax parameters have an important additional advantage over implicit identifiers: the syntax parameter identifier has the same status as other identifiers. When using a module system, it can be prefixed, renamed, and excluded just like the `forever` form, if the module system provides such functionality. This is useful in multiple ways, for example, when identifiers are translated to a different language, or if we wish to create a context where `while` is available but `abort` is not.

4.1 Implementation

Syntax parameters are not implemented directly in Racket's macro expander. Instead, they are built using other features of Racket and its macro system.

A use of `define-syntax-parameter` produces two syntax definitions. First, a fresh internal name is generated to represent the state of the syntax parameter; it is defined with the syntax parameter's initial value. Second, the syntax parameter name is defined as a syntax-parameter transformer containing the internal identifier. The syntax-parameter transformer is an *applicable structure* (a structure that can be used as an (expander) function); when the syntax parameter is used as a macro, it fetches the current value of the syntax parameter as described below and uses it to complete the macro transformation.

The current value of a syntax parameter is read and updated using `syntax-local-get-shadower`, a low-level function of the Racket macro system. Given a syntax parameter's internal name, *`internal`*, the function returns an identifier, *`shadower`*, capable of either referring to or shadowing the nearest enclosing binding that shadows *`internal`*. The syntax parameter's value can be read by accessing the *`shadower`*'s compile-time value (using Racket's `syntax-local-value`) or updated by creating a new `let-syntax` binding of *`shadower`*. Since *`shadower`* shadows *`internal`*, references to the syntax parameter within the scope of the new binding will find it as the nearest enclosing shadower of *`internal`*.

A more direct approach would be for `syntax-parameterize` to simply mutate compile-time state, or perhaps to use run-time parameters at compile time. The problem with this approach is that the side-effects are ephemeral; they are not preserved in expanded—or partly expanded—code. In particular, this interferes with Racket's use of partial expansion to implement definition contexts, both for standard forms such as `lambda` and macros such as `class`.

5. Other Uses

Although looping macros are a common example of unhygienic bindings, syntax parameters are more useful in larger, more sophisticated cases. A good example of such a case is the Racket class system.

Identifiers like `this` and `super` take on special meanings within Racket's `class` form. For example, `this` is automatically bound to the current instance object, just as in Java. To bring `this` into a method's scope, the `class` macro rewrites each method into a function with an additional first argument; `syntax-parameterize` connects the extra argument to `this`. That is, the conversion takes methods in the following form:

```
(lambda formals method-body ...)
```

and rewrites them into the following:

```
(lambda (implicit-this-arg . formals)
  (syntax-parameterize
      ((this (make-identifier-transformer
              #'implicit-this-arg)))
    method-body ...))
```

Since `this` is defined as a syntax parameter and exported from the class-system library along with `class`, modules can rename `this` on import, and macros can expand into uses of `this`. Meanwhile, attempting to use `this` outside of a `class` form, is a syntax error.

In the original `class` implementation, `this` was introduced unhygienically. Predictably, this unhygienic introduction created trouble for macros like `mixin` that expand into `class`. One partial improvement was to have a variant of `class` where the identifier for `this` is explicitly declared; macros like `mixin` could use that variant to introduce both the identifier and uses. However, `mixin` still had to do the unhygienic work of introducing the identifier (so that mixin methods could use it); furthermore, macros that expand into `mixin` needed a variant of `mixin` with an explicit binding. Aside from those problems, macros used within a `class` body could not generally introduce references to `this`, even unhygienically, since the name could be changed when specified explicitly. None of these problems occur now that `this` is based on syntax parameters.

There are many other uses of syntax parameters in the Racket code base, including:[3]

- The class system [Flatt et al. 2006] uses another syntax parameter internally to control whether a `class` form is expanded in "trace" mode [Eastlund and Felleisen 2009].
- The `match` library provides a syntax parameter called `fail` that can be used in a match clause to escape and try the next clause.
- In the `define-struct` form, `struct-field-index` converts field names to integer indexes for use with structure properties. For example, a structure instance can be made to act as a procedure or as a synchronizable event by specifying the field that implements the application or synchronization behavior.
- The contract system [Findler and Felleisen 2002] uses a syntax parameter to implement contract regions, which allow blame to be assigned at a finer granularity than modules [Strickland and Felleisen 2010].
- The contract system uses another syntax parameter internally to communicate information to nested contract forms.
- Utility macros for `slideshow` [Findler and Flatt 2004] use syntax parameters to manage implicit "pict" (slide element) combination functions and staging modes.
- In the `lexer` form [Owens et al. 2004], `return-without-pos` plays a role similar to `abort` for `loop`, and it is implemented as a syntax parameter.
- The `syntax-parse` [Culpepper and Felleisen 2010] form uses a syntax parameter internally to store its failure continuation.

Beyond Racket, syntax parameters have been adapted to a macro system for C-like syntax [Atkinson and Flatt 2011] to support implicit names such as `this`.

[3] It is worth nothing that some of these are used to communicate values in a way that is more similar to runtime parameters, rather than adjust bindings.

6. Macros are Still Hard

Syntax parameters are a great tool for solving the problem of macros that need to bind a known name. Unlike `datum->syntax`, they make a robust solution that is convenient enough to use when needed, and as a result they have become a common element of the Racket macro toolset. Indeed, when the common "how do I break hygiene when I need to?" question comes up on the Racket mailing lists, we can often reply with "you don't need to!".

However, syntax parameters are not *always* the answer to the question—there are certainly still cases where `datum->syntax` is needed. For example, the `include` macro is one that is fundamentally a tool for "near textual inclusion" of code in some lexical context, and as such it is intended to break hygiene in a `datum->syntax` style. A similar facility that intentionally breaks hygiene is the tangling process of a true literate programming tool [Flatt et al. 2009].

In addition, syntax parameters come with some subtleties that might puzzle macro writers. For example, programmers might expect the following two definitions to be equivalent:

```
(define a (lambda () (abort)))
(define-syntax a (syntax-rules () [(_) (abort)]))
```

However, they are actually different:

```
> (forever
    (define a (lambda () (abort)))
    (forever (display "inner\n") (a))
    (display "outer\n")
    (abort))
inner

> (forever
    (define-syntax a (syntax-rules () [(_) (abort)]))
    (forever (display "inner\n") (a))
    (display "outer\n")
    (abort))
inner
outer
```

This looks surprising at first sight, but on closer inspection, we can see that in the first example `a` is a thunk that holds a reference to the outer loop's `abort`, whereas in the second example it is a macro that expands to a use of `abort`—whatever the binding means in the context it appears in. Another way to see the difference is to consider what happens when the definition of `a` appears at the toplevel: in the first case we get a syntax error since we get the default "useless" binding of `abort`, but in the second we get a macro definition that abstracts over whatever `break` is—which is, in fact, a *desirable* feature (which was mentioned in Section 4).

For a related issue, prehaps less subtle, consider the following macro definition:

```
(define-syntax ten-times
  (syntax-rules ()
    [(_ body ...)
     (let loop ([n 10])
       (when (> n 0) body ... (loop (- n 1))))]))
```

Given that we have a `forever` macro, it is reasonable to refactor the macro to use it:

```
(define-syntax ten-times
  (syntax-rules ()
    [(_ body ...)
     (let ([n 10])
       (forever body ...
                (set! n (- n 1))
                (when (= n 0) (abort))))]))
```

However, this seemingly internal change to the implementation of `ten-times` can affect code that uses it—for example,

```
(forever (ten-times (display "hey\n") (abort)))
```

will loop infinitely with the second version of the macro. The reason for the difference is that once `ten-times` is implemented using `while` it effectively "inherits" `abort`, which becomes a visible part of its interface. The same would happen even if there is a chain of macro layers that eventually uses `forever`.

This is, of course, another aspect of the intended feature of syntax parameters: we usually *want* to have `abort` accessible in all macros that are derived from `forever`, otherwise we could use the `datum->syntax` solution. For such rare cases when we want to use such a macro but avoid exposing this use, Racket provides a `syntax-parameter-value` function which can be used at expansion time to get a hold of the adjusted value of a syntax parameter, to be reinstated later on. In the case of the above `ten-times`, we get the following code:

```
(define-syntax (ten-times stx)
  (syntax-case stx ()
    [(_ body ...)
     (with-syntax ([old (syntax-parameter-value #'abort)])
       #'(let ([n 10])
           (forever (syntax-parameterize ([abort old]) body ...)
                    (set! n (- n 1))
                    (when (= n 0) (abort)))))]))
```

To summarize, syntax parameters might lead to subtle behavior when we use macros to abstract over code, since it is essentially a tool that is intended to cooperate with such macro-based abstractions. Fortunately, these subtleties are not common enough to pose a problem in most practical cases.[4] Furthermore, syntax parameters are still a far better solution than the alternatives: novice macro writers get the benefit from a solution that avoids the much harder issues of hygiene, and experienced writers quickly acquire a good intuition of the resulting behavior in these cases.

7. Conclusion

Racket's syntax system, an extended dialect of the `syntax-case` system, includes many experimental extensions. Among those extensions, syntax parameters stand out as a simple improvement that solves a common problem for hygienic macro systems. It has proven itself as an indispensable tool in many situations, and is no longer considered experimental. As such, it can be a useful addition to the toolbox of Scheme macro programmers of all flavors. The "how do I break hygiene when I need to?" question is not common only in a Racket context—it is one of the oldest issues with hygienic macros, and a considerable factor in seeing `defmacro` linger on in many implementations. Having a *good* answer to most occurrences of this question is long overdue. As we have seen, it is not the answer to all such questions, but like `syntax-rules`, they provide a good answer for most requests for breaking hygiene—one that avoids the need for such breakages.

Acknowledgments

We thank the anonymous reviewers for valuable comments and suggestions, in particular, we thank the reviewer that had motivated adding Section 6. We also thank Sam Tobin-Hochstadt for his helpful feedback on drafts of this paper.

References

Kevin Atkinson and Matthew Flatt. Adapting Scheme-like macros to a C-like language. In *Proc. Workshop on Scheme and Functional Programming*, October 2011.

Will Clinger. Hygienic macros through explicit renaming. *ACM Lisp Pointers*, 4(4), 1991.

Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *Proc. ACM International Conference on Functional Programming*, pages 235–246, 2010.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*, 2009a.

R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, fourth edition, 2009b.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

Carl Eastlund and Matthias Felleisen. Sequence traces for object-oriented executions. In *Proc. Workshop on Scheme and Functional Programming*, pages 7–13, August 2009.

Marc Feeley. SRFI 39: Parameter objects, 2003. `http://srfi.schemers.org/srfi-39/`.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. ACM International Conference on Functional Programming*, pages 48–59, 2002.

Robert Bruce Findler and Matthew Flatt. Slideshow: Functional presentations. In *Proc. ACM International Conference on Functional Programming*, pages 224–235, September 2004.

Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits (invited tutorial). In *Proc. Asian Symposium on Programming Languages and Systems*, pages 270–289, 2006.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *Proc. ACM International Conference on Functional Programming*, 2009.

Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in Scheme. In *Proc. Workshop on Scheme and Functional Programming*, pages 41–52, September 2004.

Michael Sperber (Ed.). The revised^6 report on the algorithmic language Scheme, 2007.

T. Stephen Strickland and Matthias Felleisen. Nested and dynamic contract boundaries. In *Proc. The International Symposia on Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 141–158, 2010.

[4] None of the uses that were mentioned in Section 5 run into such problems.