Hygienic Macros Through Explicit Renaming

William Clinger

This paper describes an alternative to the low-level macro facility described in the *Revised*^{4} *Report on the Algorithmic Language Scheme* [1]. The facility described here is based on explicit renaming of identifiers, and was developed for the first implementation of the hygienic macro expansion algorithm described in [2]. It was the first low-level macro facility to be designed for compatibility with a high-level hygienic macro system, and it remains one of the easiest to understand.

Whereas the low-level macro facility described in the Revised^4 Report renames identifiers automatically, so that hygienic macros are obtained by default, the facility described here requires that identifiers be renamed explicitly in order to maintain hygiene.

Another difference is that, as originally implemented and as described here, there is no way to define certain hygienic macros that define other hygienic macros. The problem is that the transformation procedure for the defined macro may need to compare pieces of its first argument with the denotation of an identifier, but the only way for the defining macro to pass that denotation along to the defined macro is as part of the code for the defined macro. This problem can be solved by introducing `syntax` expressions as in the Revised^4 Report. `Syntax` is like `quote`, except that the denotation of an identifier quoted by `syntax` is preserved as part of the quoted value.

As with the low-level macro facility based on syntactic closures [3], the explicit renaming facility adds a new production for ⟨transformer spec⟩:

⟨transformer⟩ ⟶ `(transformer` ⟨expression⟩`)`

The ⟨expression⟩ is expanded in the syntactic environment of the `transformer` expression, and the expanded expression is evaluated in the standard transformer environment to yield a *transformation procedure*. The transformation procedure takes an expression and two other arguments and returns a transformed expression. For example, the transformation procedure for a `call` macro such that `(call` *proc arg* `...)` expands into `(`*proc arg* `...)` can be written as

```
(lambda (exp rename compare)
```

```
        (cdr exp))
```

Expressions are represented as lists in the traditional manner, except that identifiers may be represented by objects other than symbols. Transformation procedures may use the predicate `identifier?` to determine whether an object is the representation of an identifier.

The second argument to a transformation procedure is a renaming procedure that takes the representation of an identifier as its argument and returns the representation of a fresh identifier that occurs nowhere else in the program. For example, the transformation procedure for a simplified version of the `let` macro might be written as

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((lambda ,vars ,@body)
      ,@inits)))
```

This would not be hygienic, however. A hygienic `let` macro must rename the identifier `lambda` to protect it from being captured by a local binding. The renaming effectively creates an fresh alias for `lambda`, one that cannot be captured by any subsequent binding:

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((,(rename 'lambda) ,vars ,@body)
      ,@inits)))
```

The expression returned by the transformation procedure will be expanded in the syntactic environment obtained from the syntactic environment of the macro application by binding any fresh identifiers generated by the renaming procedure to the denotations of the original identifiers in the syntactic environment in which the macro was defined. This means that a renamed identifier will denote the same thing as the original identifier unless the transformation procedure that renamed the identifier placed an occurrence of it in a binding position.

The renaming procedure acts as a mathematical function in the sense that the identifiers obtained from any two calls with the same argument will be the same in the sense of `eqv?`. It is an error if the renaming procedure is called after the transformation procedure has returned.

The third argument to a transformation procedure is a comparison predicate that takes the representations of two identifiers as its arguments and returns true if and only if they denote the same thing in the syntactic environment that will be used to expand the transformed macro application. For example, the transformation procedure for a simplified version of the `cond` macro can be written as

```
(lambda (exp rename compare)
  (let ((clauses (cdr exp)))
    (if (null? clauses)
        `(,(rename 'quote) unspecified)
        (let* ((first (car clauses))
               (rest (cdr clauses))
               (test (car first)))
          (cond ((and (identifier? test)
                      (compare test (rename 'else)))
                 `(,(rename 'begin) ,@(cdr first)))
                (else `(,(rename 'if)
                        ,test
                         (,(rename 'begin) ,@(cdr first))
                         (cond ,@rest)))))))))
```

In this example the identifier `else` is renamed before being passed to the comparison predicate, so the comparison will be true if and only if the test expression is an identifier that denotes the same thing in the syntactic environment of the expression being transformed as `else` denotes in the syntactic environment in which the `cond` macro was defined. If `else` were not renamed before being passed to the comparison predicate, then it would match a local variable that happened to be named `else`, and the macro would not be hygienic.

Some macros are non-hygienic by design. For example, the following defines a `loop` macro that implicitly binds `exit` to an escape procedure. The binding of `exit` is intended to capture free references to `exit` in the body of the loop, so `exit` is not renamed.

```
(define-syntax loop
  (transformer
   (lambda (x r c)
     (let ((body (cdr x)))
       `(,(r 'call-with-current-continuation)
         (,(r 'lambda) (exit)
          (,(r 'let) ,(r 'f) () ,@body (,(r 'f))))))))))
```

Suppose a `while` macro is implemented using `loop`, with the intent that `exit` may be used to escape from the `while` loop. The `while` macro cannot be written as

```
(define-syntax while
  (syntax-rules ()
    ((while test body ...)
     (loop (if (not test) (exit #f))
           body ...))))
```

because the reference to `exit` that is inserted by the `while` macro is intended to be captured by the binding of `exit` that will be inserted by the `loop` macro. In other words, this `while` macro is not hygienic. Like `loop`, it must be written using the `transformer` syntax:

```
(define-syntax while
  (transformer
   (lambda (x r c)
     (let ((test (cadr x))
           (body (cddr x)))
       `(,(r 'loop)
         (,(r 'if) (,(r 'not) ,test) (exit #f))
         ,@body)))))
```

References

[1] William Clinger and Jonathan Rees, editors.
Revised^4 report on the algorithmic language Scheme.
To appear in the previous issue of *Lisp Pointers*.

[2] William Clinger and Jonathan Rees.
Macros that work.
1991 ACM Conference on Principles of Programming Languages.

[3] Chris Hanson.
A syntactic closures macro facility.
To appear in this issue of *Lisp Pointers*.