How to Write Seemingly Unhygienic and Referentially Opaque Macros with Syntax-rules

Oleg Kiselyov*
Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract

This paper details how folklore notions of hygiene and referential transparency of R5RS macros are defeated by a systematic attack. We demonstrate syntax-rules that seem to capture user identifiers and allow their own identifiers to be captured by the closest lexical bindings. In other words, we have written R5RS macros that accomplish what commonly believed to be impossible. We build on the the fundamental technique by Petrofsky of extracting variables from arguments of a macro. The present paper generalizes Petrofsky's idea to attack referential transparency.

This paper also shows how to overload the `lambda` form. The overloaded `lambda` acts as if it was infected by a virus, which propagates through the `lambda`'s body infecting other `lambda`s in turn. The virus re-defines the macro being camouflaged after each binding. This redefinition is the key insight in achieving the overall referential opaqueness. Although we eventually subvert all binding forms, we preserve the semantics of Scheme as given in R5RS.

The novel result of this paper is a demonstration that although R5RS macros are deliberately restricted in expressiveness, they still wield surprising power. We have exposed faults and the lack of precision in commonly held informal assertions about syntax-rule macros, and pointed out the need for proper formalization. For a practical programmer this paper offers an encouragement: more and more powerful R5RS macros turn out to be possible.

1 Introduction

One of the most attractive and unsurpassed features of Lisp and Scheme is the ability to greatly extend the syntax of the core language and to support domain-specific notations [14]. These syntactic extensions are commonly called macros. A special part of a Lisp/Scheme system, a macro-expander, systematically reduces the extended language to the core one.

A naive macro system that merely finds syntactic extensions and replaces them with their expansions can corrupt variable bindings and break the block structure of the program. For instance, free identifiers in user code may be inadvertently captured by macro-generated bindings, which leads to insidious bugs. This danger is very well documented, for example in [8], [1]. Lisp community has developed techniques [1] that help make macros safer, but they rely on efforts and care of an individual macro programmer. The safety is not built into the system. Furthermore, the techniques complicate the macro code and make it more bug-prone.

Scheme community has recognized the danger of the naive macro expansion to the block structure of Scheme code. The community endeavored to develop a macro system that is safe and respectful of the lexical scope by default. In limited circumstances, exceptions to the block-structure-preserving policy of macros are useful and can be allowed. These exceptions however should be statically visible. A number of experimental macro systems with the above properties have been built ([8], [9], [1], [2], [4], [13]). The least powerful and the most restrictive set of common features of these macro systems has been standardized in R5RS [7]. An earlier version of that system has been mentioned in the previous Scheme report, R4RS, and expounded in [3]. The R5RS macro system permits no exceptions to the safety policy (so-called, hygiene, see below). Furthermore, R5RS macros are specified in a restricted pattern language, which gives the macros another name: syntax-rules. The pattern language is different from the core language and therefore removes the need for the full Scheme evaluator at macro-expand time. Therefore, R5RS macros are severely limited in their ability. The strict safety policy with no exceptions has lead to claims that "Scheme's hygienic macro system is a general mechanism for defining syntactic transformations that reliably obey the rules of lexical scope" [3]. However, there has been little work in formalizing this assertion. Only [8] took upon the task of proving that the systematical renaming of introduced identifiers indeed guarantees the hygiene condition, *in the macro system* of [8]. The latter is not an implementation of R5RS macros.

Surprising discoveries of R5RS macros' latent power question commonly held beliefs about syntax-rule macros. For example, the paper [3] claims "The primary limitation of the hygienic macro system is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly.... The loop-until-exit macro that is used as an example of the low-level macro system in the Revised 4 Report is also a non-hygienic macro." In 2001, however, Al Petrofsky did express the loop-until-exit macro in the R5RS system [11] (see also [12] for more discussion). Al Petrofsky's article introduced

*Current affiliation: Fleet Numerical Meteorology and Oceanography Center, Monterey, CA 93943.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

a general technique, *Petrofsky extraction*, of writing macros that can extract a specific binding from their arguments. Al Petrofsky has also shown how to make such macros nest. The present paper generalizes Petrofsky's ideas to writing of seemingly referentially opaque R5RS macros.

A syntactic extension by its nature introduces a new language, which may differ in some aspects from the core language. Can we write a syntax-rule–based extension that looks like R5RS Scheme but allows seemingly referentially opaque and non-hygienic macros? Can such an extended language still be called R5RS Scheme? At first sight, the answer to both questions is negative. Although R5RS macros are Turing complete [6], they were regarded as "thoroughly hygienic" [3]. Furthermore, the fact that R5RS macros are written in a restricted pattern language rather than in Scheme makes them clearly incapable of certain computations (e.g., concatenating strings or symbols). It is impossible to write an R5RS macro `foo` such that `(foo` *a-symbol b-symbol*`)` expands into a `a-b-symbol`, where the latter is spelt as the concatenation of characters constituting `a-symbol` and `b-symbol`. It is not possible for an R5RS macro to tell if two identifiers have the same spelling. Ostensibly these restrictions were put in place to guarantee and enforce the rules of lexical scope for macros and their expansions (this sentiment was discussed in [1]). In this paper we demonstrate that the power of R5RS macros has been underestimated: We can indeed implement a syntax-rule extension of Scheme that permits seemingly referentially opaque and unhygienic macros [12]. Furthermore, this extended language literally complies with R5RS.

The next section briefly describes the notions of hygiene and referential transparency of macro expansions. Section 3 recalls Petrofsky extraction and its application to writing weakly non-hygienic macros. Section 4 introduces the key idea that re-defining a macro after each binding leads to the overall referential opaqueness. Carrying out such re-definitions requires overloading of all Scheme binding forms, in particular, the `lambda` itself. Section 5 accomplishes this overloading with the help of Petrofsky extraction. We demonstrate an R5RS macro that looks exactly like a careless, referentially opaque Lisp-style macro. The end result is a *library* syntax `let-leaky-syntax` that lets a programmer define a syntax-rule macro and designate a free identifier from that macro for capture by local bindings. The final section discusses what it all means: for macro writers, for macro users, and for programming language researchers.

2 Hygiene and Referential Transparency of Macro Expansions

This section introduces the terminology and the working examples that are used throughout the paper. We will closely follow [8] in our terminology. A syntactic extension, or a macro (invocation), is a phrase in an extended language distinguished by its leading token, or keyword. During the macro-expansion process the extended language is eventually reduced to the core Scheme, in one or several steps. One step in this transformation of a syntactic extension is called a (macro-) expansion step or a transcription step. A syntactic transform function (a.k.a. a macro (transformer)) is a function defined by the macro writer that expands the class of syntactic extensions introduced by the same keyword. A transcription step, which is an application of a transformer to a syntactic extension, yields a phrase in the core language or another syntactic extension. The latter will be expanded in turn. The result of an expansion step may contain identifiers that were not present in the original syntactic extension; we will call them generated identifiers.

A macro system is called hygienic, in the general sense, if it avoids inadvertent captures of free variables through systematic renaming [3]. The free variables in question can be either generated variables, or variables present in macro invocations (i.e., user variables). A narrowly defined hygiene is avoiding the capture of user variables by generated bindings. The precise definition, a hygiene condition for macro expansions (HC/ME), is given in [8]: "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step." If a macro system on the other hand specifically avoids capturing of generated identifiers, the latter always refer to the bindings that existed when the macro transformer was defined rather to the bindings that may exist at the point of macro invocations. This property is often called referential transparency.

The rest of the present section expounds sample R5RS macros chosen to illustrate HC/ME and referential transparency. We will be using the examples in the rest of the paper.

The HC/ME condition demands that bindings introduced by macros should not capture free identifiers in macro arguments. Let us define a sample macro `mbi` such that `(mbi body)` will expand into `(let ((i 10)) body)`. In the pattern language of R5RS macros, the definition reads:

```
(define-syntax mbi
 (syntax-rules ()
    ((mbi body) (let ((i 10)) body))))
```

A naive, non-hygienic expansion of `(mbi (* 1 i))` would have produced `(let ((i 10)) (* 1 i))`. The generated binding of `i` would have captured the free variable `i` occurring in the macro invocation. A hygienic expansion prevents such capture through a systematic renaming of identifiers. Therefore,

```
(let ((i 1)) (mbi (* 1 i)))
```

actually expands to

```
(let ((i~2 1))
  (let ((i~5 10)) (* 1 i~2))
```

and gives the result 1. The identifier `i~2` is different from `i~5`: we will call them identifiers of different *colors*.

The referential transparency facet demands that generated free identifiers should not be captured by local bindings that surround the expansion. To be more precise, if a macro expansion generates a free identifier, the identifier refers to the binding occurrence in the environment of the macro's definition. For example, given the definitions

```
(define foo 1)
(define-syntax mfoo
  (syntax-rules ()
    ((mfoo) foo)))
```

The form `(let ((foo 2)) (mfoo))` expands into

```
(let ((foo~1 2))
    foo)
```

and yields 1 when evaluated. The local `let` binds `foo` of a different color, and therefore, does not capture `foo` generated by the macro `mfoo`.

3 Petrofsky Extraction

In 2001 Al Petrofsky posted an article [11] that demonstrated how to circumvent a weak form of hygiene. The present paper generalizes Petrofsky's idea to attack referential transparency. For completeness and reference this section systematically derives the Petrofsky technique. We aim to write a macro `mbi` so that `(mbi 10 body)` expands into `(let ((i 10)) body)` and the binding of `i` captures free occurrences of `i` in the `body`. We assume that there are no other bindings of `i` in the scope of `(mbi 10 body)`, or `i` was defined early in the global scope and was not re-defined since. This assumption is the distinction between the weak hygiene and the true one.

Developing even weakly non-hygienic macros is challenging. We cannot just write

```
(define-syntax mbi
  (syntax-rules ()
    ((_ val body) (let ((i val)) body))))
```

because `(mbi 10 (* 1 i))` will expand into

```
(let ((i~5 10)) (* 1 i))
```

where `i` in `(* 1 i)` refers to the top-level binding of `i` or remains undefined. However, we can explicitly pass a macro the identifier to capture:

```
(define-syntax mbi-i
  (syntax-rules ()
    ((_ i val body) (let ((i val)) body))))
```

In that case,

```
(mbi-i i 10 (* 1 i))
```

expands into

```
(let ((i 10)) (* 1 i))
```

and the capture occurs. Hence to circumvent the hygiene in the weak sense, we only need to find a way to convert an invocation of `mbi` into an invocation of `mbi-i`. The macro `mbi-i` requires the explicit specification of the identifier to capture – which we can get by extracting the identifier `i`, *together* with its color, from the argument of `mbi`. That is the essence and the elegance of the Petrofsky's idea. Once we have the rightly colored occurrence of `i`, we can use it in the binding form and effect the capture.

The extraction of colored identifiers from a form is done by a macro `extract`, Fig. 1. This macro is the workhorse of the hygiene circumvention strategy. We also need a macro that extracts several identifiers, `extract*` (Fig. 2). Now we can define:

```
(define-syntax mbi-dirty-v1
  (syntax-rules ()
    ((_ _val _body)
     (let-syntax
         ((cont
           (syntax-rules ()
             ((_ (symb) val body)
              (let ((symb val)) body) ))))
       (extract i _body (cont () _val _body))))))
```

so that

```
(mbi-dirty-v1 10 (* 1 i))
```

expands into

```
(let ((i~11 10)) (* 1 i~11))
```

and evaluates to 10, as expected.

The macro `mbi-dirty-v1` seems to do the job, but it has a flaw. It does not nest:

```
(mbi-dirty-v1 10
     (mbi-dirty-v1 20 (* 1 i)))
```

expands into

```
(let ((i~16 10))
     (let ((i~17~25~28 20)) (* 1 i~16)))
```

and evaluates to 10 rather than to 20 as we might have hoped. The outer invocation of `mbi-dirty-v1` creates a binding for `i` – which violates the weak hygiene assumption. Petrofsky [11] has shown how to overcome this problem as well: we need to re-define `mbi-dirty-v1` in the scope of the new binding to `i`. Hence we need a macro that re-defines itself in its own expansion. We however face a problem: If the outer invocation of `mbi-dirty-v1` re-defines itself, this redefinition has to capture the inner invocation of `mbi-dirty-v1`. We already know how to do that, by extracting the colored identifier `mbi-dirty-v1` from the outer macro's body. We need thus to extract two identifiers: `i` and `mbi-dirty-v1`. We arrive at the following code:

```
; A macro that re-defines itself in its expansion:
;  (mbi-dirty-v2 val body)
; expands into
;  (let ((i val)) body)
; and also re-defines itself in the scope of body.
; myself-symb, i-symb are colored ids extracted
; from the 'body'

(define-syntax mbi-dirty-v2
  (syntax-rules ()
    ((_ _val _body)
     (letrec-syntax
         ((doit          ; continuation from extract*
           (syntax-rules ()
             ((_ (myself-symb i-symb) val body)
              (let ((i-symb val)) ; first bind 'i'
                (let-syntax       ; re-define oneself
                    ((myself-symb
                      (syntax-rules ()
                        ((_ val__ body__)
                         (extract*
                           (myself-symb i-symb)
                           body__
                           (doit () val__ body__))))))
                  body))))))
       (extract* (mbi-dirty-v2 i) _body
                 (doit () _val _body))))))
```

Therefore

```
(mbi-dirty-v2 10
   (mbi-dirty-v2 20 (* 1 i)))
```

now expands to

```
;    extract SYMB BODY CONT
; BODY is a form that may contain an occurrence of an identifier that
; refers to the same binding occurrence as SYMB.
; CONT is a form of the shape (K-HEAD K-IDL . K-ARGS)
; where K-IDL and K-ARGS are S-expressions representing lists or the
; empty list.
; The macro extract expands into
;   (K-HEAD (extr-id . K-IDL) . K-ARGS)
; where extr-id is the extracted colored identifier. If the symbol SYMB does
; not occur in BODY at all, extr-id is identical to SYMB.


(define-syntax extract
  (syntax-rules ()
    ((_ symb body _cont)
      (letrec-syntax
        ((tr
           (syntax-rules (symb)
              ; Found our 'symb' -- exit to continuation
              ((_ x symb tail (cont-head symb-l . cont-args))
               (cont-head (x . symb-l) . cont-args))
              ((_ d (x . y) tail cont)   ; if body is a composite form,
               (tr x x (y . tail) cont)) ; look inside
              ((_ d1 d2 () (cont-head  symb-l . cont-args))
               (cont-head (symb . symb-l) . cont-args)) ; symb does not occur
              ((_ d1 d2 (x . y) cont)
               (tr x x y cont)))))
        (tr body body () _cont)))))
```

Figure 1: Macro `extract`: Extract a colored identifier from a form

```
;    extract* SYMB-L BODY CONT
; where SYMB-L is the list of identifiers to extract, and BODY and CONT
; has the same meaning as in extract, see above.
;
; The macro extract* expands into
;   (K-HEAD (extr-id-l . K-IDL) . K-ARGS)
; where extr-id-l is the list of extracted colored identifiers. The extraction
; itself is performed by the macro extract.

(define-syntax extract*
  (syntax-rules ()
    ((_ (symb) body cont)      ; only one id: use extract to do the job
     (extract symb body cont))
    ((_ _symbs _body _cont)
     (letrec-syntax
         ((ex-aux               ; extract id-by-id
           (syntax-rules ()
             ((_ found-symbs () body cont)
              (reverse () found-symbs cont))
             ((_ found-symbs (symb . symb-others) body cont)
              (extract symb body
                       (ex-aux found-symbs symb-others body cont)))
             ))
          (reverse              ; reverse the list of extracted ids
           (syntax-rules ()     ; to match the order of SYMB-L
             ((_ res () (cont-head () . cont-args))
              (cont-head res . cont-args))
             ((_ res (x . tail) cont)
              (reverse (x . res) tail cont)))))
       (ex-aux () _symbs _body _cont)))))
```

Figure 2: Macro `extract*`: Extract several colored identifiers from a form

```
(let ((i~26 10)) (let ((i~52 20)) (* 1 i~52)))
```

and evaluates to 20.

The macro `mbi-dirty-v2` is still only weakly unhygienic. If we evaluate

```
(let ((i 1))
  (mbi-dirty-v2 10 (* 1 i)))
```

we obtain

```
(let ((i 1)) (let ((i~3~22~29 10)) (* 1 i)))
```

which evaluates to 1 rather than 10.

4 Towards the Referential Opaqueness: a `mylet` Form

In this section, we attack referential transparency by writing a macro that seemingly allows free identifiers in its expansion to be captured by the closest lexical binding. To be more precise, we want to write a macro `mfoo` that expands in an identifier `foo` in such a way so that the form `(let ((foo 2)) (let ((foo 3)) (list foo (mfoo))))` would evaluate to the list `(3 3)`. The key insight is a shift of focus from the macro `mfoo` to the binding form `let`. The macro `mfoo` is trivial:

```
(define-syntax mfoo
  (syntax-rules ()
    ((mfoo) foo)))
```

We will concentrate on re-defining the binding form to permit a referentially opaque capture. To make such redefinition easier, we introduce in this section a custom binding form `mylet`. The next section shall show how to make the regular `let` act as `mylet`.

The goal of this section is therefore developing a binding form `mylet` so that `(mylet ((foo 2)) (mylet ((foo 3)) (list foo (mfoo))))` would evaluate to the list `(3 3)`. To make this possible, the expression should expand as follows:

```
(let ((foo 2))
  (define-syntax-mfoo-to-expand-into-foo)
  (re-define-mylet-to-account-for-
      redefined-foo-and-mfoo)
  (let ((foo 3))
    (define-syntax-mfoo-to-expand-into-foo)
    (re-define-mylet-to-account-for-
        redefined-foo-and-mfoo)
    (list foo (mfoo))
 ))
```

Different bindings of a variable are typeset in different fonts. The expansion of the form `mylet` therefore binds `foo` and then re-defines the macro `mfoo` within the scope of the new binding. This `mfoo` will generate the identifier `foo` that refers to that local binding. The redefinition of `mfoo` after a binding is the key insight. It makes it possible for the expansion of the targeted macro to contain identifiers whose bindings are not inserted by the same macro. The process of defining and redefining macros during the expansion of `mylet` looks similar to the process described in the previous Section. Therefore, we take the macro `mbi-dirty-v2` as a prototype for the design of `mylet`. A generator (which helps us define and re-define the macro `mfoo`) and the macro `mylet` are given on Fig. 3. With these definitions, `(mylet ((foo 2)) (mylet ((foo 3)) (list foo (mfoo))))` expands to `((lambda (foo~47) ((lambda (foo~92) (list foo~92 foo~92)) 3)) 2)` and evaluates to `(3 3)`. The result demonstrates that `(mfoo)` indeed expanded to `foo` that was captured by the local binding. The macro `mfoo` seems to have inserted an opaque reference to the binding of `foo`. Because `mylet` constantly re-generates itself, it nests. The following test demonstrates the nesting and the capturing by the expansion of `(mfoo)` of the closest *lexical* binding:

```
(mylet ((foo 3))
    (mylet ((thunk (lambda () (mfoo))))
     (mylet ((foo 4)) (list foo (mfoo) (thunk)))))
```

This expression evaluates to `(4 4 3)`. The expansion of `(mfoo)` within the closure `thunk` refers to the variable `foo` that was lexically visible at that time.

5 Achieving the Referential Opaqueness: Redefining All Binding Forms

The previous section showed that we can indeed write a seemingly referentially opaque R5RS macro, if we resort to custom binding forms. R5RS does not prohibit us however from re-defining the standard binding forms `let`, `let*`, `letrec` and `lambda` to suit our nefarious needs. We need to 'overload' just one form: the fundamental binding form `lambda` itself.

This overloading is done by a macro `defile`, which defiles its body (Appendix B). It is worth noting a few fragments from the macro's long code. The first one

```
(letrec-syntax
  ...
   (lambda-native   ; capture the native lambda
      (syntax-rules ()
        ((_ . args) (lambda . args))))
```

does what it looks like: it captures the native `lambda`, which is needed to effect bindings. Another fragment is:

```
(letrec-syntax
...
  (let-symb     ; R5RS definition of let
    (syntax-rules ()
      ((_ . args)
       (glet (let-symb let*-symb letrec-symb
              lambda-symb) . args))))
```

A top-level macro `glet` (Appendix A) is a `let` with an extra first argument. This argument is the "environment", the list of custom-bound `let` and `lambda` identifiers for use in the macro expansion. The definition of `glet` is taken from R5RS verbatim, with the pattern modified to account for the extra first argument.

```
(define-syntax glet
  (syntax-rules ()
    ((_ (let let* letrec lambda) ; the extra arg
        ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...) val ...))
    ((_ (let let* letrec lambda)
        tag ((name val) ...) body1 body2 ...)
     ((letrec
       ((tag (lambda (name ...) body1 body2 ...)))
      tag) val ...))))
```

```
; Macro: make-mfoo NAME SYMB BODY
; In the scope of BODY, define a macro NAME that expands into an identifier SYMB

(define-syntax make-mfoo
  (syntax-rules ()
    ((_ name symb body)
      (let-syntax
         ((name
           (syntax-rules ()
             ((_) symb))))
        body))))


;   (mylet ((var init)) body)
; expands into
;   (let ((var init)) body')
; where body' is the body wrapped in the re-definitions of mylet and the macro mfoo.

(define-syntax mylet
  (syntax-rules ()
    ((_ ((_var _init)) _body)
     (letrec-syntax
       ((doit                   ; The continuation from extract*
         (syntax-rules ()       ; mylet-symb, etc. are extracted from body
           ((_ (mylet-symb mfoo-symb foo-symb) ((var init)) body)
            (let ((var init))   ; bind the 'var' first
              (make-mfoo mfoo-symb foo-symb  ; now re-generate the macro mfoo
                  (letrec-syntax
                      ((mylet-symb       ; and re-define myself
                        (syntax-rules ()
                          ((_ ((var_ init_)) body_)
                         (extract* (mylet-symb mfoo-symb foo-symb) (var_ body_)
                                   (doit () ((var_ init_)) body_))))))
                 body)))
            ))))
       (extract* (mylet mfoo foo) (_var _body)
                 (doit () ((_var _init)) _body))))))
```

Figure 3: Macros `make-mfoo` and `mylet`

The macro glet therefore relates the let form and the lambda precisely as R5RS does; glet however substitutes our custom-bound lambda. Finally, the overloaded lambda is defined as follows:

```
(letrec
...
 (lambda-symb      ; re-defined, infected lambda
   (syntax-rules ()
     ((_ _vars _body)
      (letrec-syntax
        ((doit (syntax-rules ()
          ((_ (mylet-symb mylet*-symb
               myletrec-symb mylambda-symb
               mymfoo-symb myfoo-symb)
               vars body)
            (lambda-native vars
              (make-mfoo mymfoo-symb myfoo-symb
                (do-defile  ; proliferate
                 (mylet-symb mylet*-symb
                  myletrec-symb mylambda-symb
                  mymfoo-symb myfoo-symb)
                 body)))))))
       (extract* (let-symb let*-symb letrec-symb
                  lambda-symb mfoo-symb foo-symb)
                 (_vars _body)
                 (doit () _vars _body))))))
```

We are relying on the previously captured lambda-native to create bindings. After that we immediately redefine all our macros in the updated environment. The corrupted lambda acts as if it were infected by a virus: every mentioning of lambda "transcribes" the virus and causes it to spread to other binders within the body.

The following are a few excerpts from the defile macro regression tests. An expression

```
(defile
   (let ((foo 2)) (list (mfoo) foo)))
```

expands into

```
((lambda (foo~186) (list foo~186 foo~186)) 2)
```

and predictably evaluates to (2 2). The expansion of (mfoo) has indeed captured a locally-bound identifier. All the infected lambdas are gone: the expansion result is the regular Scheme code. Furthermore,

```
(defile
   (let ((foo 2))
     (let ((foo 3) (bar (list (mfoo) foo)))
       (list foo (mfoo) bar))))
```

evaluates to (3 3 (2 2)) and

```
(defile
   (let ((foo 2))
     (list
      ((letrec
         ((bar (lambda () (list foo (mfoo))))
          (foo 3))
        bar))
      foo (mfoo))))
```

to ((3 3) 2 2). The defiled let and letrec indeed act precisely as the standard ones. Finally,

```
(defile
  (let* ((foo 2)
         (i 3)
         (foo 4)
         ; will capture binding of foo to 4
         (ft (lambda () (mfoo)))
         (foo 5)
         ; will  capture the arg of ft1
         (ft1 (lambda (foo) (mfoo)))
         (foo 6))
    (list foo (mfoo) (ft) (ft1 7) '(mfoo))))
```

evaluates to the expected (6 6 4 7 (mfoo)). In all these examples, the expansion of (mfoo) captures the closest (local) lexical binding of the variable foo. All the examples run with the Bigloo 2.4b interpreter and compiler and with Scheme48.

We must point out that the defiled examples behave as if (mfoo), unless quoted, were just the identifier foo. In other words, as if mfoo were defined as a *non-hygienic*, referentially opaque macro

```
(define-macro (mfoo) foo)
```

To be able to capture a generated identifier by a local binding, we need to know the name of that identifier and the name of a macro that generates it. We also need to effectively wrap the defile macro around victim's code. We can do that explicitly as in the examples above. We can also accomplish the wrapping implicitly, e.g., by re-defining the top-level let or other suitable form so as to insert the invocation of defile at the right spot. It goes without saying that we assume no bindings to the identifiers foo, mfoo, let, letrec, let*, and lambda between the point the macro defile is defined and the point it is invoked.

It is possible to remove the dependence of the macro defile on ad hoc identifiers such as foo and mfoo. We can pass the targeted macro and the identifier to be captured by the closest lexical binding as arguments to defile. We arrive at a form let-leaky-syntax (Appendix C), which is illustrated by the following two examples. An expression

```
(let-leaky-syntax
 bar
 ((mbar
   (syntax-rules () ((_ val) (+ bar val)))))
 (let ((bar 1)) (let ((bar 2)) (mbar 2))))
```

evaluates to 4, whereas

```
(let-leaky-syntax
  quux
  ((mquux (syntax-rules ()
     ((_ val) (+ quux quux val)))))
  (let* ((bar 1) (quux 0) (quux 2)
         (lquux (lambda (x) (mquux x)))
         (quux 3)
         (lcquux (lambda (quux) (mquux quux))))
     (list (+ quux quux) (mquux 0) (lquux 2)
           (lcquux 5))))
```

evaluates to the list (6 6 6 15). The form let-leaky-syntax is similar to let-syntax. The former takes an additional first argument, an identifier from the body of the defined syntax-rules. This designated identifier will be captured by the closest lexical binding within the body of let-leaky-syntax. The examples show that the variable is captured indeed. In particular, the macro mquux in the last example expands to an expression that adds the value of an identifier quux twice to the value of the mquux's argument. Because the identifier quux was designated for capture by the closest local binding, a procedure (lambda (quux) (mquux quux)) effectively triples its argument.

We have thus demonstrated the syntax form let-leaky-syntax that defines a macro with a specific variable excepted from the hygienic rules. The form let-leaky-syntax is a *library* syntax, developed exclusively with R5RS (hygienic) macros.

6 Discussion

In this section we will discuss the implications of the defile macro. First however we have to assure the reader that defile is legal: it fully complies with R5RS and does not rely on unspecified behavior. Indeed, the macro defile is written entirely in the pattern language of R5RS. Re-binding of syntax keywords lambda, let, let*, and letrec is not prohibited by R5RS. On the contrary, R5RS specifically states that there are no reserved keywords, and syntactic bindings may shadow variable bindings and other syntactic bindings. Furthermore, re-defined let, let*, and letrec forms relate to the lambda form precisely as the R5RS forms do. The re-defined lambda form is also in compliance with its R5RS description ([7], Section 4.1.4).

One can argue that our re-defined lambda leads to a violation of the constraint that R5RS places on the macro system: "If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro." This paragraph however applies exactly as it is to the defiled macros. In the code,

```
(define foo 1)
(defile
   (let ((foo 2)) (list (mfoo) foo)))
```

the identifier foo inserted by the expansion of the macro mfoo indeed refers to the binding of foo that was visible when the macro mfoo was defined. The twist is that the definition of the macro mfoo happened right after the local binding of foo. Despite mfoo being an R5RS, referentially transparent macro, the *overall result* is equivalent to the expansion of a referentially opaque macro.

The macro defile indeed has to surround the victim's code. One can therefore object if we merely create our own 'little language' that resembles Scheme but does not guarantee referential transparency of macro expansions. However, such a little language was presumed impossible with syntax rules [2][3]! Any macro by definition extends the language. The extended language is still expected to obey certain constraints. The impetus for hygienic macros was specifically to create a macro system with guaranteed hygienic constraints. Although syntax-rules are Turing complete, certain computations, for example, determining if two identifiers are spelled the same, are outside of their scope. It was a common belief therefore that syntax-rules are thoroughly hygienic [3].

To be more precise, the argument that syntax-rules cannot in principle implement macros such as let-leaky-syntax was informally advanced in [2]. That paper described a macro-expansion algorithm that is used in several R5RS Scheme systems, including Bigloo. Incidentally, the algorithm accounts for the possibility that the binding forms lambda and let-syntax may be redefined by the user. The paper [2] *informally* argues that the algorithm satisfies two hygiene conditions: (1) "It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro," and (2) "It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro." Unfortunately, the paper does not state the conditions with sufficient precision, which precludes a formal proof. The notion of 'inserting a binding' is particularly vague. The common folklore interpretation of the conditions is that generated bindings can capture only the identifiers that are generated at the same transcription step. Had this interpretation been true, let-leaky-syntax would have been impossible. However, the interpretation is false and Petrofsky's loop macro is a counter-example [12]. Several examples in Section 4 demonstrated the capture of generated identifiers *across* transcription steps.

It is interesting to ask if it is possible to create a macro system that is provable hygienic, which provably does not permit tricks such as the one in this paper. The paper [8] showed that if we do not allow macros to expand into the definitions of other macros, we can design a macro system that is provably hygienic. A MacroML paper [5] claimed that being generative *seems* to be a necessary condition for a macro extension to maintain strong invariants (static typing, in the context of MacroML). A generative macro can build forms from its arguments but cannot deconstruct or inspect its arguments.

We conclude that the subject of macro hygiene is not at all decided, and more research is needed to precisely state what hygiene formally means and which precisely assurances it provides.

For a practical programmer, we offer the let-leaky-syntax library form. The form lets the programmer write a new class of powerful syntactic extensions with the standard R5RS syntax-rules, without resorting to lower-level macro facilities. In general, the practical macro programmer will hopefully view the conclusions of this paper as an encouragement. We should realize the informal and narrow nature of many assertions about R5RS macros. We should not read into R5RS more than it actually says. Thus we can write more and more expressive macros than we were previously led to believe [12].

Acknowledgment

I am greatly indebted to Al Petrofsky for numerous discussions, which helped improve both the content and the presentation of the paper. Special thanks are due to Alan Bawden for extensive comments and the invaluable advice. I would like to thank Olin Shivers and the anonymous reviewers for many helpful comments and suggestions. This work has been supported in part by the National Research Council Research Associateship Program, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

References

[1] Alan Bawden and Jonathan Rees. Syntactic closures. In Proc. 1988 ACM Symposium on Lisp and Functional Programming, pp. 86-95.

[2] William Clinger and Jonathan Rees. Macros that work. In Proc. 1991 ACM Conference on Principles of Programming Languages, pp. 155-162.

[3] William Clinger. Macros in Scheme. Lisp Pointers, IV(4):25-28, December 1991.

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

[5] S. Ganz, A. Sabry, W. Taha: Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. Proc. Intl. Conf. Functional Programming (ICFP'01), pp. 74-85. Florence, Italy, September 3-5 (2001).

[6] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. Scheme and Functional Programming 2000. September 2000.

[7] R. Kelsey, W. Clinger, J. Rees (eds.), Revised5 Report on the Algorithmic Language Scheme, J. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.

[8] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Proc. 1986 ACM Conference on Lisp and Functional Programming, pp. 151-161.

[9] Eugene E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In Proc. 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 77 - 84, 1987.

[10] Al Petrofsky, Oleg Kiselyov. Re: Widespread bug (arguably) in letrec when an initializer returns twice. Messages posted on a newsgroup comp.lang.scheme on May 21, 2001 10:30:34 and 14:56:49 PST. http://groups.google.com/groups?-selm=7eb8ac3e.0105210930.21542605%40posting.google.com http://groups.google.com/groups?-selm=87ae468j7x.fsf%40app.dial.idiom.com

[11] Al Petrofsky. How to write seemingly unhygienic macros using syntax-rules. A message posted on a newsgroup comp.lang.scheme on November 19, 2001 01:23:33 PST. http://groups.google.com/groups?-selm=87oflzcdwt.fsf%40radish.petrofsky.org

[12] Al Petrofsky. Re: Holey macros! (was Re: choice for embed-
ding Scheme implementation?). A message posted on a newsgroup comp.lang.scheme on May 22 2002 10:21:31 -0700. http://groups.google.com/groups?-selm=874rh0p084.fsf%40radish.petrofsky.org

[13] Jonathan A. Rees. Implementing lexically scoped macros. Lisp Pointers. 'The Scheme of Things' (column), 1993.

[14] Olin Shivers. A universal scripting framework, or Lambda: the ultimate 'little language'. In "Concurrency and Parallelism, Programming, Networking, and Security," Lecture Notes in Computer Science 1179, pp 254-265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

Appendix A

The following are definitions of `let`, `let*` and `letrec` taken almost verbatim from R5RS. The only difference is in custom-bound `let`, `let*`, `letrec` and `lambda` identifiers, which we explicitly pass to the glet macros in the first argument.

```
(define-syntax glet
  (syntax-rules ()
    ((_ (let let* letrec lambda)
        ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...) val ...))
    ((_ (let let* letrec lambda)
        tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...) body1 body2 ...)))
             tag) val ...))))

(define-syntax glet*
  (syntax-rules ()
    ((_ mynames () body1 body2 ...)
     (let () body1 body2 ...))
    ((_ (let let* letrec lambda)
        ((name1 val1) (name2 val2) ...) body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...) body1 body2 ...)))))

; A shorter implementations of letrec [10]
(define-syntax gletrec
  (syntax-rules ()
    ((_ (mlet let* letrec lambda)
        ((var init) ...) . body)
     (mlet ((var 'undefined) ...)
       ; the native let will do fine here
       (let ((temp (list init ...)))
         (begin (set! var (car temp))
                (set! temp (cdr temp))) ...
               (let () . body))))))
```

Appendix B[1]

```
; This macro defiles its body.
; It overloads all the let-forms and the lambda, and defines a non-hygienic macro 'mfoo'. Whenever any
; binding is introduced, the let-forms, the lambdas and 'mfoo' are re-defined. The overloaded lambda acts
; as if it were infected by a virus, which keeps spreading within lambda's body to infect other lambda's there.

(define-syntax defile
  (syntax-rules ()
    ((_ dbody)
     (letrec-syntax
         ((do-defile
           (syntax-rules ()     ; all the overloaded identifiers
             ((_ (let-symb let*-symb letrec-symb lambda-symb mfoo-symb foo-symb) body-to-defile)
              (letrec-syntax
                  ((let-symb            ; R5RS definition of let
                    (syntax-rules ()
                      ((_ . args)
                       (glet (let-symb let*-symb letrec-symb lambda-symb)
                             . args))))

                   (let*-symb           ; Redefinition of let*
                    (syntax-rules ()
                      ((_ . args)
                       (glet* (let-symb let*-symb letrec-symb lambda-symb)
                              . args))))

                   (letrec-symb         ; Redefinition of letrec
                    (syntax-rules ()
                      ((_ . args)
                       (gletrec (let-symb let*-symb letrec-symb lambda-symb)
                              . args))))

                   (lambda-symb         ; re-defined, infected lambda
                    (syntax-rules ()
                      ((_ _vars _body)
                       (letrec-syntax
                           ((doit
                             (syntax-rules ()
                               ((_ (mylet-symb mylet*-symb myletrec-symb
                                               mylambda-symb mymfoo-symb
                                               myfoo-symb) vars body)
                                (lambda-native vars
                                  (make-mfoo mymfoo-symb myfoo-symb
                                    (do-defile  ; proliferate in the body
                                     (mylet-symb mylet*-symb myletrec-symb
                                                 mylambda-symb
                                                 mymfoo-symb myfoo-symb)
                                     body)))))))
                         (extract* (let-symb let*-symb letrec-symb lambda-symb
                                             mfoo-symb foo-symb)
                                   (_vars _body)
                                   (doit () _vars _body))))))

                   (lambda-native       ; capture the native lambda
                    (syntax-rules () ((_ . args) (lambda . args))))
                  )

                body-to-defile)))))

       (extract* (let let* letrec lambda mfoo foo) dbody
                 (do-defile () dbody))
       ))))
```

[1] The current implementation of the macro `defile` does not corrupt bindings that are created by internal `define`, `let-syntax` and `letrec-syntax` forms. There are no technical obstacles to corrupting those bindings as well. To avoid clutter, the present code does not detect a possible shadowing of the macro `mfoo` by a local binding. The full code with validation tests is available at `http://pobox.com/~oleg/ftp/Scheme/dirty-macros.scm`.

Appendix C

Given below is the implementation of a library syntax `let-leaky-syntax.` It is based on a slightly modified version of the macro `defile`. The latter uses parameters `leaky-macro-name`, `leaky-macro-name-gen`, and `captured-symbol` instead of hard-coded identifiers `mfoo`, `make-mfoo` and `foo`.

```
(define-syntax defile-what
  (syntax-rules ()
    ((_ leaky-macro-name leaky-macro-name-gen captured-symbol dbody)
     (letrec-syntax
       ((do-defile
         ... similar to the defile macro, Appendix B ...
  (extract*
      (let let* letrec lambda
       leaky-macro-name captured-symbol) dbody (do-defile () dbody)) ))))


(define-syntax let-leaky-syntax
  (syntax-rules ()
    ((_ var-to-capture ((dm-name dm-body)) body)
     (let-syntax
       ((dm-generator
          (syntax-rules ()
            ((_ dmg-name var-to-capture dmg-outer-body)
             (let-syntax
               ((dmg-name dm-body))
                dmg-outer-body)))))
      (defile-what
         dm-name dm-generator var-to-capture body)
))))
```