Macros that Compose: Systematic Macro Programming

Oleg Kiselyov

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

Abstract Macros are often regarded as a sort of black magic: highly useful yet abstruse. The present paper aims to make macro programming more like a craft. Using R5RS Scheme macros as an example, we develop and present a general practical methodology of building complex macros systematically, by combining simpler components by functional composition or higher-order operators.
Macro programming is complex because the systematic approach does not apply to many macros. We show that macros and other head-first normal-order re-writing systems generally lack functional composition. However, macros written in a continuation-passing style (CPS) always compose. Making CPS macros practical still requires an encoding for macro-continuations. We have found this missing piece, with an insight for anonymous macro-level abstractions.
In the specific case of R5RS macros, this paper presents a stronger result: developing R5RS macros by a translation from the corresponding Scheme procedures. We demonstrate the practical use of the technique by elaborating a real-world example.

1 Introduction

As a practical matter, macros are useful and often used, especially in large projects. Their applications range from conditional compilation to controlled inlining, to improving convenience and expressiveness of a small language core by adding syntax sugar. In the most compelling application, macros implement a domain-specific, "little" language by syntactically extending a general-purpose language [15]. Many programming systems officially or covertly include some kind of macro- or pre-processing facility. The unusual sophistication of macro systems of Scheme and Lisp is argued to be the most distinguished virtue of these languages, which is crucial for some projects [5][15]. MacroML extension for ML [4] and Camlp4 pre-processor for OCaml [13] are examples of macro facilities for modern functional languages. Macros turn out to be useful even in a non-strict language such as Haskell [16], to define and alter bindings, statements, declarations and other non-expression constructs.

However, macros have historically been poorly understood and unreliable [1]. Macros and pre-processors – i.e., abstract and concrete syntax transformers – may generate ill-typed, invalid or even unparsable code. Macros may insert bindings that accidentally capture user variables. Macros are difficult to develop and debug. Some of these issues have been addressed: The MacroML system [4], designed from the viewpoint of macros as multi-staged computations, guarantees that the expanded code is well-formed and well-typed. Capture-free (i.e., hygienic) introduction of new bindings in generated code has been extensively investigated in the Scheme community [11]. The Revised^5 Report on Scheme (R5RS) [7] defines a macro system with the assured hygiene.

This article however deals with a software-engineering aspect of writing macros. Embedding a domain-specific notation or otherwise syntactically extending a core language often require sophisticated macros. A complex artifact is easier to develop if it can be composed of smaller, separately designed and tested parts. Macro systems such as R5RS Scheme macros and Camlp4 quotations are head-first, call-by-name or normal order re-writing systems. Such systems are in general non-compositional. The familiar idioms of a function call and a functional composition – let alone higher-level combinators such as fold – do not easily apply. Therefore, macros tend to be monolithic, highly recursive, ad hoc, and requiring a revelation to write, and the equal revelation to understand. As Section 7.3 of R5RS demonstrates, we have to resort to contortions and hacks even in simple cases.

This paper pinpoints the stumbling block to macro composition and the way to overcome it. The resulting technique lets us program macros using traditional, well-understood applicative programming idioms of composition and combinators. The solution is a continuation-passing style (CPS) coupled with a *macro-level abstraction*. The latter is a first-class denotation for a future expansion, and is a novel insight of the paper.

We will limit the presentation to R5RS macros for the lack of space. However, the abstraction-passing technique is general and will apply to Camlp4, Tcl and other head-first code-generation systems. The application of the technique to R5RS macros leads to the other result of the paper: a systematic construction of R5RS macros by a translation from Scheme procedures. This is an unexpected result as typically R5RS macros look nothing like regular Scheme procedures.

In Section 2 we briefly outline R5RS macros as a representative head-first re-writing system. Section 3 demonstrates failures of macro compositions and identifies the reason. In the next section we introduce the main technical contribution, macro-level abstractions, and discuss their use to encode continuations in CPS macros. We establish the reason why CPS macros always compose. Section 5 introduces the systematic construction of R5RS macros. The section illustrates the translation technique by working out a real-world example. The last two sections discuss the related work and conclude.

2 R5RS Macros as a Head-First Re-writing System

Macro-systems are in general source-to-source transformations. Macro-phrases, or macro-applications, are introduced by a keyword. The keyword – an associated syntax transformer, to be precise – specifies how the rest of the phrase is to be parsed or interpreted. When the macro processor encounters a keyword, it invokes the corresponding transformer, passes to it the macro-phrase as it is, and re-processes the result. Because the rest of a macro phrase is interpreted solely by a syntactic transformer, the phrase may contain constructs that appear invalid or ill-formed in the core language. This fact makes macros powerful tools for the extension of language syntax [12]. Throughout the paper, we will use the terms macros and head-first re-writing systems synonymously.

We will use R5RS macros as a representative head-first pattern-based re-writing system. R5RS-macro keywords are bound to transformers by `define-macro`, `let-syntax` or `letrec-syntax` forms, for example:

```
(define-syntax id
   (syntax-rules ()
     ((id x) x)))
```

A syntax transformer, specified with a `syntax-rules` form, is a sequence of patterns and the corresponding templates. Patterns may contain pattern variables and are linear. Given a form, the macro expander checks it for macro phrases, and if found, reduces them in the normal order, i.e., leftmost first. A reduction step is matching against a set of patterns and replacing the matched expression by a template. For example, given an expression `(id (id 42))`, the expander notices that it is a macro phrase, retrieves the transformer associated with the keyword `id`, and matches the expression against the only pattern in that transformer, `(id x)`. The matching process binds the pattern variable `x` to the form `(id 42)`, which, according to the template, becomes the result of the expansion. The result is itself a macro-application and is further re-written to `42`, which is the normal form of the original expression.

As another example, let us consider a macro that expands into a character-discriminator function:

```
(define-syntax mcase1
   (syntax-rules ()
     ((mcase1 chr)
      (lambda (v)
        (case v
          ((chr) #t)
          (else #f))))))
```

A macro-application `(mcase1 #\A)` expands into `(lambda (fresh-v) (case fresh-v ((#\A) #t) (else #f)))`, a predicate that checks if its argument is equal to the character `#\A`. R5RS macros are hygienic: when a macro-expansion introduces a new binding, the bound variable, e.g., `v`, is in effect renamed to prevent accidental captures of free identifiers [7].

3 Mis-composition of Macros

In the functional world, if f and g are two appropriately typed functions we can compose them to build a more complex function. If one of the functions is the identity, the composition is equivalent to the other function: $f \cdot id \equiv f$. This property generally does not hold if f and id are macros.

Indeed, if we compose the previously introduced macros `mcase1` and `id` and evaluate `((mcase1 (id #\A)) #\A)` we get an unexpected result as if the character `#\A` is not equal to itself. The expansion of the composed form `(mcase1 (id #\A))`:

```
(lambda (vv)
  (case vv
    (((id #\A)) #t)
     (else #f)))
```

manifests the problem. According to R5RS [7], `case` is a special, library syntax form. It is defined as "(case <key> <clause1> <clause2> ...) where <key> may be any expression and each <clause> should have the form ((<datum1> ...) <expression1> <expression2> ...), where each <datum> is an external representation of some object. The last <clause> may be an `else` clause." It must be stressed that <datum1> is not an expression and therefore is not subject to macro-expansion. The expansion of the form `(mcase1 (id #\A))` then is a procedure that matches its argument against a literal two-element-list constant `(id #\A)` rather than against the character `#\A`. The composition fails because the argument of `mcase1`, `chr`, appears in a *special class position* in `mcase1`'s expansion code template. We say that a phrase occurs in a special class position if it is not subject to macro expansion. A quoted phrase, <datum> in a `case` form, bindings in `let` forms are examples of special class position phrases [12]. A macro-argument to be deconstructed inside the macro also occupies a special class position. In Scheme, phrases in special class positions are those that are not parsed as expressions.

We argue that macros that place their arguments in special class positions constitute the most compelling class of macros. The paper [16] documents several reasons for using macros in functional languages. For example, in a strict language, we often employ macros to prevent, delay or repeat the evaluation of an expression. We can achieve the same result without macros however, by turning the expression in question into a thunk. Infix and especially distfix operators [16] can add significant amount of syntactic sugar without resorting to macros. However, transformations that: introduce bindings; insert type, module or other declarations; add alternatives to enumerated type declarations cannot be accomplished without macro facilities. Such transformations give the most compelling reason for the inclusion of macros in a language standard [16]. These transformations are also non-compositional, in general.

Re-writing systems that have special-class-position phrases and that cannot force the re-writing of argument expressions are generally non-compositional. For example, a macro may be invoked with an argument that is a macro-value such as an identifier or a string. Alternatively, a macro can be invoked with the argument that is a *macro-expression*, i.e., an application of another macro. The latter will yield a macro-value when expanded. If such an argument appears in an expression position in the result of the original macro, the macro processor will expand the macro-expression when re-scanning the result. If an argument that is a macro-expression will eventually appear in a special class position, this macro-expression will *not* be further expanded. A macro that pattern-matches its argument against a known symbol or form will not work if it is applied to a macro-expression. This will cause syntax errors in the best case, or subtle semantic errors in the worst. The difference between head-first normal-order abstractions and applicative-order abstractions (functions) is that a function can force the evaluation of an expression by using it or (in strict systems) by passing it as an argument to a function. In a head-first normal-order re-writing system, a rule can cause the expansion of an expression only by returning it. In that case however, the rule loses the control and cannot obtain the expression's result.

The non-applicative nature of macros and other head-first systems has been tacitly acknowledged by language designers. To make macros functional, the designers often provide systematic or ad hoc ways for a macro-transformer to force the evaluation of its arguments. For example, Tcl has special evaluating parentheses `[]`, Lisp has a `macroexpand` form, and shell has a backquote substitution. These approaches however make it difficult for the macro expander to offer hygiene or other guarantees. Therefore, R5RS syntax transformers are rather restricted; in particularly, they cannot themselves invoke the macro expander – or other Scheme functions.

4 Macro-CPS Programming

We shall now show how to construct macros that are always compositional – even if they place their arguments in special positions. In some circumstances, we can compose with "legacy" macros written without following our methodology. As a side effect, the technique makes R5RS macros functional, restores the modularity principle, and even makes possible higher-order combinators. The present section introduces the technique and gives several small illustrations. The next section elaborates an extended example of a systematic, modular development of a real-life R5RS macro.

The first, novel ingredient to our macro programming technique is a notation for a *first-class parameterized future macro-expansion action:*

```
(??!lambda (bound-var ...) body)
```

Here `bound-var` is a variable, `body` is an expression that may contain forms `(??! bound-var)`, and `??!lambda` is just a symbol. It must be stressed that `??!lambda` is not a syntactic or bound variable. Although the `??!lambda` form may look like a macro-application, it is not. This is a critical distinction: Hilsdale and Friedman [6] looked for a macro-level lambda as a macro, but did not succeed. Our macro-level abstractions are not macros themselves. The `??!lambda` form is first class: it can be passed to and returned from R5RS macros, and can be nested.

The `??!lambda`-form is interpreted by a macro `??!apply`. To be more precise, we specify that the following macro-application

```
(??!apply (??!lambda (bound-var ...) body) arg ...)
```

expands to `body`, with all non-shadowed instances of `(??! bound-var)` hygienically replaced by `arg`. In Scheme, question and exclamation marks are considered ordinary characters. Hence `??!lambda`, `??!apply` and `??!` are ordinary symbols – albeit oddly looking, on purpose. A `??!lambda` form can bind one or several variables; the corresponding `??!apply` form should have just as many arguments. An implementation of `??!apply` that satisfies our specification is given on Fig. 1.

CPS macros first introduced in [6] are the second constituent of the technique. Our CPS macros must receive continuation argument(s), and must expand into an application of `??!apply` or another CPS macro. In particular, if a macro `foo` wants to 'invoke' a macro `bar` on an argument `args`, `foo` should expand into `(bar args (??!lambda (res) continuation))`. Here the form `continuation` includes `(??! res)` forms and thus encodes whatever processing needs to be done with the `bar`'s result.

This technique easily solves the problem of composing macros `id` and `mcase1`. We will first re-write the macro `id` in CPS:

```
(define-syntax id-cps
  (syntax-rules ()
    ((_ x k)
     (??!apply k x))))
```

This macro immediately passes its argument to its continuation. The composition of `id-cps` with `mcase1` will use a macro-level abstraction to encode the continuation:

```
(id-cps #\A (??!lambda (x) (mcase1 (??! x))))
```

We can even compose `mcase1` and `id-cps` twice:

```
(let ((discriminator
        (id-cps #\A
             (??!lambda (x)
                (id-cps (??! x)
                     (??!lambda (x) (mcase1 (??! x))))))))
(display (discriminator #\A)))
```

When evaluated, this expression yields the expected result `#t`.

We observe that given the definition of `id-cps` and the specification of `??!apply`, the expression

```
(id-cps datum (??!lambda (x) (exp (??! x))))
```

```
(define-syntax ??!apply (syntax-rules (??!lambda)
    ((_ (??!lambda (bound-var . other-bound-vars) body)
        oval . other-ovals)
     (letrec-syntax
         ((subs
           (syntax-rules (??! bound-var ??!lambda)
             ((_ val k (??! bound-var))  (appl k val))
             ; check if bound-var is shadowed in int-body
             ((_ val k (??!lambda bvars int-body))
              (subs-in-lambda val bvars (k bvars) int-body))
             ((_ val k (x))     ; optimize single-elem list substitution
              (subs val (recon-pair val k ()) x))
             ((_ val k (x . y))
              (subs val (subsed-cdr val k x) y))
             ((_ val k x)       ; x is an id other than bound-var, number&c
              (appl k x))))
          (subsed-cdr           ; we've done the subs in the cdr of a pair
           (syntax-rules ()     ; now do the subs in the car
             ((_ val k x new-y)
              (subs val (recon-pair val k new-y) x))))
          (recon-pair           ; reconstruct the pair out of subs comp
           (syntax-rules ()
             ((_ val k new-y new-x) (appl k (new-x . new-y)))))
          (subs-in-lambda ; substitute inside the lambda form
           (syntax-rules (bound-var)
             ((_ val () kp  int-body)
              (subs val (recon-l kp ()) int-body))
              ; bound-var is shadowed in the int-body: no subs
             ((_ val (bound-var . obvars) (k bvars) int-body)
              (appl k (??!lambda bvars int-body)))
             ((_ val (obvar . obvars) kp int-body)
              (subs-in-lambda val obvars kp int-body))))
          (recon-l      ; reconstruct lambda from the substituted body
           (syntax-rules ()
             ((_ (k bvars) () result)
              (appl k (??!lambda bvars result)))))
          (appl         ; apply the continuation
           (syntax-rules ()     ; add the result to the end of k
             ((_ (a b c d) result) (a b c d result))
             ((_ (a b c) result)   (a b c result))))
          (finish
           (syntax-rules ()
             ((_ () () exp)  exp)
             ((_ rem-bvars rem-ovals exps)
              (??!apply (??!lambda rem-bvars exps) . rem-ovals))))
          )
       ; In the following, finish is the continuation...
       (subs oval (finish other-bound-vars other-ovals) body)))))
```

Figure 1. The implementation of the ??!apply macro

yields (exp datum) for any expression exp with no free occurrence of (??! x). This observation proves that id-cps, in contrast to the macro id, always composes with any macro. Furthermore, id-cps is the unit of the macro composition. Furthermore, the expression (exp (??! x)) may even be an application of a legacy, non-CPS-style macro, e.g., mcase1. The last macro in a compositional chain may be a "third-party" macro, written without our methodology.

More examples of developing complex macros by macro composition can be found in [8]. One of the examples is a compile-time implementation of a factorial over Peano-Church numerals. No computer science paper is complete without working out the factorial. Figure 2 shows the factorial macro, excerpted from [8].

```
(define-syntax ?plc-fact
  (syntax-rules ()
    ((?plc-fact co k)           ; pattern
     (?plc-zero? co
       (??!lambda (c) (?plc-succ (??! c) k)) ; k on c being zero
       (??!lambda (c)           ; k on c being non-zero
          (?plc-pred (??! c)    ; the predecessor
             (??!lambda (c-1)
               (?plc-fact (??! c-1)
                 (??!lambda (c-1-fact)
                     (?plc-mul (??! c) (??! c-1-fact) k))))))))))
```

Figure 2. The factorial CPS macro

This fragment is quoted here to illustrate modularity: the factorial macro is built by composition of separately defined and tested arithmetic (?plc-pred, ?plc-mul) and comparison macros. Selection forms such as ?plc-zero? take several continuation arguments but continue only one. A paper [9] develops even more challenging R5RS macro: the normal-order evaluator for lambda calculus. The challenge comes from capture-free substitutions and repeated identifications of left-most redexes. The evaluator is truly modular: it relies on a separately designed and tested one-step beta-substitutor, and on a higher-order macro-combinator map. Our technique of CPS and macro-level abstractions guarantees composability of all pieces.

The reasons for such a guarantee are Plotkin's simulation and indifference theorems discussed in [14][2]. Formally, composability can be defined as an observational equivalence of $f\,(g\,v)$ and $f\,eval(g\,v)$, where f, g, and v are values. The equivalence is a tautology in a call-by-value language: A call-by-value evaluator always evaluates arguments of an application. The equivalence is not assured in a call-by-name language (e.g., a macro system). For CPS terms, the equivalence that defines composability can be written as:

$$\mathcal{F}[\![f\,(g\,v)]\!]\;\lambda k.k\;\equiv \mathcal{F}[\![g\,v]\!]\;\lambda n.\Psi[\![f]\!]\;(\lambda k.k)\,n$$

$$\cong \ \Psi[\![f]\!]\,(\lambda k.k)\,eval(\mathcal{F}[\![g\,v]\!]\,\lambda k.k)$$

where $\mathcal{F}$ and Ψ are Fisher's CPS transformers for expressions and values correspondingly [14]. The latter equivalence holds for both call-by-name $eval_n$ and call-by-value $eval_v$ evaluators, which means that CPS terms compose in both calculi. The proof of that assertion is based on the Plotkin's indifference and simulation theorems combined with a lemma

$eval_v M$ defined $\Rightarrow \mathcal{F}[\![M]\!] =_v \lambda k.k\Psi[\![eval_v M]\!]$
$eval_v M$ undefined $\Rightarrow eval_v \mathcal{F}[\![M]\!]k$ undefined, $eval_n \mathcal{F}[\![M]\!]k$ undefined

The lemma is a re-statement of Plotkin's colon-translation (cf. also lemmas 3 and 4 of [2] for a more efficient CPS transformation).

5 Systematic Construction of R5RS Macros

The `?plc-fact` macro on Fig. 2 curiously looked rather similar to a regular Scheme procedure written in CPS. This observation raises a question if R5RS macros can be written systematically – by translating from the corresponding Scheme functions. This section answers affirmatively and elaborates an example of such a translation.

The problem, which was originally posed on a newsgroup comp.lang.scheme, is to write a macro `quotify` that should work as follows:

```
(quotify (i j k) (s4 k s5 l () (m i) ((('i)))))
==> `(s4 ,k s5 l () (m ,i) (((',i))))
```

In other words, the problem is a non-destructive modification of selected leaves in an S-expression tree. We start by writing a regular Scheme procedure that implements the required transformation:

```
(define (quotify-fn symbs-l tree)
  (define (doit tree)
    (map
     (lambda (node)
       (if (pair? node)
         ; recurse to process children
           (doit node)
           (if (memq node symbs-l)
             ; replace the leaf
               (list 'unquote node)
               node)))
     tree))
  (list 'quasiquote (doit tree)))
```

After the procedure is verified we *mechanically* convert it to a CPS macro, using a Scheme–to–syntax-rules translator [10]. For clarity, this section will be developing the CPS version of `quotify-fn` and the corresponding macro manually and side-by-side. A set of tables through the end of this section document all the steps. The tables juxtapose CPS procedures (in the left column) and the corresponding macros. Before we proceed with the CPS conversion of `quotify-fn`, we need CPS versions of Scheme primitives such as `car` and `cdr`. Table 1 shows these primitives and their macro counterparts. We distinguish the names of CPS macros with the leading character '?'.

`(define (cons-cps a b k)`
` (k (cons a b)))`
	`(define-syntax ?cons`
	`  (syntax-rules ()`
	`    ((_ x y k)`
	`     (??!apply k (x . y)))))`

`(define (car-cps x k)`
`  (k (car x)))`
	`(define-syntax ?car`
	`  (syntax-rules ()`
	`    ((_ (x . y) k) (??!apply k x))))`

`(define (cdr-cps x k)`
`  (k (cdr x)))`
	`(define-syntax ?cdr`
	`  (syntax-rules ()`
	`    ((_ (x . y) k) (??!apply k y))))`

`(define (ifpair? x kt kf)`
`  (if (pair? x) (kt x) (kf x)))`
	`(define-syntax ?ifpair?`
	`  (syntax-rules ()`
	`    ((_ (a . b) kt kf)`
	`     (??!apply kt (a . b)))`
	`    ((_ non-pair kt kf)`
	`     (??!apply kf non-pair))))`

Table 1. Primitive CPS functions and macros. The primitive `ifpair?` receives a value `x` and two continuations `kt` and `kf`. The primitive passes `x` to `kt` if `x` is a pair. Otherwise, `x` is passed to `kf`.

The equality-comparison–and–branching primitive is a bit more complex, Table 2. The `?ifeq?` macro is the only one in this section that requires care. However, the macro does a simple task and can be written and tested independently of other code.

We can combine the primitives into more complex functions, for example, a CPS function that checks the membership of an item in a list, Table 3. The similarity between the function `memq-cps` and the corresponding macro `?memq` is striking. We also need another complex function: `map`, which is a higher-order combinator (Table 4). This gives us all the pieces to write the `quotify-cps` function and the corresponding CPS macro, Table 5.

The macro `?quotify` is the sought answer. It is a modular macro and can be freely composed with other CPS macros. It is a R5RS, hence, hygienic macro. It is, however, ungainly, e.g.: `(?quotify (i j k) (s4 k s5 l () (m`

```scheme
(define (ifeq? a b kt kf)
  (if (eq? a b) (kt a) (kf a)))
```

```scheme
(define-syntax ?ifeq?
  (syntax-rules ()
    ((_ (x . y) b kt kf)
     (??!apply kf (x . y)))
    ((_ () b kt kf)
     (??!apply kf ()))
    ((_ a b _kt _kf)
     (let-syntax
         ((aux
           (syntax-rules (a)
             ((_ a kt kf)
              (??!apply kt a))
             ((_ other kt kf)
              (??!apply kf a)))))
       (aux b _kt _kf)))))
```

Table 2. The compare-and-branch primitive. It receives two values and two continuations, `kt` and `kf`. The primitive passes the first value to `kt` if the two values are the same. Otherwise, the first value is given to `kf`.

```scheme
; if a occurs in lst, pass
; the sublist to kt. Otherwise,
; pass () to kf
(define (memq-cps a lst kt kf)
  (ifpair? lst
    (lambda (lst) ; it's a pair
      (car-cps lst
        (lambda (x)
          (ifeq? a x
            ; match
            (lambda (_)
               (kt lst))
            ; mismatch
            (lambda (_)
              (cdr-cps lst
                (lambda (tail)
                  (memq-cps
                     a tail kt kf)
      )))))))
    (lambda (empty)
      (kf empty))))
```

```scheme
(define-syntax ?memq
  (syntax-rules ()
    ((_ a lst kt kf)
     (?ifpair? lst
        (??!lambda (lst) ; it's a pair
         (?car lst
          (??!lambda (x)
           (?ifeq? a (??! x)
            ; match
            (??!lambda (_)
               (??!apply kt (??! lst)))
            ; mismatch
            (??!lambda (_)
              (?cdr lst
                (??!lambda (tail)
                  (?memq
                    a (??! tail) kt kf)
       )))))))
    (??!lambda (empty)
      (??!apply kf (??! empty)))))))
```

Table 3. Testing the list membership.

```
(define (map-cps f lst k)
  (ifpair? lst
     ; lst still has elements
     (lambda (lst)
       (car-cps lst
          (lambda (x)
            (f x
               (lambda (fx)
                 (cdr-cps lst
                    (lambda (tail)
                      (map-cps f tail
                           (lambda (res)
                             (cons-cps fx
                               res k))
       ))))))))
     ; lst is empty
     (lambda (empty)
       (k empty))))
```

```
(define-syntax ?map
  (syntax-rules ()
    ((?map f lst k)
     (?ifpair? lst
        ; lst still has elements
        (??!lambda (lst1)
          (?car (??! lst1)
            (??!lambda (x)
              (??!apply f (??! x)
                (??!lambda (fx)
                  (?cdr (??! lst1)
                    (??!lambda (tail)
                      (?map f (??! tail)
                        (??!lambda (res)
                          (?cons (??! fx)
                            (??! res) k))
          ))))))))
        ; lst is empty
        (??!lambda (empty)
          (??!apply k (??! empty)))))))
```

Table 4. Higher-order CPS combinator `map-cps` and its R5RS macro counterpart `?map`

`i) ((('i))) (??!lambda (r) (begin (??! r))))`. The ease of use is an important issue for macros, since making the code look pretty is the principal reason for their existence. Therefore, as the last step we wrap `?quotify` into a non-CPS macro:

```
(define-syntax quotify (syntax-rules ()
   ((_ args ...)
      (?quotify args ...
         ; the identity continuation: (lambda (x) x)
         (??!lambda (r) (begin (??! r)))))))
```

The generic nature of the wrapper is noteworthy. The following expression is a usage example and a simple test of `quotify`:

```
(let ((i 'symbol-i) (j "str-j") (k "str-k"))
  (display (quotify (i j k) (s4 k s5 l () (m i) ((('i))))))
  (newline))
```

When evaluated, the expression prints the expected result:

```
(s4 str-k s5 l () (m symbol-i) ((((quote symbol-i)))))
```

The macro `quotify` fulfills the original specification. It is easy to use on its own. If we want to compose it with other macros we should use its CPS version,

```
(define (quotify-cps symbs-l tree k)
  (define (doit tree k)
    (map-cps
     (lambda (node k)
       (ifpair? node
         ; recurse to process children
         (lambda (node1)
           (doit node1 k))
         (lambda (node1)
           ; node1 is not a pair
           (memq-cps node1 symbs-l
             ; matches
             (lambda (_)
               (k (list
                    'unquote
                    node1)))
             ; mis-matches: leave the node alone
             (lambda (_)
               (k
                 node1))))))
     tree k))
  (doit tree
       (lambda (conv-tree)
         (k (list
              'quasiquote
              conv-tree)))))
```

```
(define-syntax ?quotify
  (syntax-rules (quasiquote unquote)
  ((_ symbs-l _tree _k)
   (letrec-syntax
       ((doit
         (syntax-rules ()
           ((_ tree k)
            (?map
             (??!lambda (node mk)
                (?ifpair? (??! node)
                ; recurse to process children
                (??!lambda (node1)
                    (doit (??! node1) (??! mk)))
                (??!lambda (node1)
                    ; node1 is not a pair
                    (?memq (??! node1) symbs-l
                      ; matches
                      (??!lambda (_)
                        (??!apply (??! mk)
                          (unquote
                            (??! node1))))
                      ; mis-matches
                      (??!lambda (_)
                        (??!apply (??! mk)
                          (??! node1)))))))
                tree k)))))
     (doit _tree
           (??!lambda (conv-tree)
              (??!apply _k
                (quasiquote
                  (??! conv-tree))))))))))
```

Table 5. CPS function `quotify-cps` and its R5RS macro counterpart `?quotify`

`?quotify`, instead. As Tables 1-5 show, R5RS macros can indeed look just like regular Scheme procedures, and – more importantly – be systematically developed as regular, tail-recursive Scheme procedures.

6 Related Work

Employing the continuation-passing style for writing R5RS macros is not a new idea. This was the impetus of a paper "Writing macros in continuation-passing style" by Erik Hilsdale and Daniel P. Friedman [6]. The paper has noted that "The natural way to transform these [CPS] procedures into macros is to assume the existence of two macros that behave like lambda and apply. ... Unfortunately, it doesn't work because of hygiene ... So it seems we cannot write a macro to express these syntactic continuations." Therefore, Hilsdale and Friedman chose a different approach, which turns normally anonymous continuation abstractions into regular, named macros. The authors did not seem to regard that approach entirely satisfactory: they wrote that their technique is "akin to programming in an assembly language for macros, where we have given up not only all but one control structure (pattern selection) but also lexical closures." The present paper demonstrated how to regain what was considered lost, including anonymous syntactic abstractions, branching and other control structures, and even higher-order combinators.

Keith Wansbrough [16] has made a compelling case for macros even in a non-strict functional language. The body of his paper however shuns function-like (i.e., parameterized) macros, which are only mentioned in a section discussing future research. The paper [16] never mentions macro compositions and the accompanying problems.

Camlp4 [13] is a Pre-Processor-Pretty-Printer for OCaml. One of its applications is extending the OCaml syntax by means of a *quotation*, which is essentially a macro. A quotation is a string-to-string or a string-to-AST transformation. The Camlp4 documentation mentions the possibility of nesting quotations but does not discuss composability. It is easy to show that a quotation that places its argument in a non-expression position cannot be composed with other quotations. The situation is analogous to that of R5RS macros. Therefore, the solution to the composability problem proposed in the present paper applies to quotations as well.

Steve Ganz, Amr Sabry, and Walid Taha [4] introduced a novel macro system for a modern functional language that guarantees that the result of a macro-expansion is well-formed and *well-typed*. This is a highly appealing property of their MacroML system. The MacroML paper however does not address the issues of modularity and composability. It is not clear from the examples given in [4] if MacroML macros always compose.

Almost all implementations of Scheme offer another macro system, called defmacro or low-level macros. Like R5RS macros, defmacro is a head-first re-writing system. A defmacro syntax transformer however is a regular Scheme procedure that produces an S-expression. We can write defmacro transformers using all standard Scheme procedures and forms. Furthermore, some Scheme systems provide a non-standard procedure `macroexpand`, which a syntax transformer can use to force the expansion of arbitrary forms. Low-level macros in such systems are therefore composable. If the procedure `macroexpand` is not available, low-level macros can still be developed in a modular fashion, with the methodology proposed in the present paper.

Chez Scheme [3] generalized R5RS and low-level macro systems of Scheme into a syntax-case system. The latter is also a head-first re-writing system. Syntax-case transformers operate on an abstract datatype of code (called syntax-object) rather that strings or parsed trees. The transformers can analyze syntax objects with pattern matching; the transformers can also use standard Scheme procedures and syntax-object methods. Whether syntax-case macros are composable depends on the availability of a non-standard procedure `expander` for use in a transformer. The procedure `expander` is mentioned in the implementation section of [3]; however, this procedure is not a part of the syntax-case specification.

7 Conclusions

We have demonstrated that macro programming is in general non-functional. Macros that place their arguments in non-expression positions and cannot force expansion of their arguments do not compose. The fundamental engineering principle of modular design therefore does not generally apply to macros.

We have presented a methodology that makes macro programming applicative and systematic. The methodology is centered on writing macros in continuation-passing style. The novel element is the denotation for parameterized future macro-expansion actions: anonymous first-class syntactic abstractions. The insight that syntactic abstractions are not themselves macros made the development of such abstractions possible. Writing macros in CPS removes the obstacles to composability; anonymous macro abstractions encode macro-continuations and make the methodology practical. Therefore, complex macros can be constructed by combining separately developed and tested components, by composition and higher-order operators. The final result built by the CPS-composition can be applied to the identity macro-continuation to yield an easy-to-use macro.

R5RS macros specifically admit a stronger result: a mechanical translation from the corresponding Scheme procedures. The technique consists of three steps: (1) write the required S-expression transformation in regular Scheme; (2) mechanically convert the Scheme procedure into CPS; and (3) translate the CPS procedure into a macro, replacing regular `lambda`s with macro-lambdas and annotating occurrences of bound variables. Coding of R5RS macros becomes no more complex than programming of regular Scheme functions. A real-world example elaborated in the paper has demonstrated the practical power and the convenience of the technique.

The approach of this paper makes programming of Scheme macros and of other related head-first re-writing systems more like a craft than a witchcraft.

Acknowledgments I would like to thank anonymous reviewers for their helpful comments. 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. Clinger, W.: Macros in Scheme. Lisp Pointers, IV(4) (December 1991) 25-28
2. Danvy, O., Filinski, A.: Representing Control: A study of the CPS transformation. Mathematical Structures in Computer Science, vol. 2, No. 4 (1992) 361-391
3. Dybvig, R. K.: Writing Hygienic Macros in Scheme with Syntax-Case. Computer Science Department, Indiana University (1992)
4. Ganz, S., Sabry, A., Taha, W.: 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)
5. Graham, P.: Beating the Averages. April 2001. http://www.paulgraham.com/avg.html
6. Hilsdale, E., Friedman, D. P.: Writing macros in continuation-passing style. Scheme and Functional Programming 2000. September 2000. http://www.ccs.neu.edu/home/matthias/Scheme2000/hilsdale.ps
7. Kelsey, R., Clinger, W., Rees J. (eds.): Revised5 Report on the Algorithmic Language Scheme. J. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September 1998
8. Kiselyov, O.: Transparent macro-CPS programming. November 30, 2001. http://pobox.com/~oleg/ftp/Scheme/syntax-rule-CPS-lambda.txt
9. Kiselyov, O.: Re-writing abstractions, or Lambda: the ultimate pattern macro. December 16, 2001. http://pobox.com/~oleg/ftp/Computation/rewriting-rule-lambda.txt
10. Kiselyov, O.: A Scheme–to–syntax-rules compiler. July 5, 2002. http://pobox.com/~oleg/ftp/Scheme/cps-macro-conv.scm
11. Kohlbecker, E. E.: Syntactic Extensions in the Programming Language Lisp. Ph.D. Thesis. Indiana Classics University (1986)
12. Kohlbecker, E. E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In Proc. ACM Conference on Lisp and Functional Programming, pp. 151-161 (1986)
13. de Rauglaudre, D.: Camlp4. Version 3.04+3, January 20, 2002. http://caml.inria.fr/camlp4/
14. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. Proc. 1992 ACM Conference on Lisp and Functional Programming, pp. 288-298. Technical Report 92-180, Rice University
15. Shivers, O.: A universal scripting framework, or Lambda: the ultimate 'little language'. In: Jaffar, J., Yap, R. H. C. (eds.): Concurrency and Parallelism, Programming, Networking, and Security. Lecture Notes in Computer Science, Vol. 1179. Springer-Verlag, Berlin Heidelberg New York (1996) 254-265
16. Wansbrough, K.: Macros and Preprocessing in Haskell (1999). http://www.cl.cam.ac.uk/~kw217/research/misc/hspp-hw99.ps.gz