Writing Macros in Continuation-Passing Style

Erik Hilsdale Daniel P. Friedman*

Computer Science Department
Indiana University
Bloomington, Indiana 47405
{ehilsdal, dfried}@cs.indiana.edu

Abstract

The Scheme programming language has a standard mechanism for syntactic extension that is little used because it is perceived to have not enough expressive power. While there are useful transformations the standard mechanism cannot do, it is possible to create powerful and portable syntactic extensions by writing syntax transformers in a continuation-passing style. We introduce this style, and show how it may be used to perform arbitrary Turing-complete computation over expression shapes during the expansion process. We conclude with a real-world use of the technique, and evaluate the loss of clarity necessitated by the standard mechanism.

1 Introduction

The Revised[5] Report on the Algorithmic Language Scheme includes within it a high-level language for defining syntactic extensions, one based on the `syntax-rules` form. Though traditional syntactic extension in Lisp systems involves representing syntax as Lisp data and allowing the full Lisp runtime system to be used to process that data during expansion, much work has been done in the Scheme community, through pattern rewriting systems and the concept of hygiene, to make syntactic extension more tractable and to enforce a cleaner separation between expansion and evaluation. Yet because of the restrictions of this pattern language, it has been viewed as inadequate for complex language extension. Most implementations of the Scheme language include some other mechanism. This paper discusses the computational power of the Scheme macro system as a functional language in itself, considers its limitations in expressiveness, and relates how the technique of writing macros in a continuation-passing style—one that has been used by serious practitioners for some time—provides a way to get around some of these limitations.

Using the syntactic extension mechanisms described in the Revised[5] Report, keywords can be bound to syntax transformers, but unlike traditional quasiquotation systems, these transformers are pattern based. While they can be viewed as functions, their domain and codomain are not S-expressions or character sequences, but rather abstract syntax objects. The only operations allowed on syntax objects are matching and binding through a pattern language, and construction through a related template language.

*This work was supported in part by the National Science Foundation under grant CCR-9633109.

This means that a large amount of functionality the Lisp world takes for granted is simply not available. We cannot apply procedures from the run-time environment to syntax objects. We cannot create syntax objects by consing together lists or reading from external files. The syntax expander simply does not have access to the run-time system. Because of the declarative nature of syntax expanders, it is difficult to build syntax objects from pieces in a normal recursive fashion.

We can get Turing-complete computation out of such a restricted system: Though a syntax transformer can only construct syntax objects, the expansion system redispatches on the operator of the newly constructed syntax object. If the operator is a keyword with a syntax transformer bound to it, the new syntax object is transformed as well. But more importantly, we can use this computational power to write syntax expanders in a continuation-passing style, allowing some ability to build expressions in separate pieces.

In Section 2 of this paper we give a short introduction to the standard Scheme macro system, `syntax-rules`, and introduce some of its limitations. In Section 3 we first derive continuation-passing macros from continuation-passing functions, and from this derivation choose an appropriate representation of continuations. In Section 4 we show four uses for this style of macros: an expansion inspector, a structural equivalence checker over syntax, a reducer for a call-by-name lambda calculus, and a robust implementation of disjoint sums. We discuss related work in Section 5 and present our conclusions in Section 6.

2 Syntactic Extension in Scheme

Keywords in Scheme are bound to syntax transformers either globally, with the `define-syntax` form, or locally, with the `let-syntax` and `letrec-syntax` forms. They are specified, however, with the `syntax-rules` form. This form lists a number of patterns to match, and for each pattern gives a template with which to construct new syntax. For example, here is the binding for Scheme's `and` expression:

```
(define-syntax and
  (syntax-rules ()
    ((and)
     #t)
    ((and Exp)
     Exp)
    ((and Exp0 Exp ...)
     (if Exp0 (and Exp ...) #f))))
```

The `define-syntax` form gives a global binding to the keyword `and`. The `syntax-rules` form specifies that `(and)` expands into boolean true, `(and` α`)` expands into α, and `(and` α β`*)` expands into `(if` α `(and` β`*) #f)`. Because this output is reexpanded, though, it is safe to expand into another instance of an `and` expression; the inner `and` will be subsequently expanded.

If it is the case that a definition of the `and` keyword is necessary only in one expression, we could have bound it with the `letrec-syntax` form. This expression

```
(letrec-syntax ((and (syntax-rules ()
                       ((and)
                        #t)
                       ((and Exp)
                        Exp)
                       ((and Exp0 Exp ...)
                        (if Exp0 (and Exp ...) #f)))))
  (and alpha beta))
```

expands into

```
(if alpha beta #f)
```

without exporting the `and` macro to the surrounding scope. The `let-syntax` form can be used when the expander for a keyword need not expand into references to that keyword.

Though `syntax-rules` transformers may also have other kinds of patterns and templates—such as those for vectors—we limit ourselves to consideration of list patterns for this paper. However, it is important to realize that these patterns are patterns for equality. Though a pattern can test that a certain expression is a literal 42, it cannot test for "any literal less than 42" or for "any identifier that will eventually be bound to a number".

There is no means to "partially expand" a subform for use in the resulting template, such as a `with` or `with-syntax` form [1]. That is, there is no way to expand a portion of the syntax, and then use that expanded code to decide how to expand another subform. This makes syntax expanders difficult to abstract as there is no primitive notion of a recursive call to build a portion of an output expression. It is this problem that continuation-passing style macros can solve.

3 Defining the Style

First-order accumulator-style macros are a common way of processing data during expansion

```
(define-syntax syn-reverse
  (syntax-rules ()
    ((syn-reverse () Acc)
     (do-something-with Acc))
    ((syn-reverse (X Y ...) (Acc ...))
     (syn-reverse (Y ...) (Acc ... X)))))
```

This technique is used, for example, to generate temporary identifiers for the expansion of `letrec` [4]. While this style of writing "tail-recursive" syntax expanders can handle linear problems, when faced with tree-recursive problems we need more power than this technique provides.

3.1 Continuation Passing Style

The key insight that gives us the power we need is that recurring computation is done by transforming into a shape where the operator of the resulting shape determines what happens next. That seems familiar. "What happens next" sounds a lot like "continuation," and in fact the key realization is that continuation-passing style maps directly onto syntactic extension.

It is useful to derive continuation-passing style for syntactic transformers from continuation-passing style for ordinary programs. Here is a definition of `reverse*`, a procedure that accepts a possibly deep list, and reverses the list and all of its sublists:

```
(define reverse*
  (lambda (x)
    (cond
      ((null? x) '())
      ((list? (car x))
       (snoc (reverse* (cdr x)) (reverse* (car x))))
      (else
        (snoc (reverse* (cdr x)) (car x))))))
```

Its helper, `snoc`, can be defined with:

```
(define snoc
  (lambda (ls x)
    (append ls (list x))))
```

We can convert this to a continuation-passing style that uses closures to represent our continuations. We will find it helpful, though, to define an `apply-cont` procedure rather than applying the continuations directly.

```
(define apply-cont
  (lambda (k v)
    (k v)))
```

With this definition, our continuation-passing style versions of `reverse*` and `snoc` look like

```
(define reverse*-cps
  (lambda (x k)
    (cond
      ((null? x) (apply-cont k '()))
      ((list? (car x))
       (reverse*-cps (cdr x)
         (lambda (newTail)
           (reverse*-cps (car x)
             (lambda (newHead)
               (snoc-cps newTail newHead k))))))
      (else
        (reverse*-cps (cdr x)
          (lambda (newTail)
            (snoc-cps newTail (car x) k)))))))

(define snoc-cps
  (lambda (ls x k)
    (apply-cont k (append ls (list x)))))
```

With the procedures in this form, we can transform both of them into macros.

3.2 A Wrong Turn

The natural way to transform these procedures into macros is to assume the existence of two macros that behave like lambda and apply. This syntactic lambda, or (since it is only used to create continuations) `syn-cont`, might take a formal argument and an expression body, and freeze expansion of the body until the syn-cont is applied, with apply-syn-cont, to an expression. Assuming these macros are defined, we could write

```
(define-syntax syn-reverse*
  (syntax-rules ()
    ((syn-reverse* ()                   K)
     (apply-syn-cont K ()))
    ((syn-reverse* ((Head0 ...) . Tail) K)
     (syn-reverse* Tail
       (syn-cont (NewTail)
         (syn-reverse* (Head0 ...)
           (syn-cont (NewHead)
             (syn-snoc NewTail NewHead K))))))
    ((syn-reverse* (Head . Tail)        K)
     (syn-reverse* Tail
       (syn-cont (NewTail)
         (syn-snoc NewTail Head K))))))

(define-syntax syn-snoc
  (syntax-rules ()
    ((syn-snoc (X ...) Y K)
     (apply-syn-cont K (X ... Y)))))
```

thus leaving only syn-cont and apply-syn-cont. The latter is easier: Applying a continuation to a value, in the general case, involves making the value available to the continuation and transferring control to the continuation. With macros, the only way to transfer control to a form is to expand into it, and the only way to make syntactic values available is to include those values in the expansion. So apply-syn-cont can be defined as

```
(define-syntax apply-syn-cont
  (syntax-rules ()
    ((apply-syn-cont (Op Arg ...) Value ...)
     (Op Arg ... Value ...))))
```

Thus, whenever the apply-syn-cont macro is expanded, it splices its second and further arguments into its first argument, and expansion continues with the newly spliced expression.

As for syn-cont, it must accept the extra values passed to it by apply-syn-cont, make those values available to its body, and transfer control to that body. Since $\mathrm{R^5RS}$ supports let-syntax, we might think to use this form to bind a helper macro that, when expanded, will substitute whatever expressions given it for the continuation formals in the body, and we expand directly into that form.

```
(define-syntax syn-cont
  (syntax-rules ()
    ((syn-cont (Formal ...) Body Val ...)
     (let-syntax ((f (syntax-rules ()
                        ((f Formal ...) Body))))
       (f Val ...)))))
```

This can be seen as a rather peculiar way to regain lambda from let. Unfortunately, it doesn't work because of hygiene. Consider the macro defined with let-syntax:

```
(syntax-rules ()
  ((f Formal ...) Body))
```

The $\mathrm{R^5RS}$ macro system guarantees that the introduced Formal identifiers will not be captured by Body, dashing our hopes for a general syn-cont macro.

Even if it had worked, though, the substitution of the argument would naïvely use the standard macro-template substitution mechanism of the $\mathrm{R^5RS}$ macro system. This mechanism is unaware of our intended scopes; it substitutes the values for *all* occurrences of the formals in the body, even if such an occurrence should be shadowed by another syn-cont. Unfortunately, it's worse than that: It would actually substitute a value in place of the *formal parameter* of a contained syn-cont, if allowed:

```
(apply-syn-cont
  (syn-cont (X)
    (context
      (syn-cont (X)
        (context X))))
  (V W))
```

would expand into

```
(context
  (syn-cont ((V W))
    (context (V W))))
```

So it seems we cannot write a macro to express these syntactic continuations. But all is not lost.

3.3 Representing Continuations

What we have seen is that we cannot use lexical closures when we represent our syntactic continuations. But by taking our continuation-passing style transformation one step further and abstracting out our representation of continuations, we can avoid these problems completely. In Scheme, we can write reverse*-cps and name the constructor of every continuation rather than leaving them as anonymous procedures:

```
(define reverse*-cps
  (lambda (x k)
    (cond
      ((null? x) (apply-cont k '()))
      ((list? (car x))
       (reverse*-cps (cdr x)
         (deep-tail-cont (car x) k)))
      (else
        (reverse*-cps (cdr x)
          (shallow-tail-cont (car x) k))))))

(define snoc-cps
  (lambda (ls x k)
    (apply-cont k (append ls (list x)))))

(define deep-tail-cont
  (lambda (head k)
    (lambda (newTail)
      (reverse*-cps head
        (deep-head-cont newTail k)))))

(define deep-head-cont
  (lambda (newTail k)
    (lambda (newHead)
      (snoc-cps newTail newHead k))))

(define shallow-tail-cont
  (lambda (head k)
    (lambda (newTail)
      (snoc-cps newTail head k))))
```

So too can we write **syn-reverse*** as a syntactic extension without the use of **syn-cont**. We represent each of the continuation creators as individual macros. Each such continuation is first given its free pattern variables, and then the form is passed to an expansion of another continuation-accepting macro. When the continuation is applied, **apply-syn-cont** splices in its "bound" pattern variables as before:

```
(define-syntax syn-reverse*
  (syntax-rules ()
    ((syn-reverse* ()                      K)
     (apply-syn-cont K ()))
    ((syn-reverse* ((Head0 ...) . Tail) K)
     (syn-reverse* Tail
       (syn-deep-tail-cont (Head0 ...) K)))
    ((syn-reverse* (Head . Tail)         K)
     (syn-reverse* Tail
       (syn-shallow-tail-cont Head K)))))

(define-syntax syn-snoc
  (syntax-rules ()
    ((syn-snoc (X ...) Y K)
     (apply-syn-cont K (X ... Y)))))

(define-syntax syn-deep-tail-cont
  (syntax-rules ()
    ((syn-deep-tail-cont Head K   NewTail)
     (syn-reverse* Head
       (syn-deep-head-cont NewTail K)))))

(define-syntax syn-deep-head-cont
  (syntax-rules ()
    ((syn-deep-head-cont NewTail K   NewHead)
     (syn-snoc NewTail NewHead K))))

(define-syntax syn-shallow-head-cont
  (syntax-rules ()
    ((syn-shallow-head-cont Head K   NewTail)
     (syn-snoc NewTail Head K))))
```

We could even dispense with the helper macro **syn-snoc** completely when defining all of our continuations separately, as we can use the pattern-matching facilities of **syntax-rules** to append the values.

```
(define-syntax syn-deep-head-cont
  (syntax-rules ()
    ((syn-deep-head-cont (NewTail0 ...) K   NewHead)
     (apply-syn-cont K (NewTail0 ... NewHead)))))

(define-syntax syn-shallow-head-cont
  (syntax-rules ()
    ((syn-shallow-head-cont Head K   (NewTail0 ...))
     (apply-syn-cont K (NewTail0 ... Head)))))
```

Admittedly, this places a large burden on the programmer and maintainer of the code. It seems 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. But the style does allow, with some effort, the expression of fairly complex systems.

4 Using the Style

We present four examples of continuation-passing macros. First, we show a trivial example of the style, by implementing an expansion inspector. Then we show how we can write an expansion-time structural equality tester for expressions. Next we use a part of this tester to implement a beta reducer for a lambda calculus. And finally, we show how this technique can be used to write a portable disjoint sum package with some safety considerations.

In order to save space and aid readability, we take advantage of the fact that applying a nullary continuation involves simply expanding into it. So the remainder of this paper will elide creating a separate continuation macro for nullary continuations (and will often forgo using the **apply-syn-cont** macro on them) in preference for direct expansion.

4.1 Inspection

While the procedure **syntax-expand**—or some other procedure for inspecting expanded expressions—is present in many Lisp and Scheme systems, it is not standard. If, however, we've defined a form (or set of forms) in continuation-passing style, we can pass in a very simple continuation that expands to the output as a literal datum

```
(define-syntax inspect-cont
  (syntax-rules ()
    ((inspect-cont Value)
     'Value)))
```

Since we may have syntactic continuations that accept multiple values, it may be a better idea to have **inspect-cont** accept them.

```
(define-syntax inspect-cont
  (syntax-rules ()
    ((inspect-cont Value ...)
     '(the-values-are Value ...))))
```

We can use this to test **syn-reverse***:

```
> (syn-reverse* (a (b c) d e f)
    (inspect-cont))
(the-values-are (f e d (c b) a))
```

Of course, this inspector only works with macros (or, more likely, helper macros) written explicitly in continuation-passing style.

4.2 Equality Testing

A macro to check for equality of shapes is fairly easy to define using success and failure continuations:

```
(define-syntax syn-shape-equal
  (syntax-rules ()
    ((syn-shape-equal (Head0 . Tail0) (Head1 . Tail1)
       SK FK)
     (syn-shape-equal Head0 Head1
       (syn-shape-equal Tail0 Tail1 SK FK)
       FK))
    ((syn-shape-equal NonPair0        (Head1 . Tail1)
       SK FK)
     (apply-syn-cont FK))
    ((syn-shape-equal (Head0 . Tail0) NonPair1
       SK FK)
     (apply-syn-cont FK))
    ((syn-shape-equal NonPair0        NonPair1
       SK FK)
     (apply-syn-cont SK))))
```

Checking the equality of identifiers, however, requires us to use a trick of the macro system. The **syntax-rules** form allows us to specify a number of *literals* that must match exactly if a pattern containing one of the literals is to match.

Thus, given two identifiers X and Y, a success continuation, and a failure continuation, we expand into an expression that locally binds a **syntax-rules** form with the literal X. This local expander then dispatches on its argument to the success or failure continuation. Applying the keyword bound to this local expander to Y completes the test.

```
(define-syntax syn-eq
  (syntax-rules ()
    ((syn-eq X Y SK FK)
     (let-syntax ((f (syntax-rules (X)
                       ((f X _SK _FK)
                        (apply-syn-cont _SK))
                       ((f NonX _SK _FK)
                        (apply-syn-cont _FK)))))
       (f Y SK FK)))))
```

4.3 Lambda Calculi

Once we have the ability to check for identifier equality, we can use it to implement the substitution pass of a simple lambda calculus of abstraction, application, and variables.

```
Exp ::= (lambda (Id) Exp)
      | (Exp Exp)
      | Id
```

First we implement substitution.

```
(define-syntax lc-subst
  (syntax-rules (lambda)
    ((lc-subst New Old (lambda (Formal) Body) K)
     (syn-eq Old Formal
       (apply-syn-cont K (lambda (Formal) Body))
       (lc-subst temp Formal Body
         (lc-subst-newbody1-k New Old temp K))))
    ((lc-subst New Old (Op Arg) K)
     (lc-subst New Old Op
       (lc-subst-rator-k New Old Arg K)))
    ((lc-subst New Old Var K)
     (syn-eq Old Var
       (apply-syn-cont K New)
       (apply-syn-cont K Var)))))

(define-syntax lc-subst-newbody1-k
  (syntax-rules ()
    ((lc-subst-newbody1-k New Old Temp K NewBody1)
     (lc-subst New Old NewBody1
       (lc-subst-newbody2-k Temp K)))))

(define-syntax lc-subst-newbody2-k
  (syntax-rules ()
    ((lc-subst-newbody2-k Temp K NewBody2)
     (apply-syn-cont K (lambda (Temp) NewBody2)))))

(define-syntax lc-subst-rator-k
  (syntax-rules ()
    ((lc-subst-rator-k New Old Arg K   NewOp)
     (lc-subst New Old Arg
       (lc-subst-rand-k NewOp K)))))

(define-syntax lc-subst-rand-k
  (syntax-rules ()
    ((lc-subst-rand-k NewOp K   NewArg)
     (apply-syn-cont K (NewOp NewArg)))))
```

We have taken advantage of hygiene to generate a fresh variable **temp** as we substitute through lambda contours: Because it will eventually be used as an introduced variable in a binding form (**lambda**), the **temp** identifier is guaranteed to be fresh for every expansion of the **lc-subst** macro.[1]

With this implementation of substitution we implement a single-step reducer that takes two continuations; a success continuation of one argument that is applied if there is a pending reduction, and a nullary failure continuation if the expression is in normal form. Though we have chosen a normal-order reduction strategy, we can obtain applicative-order through a simple rewrite, included as a comment.

```
(define-syntax beta1
  (syntax-rules (lambda)
    ((beta1 (lambda (Formal) Body)       SK FK)
     (beta1 Body
       (beta1-lambda-k SK Formal)
       FK))
    ((beta1 ((lambda (Formal) Body) Arg) SK FK)
     ;; for applicative-order
     ;; (beta1 Arg
     ;;   (beta1-applicative-k SK Formal Body)
     ;;   (lc-subst Arg Formal Body SK))
     (lc-subst Arg Formal Body SK))
    ((beta1 (Op Arg)                     SK FK)
     (beta1 Op
       (beta1-op-k SK Arg)
       (beta1 Arg
         (beta1-arg-k SK Op)
         FK)))
    ((beta1 X                            SK FK)
     (apply-syn-cont FK))))

(define-syntax beta1-lambda-k
  (syntax-rules ()
    ((beta1-lambda-k SK Formal   NewBody)
     (apply-syn-cont SK (lambda (Formal) NewBody)))))

(define-syntax beta1-applicative-k
  (syntax-rules ()
    ((beta1-applicative-k SK Formal Body   NewArg)
     (apply-syn-cont SK
       ((lambda (Formal) Body) NewArg)))))

(define-syntax beta1-op-k
  (syntax-rules ()
    ((beta1-op-k SK Arg   NewOp)
     (apply-syn-cont SK (NewOp Arg)))))

(define-syntax beta1-arg-k
  (syntax-rules ()
    ((beta1-arg-k SK Op   NewArg)
     (apply-syn-cont SK (Op NewArg)))))
```

The general beta reducer, then, simply applies **beta1** until there are no further reductions.

```
(define-syntax beta*
  (syntax-rules ()
    ((beta* Exp K)
     (beta1 Exp
       (beta*-k K)
       (apply-syn-cont K Exp)))))

(define-syntax beta*-k
  (syntax-rules ()
    ((beta*-cont K   NewExp)
     (beta* NewExp K))))
```

[1] There is a slight fudge here. If the final continuation is something like our `inspect-cont`, above, which quotes its result, then `temp` would not be an introduced variable in a binding form. It would be a literal symbol, and R^5RS does not require that literal symbols be fresh. Because we feel the main purpose of macros is to generate programs rather than data, we have allowed ourselves this inconsistency.

Thus, using `inspect-cont` to view our reductions, we see that.

```
> (beta* ((lambda (x) (x x)) (lambda (y) (y z)))
    (inspect-cont))
(the-values-are (z z))
```

This facility may be taken as a proof that Scheme's macros, which seem so constrained, are capable of performing arbitrary computations.

4.4 Disjoint Sums

A serious problem with writing Scheme macro packages in a straightforward fashion is handling non-local constraints. An example of this is a naïve implementation of a sum form—the kind of thing used to implement ML style sum-of-products datatypes. A Scheme implementation of sums might include a definition form:

```
(define-sum Sum-type Variant ...)
```

and a form used to dispatch on the variant of a particular sum:

```
(sum-case Sum-type Expression
  (Variant Expression)
  ...)
```

For example:

```
(define-sum Direction
  North South East West)

(define move-point
  (lambda (point dir)
    (let ((x (car point))
          (y (cdr point)))
      (sum-case Direction dir
        (North (cons x (+ y 1)))
        (South (cons x (- y 1)))
        (East (cons (+ x 1) y))
        (West (cons (- x 1) y))))))

(define move-point-north
  (lambda (point)
    (move-point point North)))
```

We can provide such a package without recourse to continuation-passing style by using two simple macros:

```
(define-syntax define-sum
  (syntax-rules ()
    ((define-sum Name Variant ...)
     (begin
       (define Variant (cons 'Variant ()))
       ...))))

(define-syntax sum-case
  (syntax-rules ()
    ((sum-case Name X (Variant-name Exp) ...)
     (let ((v X))
       (cond
         ((eqv? v Variant-name) Exp)
         ...)))))
```

This solution lacks security, however. We might expect that a `sum-case` expression expecting a variant of a sum S would not be *syntactically* valid if variants not in S were listed, if there were duplicate entries, or if not all of the variants of S were covered.

This means we need to somehow transfer information from the expansion of a `define-sum` form to the expansion of a `sum-case` form, but that information must only be seen through the expansion process. To transfer this information we have `define-sum` itself define a macro—in continuation-passing style—that simply passes the defined variant names to its continuation.

```
(define-syntax define-sum
  (syntax-rules ()
    ((define-sum Name Variant ...)
     (begin
       (define Variant (cons 'Variant ()))
       ...
       (define-syntax Name
         (syntax-rules ()
           ((_ K)
            (apply-syn-cont K (Variant ...)))))))))
```

The `sum-case` macro, then, can use the generated macro for `Name` to get the variant names we wish to match. If the `sum-case` form has coverage of all the cases without duplication, it simply expands into the `let` that binds the object and the `cond` that performs the run-time dispatch. Otherwise it expands into a non-expression.

```
(define-syntax sum-case
  (syntax-rules ()
    ((sum-case Name X (Variant-name Exp ...) ...)
     (Name (sum-case-k Name X
             ((Variant-name Exp ...) ...))))))

(define-syntax sum-case-k
  (syntax-rules ()
    ((sum-case-k Name X ((Variant-name Exp ...) ...)
       Real-variants)
     (check-coverage (Variant-name ...) Real-variants
       (let ((v X))
         (cond
           ((eqv? v Variant-name) Exp ...)
           ...))
       (syn-error
         (sum-case Name X
           (Variant-name Exp ...)
           ...))))))
```

The code for `check-coverage` uses some simple helpers, `syn-member?` and `syn-remove`.

```
(define-syntax check-coverage
  (syntax-rules ()
    ((check-coverage () () SK FK)
     (apply-syn-cont SK))
    ((check-coverage (Head Tail ...) Xs SK FK)
     (syn-member? Head Xs
       (syn-remove Head Xs
         (check-coverage-k (Tail ...) SK FK))
       FK))
    ((check-coverage () Xs SK FK)
     (apply-syn-cont FK))))

(define-syntax check-coverage-k
  (syntax-rules ()
    ((check-coverage-k (Tail ...) SK FK   NewXs)
     (check-coverage (Tail ...) NewXs SK FK))))

(define-syntax syn-member?
  (syntax-rules ()
    ((syn-member? X () SK FK)
     (apply-syn-cont FK))
    ((syn-member? X (Y Z ...) SK FK)
     (syn-equal? X Y SK
       (syn-member? X (Z ...) SK FK)))))
```

```
(define-syntax syn-remove
  (syntax-rules ()
    ((syn-remove X () K)
     (apply-syn-cont k ()))
    ((syn-remove X (Y Z ...) K)
     (syn-remove X (Z ...)
       (syn-remove-k X Y K)))))

(define-syntax syn-remove-k
  (syntax-rules ()
    ((syn-remove-k X Y K   NewZs)
     (syn-equal? X Y
       (apply-syn-cont K NewZs)
       (apply-syn-cont K (Y . NewZs))))))
```

Finally, we need to expand into something when we want to signal failure. That something should cause a failure at expansion time. Though we could expand into `(lambda 3)` or the like, it's more elegant to define our own keyword with no patterns.

```
(define-syntax syn-error
  (syntax-rules ()))
```

This makes the check for coverage completely within expansion time, thus allowing for much earlier detection of coverage errors.

5 Related Work

There have been several attempts at expanding the generality and expressiveness of special forms. These typically have approached the problem through removal of unnecessary features rather than redesign. Kent Pitman [7] recognized that macros obviated the need for FEXPRs and NLAMBDAs. As a result, macros emerged as a central focus of research. Eugene Kohlbecker [5, 6] recognized that quasiquotation, with its ties to the run-time system, was unnecessary to express most macros. In addition, his pioneering work on hygiene freed macro-writers from thinking about inadvertent variable capture. The Revised[5] Report on the Algorithmic Language Scheme [4] extended Kohlbecker's model for writing macros.

Dybvig, Friedman, and Haynes [2] took an alternate route by giving the macro-writer control over what function to use for a particular expansion. This expansion-passing style was expected to be used with a quasiquotation model of macros. While their notion of expanders corresponds to an expansion's continuation, this continuation was under the control of the run-time expansion machinery, and could not be used without access to the run time. This model of expansion violates the central premise of the symbolic character of our work.

In Paul Graham's book, "On Lisp" [3], there is a chapter on continuation-passing macros. These macros are specifically designed to give the Lisp user the expressiveness of `call-with-current-continuation`. This should not be confused with our notion of writing macros in continuation-passing style.

6 Conclusions

In this paper we have presented a powerful technique for writing programs in a functional language that happens to be the macro-specification language for Scheme. This technique, derived directly from continuation-passing style of writing procedures, gives us much more control over syntax expansion than direct-style macro definitions allow. We have used this control not only to show the computational power of the Scheme macro system—by implementing a beta reducer for a lambda calculus—but also to deal with the non-local constraints found in a fairly simple pair of macros intended to implement a sum package.

But this power came with a great cost in clarity and expressiveness. Continuation-passing style was necessary because `syntax-rules` lacks control mechanisms apart from pattern matching. Due to hygiene, continuations could not be represented in a convenient form: instead, they had to be represented as a sort of records, with the description their behavior separated from their loci of creation.

So one thing this exercise should do is validate the decision of many Scheme implementors to provide either lower-level macro systems or extensions to the `syntax-rules` system. Until there is a consensus on the shape of such a more direct system, however, writing macros in continuation-passing style has one powerful benefit: for those Schemes that adhere to the the Revised[5] Report, it is a portable way of creating complex Scheme extensions.

7 Acknowledgements

We gratefully acknowledge Jonathan Sobel and Kevin Millikin for critical comments on and enthusiasm for earlier drafts. In addition, we are grateful to Mitch Wand for his spontaneous encouragement.

References

[1] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, Inc, second edition, 1996.

[2] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, June 1988.

[3] Paul Graham. *On Lisp*. Prentice Hall, 1994.

[4] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised[5] report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[5] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. Indiana University, 1986.

[6] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, page 77. ACM Press, 1987.

[7] Kent M. Pitman. Special forms in LISP. In *ACM Conference on LISP and Functional Programming, Stanford, California*, pages 179–187, Stanford, CA, August 1980. ACM Press.