Rewriting Macros on the Fly

A Modular Approach to Administrative Reduction During Expansion

Paul Downen
University of Massachusetts Lowell
Lowell, USA
Paul_Downen@uml.edu

Abstract

Administrative reductions are extra steps introduced by program transformations, usually as a form of bookkeeping new information needed by the transformation. While this can keep transformations themselves simple and small, they have the unfortunate effect of making the programs they produce more costly in terms of run-time resources and—worse still—cognitive resources used by humans trying to read and debug their code. This paper introduces a general-purpose technique for eliminating administrative expressions in macros using localized rewriting rules. The result is a system of writing and rewriting compositional macros (both in the sense that the macros are defined as compositional functions, and that several different macros can be combined without administrative reductions) that can be extended by the programmer with new functionality.

CCS Concepts: • **Software and its engineering** → **Macro languages**; • **Theory of computation** → *Rewrite systems.*

Keywords: Racket, Macro Expansion, Code Transformation, Administrative Reductions, Continuation-Passing Style

ACM Reference Format:
Paul Downen. 2025. Rewriting Macros on the Fly: A Modular Approach to Administrative Reduction During Expansion. In *Proceedings of the 26th ACM SIGPLAN International Workshop on Scheme and Functional Programming (Scheme '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759537.3762693

1 Introduction

Many program transformations are plagued by *administrative reductions*: additional bookkeeping steps that are left over in the new code produced by the transformation, but were not present in the original program. For example, consider how one might convert the application $(f\ (g\ x))$ into a standard *continuation-passing style* (CPS), which introduces an additional parameter representing the evaluation context as a function that gets passed to every call. Manually unfolding the transformation $[\![f\ (f\ x)]\!]$ by hand, an experienced CPS expert might produce the following code:

$$[\![f\ (g\ x)]\!] = \lambda k.\ g\ x\ (\lambda y.\ f\ y\ k)$$

The transformed code elucidates the following details of program execution:

- The new parameter k represents the evaluation context of the whole expression.
- The inner call $g\ x$ happens first, and its previously anonymous result is now given an explicit name (y) in the following continuation.
- After $g\ x$ returns a value y—*i.e.,* passes a value to its continuation $(\lambda y \dots)$—the next call is to $f\ y$.
- The return value of $f\ y$—*i.e.,* the value $f\ y$ passes to k—is the final value returned by the whole expression.

This example of CPS-transformed code for $(f\ (g\ x))$ is as simplified as possible. Formally speaking, it is a *normal form*: no possible reductions (*i.e.,* β reductions in the λ-calculus) can apply without inlining definitions for functions f or g. However, it was produced by a human with the goal of avoiding any unnecessary steps. Most definitions of CPS transformations will not reach this form right away because they are defined *compositionally*: each construct of the programming language (*e.g.,* application, function abstraction, *etc.*) is always handled in the same way no matter the sub-expressions they are made of. This forces a "worst-case scenario" attitude in the transformation process that is often unnecessary. For this example, a typical compositional CPS transformation would begin like so:

$$[\![f\ (g\ x)]\!] = \lambda k.\ (\lambda k'.\ k'\ f)\ (\lambda f'.\ [\![g\ x]\!]\ (\lambda y.\ f'\ y\ k))$$

The first function expression (f) *might* perform some computational steps, so it is given a continuation $(\lambda f'.\ \dots)$ explaining what to do when it is done. However, f is already done, and so the translation of f immediately passes it to that continuation as $[\![f]\!] = \lambda k'.\ k'\ f$. This application $(\lambda k'.\ \dots)\ (\lambda f'.\ \dots)$ is an administrative reduction, and is undesirable for several reasons. First, it represents an extra step at run-time (or at best optimized by a sufficiently advanced implementation). Perhaps worse, administrative steps quickly build up in even modest examples to make the transformed code significantly larger and more difficult to understand.

ACM ISBN 979-8-4007-2162-5/2025/10
https://doi.org/10.1145/3759537.3762693

Administrative reductions are bad enough in theory, but worse in practice, such as in macro expansions in a language like Scheme or Racket. When developing a new macro transformation, the ability to expand and read the code is a vital asset for debugging, and a pile-up of administrative reductions can make this task nearly impossible. An abstract setting like the λ-calculus affords us the ability to hand wave the problem away by marking some λs introduced by a transformation as administrative and then rewriting the term into the normal form where they are eliminated if possible [15]. However, if we were to write the same transformation as a Scheme macro, there is no general "reduce only these lambdas and ignore others" procedure available out of the box during macro expansion. Alternatively, we take efforts to avoid introducing them in the first place during the translation process [5]. However, this makes for monolithic macros that are difficult to change or extend with new features, some of which might be defined by the programmer.

This paper is about a general technique for developing Scheme-like macros that are compositional, avoid administrative reductions, and are free from the above compromises. The key idea is that there is a brief moment in time during macro expansion—right before the final form is set into stone—that administrative reductions become visible and can be rewritten away. Inserting rewriting rules in just that moment can turn an otherwise administrative-full transformation into an administrative-free one without compromising the rest of the code. This can be used to combine the results from several independent transformations without having to fuse them together into a big monolithic macro. Better yet, because all that rewriting logic is localized into certain key places, it can be exposed as an API that the programmer can add new rewriting rules to incorporate their own code into the administrative-reduction process.

More specifically, we begin with a review of the common approaches in sections 2 and 3 and give our technical contributions in sections 4 to 6 as follows:

- Section 2 reviews common naïve and one-pass CPS transformations on the λ-calculus with different amounts of administrative reduction, and how they can be encoded as Racket macros.
- Section 3 reviews an alternative approach to CPS based on the continuation monad, with the advantage of more modularity and features but at the cost of producing code with many administrative reductions.
- Section 4 introduces the new technique of this paper. It shows how the monad-based macros can be enhanced with rewriting rules to eliminate administrative steps at macro-expansion time, producing administrative-normal forms for select continuation operations.
- Section 5 generalizes the technique to accommodate arbitrary user-defined rewriting rules, which allows for adding new macros and programming features in client code that can participate in the elimination of the administrative reductions.
- Section 6 evaluates the impact on the size and complexity of expanded code. This includes a larger case study on a Racket library for copattern matching [9], which has an average 55% reduction in number of tokens and 61% depth in the syntax tree of expanded code compared to a naïve implementation that doesn't rewrite administrative steps.

The code examples and collected data presented here can be found at https://github.com/pdowen/rewrite-macros.

2 Administrative Reductions of CPS Transformation

2.1 A naïve transformation

An early example of a CPS transformation is given by Plotkin [13] for translating the call-by-value λ-calculus:

$$\begin{aligned} C[\![x]\!] &= \lambda k.\ k\ x \\ C[\![\lambda x.M]\!] &= \lambda k.\ k\ \lambda x.C[\![M]\!] \\ C[\![M\ N]\!] &= \lambda k.\ C[\![M]\!]\ \lambda f.\ C[\![N]\!]\ \lambda x.\ f\ x\ k \end{aligned}$$

Value expressions—like variables x and functions $\lambda x.M$—need to be passed to the continuation k they are given, whereas applications—$M\ N$—need to evaluate the two sides M and N before the function can be called.

This theoretical program transformation can be made real in terms of a macro in Racket written in terms of `syntax-rules`:

```
(define-syntax cps
  (syntax-rules (λ)
    ;; Make sure to handle lambdas first
    [(cps (λ(x) body))
     (λ(k) (k (λ(x) (cps body))))]
    ;; Other binary lists are applications
    [(cps (fun arg))
     (λ(k) ((cps fun)
            (λ(f) ((cps arg)
                   (λ(x) ((f x) k))))))]
    ;; Otherwise, assume it is an identifier
    [(cps x)
     (λ(k) (k x))]))
```

The `cps` macro needs to handle the special lambda form `(λ(x) body)` which begins specifically with the symbol λ followed by a single parameter `x` (in a list) and then the `body` expression that calculates the result. The first parameter to `syntax-rules` lists λ as a literal identifier—like a keyword—not to be confused with a variable, so that the first syntax rule only matches on lists that begin with exactly λ. After that, the `cps` macro has rules for handling single-argument applications `(fun arg)` and variables.[1]

[1]Note that the variable `x` in the syntax pattern `(cps x)` could actually match with *any* expression not already matched by the above two rules. However, assuming that `cps` is only given expressions that represent this simple λ-calculus—only `(λ(x) expr)` or `(expr expr)` or `identifier`—then the only remaining case is an identifier.

Trying this new `cps` macro, we can transform the (curried) function that applies its first argument twice to the second:

```
(define ex1
  (cps (λ(f) (λ(x) (f (f x))))))
```

To test out that the transformation is correct, we need to iteratively run the example code, apply the function value it returns, and then run the body again until the final result is returned. We can set up this infrastructure to `run` a CPS program by passing the `identity` function as the initial continuation, and inject a pure function as a CPS value which returns its result:

```
(define (run m) (m identity))

(define (pure-fun f)
  (λ(x) (λ(k) (k (f x)))))
```

Using these, one possible way to run the example is to pass the pure function representing $\lambda x.\, 2*(1+x)$ and the number 9, running each step of CPS along the way like so:

```
(define (run-example ex-cpsd)
  (run
   ((run
     ((run ex-cpsd)
      (pure-fun (λ(x) (* 2 (+ 1 x))))))
    9)))
```

The twofold function application is $2*(1+(2*(1+9))) = 42$, which is exactly what `(run-example ex1)` returns.

However, a curious programmer might want to understand how `cps` works by examining the expanded form of the expression. Peeking inside, they would see:

```
(define ex1*
  (λ(k)
    (k (λ(f)
         (λ(k) (k (λ(x)
                    (λ(k) ((λ(k) (k f))
                           (λ(f1) ((λ(k)
                                     ((λ(k) (k f))
                                      (λ(f2)
                                        ((λ(k) (k x))
                                         (λ(y)
                                           ((f2 y)
                                                k))))))
                                   (λ(z) ((f1 z) k))
                                  )))))))))))
```

Unacceptable!

2.2 Plotkin's "colon" translation

The naïve CPS transformation introduces far too many steps that have little relation to what the original code is trying to do. Rather than trying to optimize the output of the macro—somehow traversing and rewriting the expression it returns into a better form—we can avoid creating these administrative steps in the first place during the transformation. One way to do so has been dubbed the "colon" translation due to Plotkin's original notation [13]. The main idea is to define a new translation C_a as an optimization of the previous C by fusing it together with the application of the continuation:

$$C[\![M]\!]\ k =_\beta C_a[\![M]\!]\,[k]$$

where the transformation $C_a[\![M]\!]\,[k]$ takes *two* parameters—both the expression to transform as well as the continuation it is to be applied to—as such it avoids the creation of so many abstractions $(\lambda k.\ \dots)$ at each step. This simplifying translation can be written as:

$$\begin{aligned}
C_a[\![x]\!]\,[k] &= k\ x \\
C_a[\![\lambda x.M]\!]\,[k] &= k\ \lambda x. C_a[\![M]\!] \\
C_a[\![M\ N]\!]\,[k] &= C_a[\![M]\!]\,[\lambda f.\, C_a[\![N]\!]\,[\lambda x.\, f\ x\ k]] \\
C_a[\![M]\!] &= \lambda k.\, C_a[\![M]\!]\,[k]
\end{aligned}$$

The last line is an auxiliary one-parameter definition of how to handle cases where the continuation is not explicitly given—by intuitively η-expanding as $(\lambda k.\ C_a[\![M]\!]\ k)$ and then tucking the new continuation k deep inside the translation of M where it is needed—and is only used inside the body of a source-program λ-abstraction.

As before, this transformation can be encoded as a Racket macro that now takes either two arguments (corresponding to the "applied" form $C_a[\![M]\!]\,[k]$) or one (corresponding to the "unapplied" $C[\![M]\!]$).

```
(define-syntax cps-apply
  (syntax-rules (λ)
    [(cps-apply (λ(x) body) k)
     (k (λ(x) (cps-apply body)))]
    [(cps-apply (fun arg) k)
     (cps-apply fun
      (λ(f) (cps-apply arg
             (λ(x) ((f x) k)))))]
    [(cps-apply val k)
     (k val)]
    [(cps-apply expr)
     (λ(k) (cps-apply expr k))]))
```

Expanding this new macro on the same example

```
(define ex1-ap
  (cps-apply (λ(f) (λ(x) (f (f x))))))
```

leads to the following transformed code

```
(define ex1-ap*
  (λ(k)
    (k (λ(f)
         (λ(k) (k (λ(x)
                    (λ(k) ((λ(f1)
                             ((λ(f2)
                                ((λ(x1)
                                   ((f2 x1)
                                    (λ(y)
                                      ((f1 y) k)))) x))
                              f))
                           f)))))))))
```

Slightly better, but not good enough yet. There are still applications of continuations to values, such as `((λ(x1) ...) x)` that shouldn't be in the expanded code.

2.3 Fully administrative-normal translation

To squeeze out the last few administrative reductions, we need to be careful to *not* introduce a new continuation abstraction if it will be given to a value form, which intends to immediately apply it in the next step. Avoiding these administrative steps is a little trickier, as it requires us to break compositionality slightly and "look ahead" to the next forms to see if they look like syntactic values: either a variable x or a function $\lambda x.M$. These values are handled by a specialized translation of values $C_v[\![V]\!]$ that knows how to descend into the body of a function, and the main transformation of computations $C_c[\![M]\!]\,[k]$ will now check when sub-expressions are values or not. Theoretically, this new transformation is:

$$
\begin{aligned}
C_v[\![x]\!] &= x \\
C_v[\![\lambda x.M]\!] &= \lambda x.C_c[\![M]\!] \\[1ex]
C_c[\![V]\!]\,[k] &= k\; C_v[\![V]\!] \\
C_c[\![V\;W]\!]\,[k] &= C_v[\![V]\!]\; C_v[\![W]\!]\; k \\
C_c[\![V\;N]\!]\,[k] &= C_c[\![N]\!]\,[\lambda x.\, C_c[\![V\;x]\!]\,[k]] && (N \notin \mathit{Value}) \\
C_c[\![M\;N]\!]\,[k] &= C_c[\![M]\!]\,[\lambda f.\, C_c[\![f\;N]\!]\,[k]] && (M, N \notin \mathit{Value}) \\[1ex]
C_c[\![M]\!] &= \lambda k.\, C_c[\![M]\!]\,[k]
\end{aligned}
$$

And in practice, the corresponding Racket macro is given in figure 1. Note that we now have to switch from simple `syntax-rules` macros to a more complex `syntax-case` definition to use guards (*a.k.a fenders*) as side conditions to check if a rule should apply before committing. For example, the first rule of `cps-comp` checks if the given expression has the form of a syntactic value (as defined by the `syntactic-value?` predicate) before applying. Similar checks are done when translating an application for the different cases of applying two values, two non-values, or a mix.

While the definition of our CPS macro is now significantly less simple than what we started with, its output is simpler. Revising, the same example

```
(define ex1-ad
  (cps-comp (λ(f) (λ(x) (f (f x))))))
```

now expands to

```
(define ex1-ad*
  (λ(k) (k (λ(f)
            (λ(k) (k (λ(x)
                       (λ(k) ((f x)
                              (λ(y) ((f y) k))
                             )))))))))
```

Finally! This is the simplest form of CPS for the example code, with no more internal redexes left over.

3 Modular, Monadic CPS Macros

So we can now eliminate the administrative steps of CPS at macro-expansion time. However, the approach seen so far is quite monolithic. The `cps` macro tries to transform the *entire* expression in one shot, or bust. This rules out nested sub-expressions that are handled outside of `cps`. The monolithic nature also makes the `cps` macro harder to extend with new features—for example, multi-argument functions and calls are not even supported—which limits the CPS sub-language.

```
(define-for-syntax (syntactic-value? stx)
  (syntax-case stx (λ)
    [(λ(x) body) #t]
    [x (identifier? #'x)]))

(define-syntax (cps-value stx)
  (syntax-case stx (λ)
    [(cps-value (λ(x) body))
     #'(λ(x) (cps-comp body))]
    [(cps-value x)
     (identifier? #'x)
     #'x]))

(define-syntax (cps-comp stx)
  (syntax-case stx ()
    [(cps-comp val k)
     (syntactic-value? #'val)
     #'(k (cps-value val))]
    [(cps-comp (fun arg) k)
     (and (syntactic-value? #'fun)
          (syntactic-value? #'arg))
     #'(((cps-value fun) (cps-value arg)) k)]
    [(cps-comp (fun arg) k)
     (syntactic-value? #'fun)
     #'(cps-comp arg
                 (λ(x) (cps-comp (fun x) k)))]
    [(cps-comp (fun arg) k)
     #'(cps-comp fun
                 (λ(f) (cps-comp (f arg) k)))]
    [(cps-comp expr)
     #'(λ(k) (cps-comp expr k))]))
```

Figure 1. An administrative-normal CPS macro.

Instead of pushing this monolithic approach further, we switch to a more modular style based on monadic combinators: rather than being forced to transform a whole expression at once, the expression will be built from smaller, basic operations that combine values together to make the whole.

A stylistic choice we make now—which will become *absolutely crucial* in the next section 4—is to treat the values representing monadic computations as black boxes. The only way we will actually inspect computation values is with the operation (`after` comp cont) which says that *after* the computation comp is done, it should finish with the continuation cont. Continuations, too, will be treated as black boxes, only usable with the operation (`resume` cont val ...) which says to resume the continuation cont with the (multiple) values as inputs. Of course, in reality, both computations and continuations are just function values, and so `after` and `resume` can be defined like so:

```
(define (after comp cont) (comp cont))

(define-syntax resume
  (syntax-rules ()
    [(resume cont val ...)
     (cont val ...)]
    [(resume cont val ... . rest)
     (apply cont val ... rest)]))
```

where `resume` can take an optional rest parameter—a value containing a list of the remaining arguments—at the end of the given arguments, as the syntactic counterpart to the function abstraction form (λ(x ... . rest) body). To foreshadow, their use, the top-level run function is now written:

```
(define (run m) (after m identity))
```

3.1 Monadic operations

Every good monadic library starts with the primitive monadic combinators return and bind that glue everything together. The (return val) function says how to make a computation that just produces val immediately with no side effects, and (bind m f) says how to pass the value returned by m as the parameter to function f, and continue running where m left off. The standard definitions for the continuation monad are:

```
(define (return val) (λ(k) (resume k val)))

(define (bind m f)
  (λ(k) (after m (λ(x) (after (f x) k)))))
```

A common pattern is to call bind directly with a lambda abstraction as the second argument, with the intention to "bind" the return value to the function parameter like a let-expression. To make examples clearer, we can canonize this pattern as a more pleasant macro:

```
(define-syntax-rule
  (let-val [name comp] body)
  (bind comp (λ(name) body)))
```

3.2 Returning lambdas and calling functions

We can now use the basic monadic glue to create shorthands that make it easier to write CPS code. For functions, two of the most common operations are to return a lambda value or to call an effectful function with the results of effectful argument computations. These two shorthands are given by these two simple syntax rules:

```
(define-syntax-rule
  (ret-λ params body)
  (return (λ params body)))

(define-syntax-rule
  (call fun . args)
  (λ(k) (after fun (call-cont () args k))))
```

where most of the heavy work is done by call-cont which builds the continuation for calling a function. Its general form is (call-cont (fun val ...)(comp ...)cont) where the first list (fun val ...) contains the prepared call so far where the function (fun) and each argument (val ...) have already been evaluated. The second list (comp ...) contains the computations that will produce the remaining arguments. Finally, cont gives the continuation where the call should return to. The call-cont macro is defined like so:

```
(define-syntax call-cont
  (syntax-rules ()
    [(call-cont () ms k)
     (λ(f) (call-cxt (f) ms k))]
    [(call-cont (f arg ...) ms k)
     (λ(x) (call-cxt (f arg ... x) ms k))]))

(define-syntax call-cxt
  (syntax-rules ()
    [(call-cxt (f x ...) (m . ms) k)
     (after m (call-cont (f x ...) ms k))]
    [(call-cxt (f x ...) () k)
     (after (f x ...) k)]
    [(call-cxt (f x ...) rest k)
     (after rest
            (call-rest-cont (f x ...) k))]))

(define-syntax-rule
  (call-rest-cont (fun arg ...) k)
  (λ xs (after (apply fun arg ... xs) k)))
```

This uses the helper macros call-cxt for building the next calling context and call-rest-cont for supporting "rest" arguments as in (call fun arg ... . rest), similar to resume.

3.3 Automatic CPS transformation

With these shortcuts in hand, it becomes much easier to write a cps transformation that supports more language features like multi-argument functions. The monad-based cps macro is defined based on ret-λ, call, and return as follows:

```
(define-syntax cps
  (syntax-rules (λ return)
    [(cps (λ params body))
     (ret-λ params (cps body))]
    [(cps (fun arg ...))
     (call (cps fun) (cps arg) ...)]
    [(cps (return expr))
     (return expr)]
    [(cps other)
     (return other)]))
```

In addition to supporting multi-argument lambdas and calls, this cps macro has another feature: it recognizes the special form (return expr) which explicitly signals that the transformation should stop, no matter what expr looks like.

We can now encode our example $\lambda f.\lambda x.\, f\ (f\ x)$ in three different ways: (ex1) directly using return and bind, (ex2) using the shorthands ret-λ and let-val, or using cps.

```
(define ex1
  (return
   (λ(f)
     (return
      (λ(x)
        (bind (f x) (λ(y) (f y))))))))

(define ex2
  (ret-λ(f)
    (ret-λ(x)
      (let-val [y (f x)]
        (f y)))))
```

```
(define ex3
  (cps (λ(f) (λ(x) (f (f x))))))
```

While the new suite of macros is more modular, their output leaves a lot to be desired. The more explicit `ex2` expands exactly to `ex1`, but `ex3` is less fortunate, expanding to:

```
(define ex3*
  (return
   (λ(f)
     (return
      (λ(x)
        (λ(k) ((return f)
               (λ(f1) ((λ(k) ((return f)
                              (λ(f2)
                                ((return x)
                                 (λ(x1) ((f2 x1) k))))))
                       (λ(y) ((f1 y) k)))))))))))
```

3.4 Chains of commands

One common way to make monadic code more understandable is to organize it into chains of commands, similar to statements in imperative languages. The trick is that the command chain is actually nested one after the other, so that each line can bind variables that are in scope for the following ones. We can easily define this notation as a macro for *any* operations that take the "next step" as their final argument like `(let-val [name comp] next)`:

```
(define-syntax chain
  (syntax-rules ()
    [(chain (op arg ...) cmd1 cmd ...)
     (op arg ... (chain cmd1 cmd ...))]
    [(chain end)
     end]))
```

Because `chain` does not discriminate which operations are being used, it works just as well for stringing together sequences of the monadic `let-val`, the native `let` or `letrec`. For example, we can simplify the code for running examples `ex1`, `ex2`, and `ex3` into a more linear organization like so:

```
(define (run-example ex-comp)
  (run
   (chain
    (let-val [h ex-comp])
    (let ([double-inc
           (λ(x) (return (* 2 (+ 1 x))))]))
    (let-val [g (h double-inc)])
    (let-val [y (g 9)])
    (return y))))
```

This shows that all three examples are the same, yielding `42`.

3.5 Multi-argument functions

To test out that multi-argument functions work as intended, here is an automatically-derived example of a `cps` function taking three arguments—the first being a function that gets composed with itself like so:

```
(define ex4
  (cps (λ(f x y) (f x (f x y)))))
```

```
(run
 (chain
  (let-val [h ex4])
  (let ([mult (λ(x y) (return (* x y)))]))
  (h mult 2 3)))
```

The combined operations in the running context simplify to $2 * (2 * 3) = 12$, which is what the computation returns.

3.6 Multiple-value returns

Our syntax for `resume` foreshadowed the ability to pass multiple values to a continuation. We can make use of that functionality by extending the basic monadic glue with multi-value versions like so:

```
(define (returns . vals)
  (λ(k) (resume k . vals)))

(define (binds m f)
  (λ(k)
   (after m (λ xs (after (apply f xs) k)))))

(define-syntax-rule
  (let-vals [params comp] body)
  (bind comp (λ params body)))
```

That way, a single computation can return multiple values at the same time, like `ex5` below:

```
(define ex5
  (λ(f x) (returns x (f x) (f (f x)))))

(run
 (chain
  (let ([inc (λ(x) (+ 1 x))]))
  (let-vals [(x y z) (ex5 inc 10)])
  (return (list x y z))))
```

The running context produces the list `'(10 11 12)`.

4 Rewriting Administrative Steps Across Macro Boundaries

The modularity of the monadic approach is hard to match with a single monolithic transformation. Not only can the programmer freely glue together their code with reusable, nestable parts, but they can also seamlessly add new operations and features after the fact in client code, like multiple-value returns and chaining commands. However, the cost is a grisly impact on the code produced by transformation, like the expanded form of `ex3*` which is far, far, from optimal for computers to run and humans to read. If only we could have both modularity and simple output.

We can! Although each macro, defined independently, is unaware of the others, there is a brief moment in time during expansion where the different macros are nested in the expression-to-be but before the final expression is set in stone. We can exploit that moment to look for unfortunate nestings and rewrite them into a more fortuitous form with less redundancy. For example, if we ever see

```
(after (return val) cont)
```

it would be better to rewrite the expression to skip the return:

```
(resume cont val)
```

Thankfully, the patterns we need to look for are quite predictable: they are all an elimination style form (like `after` or `resume`) followed by an introduction style form (like `return`, `bind`, `let-val`, *etc.*). Therefore, the only time we need to look ahead for potential rewrites is in the elimination forms that otherwise don't do much, consolidating the rewrite logic inside `after` and `resume`. On the introduction side, we then need to think about what kind of shortcuts are possible if they appear in one of those contexts, and produce the more direct expression instead.

4.1 Following monadic operations by a continuation

First, consider how we can simplify `return`. Using the rewrite shown above, if it appears in the context of an `after`, it should go directly to resume its given continuation. In code, we write a macro that defines the combination of the two:

```
(after (return x) k) = (after-return (x) k)
```

and can derive the definition of `return` in all other contexts from the special case by η-expansion like so:

```
(define (return val)
  (λ(k) (after-return (val) k)))

(define-syntax-rule
  (after-return (val) k)
  (resume k val))
```

Likewise, we can shortcut the `bind` operation if it appears in the context of an `after` to place the given continuation directly where it needs to go:

```
(after (bind m f) k) = (after-bind (m f) k)
```

and the general definition of `bind` used in other contexts is derived by η-expanding the special case as:

```
(define (bind m f)
  (λ(k) (after-bind (m f) k)))

(define-syntax after-bind
  (syntax-rules (λ)
    [(after-bind (m (λ(x) body)) k)
     (after (let-val [x m] body) k)]
    [(after-bind (m f) k)
     (after m (λ(x) (after (f x) k)))]))
```

Note that `bind`, in a way, is *both* introducing a CPS computation as well as eliminating a function argument. Therefore, we can give an additional special case that simplifies the common case `(bind m (λ(x)body))` directly to a let-expression `(let-val [x m] body)`, defined below.

Of all the glue, the trickiest to get right is `let-val`. When in the context of `after`, we would ideally like to avoid binding the continuation at all, tucking it inside the implied continuation given to the computation like so:

```
(after (let-val [x comp] body) cont)
= (after comp (λ(x) (after body cont)))
```

This is perfectly fine in theory, where we assume all code transformations are performed up to valid α-renaming, so that the binding of `x` never captures free variables in `cont`. But even the hygiene checks of the macro system do not prevent this capture! To correctly perform this code motion, we have to check manually that it is safe to do—if not, we fall back to an explicit let-binding that avoids capture:

```
(define-syntax-rule
  (let-val [id comp] body)
  (λ(k) (after-let-val ([id comp] body) k)))

(define-syntax (after-let-val stx)
  (syntax-case stx ()
    [(after-let-val ([id m] bd) cont)
     (noncapture? #'id #'cont)
     #'(after m (λ(id) (after bd cont)))]
    [(after-let-val ([id m] bd) cont)
     #'(let ([k cont])
         (after m (λ(id) (after bd k))))]))
```

As a confirmation, we can check that the apparent static binding of `y` in the following example

```
(run
 (let ([y 1])
   (let-val [z (let-val [y (return 2)]
                 (return y))]
     (return (list y z)))))
```

produces the expected result `'(1 2)`, instead of `'(2 2)`.

It is always safe to fall back to the explicit let-binding, though it does increase the final code size. So there's a choice on how thoroughly we want to check that the continuation is safe from capture. A simple shallow check, that is fast to perform, is to allow for simple atomic literals (numbers, strings, *etc.*), or a single identifier expression which is *not* among the parameters being bound. This is enough to simplify many examples, and is given as:

```
(define-for-syntax (atomic-literal? expr)
  (or (number? expr)
      (boolean? expr)
      (string? expr)))

(define-for-syntax
    (capture-identifier? params expr)
  (syntax-case params ()
    [id (identifier? #'id)
     (bound-identifier=? #'id expr)]
    [() #f]
    [(id . params) (identifier? #'id)
     (or (bound-identifier=? #'id expr)
         (capture-identifier? #'params
           expr))]))

(define-for-syntax (noncapture? params expr)
  (or (atomic-literal? (syntax->datum expr))
      (and (identifier? expr)
           (not (capture-identifier? params
                                     expr)))))
```

4.2 Eliminating steps in lambdas and calls

Continuing on, we repeat this same pattern for all the other combinators: define the special case where they are immediately eliminated, and derive the general case from it via expansion if necessary. For returning a lambda, we have:

```
(define-syntax-rule
  (ret-λ params body)
  (return (λ params body)))

(define-syntax-rule
  (after-ret-λ (params body) cont)
  (resume cont (λ params body)))
```

Similarly, calling an effectful function becomes:

```
(define-syntax-rule
  (call fun . args)
  (λ(k) (after-call (fun . args) k)))

(define-syntax-rule
  (after-call (fun . args) cont)
  (after fun (call-cont () args cont)))
```

where `call-cont` is unchanged.

4.3 Rewriting macros to eliminate redexes

Now is the time where have to pay the piper. All of the rewriting we *intended* to do above has to happen somewhere, and that is in the elimination operations. Our poor, simple `after` function now houses all the rewrite rules for how to shortcut through sub-expressions. It looks ahead one step to see if the computation is formed by one of the introductions we know about, and replaces it with the combined form if so. Otherwise, it can always fall back to calling the computation like a function if no special rewrite applies.

```
(define-syntax after
  (syntax-rules
      (return bind let-val ret-λ call cps)
    [(after (return val) cont)
     (after-return (val) cont)]
    [(after (bind comp fun) cont)
     (after-bind (comp fun) cont)]
    [(after (let-val binding body) cont)
     (after-let-val (binding body) cont)]
    [(after (ret-λ params body) cont)
     (after-ret-λ (params body) cont)]
    [(after (call fun . args) cont)
     (after-call (fun . args) cont)]
    [(after (cps expr) cont)
     (after-cps (expr) cont)]
    [(after comp cont)
     (comp cont)]))
```

The code for `resume` is similar. Here, we can notice when the given continuation comes from `call-cont` or `call-rest-cont`. Instead of forming the continuation as a lambda that is applied to values, `resume` will shortcut to the next step that the body of that function produces like so:

```
(define-syntax resume
  (syntax-rules (call-cont call-rest-cont)
    [(resume (call-cont () ms k) f)
     (call-cxt (f) ms k)]
    [(resume (call-cont (f x ...) ms k) val)
     (call-cxt (f x ... val) ms k)]
    [(resume (call-rest-cont (f x ...) k)
             y ...)
     (after (f x ... y ...) k)]
    [(resume (call-rest-cont (f x ...) k)
             y ... . zs)
     (after (apply f x ... y ... zs) k)]
    [(resume k val ...)
     (k val ...)]
    [(resume k val ... . rest)
     (apply k val ... rest)]))
```

4.4 Automatic, administrative-free CPS transform

Now, we get to the automatic `cps` transformation. This time, it looks almost the same as the naïve version, except that the main transformation takes the continuation that comes after, which is just threaded along with the `after` operation.

```
(define-syntax-rule
  (cps expr)
  (λ(k) (after-cps (expr) k)))

(define-syntax after-cps
  (syntax-rules (λ return)
    [(after-cps ((λ params body)) cont)
     (after (ret-λ params (cps body)) cont)]
    [(after-cps ((fun arg ...)) cont)
     (after (call (cps fun) (cps arg) ...)
        cont)]
    [(after-cps ((return expr)) cont)
     (after (return expr) cont)]
    [(after-cps (other) cont)
     (after (return other) cont)]))
```

Even though this `cps` macro has no logic for avoiding administrative reductions, this all gets handled already by `after`. For example, the expansion of ex2 does not change much, only the definition of `bind` has been inlined.

```
(define ex2*
  (return
   (λ(f)
    (return
     (λ(x) (λ(k) ((f x)
                  (λ(y) ((f y) k)))))))))
```

However, the expansion of ex3 below is vastly shorter, actually reaching the final administrative-normal form!

```
(define ex3*
  (λ(k) (k (λ(f)
            (λ(k) (k (λ(x)
                      (λ(k) ((f x)
                             (λ(y) ((f y)
                                 k)))))))))))
```

5 Extensible Macro Rewriting Rules

We've seen how to eliminate administrative reductions at expansion time by rewriting macros to fuse together elimination-introduction macro pairs into a more direct form. The cost of this methodology is:

1. Each introduction macro has two different forms: the special case where it is immediately eliminated, and a general form for all other contexts that is usually derived automatically from the special case.
2. Each elimination macro checks if the relevant sub-expression is an introduction form, and if so, rewrites it to the special case.

Part 1 amounts to a little extra work—typically just a wrapper that goes along with a definition—that is easy to combine with other macros since each one is defined independently of the others. However, part 2 is the real pain point, where an elimination form like `after` has to include special cases for *every* other introduction macro in the system. Not only is this redundant and tedious—because the shortcut logic is essentially the same in every case—but it is not very extensible. In section 4 we included some special cases for eliminating administrative steps, but if we wanted to extend it with multi-value returns, we would have to change the definition of `after` by adding new rules for more special cases. Yuck. There should be a way to update `after` without changing its code after the fact when new functionality is added, so that it gets included in the rewriting machinery.

5.1 Generalized rewriting macros

The saving grace to the rewriting technique comes from the fact that each rewriting rule is phrased in a highly similar way. Because we have defined the special cases as their own macros, each rewrite replaces the general operation `op` with its specialized `after-op` in an expression like so:

```
(after (op x ...) k) = (after-op (x ...) k)
```

We should factor this pattern out once and for all!

To do so, we first need to collect some information about which introduction macros to look for and—when found—what to replace them with. This can be done through a simple lookup table (here represented as an association list between names of macros) available at expansion-time:

```
(define-for-syntax after-operation-rewrite
  (list [cons #'return   #'after-return]
        [cons #'bind     #'after-bind]
        [cons #'let-val  #'after-let-val]
        [cons #'ret-λ    #'after-ret-λ]
        [cons #'call     #'after-call]))
```

We can then use that table to determine (1) when the elimination-introduction expression pattern has been found, and (2) what shortcut expression should it be rewritten to. This leads to the following definition of `after` with a single rule that uses the lookup table to drive all the special-case rewrites:

```
(define-syntax (after stx)
  (syntax-case stx ()
    [(after (op . args) cont)
     (and (identifier? #'op)
          (assoc #'op
                 after-operation-rewrite
                 free-identifier=?))
     (with-syntax
         ([after-op
           (cdr (assoc
                 #'op
                 after-operation-rewrite
                 free-identifier=?))])
       #'(after-op args cont))]
    [(after comp cont)
     #'(comp cont)]))
```

Now, there are only two rules. The second rule is the usual default case where there is no shortcut, and the first rule handles every rewrite with the following steps:

1. Check for any expression where the computation given to `after` is an application `(op . args)`.
2. If `op` is an identifier, search the `after-operation-rewrite` lookup table to see if `op` has an entry.
3. If it is and it does, then lookup the new operation name it should be replaced with, called `after-op` locally, and change the expression to use the fused `(after-op args cont)` instead.

We should do the same reorganization with the other elimination form, `resume`, too. However, its definition looks different: the logic of how to rewrite each special case was written directly into the syntax rules of `resume`. That is no problem! We can just refactor the special-case logic into separate macros that spell out each rewrite:

```
(define-syntax resume-call-cont
  (syntax-rules ()
    [(resume-call-cont (() ms k) fun)
     (call-cxt (fun) ms k)]
    [(resume-call-cont ((f x ...) ms k) val)
     (call-cxt (f x ... val) ms k)]))

(define-syntax resume-call-rest-cont
  (syntax-rules ()
    [(resume-call-rest-cont ((f x ...) k)
                             y ...)
     (after (f x ... y ...) k)]
    [(resume-call-rest-cont ((f x ...) k)
                             y ... . zs)
     (after (apply f x ... y ... zs) k)]))
```

With these, we can then group all of `resume`'s rewrites into a general rule that uses the lookup table like before:

```
(define-for-syntax
    resume-continuation-rewrite
  (list [cons #'call-cont
      #'resume-call-cont]
        [cons #'call-rest-cont
           #'resume-call-rest-cont]))
```

```
(define-syntax (resume stx)
  (syntax-case stx ()
    [(resume (op . args) . vals)
     (and (identifier? #'op)
          (assoc #'op
                 resume-continuation-rewrite
                 free-identifier=?))
     (with-syntax
         ([resume-cont
           (cdr (assoc
                 #'op
                 resume-continuation-rewrite
                 free-identifier=?))])
       #'(resume-cont args . vals))]
    [(resume cont val ...)
     #'(cont val ...)]
    [(resume cont val ... . rest)
     #'(apply cont val ... rest)]))
```

Besides removing the redundancy in elimination macros and shortening their definition, this refined approach is far more extensible. Since every rewrite is controlled through a lookup table, we can easily update it in client code if we ever define new functionality that should be integrated into the simplification process. For example, notice that we forgot to include the automated `cps` macro in the rewrite table. No problem! It can be added afterwards through an update to the association list:

```
(begin-for-syntax
  (set! after-operation-rewrite
        (cons
         [cons #'cps #'after-cps]
         after-operation-rewrite)))
```

And now we get the same administrative-normal form of `ex3` as before. Updating the rewrite table lets us do more administrative-normal expansion of examples using other features—such as chains of commands or multi-value returns—that were previously omitted from the rewrite logic.

5.2 Chains of commands

Getting the continuation to tuck inside a `chain` of commands doesn't require much. In the specialization

```
(after (chain cmd ...) k) = (after-chain
    (cmd ...) k)
```

can be defined by just nesting each operation inside of `after`; if there is a special case for the given operation, then `after` will take care of it.

```
(define-syntax after-chain
  (syntax-rules ()
    [(after-chain ((op x ...) c1 c ...) k)
     (after (op x ... (chain c1 c ...)) k)]
    [(after-chain (end) k)
     (after end k)]))
```

While it may seem like `after-chain` isn't doing much compared to `chain`, it is important to make sure that `(after (chain cmd ...)k)` doesn't fall through to the default syntactic application `((chain cmd ...)k)`, which would prevent threading the continuation `k` through each command.

We should also make sure that the continuation can flow into plain let-bindings, which don't actively participate in continuation-passing style, like so:

```
(after (let ([x expr] ...) body) cont)
= (let ([x expr] ...) (after body cont))
```

This rewrite is captured by the following specialized macro combining the two steps, taking care to watch out for accidental capture similar to `after-let-val`:

```
(define-syntax (after-let stx)
  (syntax-case stx ()
    [(after-let (([x expr] ...) body) cont)
     (noncapture? #'(x ...) #'k)
     #'(let ([x expr] ...)
         (after body cont))]
    [(after-let (([x expr] ...) body) cont)
     #'(let ([x expr] ... [k cont])
         (after body k))]))
```

Now we can update the rewriting table with the new cases:

```
(begin-for-syntax
  (set! after-operation-rewrite
        (append
         (list [cons #'chain #'after-chain]
               [cons #'let   #'after-let])
         after-operation-rewrite)))
```

This is enough to reach an administrative-normal form for the `run-example*` code that mixes multiple `let`s and `let-val`s in a `chain`, which expands as follows:

```
(define (run-example* ex-comp)
  (run
   (λ(k)
     (ex-comp
      (λ(h)
        (let ([double-inc
               (λ(x) (return
                      (* 2 (+ 1 x))))])
          ((h double-inc)
           (λ(g) ((g 9)
                  (λ(y) (k y)))))))))))
```

If we have code that uses other native let-binding forms, like `letrec` or `let*`, those can be added rewrite table, too, on an as-needed basis.

5.3 Multiple-value returns

To simplify the expansion of code using multiple-value returns, we just need to define their shortcut steps similar to the single-value counterparts:

```
(define-syntax-rule
  (after-returns vals k)
  (resume k . vals))


(define (returns . vals)
  (λ(k) (after-returns vals k)))
```

```
(define-syntax after-binds
  (syntax-rules (λ)
    [(after-binds (m (λ params body)) k)
     (after (let-vals [params m] body) k)]
    [(after-binds (m f) k)
     (after m (λ xs (after (apply f xs)
         k)))]))

(define (binds m f)
  (λ(k) (after-binds (m f) k)))
```

As always, the trickiest expression to simplify is the multiple-value binding `let-vals`, which has to avoid capturing free variables in the continuation.

```
(define-syntax-rule
  (let-vals [ids m] bd)
  (λ(k) (after-let-vals ([ids m] bd) k)))

(define-syntax (after-let-vals stx)
  (syntax-case stx ()
    [(after-let-vals ([ids m] bd) cont)
     (noncapture? #'ids #'cont)
     #'(after m (λ ids (after bd cont)))]
    [(after-let-vals ([ids m] bd) cont)
     #'(let ([k cont])
         (after m (λ ids (after bd k))))]))
```

To use the rewriting logic, we just need to update the table

```
(begin-for-syntax
  (set!
   after-operation-rewrite
   (append
    (list [cons #'returns  #'after-returns]
          [cons #'binds    #'after-binds]
          [cons #'let-vals
              #'after-let-vals])
    after-operation-rewrite)))
```

so that the code for running example `ex5` will expand into the following administrative-normal form:

```
(run
 (let ([inc (λ (x) (+ 1 x))])
   (λ(k) ((ex5 inc 10)
          (λ(x y z) (k (list x y z)))))))
```

6 Case Study of Rewriting Copatterns

To see the impact of the simplifications done by rewriting macros, we can measure the change with respect to the naïve version. Rather than measuring run-time performance—which will depend heavily on the implementation used and how well it can optimize administrative reductions left in naïve transformation—we focus on static measures of expansion-time code size. Since this can be relevant to the programmer or macro-writer trying to understand and debug their code, we focus on measures that account for human concerns that impact how easy (or hard!) is to read code, including:

- *Height:* More deeply nested code is usually harder to properly read, so larger height can correspond to more confusing expressions. The height of a syntax tree is measured in the usual way, where the empty list is 0, non-empty lists are 1 plus the maximum height of all elements, and everything else is 1.
- *Atoms:* More individual parts create more information for the reader to digest, so code made of fewer atoms can be simpler. The total number of atoms in a syntax tree is measured by flattening the tree to a list, so that the number of atoms is the length of the list.
- *Tokens:* In reality, a human reader must process a combination of tree structure and the atomic leaves that make up the overall code. We measure the total number of tokens in a syntax tree as an approximation of these two factors. Each atom counts as 1 token. A list counts as 2 tokens (the two parentheses) and the sum of its elements' tokens. Dotted lists are similar, but count as 3 tokens (the two parentheses and one dot marking the tail) plus the sum of its parts. We also special-case quoting forms (`quote`, `quasiquote`, and `unquote`) which are typically printed as just one token.

Using these measures gives an objective way to evaluate the impact of macro-time administrative rewriting. To start, we could consider the examples seen here by comparing the sizes original source program the expansion of the naïve monadic macros in section 3, and the administrative-eliminating rewriting macros in section 5. On average, the administrative-reducing rewriting macros have similar height to the naïve ones, 8% fewer atoms, and 5% fewer tokens. However, these toy examples are so small, where the difference either way is usually minor, and not representative of serious code. This leads to mixed results with large savings in some examples (like `ex3` and `ex4`) and small losses elsewhere. A reasonable person might prefer the naïvely-expanded version of `ex2`—which looks exactly like the hand-written `ex1` using `return` and `bind`—to the "simplified" version that inlined the definition of `bind`.

To get a more realistic measurement of the impact of this technique—and how rewriting away administrative steps can change expanded code size—we need to test larger and more serious examples. For this, we applied the same technique of rewriting macros to create an alternativeimplementation of copatterns in Racket and Scheme based on [9]. The implementationcontains many example uses of the copattern-matching macros, including stream-processing operations, and several variations on arithmetic and algebraic expression evaluators. To measure the impact on code size, we compared the expanded forms generated by the naïve Racket library (`composable.rkt`) versus one that uses macro rewriting to skip administrative steps (`composable-inline.rkt`), whose results are shown in table 1.

It is no surprise that both expanded forms are always larger than the original source—the point of using a macro is usually to save on code. However, on the larger examples, the rewritten version is *always* smaller than the naïve version in every measure, averaging 61% shorter height, 60% fewer

	Source			Naïve			vs Source			Rewriting			vs Source			vs Naïve		
Example	H	A	T	H	A	T	H%	A%	T%	H	A	T	H%	A%	T%	H%	A%	T%
`streams`	7	214	432	32	1235	3267	*4.57*	*5.77*	*7.56*	17	630	1520	*2.43*	*2.94*	*3.52*	**0.53**	**0.51**	**0.47**
`arith`	7	68	140	23	284	758	*3.29*	*4.18*	*5.41*	14	142	346	*2.00*	*2.09*	*2.47*	**0.61**	**0.50**	**0.46**
`arith-ext`	8	147	287	23	390	994	*2.88*	*2.65*	*3.46*	14	240	556	*1.75*	*1.63*	*1.94*	**0.61**	**0.62**	**0.55**
`algebra`	8	65	131	28	201	533	*3.50*	*3.09*	*4.06*	19	152	390	*2.36*	*2.34*	*2.98*	**0.67**	**0.76**	**0.73**
`partial`	7	157	279	23	477	1181	*3.29*	*3.04*	*4.23*	14	294	646	*2.00*	*1.87*	*2.32*	**0.61**	**0.62**	**0.55**
Average	7	130	254	26	517	1347	*3.50*	*3.75*	*4.95*	16	292	692	*2.11*	*2.18*	*2.64*	**0.61**	**0.60**	**0.55**

Table 1. Code size of copattern-matching examples — source code before expansion, expanded naïve macros, and expanded rewriting macros — comparing maximum height (H) total number of atoms (A), and total number of tokens (T) in syntax trees. The examples include infinite streams and stream processing functions (`streams`), a small arithmetic expression evaluator (`arith`), compositional extensions of the arithmetic expression evaluator with new operations (`arith-ext`), an extended evaluator with variable expressions and environments (`algebra`), and partially evaluating around free variables (`partial`).

atoms, and 55% fewer overall tokens. In some of the more extreme cases, macro rewriting reduces code size by half or more; a significant reduction. And while not a quantitative measurement, the macro-rewritten output is often closer to human-written code in a qualitative sense. Whereas the naïve output produces extremely higher-order code (think: lambdas applied to lambdas), the macro-rewritten output more often simplifies away some levels of abstraction.

7 Related Work

There is ample previous work on how to eliminate administrative reductions in CPS transformations, *e.g.,* [1–4, 6–8, 10, 12–15]; for a more in-depth review of past work, see [5]. These can mainly be thought of in two distinct styles. The first, like [15] is based on a naïve translation that marks administrative functions, and uses a secondary rewriting process to eliminate them after. The second, like [1], organizes translation to never produce administrative reductions in the first place. Our approach is in between the two: translation will create administrative redexes, which then get rewritten in a post-hoc process *during* expansion. Alternatively, one could implement a full normalizer as a macro to run in a second phase after the main macro's expansion [11]. This could potentially give a more thorough amount of normalization—especially taking into consideration redexes introduced directly by the programmer—which is not covered by rewriting internal steps introduced by the macro-expansion process. However, in place of the small-step, local rewrites used here, this alternative requires a correct and complete implementation in the macro system of the target expression language—typically an extension of λ-calculus up to, and including, all of Scheme—which handles capture-avoiding substitution and traversal of the fully expanded expression.

In contrast to most previous work that focuses on the number of administrative steps reduced during expansion or the run-time performance of the transformed program, we focus here on the size and quality of the fully-expanded code because it was the original use-case for which the technique was developed. The first application of rewriting macros was done alongside the design of the copattern library in Scheme [9], which produces extremely higher-order code that is difficult to print and inspect directly. Instead, it became necessary to read and understand the macro-expanded code to debug the library while it was being developed. The technique for rewriting macros was indispensable for producing simpler, human-readable output in a real-world macro library built on top of many independent, composable macros. Only afterward was this technique applied to CPS transformations.

8 Conclusion

Crafting a good macro library has many tensions: simplicity of macro definitions versus the expressions they output; dynamic extensibility versus static predictability; a suite of small, modular components composed locally or a big monolithic macro to process the whole expression. This paper describes a method of rewriting macros—through an example based on well-known continuation-passing style transformations—that aims to combine the advantages of these two divergent styles, externally presenting a library of flexible, reusable, extensible parts, but internally taking advantage of whole-expression information to simplify code as a single cohesive unit. Not only does this give the the macro user a better experience of having more readable expanded forms made from bite-sized pieces, it also keeps the macro writer sane by keeping separate concerns separate in the transformation process. The general-purpose rewriting framework also creates clear entry points where the client can integrate their own extensions into this process without modifying the library's code. Applied to a larger-scale example on copattern-matching objects in Racket, this technique gives significant reductions in expanded code size, leading to more understandable and maintainable programs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2245516.

References

[1] Andrew W. Appel. 2007. *Compiling with Continuations*. Cambridge University Press, USA.

[2] Daniel Damian and Olivier Danvy. 2003. CPS transformation of flow information, part II: Administrative reductions. *Journal of Functional Programming* 13, 5 (2003), 925–933. doi:10.1017/S0956796803004702

[3] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 151–160. doi:10.1145/91556.91622

[4] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. doi:10.1017/S0960129500001535

[5] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. 2007. On one-pass CPS transformations. *Journal of Functional Programmingt* 17, 6 (2007), 793–812. doi:10.1017/S0956796807006387

[6] Olivier Danvy and Lasse R. Nielsen. 2001. A Higher-Order Colon Translation. In *Functional and Logic Programming*, Herbert Kuchen and Kazunori Ueda (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–91. doi:10.1007/3-540-44716-4_5

[7] Olivier Danvy and Lasse R. Nielsen. 2003. A first-order one-pass CPS transformation. *Theoretical Computer Science* 308, 1-3 (2003), 239–257. doi:10.1016/S0304-3975(02)00733-8

[8] Olivier Danvy and Lasse R. Nielsen. 2005. CPS transformation of beta-redexes. *Inform. Process. Lett.* 94, 5 (2005), 217–224. doi:10.1016/J.IPL.2005.02.002

[9] Paul Downen and Adriano Corbelino II. 2025. CoScheme: Compositional Copatterns in Scheme. In *Trends in Functional Programming*. Springer, Cham, 56 pages. doi:10.1007/978-3-031-99751-8_10

[10] Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. 1992. *Essentials of programming languages*. MIT Press, Cambridge, MA, USA.

[11] Oleg Kiselyov. 2004. Normal-order direct-style beta-evaluator with syntax-rules, and the repeated applications of call/cc. In: presentation at the Workshop "Daniel P. Friedman: A Celebration." December 4, 2004. Bloomington, IN. https://okmij.org/ftp/Scheme/callcc-calc-page.html

[12] Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. Back to Direct Style: Typed and Tight. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 104 (April 2023), 28 pages. doi:10.1145/3586056

[13] Gordon D. Plotkin. 1975. Call-By-Name, Call-By-Value and the λ-Calculus. *Theoretical Computer Science* 1 (1975), 125–159. Issue 2. doi:10.1016/0304-3975(75)90017-1

[14] Christian Queinnec and Kathleen Callaway. 1996. *Lisp in small pieces*. Cambridge University Press, USA.

[15] Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (1993), 289–360. doi:10.1007/BF01019462

Received 17 July 2025