Rewriting Macros on the Fly

A Modular Approach to Administrative Reduction During Expansion

Paul Downen
University of Massachusetts Lowell

Scheme — Thursday, October 16, 2025

CPS and Administrative Redexes

CPS in Theory

As an abstract code transformation

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

CPS in Practice

As an executable Scheme / Racket[1] macro!

```
(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))]))
```

[0]For this talk, everything applies equally well to both Racket and R[6]RS

The Ugly Reality of Expansion

A simple program

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

expands to ...

The Ugly Reality of Expansion

A simple program

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

expands to

```
(define ex
  (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))))))))))))
```

Who can read this???

The Problem With Administrative Redexes

Speed is not the only metric

- Slower, sure
  - More steps in expanded code $\implies$ more to do at run-time
  - Really relying on host to optimize code

The Problem With Administrative Redexes

Speed is not the only metric

- Slower, sure
  - More steps in expanded code $\implies$ more to do at run-time
  - Really relying on host to optimize code
- Also inscrutable to mere mortals
  - Large examples quickly become impenetrable to human understanding
  - For very higher-order code (like CPS), printing is less useful for debugging
  - Instead, looking at the expanded code may be best option for macro-developers

An Administrative-Reducing CPS Transform

C_v = translate value; C_c = translate computation

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

Dutifully Transcribing into Real Code

```
(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 () ... ))
```

An Administrative-Normal Output

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

expands to …

An Administrative-Normal Output

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

expands to

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

Not great, but reasonable! In fully β-normal form.

Modular, Monadic CPS Macros

The Story So Far

- Fine in this small example
- Style is monolithic, doesn't extend well
  - New language features case a blow-up in specialized cases
  - Can't combine with user-defined code to eliminate those administrative redexes, too
- A more modular presentation would be better
  - Easier to maintain for macro-writer
  - Easier to use for client programmer

A More Modular Macro Library


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

(define-syntax-rule
  (after comp cont)
  (comp cont))

(define-syntax-rule
  (resume cont val ...)
  (cont val ...))
```

A More Modular Macro Library

Monads to the rescue!

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

(define-syntax-rule
  (after comp cont)
  (comp cont))

(define-syntax-rule
  (resume cont val ...)
  (cont val ...))

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

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

Separating Concerns

Separate features $\Longrightarrow$ separate macros

- Easy to extend with more features
- For example,
  - Multi-argument functions
  - Multiple return values
  - Chaining commands like “do”-notation

Combining Macros

Makes it easier to write code

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

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

Combining Macros

Makes it easier to write code

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

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

(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))))
```

Recovering Big-Step CPS Transform

```
(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)]))
```

Recovering Big-Step CPS Transform

```
(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)]))

(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))))
```

Administrative Redexes Return

Same example

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

Same problem ...

Administrative Redexes Return

Same example

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

Same problem …

```
(define ex
  (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))))))))))
```

Oof…

Rewriting Steps Across Macro Boundaries

The Problem with Compositionality

Separate concerts are blind to each other

Separate macros/functions expand and run separately

```
(after (return val) cont)
= ((return val) cont)
= ((λ(k) (k val)) cont)
```

Instead, it would be more direct to shortcut directly past the β-reduction

```
(after (return val) cont)
= (resume cont val)
= (cont val)
```

But how can **after** “look ahead” to anticipate the next step?

Fusing Macro Pairs Into Shortcuts: after a return

One combination we want to look out for:

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

Fusing Macro Pairs Into Shortcuts: after a return

Get close to your neighbors

One combination we want to look out for:

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

The shortcut for skipping the administrative step:

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

Fusing Macro Pairs Into Shortcuts: after a return

Get close to your neighbors

One combination we want to look out for:

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

The shortcut for skipping the administrative step:

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

Easy to derive the general case when **return** is used in any other context as:

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

Fusing Macro Pairs Into Shortcuts: after a bind

Getting a little more fancy, looking an extra step ahead

Another combination we want to look out for:

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

Fusing Macro Pairs Into Shortcuts: after a bind

Getting a little more fancy, looking an extra step ahead

Another combination we want to look out for:

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

The shortcut for skipping the administrative step (even skipping the common β-step for λ!):

```
(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)))]))
```

Fusing Macro Pairs Into Shortcuts: after a bind

Getting a little more fancy, looking an extra step ahead

Another combination we want to look out for:

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

The shortcut for skipping the administrative step (even skipping the common β-step for λ!):

```
(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)))]))
```

Easy to derive the general case when **bind** is used in any other context as:

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

For a new macro (op . args) look out for shortcuts with

```
(after (op . args) cont)
```

if it expects a continuation. Or if it is a continuation,

```
(resume (op . args) val ...)
```

Then define the appropriate fused shortcut

```
(after-op args cont)
(resume-op args val ...)
```

and derive the general case from the fusion via η-expansion.

Rewriting Macros on the Fly

The whole trick that makes this possible

- Every rewrite we want starts with an elimination form **after** or **resume** around an introduction form (e.g., **return**, **bind**, **let–val**, etc.)
- While **after** or **resume** is expanding, it can look ahead arbitrarily deep into sub-expressions before deciding what code to write (e.g., see **after-bind**)
  - That is, macro expansion itself is done lazily
  - Inner macro calls don't expand until the outer one is completely finished
- Even though macros are written separately, the macro-expansion process can break the barrier and rewrite chains of macro calls into simpler forms
- All of the rewrite logic can be isolated into the elimination forms **after** and **resume**
- Only need to explain how they fuse with various intro forms

Automatic Optimization via Macro Rewrites

Same example of the "unoptimized" monadic **cps** macro

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

using rewriting **after** and **resume** expands to β-normal

```
(define ex
  (λ(k) (k (λ(f)
               (λ(k) (k (λ(x)
                            (λ(k) ((f x)
                                      (λ(y) ((f y) k)))))))))))
```

just like the specialized **cps-comp** macro before!

Extensible Macro Rewriting Rules

The Rewriting after Macro

Have to pay the piper at some point ...

```
(define-syntax after
  (syntax-rules
      (return bind let-val ...)
    ;; Special cases for rewrites
    [(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)]
     ...
    ;; Final general case when there's no rewrite to do
    [(after comp cont)
     (comp cont)]))
```

The Rewriting after Macro

Have to pay the piper at some point ...

```
(define-syntax after
  (syntax-rules
      (return bind let-val ...)
    ;; Special cases for rewrites
    [(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)]
     ...
    ;; Final general case when there's no rewrite to do
    [(after comp cont)
     (comp cont)]))
```

- Gets the job done... but is highly redundant
- Even worse, scales poorly! (a new clause for every new feature)

Writing Down the Table of Rewrites

- Because all of **after**'s rewrites have a similar form

```
(after (op . args) cont) = (after-op args cont)
```

- The only interesting information is the pair of op and after–op names

```
(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]))
```

The Macro Rewriting Engine

Capturing all rewrites with the same logic

- Can now summarize all the rewriting steps as one case (worst macro-magic of the talk)

```
(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)]))
```

- First case looks for operations to rewrite in the table, and does the replacement
- Second case doesn't do anything special

Extensible Macro Rewriting Rules

- Using a table of syntax-time metadata is shorter, sure
- It is also more maintainable and extensible!
- If the library-writer adds a new operation, just update the table
- If a client wants to make a custom operation, they can update the table, too

A Case Study: Rewriting Copatterns

Rewriting Macros in Copatterns

Exercising the trick at scale

- CPS is a single, well-studied use-case
- Examples here are small, so impact is debatable
- Same technique applied to a macro library for copattern-matching[2]
- Using rewriting macros to eliminate administrative steps expands to copattern-matching code with
  - Quantitatively, 39% less nesting and 45% fewer tokens
  - Qualitatively, more direct with less indirection and higher-order arguments
  - Much easier to debug and develop new macros!

[2] CoScheme: Compositional Copatterns in Scheme. Downen & Corbelino II, TFP '25.

Take Away

- Ease the tension between compositional and optimization!
- Scheme/Racket supports custom rewrite rules across macro boundaries
- Shortcut elimination-introduction pairs during expansion

If you want to play with these toys for yourself

CPS transformation macros

`https://github.com/pdownen/rewrite-macros`

Copatterns for Racket & R^{6}RS

`https://github.com/pdownen/CoScheme`