CoScheme: Compositional Copatterns in Scheme

Or, "Equal" Means Equal

Paul Downen and Adriano Corbelino II
University of Massachusetts Lowell

TFP — Thursday, January 16, 2025

Why Copatterns in Scheme?

Composition, Composition, Composition!

- Programs defined by equational reasoning on their context (à la ML, Haskell)
- Composition of extensible fragments at run-time
  - Vertical — *either or* — compose alternative options, handling failure
  - Horizontal — *and then* — compose sequence of steps, parameters, matching, guards
  - Circular — *self* — recursion back on the entire composition itself
- Library of composable macros
- Side benefit: supports infinite objects, some OO-style designs

Copatterns in Scheme

Encoding Functional Equations

Procedural style via manual operations

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip xs     ys     = []
```

Encoding Functional Equations

Procedural style via manual operations

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip xs     ys     = []
```

```
(define (zip xs ys)
  (cond
    [(and (pair? xs) (pair? ys))
     (cons (cons (car xs) (car ys))
           (zip (cdr xs) (cdr ys)))]
    [else '()]))
```

Encoding Functional Equations

Hybrid style via pattern matching

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip xs     ys     = []
```

```
(define (zip xs ys)
  (match xs
    [`(,x . ,xs-rest)
     (match ys
       [`(,y . ,ys-rest)
        `((,x . ,y) . ,(zip xs-rest ys-rest))]
       [_ '()])]
    [_ '()]))
```

Encoding Functional Equations

Equational style via copattern matching

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip xs     ys     = []
```

```
(define*
  [(zip `(,x . ,xs) `(,y . ,ys)) = `((,x . ,y) . ,(zip xs ys))]
  [(zip xs          ys)          = `()])
```

Encoding Infinite Objects

Streams observed through head and tail projecections

Stuttering stream from 0:

$$\mathit{stutter}\ 0 = 0, 0, 1, 1, 2, 2, 3, 3, \ldots$$

Stuttering stream from n:

$$\mathit{stutter}\ n = n, n, n+1, n+1, n+2, n+2, n+3, n+3, \ldots$$

Encoding Infinite Objects

Streams observed through head and tail projecections

Stuttering stream from 0:

$$stutter\ 0 = 0, 0, 1, 1, 2, 2, 3, 3, \ldots$$

Stuttering stream from n:

$$stutter\ n = n, n, n+1, n+1, n+2, n+2, n+3, n+3, \ldots$$

$\text{stream}\ a = (\text{'}\texttt{head} \to a)\ \&\ (\text{'}\texttt{tail} \to \text{stream}\ a)$

```
(define*
  [ ((stutter n) 'head)        = n]
  [(((stutter n) 'tail) 'head) = n]
  [(((stutter n) 'tail) 'tail) = (stutter (+ n 1))])
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) =
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) = 5
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) = 5

(((counter 5) 'add 6) 'get)
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) = 5

(((counter 5) 'add 6) 'get)

= ((counter 11) 'get)  = 11
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) = 5

(((counter 5) 'add 6) 'get)

= ((counter 11) 'get)  = 11

(define c (counter 10))

(list ((c 'add 2) 'get) ((c 'add 4) 'get))
```

Equational Reasoning Pop Quiz


```
(define*
  [((counter x) 'add y) = (counter (+ x y))]
  [((counter x) 'get)   = x])

((counter 5) 'get) = 5

(((counter 5) 'add 6) 'get)

= ((counter 11) 'get)  = 11

(define c (counter 10))

(list ((c 'add 2) 'get) ((c 'add 4) 'get))

= (list ((counter 12) 'get) ((counter 14) 'get))

= '(12 14)
```

The Expression Problem

A Simple Evaluator

Arithmetic expressions with just numbers and addition

```
;; Expr = `(num ,Number)
;;      | `(add ,Expr ,Expr)

;; expr0 : Expr
(define expr0
  '(add (num 1) (add (num 2) (num 3))))

;; eval : Expr -> Number
(define*
  [(eval `(num ,n))     = n]
  [(eval `(add ,l ,r)) = (+ (eval l) (eval r))])
```

The Expression Problem

Easy to add new operations:

```
;; expr? : Any -> Bool
(define*
  [(expr? `(num ,n))     = (number? n)]
  [(expr? `(add ,l ,r)) = (and (expr? l) (expr? r))]
  [(expr? _)             = #f])

;; list-nums : Expr -> List Number
(define*
  [(list-nums `(num ,n))     = (list n)]
  [(list-nums `(add ,l ,r)) = (append (list-nums l) (list-nums r))])
```

Hard to add new expression cases:

```
;; Expr = ... | `(mul ,Expr ,Expr)
???
```

A De/Re-composed Evaluator

As extensible objects

```
(define-object
  [(eval-num `(num ,n)) = n])

(define-object
  [(eval-add `(add ,l ,r)) = (+ (eval-add l) (eval-add r))])

;; same as before, but now defined via composition
(define eval-obj (eval-num 'compose eval-add))
```

A De/Re-composed Evaluator

As extensible objects

```
(define-object
  [(eval-num `(num ,n)) = n])

(define-object
  [(eval-add `(add ,l ,r)) = (+ (eval-add l) (eval-add r))])

;; same as before, but now defined via composition
(define eval-obj (eval-num 'compose eval-add))

eval-obj expr = eval expr

eval-obj = (object
             [(eval-num `(num ,n))     = n]
             [(eval-add `(add ,l ,r)) = (+ (eval-add l) (eval-add r))])
```

```
;; Expr = ... | `(mul ,Expr ,Expr)

(define-object
  [(eval-mul `(mul ,l ,r)) = (* (eval-mul l) (eval-mul r))])

(define eval-arith
  (eval-obj 'compose eval-mul))
```

```
;; Expr = ... | `(mul ,Expr ,Expr)

(define-object
  [(eval-mul `(mul ,l ,r)) = (* (eval-mul l) (eval-mul r))])

(define eval-arith
  (eval-obj 'compose eval-mul))

eval-arith
=
(object
  [(eval-num `(num ,n))     = n]
  [(eval-add `(add ,l ,r)) = (+ (eval-add l) (eval-add r))]
  [(eval-mul `(mul ,l ,r)) = (* (eval-mul l) (eval-mul r))])
```

Challenge: Adding Variables & Environments

Evaluating variable expressions requires an environment

```
;; Expr = ... | `(var ,Symbol)

(define-object
  [(eval-var env `(var ,x)) = (dict-ref env x)])

(eval-var '((x . 10) (y . 20)) '(var y)) = 20
```

How to compose (binary) `eval-var` with (unary) `eval-arith`?

Solution 1: Fixing the Environment


```
(define (fix-environment alg-evaluator env)
  (object
    [(_ expr)
     (try-apply-forget alg-evaluator env expr)]))
;; try-apply-forget attemps an application,
;; if it fails to match, continue with next option

(define-object
  [(eval-alg env expr)
  = (((fix-environment (eval-var 'unplug) env)
      'compose eval-arith)
     expr)])
;; 'unplug is an inherited-by-default method of objects
;; that converts it to a composable extension
```

Solution 2: Threading the Environment

Horizontal composition & first-class failure continuation

$$
\begin{aligned}
\mathit{eval}\ \mathit{expr} &= \ldots \mathit{eval}\ \mathit{subexpr} \ldots \\
&\Downarrow \\
\mathit{eval}\ \mathit{env}\ \mathit{expr} &= \ldots \mathit{eval}\ \mathit{env}\ \mathit{subexpr} \ldots
\end{aligned}
$$

Solution 2: Threading the Environment

Horizontal composition & first-class failure continuation

$$\begin{aligned} \mathit{eval}\ \mathit{expr} &= \ldots \mathit{eval}\ \mathit{subexpr} \ldots \\ &\Downarrow \\ \mathit{eval}\ \mathit{env}\ \mathit{expr} &= \ldots \mathit{eval}\ \mathit{env}\ \mathit{subexpr} \ldots \end{aligned}$$

```
(define (with-environment arith-evaluator)
  (object
   [(self env expr)
    (with-self
        (override-λ* self
          [(_ sub-expr) = (self env sub-expr)])
      (try-apply-forget arith-evaluator expr))]))
;; Implement environment extension by hiding environment
;; and overriding its concept of "self" to bring it back

(define eval-alg
  ((with-environment (eval-arith 'unplug))
   'compose eval-var))
```

Challenge: Constant Folding

Instead of using an environment to evaluate variables, just leave them alone:

```
(constant-fold '(mul (add (num 1) (num 2))
                     (add (var x) (num 4))))
=
               '(mul (num 3)
                     (add (var x) (num 4)))
```

Intermezzo: More Cautious Error Handling


```
(define-object eval-add-safe
  [(self `(add ,l ,r))
  = (self 'add (self l) (self r))]
  [(self 'add x y) (try-if (and (number? x) (number? y)))
  = (+ x y)])

(define-object eval-mul-safe
  [(self `(mul ,l ,r))
  = (self 'mul (self l) (self r))]
  [(self 'mul x y) (try-if (and (number? x) (number? y)))
  = (* x y)])

(define eval-arith-safe
  (eval-num 'compose eval-add-safe eval-mul-safe))
```

Leave Variables Alone

And reform blocked expressions

```
(define-object
  [(leave-variables `(var ,x)) = `(var ,x)])
```

Leave Variables Alone

And reform blocked expressions

```
(define-object
  [(leave-variables `(var ,x)) = `(var ,x)])


(define (operation? s)
  (or (equal? s 'add) (equal? s 'mul)))

(define-object reform-operations
  [(reform op l r) (try-if (operation? op)) (try-if number? l)
  = (reform op `(num ,l) r)]
  [(reform op l r) (try-if (operation? op)) (try-if number? r)
  = (reform op l `(num ,r))]
  [(reform op l r) (try-if (operation? op))
  = (list op l r)])
```

Solution: Constant Folding Made Easy


```
(define constant-fold
  (eval-arith-safe 'compose
                   leave-variables
                   reform-operations))
```

Solution: Constant Folding Made Easy


```
(define constant-fold
  (eval-arith-safe 'compose
                   leave-variables
                   reform-operations))

(define expr3
  '(add (add (num 1) (num 1))
        (mul (var x)
             (mul (num 2) (add (num 2) (num 3))))))

(constant-fold expr3)
= '(add (num 2) (mul (var x) (num 10)))
```

(How)
Does It Work?

$$Object = \text{some type of usable function}$$

$$Template = Object \to Object'$$

$$\begin{aligned} Extension &= Template \to Template' \\ &= Template \to Object \to Object' \end{aligned}$$

$$Object = \text{some type of usable function}$$

$$Template = (self : Object) \to Object'$$

$$\begin{aligned} Extension &= Template \to Template' \\ &= Template \to Object \to Object' \end{aligned}$$

$$Object = \text{some type of usable function}$$

$$Template = (self : Object) \to Object'$$

$$\begin{aligned} Extension &= (next : Template) \to Template' \\ &= (next : Template) \to (self : Object) \to Object' \end{aligned}$$

Expanding Definition Macros

```
(define* name clause ...) = (define name (λ* clause ...))

(define-object name clause ...) = (define name (object clause ...))
```

Expanding Definition Macros

```
(define* name clause ...) = (define name (λ* clause ...))

(define-object name clause ...) = (define name (object clause ...))

(λ* clause ...) = (introspect (template clause ...))

(object (<: mod) clause ...) = (plug (mod (extension clause ...)))

(object clause ...) = (object (<: (default-object-modifier)) clause ...)
```

Expanding "Big" Macros

```
(template clause ...)
= (closed-cases (extension clause ...))

(extension [copat step ...] ...)
= (compose [chain (comatch copat) step ...] ...)
```

Expanding "Big" Macros

```
(template clause ...)
= (closed-cases (extension clause ...))

(extension [copat step ...] ...)
= (compose [chain (comatch copat) step ...] ...)

(chain (op ...) step ... ext) = (op ... (chain step ... ext))

(chain = expr) = (always-is expr)
```

Some "Small" Macros

```
(try next self expr) = (λ(next) (λ(self) expr))

(always-is expr) = (try _ _ expr)

(try-if check ext)
= (try next self
       (if check
           ((ext next) self)
           (next self)))
```

Some "Small" Macros

```
(try next self expr) = (λ(next) (λ(self) expr))

(always-is expr) = (try _ _ expr)

(try-if check ext)
= (try next self
       (if check
           ((ext next) self)
           (next self)))

(try-λ x ext)
= (try next self
       (λ x
         ((ext (λ(s) (apply (next s) x)))
          self)))
```

Soundness of Equational Reasoning

- Model macros as a (selective) CPS-like translation
- Target: λ-calculus + recursion + symbols + lists + patterns
- Source: Target + copatterns + templates + extensions + λ^*
- Translations from Source to Target:

$$
\begin{array}{rll}
[\![\_]\!] & : \mathit{Term} \to \mathit{Target} & \text{(only translates new forms)} \\
T[\![\_]\!] & : \mathit{Template} \to \mathit{Target} & \text{(always a unary function)} \\
E[\![\_]\!] & : \mathit{Extension} \to \mathit{Target} & \text{(always a binary function)}
\end{array}
$$

Soundness of Equational Reasoning

- Model macros as a (selective) CPS-like translation
- Target: λ-calculus + recursion + symbols + lists + patterns
- Source: Target + copatterns + templates + extensions + λ^*
- Translations from Source to Target:

$$
\begin{aligned}
[\![\_]\!] &: \mathit{Term} \to \mathit{Target} && \text{(only translates new forms)} \\
T[\![\_]\!] &: \mathit{Template} \to \mathit{Target} && \text{(always a unary function)} \\
E[\![\_]\!] &: \mathit{Extension} \to \mathit{Target} && \text{(always a binary function)}
\end{aligned}
$$

Theorem (Conservative Extension)
If $M = N$ in the target theory, then $M = N$ in the source theory.

Theorem (Soundness)
If $M = N$ in the source theory, then $[\![M]\!] = [\![N]\!]$ in the target theory.

(Co)Pattern Matching Equalities

- Patterns P against values V:
  - Matching iff $P[W\ldots/x\ldots] = V$
  - Apart iff $P \# V$ (inductively defined over structure of P)

(Co)Pattern Matching Equalities

- Patterns P against values V:
  - Matching iff $P[W.../x...] = V$
  - Apart iff $P \# V$ (inductively defined over structure of P)
- Copatterns Q against contexts C
  - Matching iff $Q[W.../x...] = C$
  - Apart iff $Q \# C$ (inductively defined over structure of Q)

(Co)Pattern Matching Equalities

- Patterns P against values V:
  - Matching iff $P[W.../x...] = V$
  - Apart iff $P \# V$ (inductively defined over structure of P)
- Copatterns Q against contexts C
  - Matching iff $Q[W.../x...] = C$
  - Apart iff $Q \# C$ (inductively defined over structure of Q)

```
(try-match P V ext) = ext[W.../x...]      ; if P[W.../x...] = V
(try-match P V ext) = empty-extension     ; if P # V

C[(λ* [Q[y] = expr] clause ...)]   = expr[W.../x...]  ; if Q[W.../x...] = C
C[(λ* [Q[y] = expr] [else deflt])] = C[deflt]         ; if Q # C
```

CoScheme: Composable, Equational, Copatterns!

Try It Yourself!

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

- Racket
- R^{6}RS

Factoring Copatterns

Nested definitions

Sharing constant parameters of a loop

```
map f []     = []
map f (x:xs) = f x : map f xs

map f xs = go xs                    -- map f = go
  where go []     = []
        go (x:xs) = f x : go xs
```

Nested definitions

Sharing constant parameters of a loop

```
map f []     = []
map f (x:xs) = f x : map f xs

map f xs = go xs                    -- map f = go
  where go []     = []
        go (x:xs) = f x : go xs
```

```
(define*
  [(map f xs) = ((map f) xs)]  ; (un)curried forms equal
  [(map f) (nest)              ; map f = go
    (extension
     [(go `())          = `()]
     [(go `(,x . ,xs)) = `(,(f x) . ,(go xs))])])
```

Refactoring a Common Prefix


```
(define-object reform-operations
  [(reform op l r) (try-if (operation? op)) (try-if number? l)
  = (reform op `(num ,l) r)]
  [(reform op l r) (try-if (operation? op)) (try-if number? r)
  = (reform op l `(num ,r))]
  [(reform op l r) (try-if (operation? op))
  = (list op l r)])
```

common prefix: `(reform op l r) (try-if (operation? op))`

Refactoring a Common Prefix


```
(define-object reform-operations
  [(reform op l r) (try-if (operation? op)) (try-if number? l)
  = (reform op `(num ,l) r)]
  [(reform op l r) (try-if (operation? op)) (try-if number? r)
  = (reform op l `(num ,r))]
  [(reform op l r) (try-if (operation? op))
  = (list op l r)])
```

common prefix: `(reform op l r) (try-if (operation? op))`

```
(define-object reform-operations
  [(reform op l r) (try-if (operation? op))
   (extension
    [_ (try-if (number? l)) = (reform op `(num ,l) r)]
    [_ (try-if (number? r)) = (reform op l `(num ,r))]
    [_                      = (list op l r)])])
```