# CoScheme:
# Compositional Copatterns in Scheme

Or, "Equal" Means Equal

**Paul Downen**
University of Massachusetts Lowell

Northeastern Software Seminar — Friday, January 30, 2026

# “Copatterns?”

# What's in a Copattern?

"Why not just patterns?"

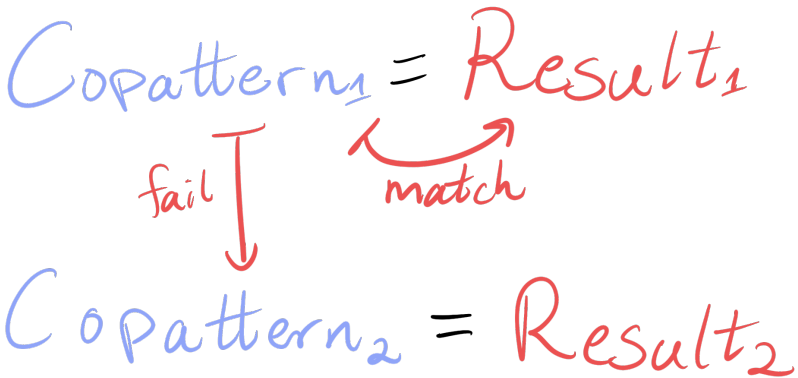
- Copatterns give a more general method for programs to interact with their context
- They are
  - Declarative (based on equalities, not operations)
  - Structural (describe the shapes of their context)
- Famous use case: Modeling infinite objects
- Also just a clean way to describe many coding patterns!
- Pattern-matching functions are a special case

# How to Read Copatterns

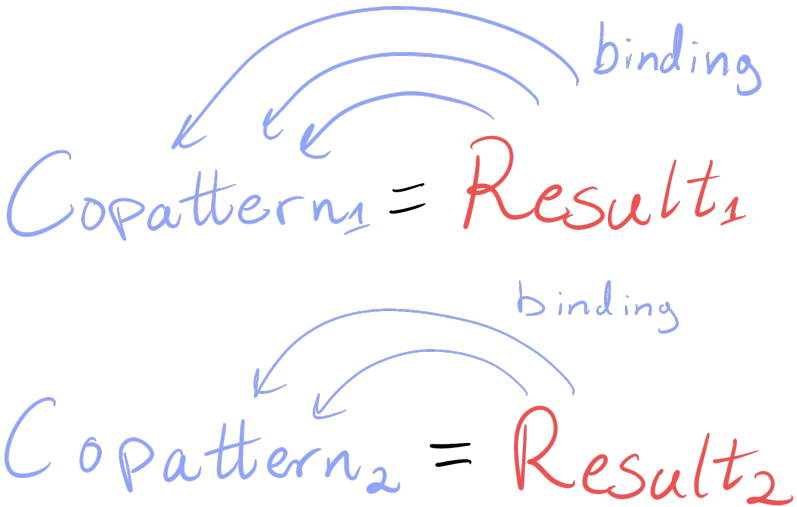
$$\mathrm{Copattern}_1 = \mathrm{Result}_1$$

$$\mathrm{Copattern}_2 = \mathrm{Result}_2$$

# How to Read Copatterns



# How to Read Copatterns



# How to Read Copatterns

## And some more advanced flow

Copattern

if $guard_1$ = $Result_1$

if $guard_2$ = $Result_2$

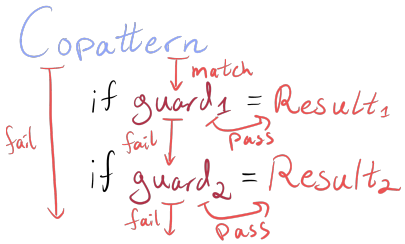
InnerCo

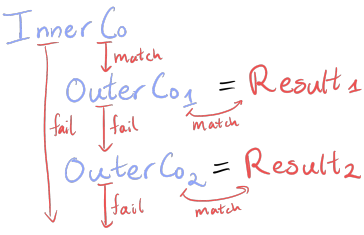
$OuterCo_1$ = $Result_1$

$OuterCo_2$ = $Result_2$

# How to Read Copatterns

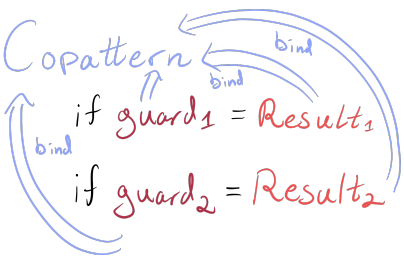
## And some more advanced flow

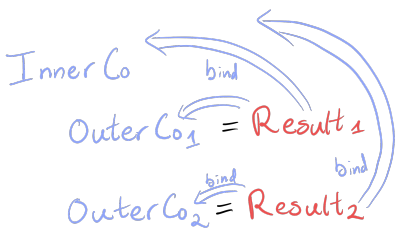


# How to Read Copatterns

## And some more advanced flow



# Where Do Copatterns Come From?

- A deep study of duality of classical logic and polarity in proof theory
  - Noam Zeilberger. On the Unity of Duality. Annals of Pure and Applied Logic 153:1 (2008).
- A fix to problems with coinduction in proof assistants
  - Abel, Pientka, Thibodeau, and Setzer. Copatterns: programming infinite structures by observations. POPL '13.
- Lurking in Agda for a while…
  - And other proof assistants, like Beluga

# Why Copatterns in Scheme/Racket?

I'm a Programmer, too, not just a Proof Monkey!

- I want to write real programs
  - …and actually run them.
- Lets me pretend I'm writing declarative, equational Haskell/SML code
- Still get to use advanced control flow effects (e.g., control operators)
- An easy way to handle complex infinite objects in a strict language
- Dynamic types: Interesting ways to compose programs!
- Side benefit: sneak in purely functional OOP

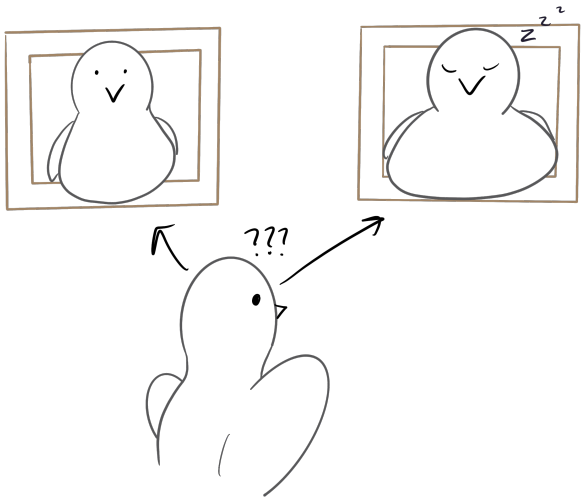
Don’t miss the “What”
by obsessing over “How”

# Let's Play!

# Intermezzo:
# Infinite Pigeons!!!

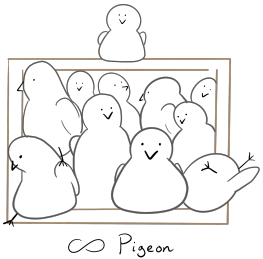
# The Pigeonhole Principle

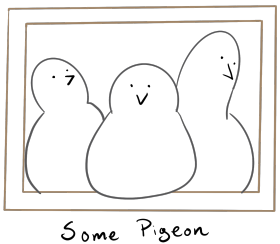
Too many pigeons, not enough holes



# The Infinite Pigeonhole Principle

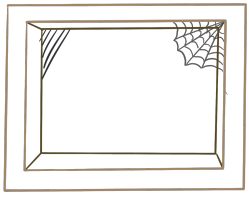
One hole must be infinitely full!

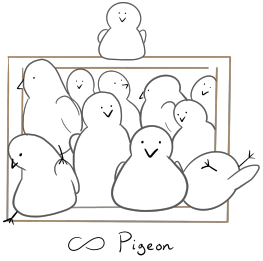


# The Infinite Pigeonhole Principle

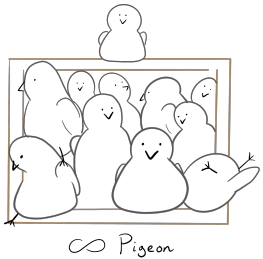
One hole must be infinitely full!

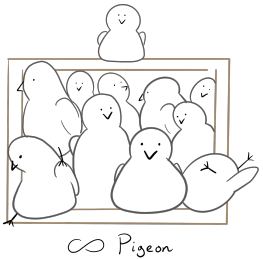


# The Infinite Pigeonhole Principle

One hole must be infinitely full!



# Encoding The Infinite Pigeonhole Principle

- 2 holes = Boolean choice (true or false, 1 or 0)
- $\infty$ pigeons = infinite stream

**Corollary (Infinite bits)**
*Any infinite binary stream (of `#t` or `#f`) must have infinite `#t`s or infinite `#f`s.*

*Concretely, there exists an infinite stream of ascending indexes which always point to all `#t`s or always point to all `#f`s in the given binary stream.*

# Encoding The Infinite Pigeonhole Principle

- 2 holes = Boolean choice (true or false, 1 or 0)
- $\infty$ pigeons = infinite stream

**Corollary (Infinite bits)**
*Any infinite binary stream (of `#t` or `#f`) must have infinite `#t`s or infinite `#f`s.*

*Concretely, there exists an infinite stream of ascending indexes which always point to all `#t`s or always point to all `#f`s in the given binary stream.*

**Lemma (Infinite filter)**
*For any infinite stream and any predicate on elements of the stream, there must be infinite elements of that stream for which the predicate returns `#t` or for which the predicate returns `#f`.*

*Concretely, there exists a substream — consisting of some of the original elements in the same order — for which every element passes the predicate, or every element fails the predicate.*

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating `(infinite-filter square? nats)`

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? |
|---|---|
| YES | 0 |
| no | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating `(infinite-filter square? nats)`

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? |
|---|---|---|
| YES | 0 | 1 |
| no | | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating (`infinite-filter square? nats`)

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? | @ 3? |
|---|---|---|---|
| yes | 0 | 1 | |
| NO | 2 | | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating (`infinite-filter square? nats`)

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? | @ 3? |
|---|---|---|---|
| yes | 0 | 1 | |
| NO | 2 | 3 | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating (`infinite-filter square? nats`)

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? | @ 3? |
|---|---|---|---|
| YES | 0 | 1 | 4 |
| no | 2 | 3 | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating (`infinite-filter square? nats`)

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? | @ 3? | @ 4? |
|---|---|---|---|---|
| yes | 0 | 1 | 4 | |
| NO | 2 | 3 | 5 | |

# Generating the Infinite Filter

## Choosing the answer based on the question

Calculating (`infinite-filter square? nats`)

Original stream: 0, 1, 2, 3, 4, 5, 6, 7, …

Answer:

| Square? | @ 0? | @ 2? | @ 3? | @ 4? | @ 5? | … |
|---|---|---|---|---|---|---|
| yes | 0 | 1 | 4 | … | | |
| NO | 2 | 3 | 5 | 6 | 7 | … |

# How?!?!

$$\begin{aligned} \mathit{Object} &= \text{some type of usable function} \\ \mathit{Template} &= \mathit{Object} \to \mathit{Object}' \\ \mathit{Extension} &= \mathit{Template} \to \mathit{Template}' \\ &= \mathit{Template} \to \mathit{Object} \to \mathit{Object}' \end{aligned}$$

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

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

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

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

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

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

# 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)))
```

# Rewriting Macros in Copatterns

## Eliminating administrative reductions at expansion time

- The expanded macros are too hard to understand!
  - Bad for macro development
  - Bad for debugging
  - Questionable for performance
- Use a technique for rewriting nested macro calls to inline & skip administrative steps
- “Rewritten” macros expand 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!

# What…?

# Soundness of Equational Reasoning

- Model macros as a (selective) CPS-like translation
- Target: $\lambda$-calculus + recursion + symbols + lists + patterns
- Source: Target + copatterns + templates + extensions + $\lambda^*$
- 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: $\lambda$-calculus + recursion + symbols + lists + patterns
- Source: Target + copatterns + templates + extensions + $\lambda^*$
- 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
```

# A Naïve Semantics for Monolithic Copatterns



$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns



$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \le i \le n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns



$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns



$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns



$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns

## Copatterns = equational reasoning on the context

$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leqslant i \leqslant n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Naïve Semantics for Monolithic Copatterns

## Copatterns = equational reasoning on the context

$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \lambda\{L \to M \ldots\} \\ \mathit{Copat} \ni \quad L &::= \varepsilon \mid x\ L \mid X\ L \end{aligned}$$

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}] \quad \begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j, & \nexists \vec{N},\ C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

# A Calculus for Compositional Copatterns with Control

$$
\begin{aligned}
\textit{Response} \ni R &::= q \mid \varepsilon \mid M \,!\, R \\
\textit{Term} \ni M, N &::= x \mid M\ N \mid M\ X \mid \mathbf{raise} \mid O \,?\, M \mid !q \to R \\
\textit{Option} \ni O &::= x \to O \mid X \to O \mid ?x \to M
\end{aligned}
$$

Old monolithic syntax now just sugar (proved correct by CPS!) on smaller primitives:

$$
\begin{aligned}
\lambda\{O_1 \mid \cdots \mid O_n\} &:= O_1 \,?\, (\cdots ?\, (O_n \,?\, \mathbf{raise})) \\
\varepsilon \to M &:= ?\_ \to M \\
(x\ L) \to O &:= x \to (L \to O) \\
(X\ L) \to O &:= X \to (L \to O)
\end{aligned}
$$

# A Calculus for Compositional Copatterns with Control

$$
\begin{aligned}
\mathit{Response} \ni R &::= q \mid \varepsilon \mid M\,!\,R \\
\mathit{Term} \ni M, N &::= x \mid M\,N \mid M\,X \mid \textbf{raise} \mid O\,?\,M \mid !q \to R \\
\mathit{Option} \ni O &::= x \to O \mid X \to O \mid ?x \to M
\end{aligned}
$$

Old monolithic syntax now just sugar (proved correct by CPS!) on smaller primitives:

$$
\begin{aligned}
\lambda\{O_1 \mid \cdots \mid O_n\} &:= O_1\,?\,(\cdots\,?\,(O_n\,?\,\textbf{raise})) \\
\varepsilon \to M &:= ?\_ \to M \\
(x\ L) \to O &:= x \to (L \to O) \\
(X\ L) \to O &:= X \to (L \to O)
\end{aligned}
$$

But smaller primitives now give more functionality, such as vertical composition

$$
\begin{aligned}
\texttt{object}\ O &:= \lambda\{O \mid \mathit{self}\ \mathit{Open} \to \lambda\{x \to O\,?\,x\}\} \\
\mathit{compose} &:= \lambda o\ o' \to \texttt{object}\ \{?x \to o.\mathit{Open}(o'.\mathit{Open}\ x)\}
\end{aligned}
$$

$$
\mathit{compose}\ \texttt{object}\{O\}\ \texttt{object}\{O'\} = \texttt{object}\{O \mid O'\}
$$

# An Operational Semantics



$$\mathit{DelimCxt} \ni D ::= \Box \mid M \,!\, D \qquad \mathit{EvalCxt} \ni E ::= \Box \mid E\ N \mid E\ X$$

$$\frac{M \mapsto M'}{E[M] \mapsto E[M']} \qquad \frac{M \mapsto M'}{M \,!\, \varepsilon \mapsto M' \,!\, \varepsilon} \qquad \frac{R \mapsto R'}{D[R] \mapsto D[R']}$$

$$\begin{aligned}
(?x \to N) \,?\, M &\mapsto N[M/x] \\
((x \to O) \,?\, M)\ N &\mapsto O[N/x] \,?\, (M\ N) \\
((X \to O) \,?\, M)\ X &\mapsto O \,?\, (M\ X) \\
(P \,?\, M)\ X &\mapsto M\ X && \text{(otherwise)} \\
(P \,?\, M)\ N &\mapsto M\ N && \text{(otherwise)}
\end{aligned}$$

$$\begin{aligned}
E[!k \to R] \,!\, \varepsilon &\mapsto R[(E[\mathbf{raise}] \,!\, \varepsilon)/k] \\
M \,!\, (E[\mathbf{raise}] \,!\, \varepsilon) &\mapsto E[M] \,!\, \varepsilon \\
(P \,?\, M) \,!\, \varepsilon &\mapsto M \,!\, \varepsilon
\end{aligned}$$

# Future Work

TODO

- Better error messages!!!
  - Point to correct place in user code, not library
  - Show enough context in stack trace to explain copattern-match failure
  - Give a way to produce more meaningful errors based on expectations
- Better display of anonymous objects
  - Point to definition site of user code, not library
- Memoization

# CoScheme: Composable, Equational, Copatterns!

## Try It Yourself!

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

- Racket
- R$^6$RS

“Copatterns?”

Let’s Play!

Intermezzo:
Infinite Pigeons!!!

How?!?!

What…?

Copatterns in Scheme

The
Expression Problem

Factoring Copatterns

# Copatterns in Scheme

# Encoding Functional Equations

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

# Encoding Functional Equations

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

```racket
(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

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

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

# Encoding Functional Equations

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

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

# Encoding Infinite Objects

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

Stuttering stream from 0:

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

Stuttering stream from n:

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

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

```
(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

```
;; 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

```
(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

```
(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))])
```

# Simple Extensions of the Expression Language

```
;; 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))
```

# Simple Extensions of the Expression Language

```
;; 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

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

# Solution 2: Threading the Environment

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

```
(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))
```

# Solution 2: Holding the Environment

```
; ((alg Env) `env)        : Env
; ((alg Env) `eval Expr) : number
(define (alg dict)
  (arith-ext `compose
   (object
    [(self `env)           = dict]
    [(self `eval x)
     (try-if (symbol? x)) = (dict-ref (self `env) x)])))

((alg env-xy) `eval expr0) = 7
((alg env-xy) `eval expr1) = -5
((alg env-xy) `eval expr2) = -30
```

# 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

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

# Leave Variables Alone

```
(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)))
```

# Factoring Copatterns

# Nested definitions

```haskell
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

```haskell
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
```

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