Controlling Copatterns

There and Back Again

Paul Downen
University of Massachusetts Lowell

OlivierFest — Wednesday, October 15, 2025

The Context

Quick Primer on Copatterns

In Scheme / Racket

Defining the 2×2 matrix

$$\text{quad} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

as the nested pair $((1, 2), (3, 4))$ by copattern matching:

```
(define*
  [((quad 'fst) 'fst) = 1]
  [((quad 'fst) 'snd) = 2]
  [((quad 'snd) 'fst) = 3]
  [((quad 'snd) 'snd) = 4])
```

Mixing Copatterns of Different Depths

Overriding the diagonals (upper left, lower right): $\left(\mathtt{diag}\ x\ y\ \begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{bmatrix} x & b \\ c & y \end{bmatrix}$

```
(define*
  [(((diag x y z) `fst) `fst) = x]
  [(((diag x y z) `snd) `snd) = y]
  [   (diag x y z)              = z])
```

is equivalent to (using equational reasoning) expanding the remaining options

```
(define*
  [(((diag x y z) `fst) `fst) = x]
  [(((diag x y z) `snd) `snd) = y]
  [(((diag x y z) `fst) `snd) = ((z `fst) `snd)]
  [(((diag x y z) `snd) `fst) = ((z `snd) `fst)])
```

Composable Copatterns

Combining programs along different dimensions

- 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
- Side benefit: supports infinite objects, some OO-style designs

An Equational Arithmetic Evaluator

Is this ML, or Scheme?

```
; Expr = number | `(add ,Expr ,Expr) | `(mul ,Expr ,Expr)

; expr0 : Expr
(define expr0 `(add 1 (mul 2 3)))

; (arith `eval Expr) : number
(define-object
  [(arith `eval n) (try-if (number? n))
   = n]
  [(arith `eval `(add ,l ,r))
   = (+ (arith `eval l) (arith `eval r))]
  [(arith `eval `(mul ,l ,r))
   = (* (arith `eval l) (arith `eval r))])

(arith `eval expr0) = 7
```

Extending Evaluation with a New Operator

The wrong way

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

; expr1 : Expr
(define expr1 `(add 1 (neg (mul 2 3))))

(define-object
  [(arith-wrong `eval `(neg ,e))
   = (- (arith-wrong `eval e))]
  [(arith-wrong `eval e) = (arith `eval e)])

(arith-wrong `eval expr1)
=
(+ 1 (arith `eval `(neg (mul 2 3))))
=/=
```

Extending Evaluation with a New Operator

The correct way, using vertical composition

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

; expr1 : Expr
(define expr1 `(add 1 (neg (mul 2 3))))
```

Correct vertical composition:

```
; (arith-ext `eval Expr) : number
(define arith-ext
  (arith `compose
   (object
    [(self `eval `(neg ,e))
     = (- (self `eval e))])))

(arith-ext `eval expr1)
=
(+ 1 (arith-ext `eval `(neg (mul 2 3))))
=
-5
```

Understanding the Result of Composition

Expanding the vertical composition:

```
(define-object
  [(arith-ext ‘eval n) (try-if (number? n))
   = n]
  [(arith-ext ‘eval ‘(add ,l ,r))
   = (+ (arith-ext ‘eval l) (arith-ext ‘eval r))]
  [(arith-ext ‘eval ‘(mul ,l ,r))
   = (* (arith-ext ‘eval l) (arith-ext ‘eval r))]
  [(arith-ext ‘eval ‘(neg ,e))
   = (- (arith-ext ‘eval e))])
```

How Do We Extend Arithmetic to Algebra?

A more serious extension, needs another parameter

```
; Expr = ... | symbol

; expr2 : Expr
(define expr2 `(add x (neg (mul 2 y))))

; Env = ((symbol . number ) ...)

; env-xy : Env
(define env-xy `((x . 10) (y . 20)))

(define-object alg ...?)
```

Extending the Evaluator with an Environment!

Is this functional, or object-oriented?

```
; ((with-env Env) `env) : Env
(define (with-env dict)
  (object [(_ ‘env) = dict]))

; ((alg Env) `env)        : Env
; ((alg Env) `eval Expr) : number
(define (alg dict)
  (arith-ext ‘compose
   (with-env dict)
   (object
    [(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
```

The Problem

The story so far

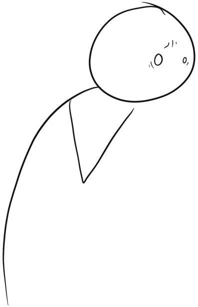
- Compositional copatterns implemented as Scheme / Racket macros
- For free: semantics as (selective) CPS
- This gives some rules for equational reasoning...

The story so far

- Compositional copatterns implemented as Scheme / Racket macros
- For free: semantics as (selective) CPS
- This gives some rules for equational reasoning…
  - …but not enough to calculate every answer
- Really need a standard operational semantics
  - Reason about copattern-matching programs directly (no translation)
  - State and prove type safety problems

Operational Semantics of Compositional Copatterns

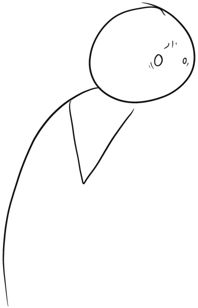
What to do, what to do…?



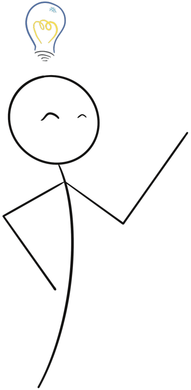
How the heck to deal with all the recursive generality and edge cases?!

Operational Semantics of Compositional Copatterns

What to do, what to do…?



How the heck to deal with all the recursive generality and edge cases?!



Oh wait! I remember! Olivier taught me exactly how to deal with this kind of thing!

The Unity of Semantic Artifacts

- Technique for mechanically deriving one style of semantics from another
- Using only off-the-shelf, semantics-preserving program transformations
- Correspondence between semantic artifacts for free!
- Many good intros: "A Walk in the Semantic Park"[1] is quite refreshing

[1] Olivier Danvy, Jacob Johannsen, and Ian Zerny, PEPM 2011.

The Unity of Semantic Artifacts

- Technique for mechanically deriving one style of semantics from another
- Using only off-the-shelf, semantics-preserving program transformations
- Correspondence between semantic artifacts for free!
- Many good intros: "A Walk in the Semantic Park"[1] is quite refreshing
- This was a key part of my second academic paper:

Classical Call-by-Need Sequent Calculi:
The Unity of Semantic Artifacts

Zena M. Ariola[1], Paul Downen[1], Hugo Herbelin[2], Keiko Nakata[3],
and Alexis Saurin[2]

Abstract. We systematically derive a classical call-by-need sequent calculus, which does not require an unbounded search for the standard redex, by using the unity of semantic artifacts proposed by Danvy *et al.* The calculus serves as an intermediate step toward the generation of an environment-based abstract machine. The resulting abstract machine is context-free, so that each step is parametric in all but one component. The context-free machine elegantly leads to an environment-based CPS

[1] Olivier Danvy, Jacob Johannsen, and Ian Zerny, PEPM 2011.

Two Roads Through the Semantic Woods

Search & Reduce

CPS

Defunctionalize

Simplify

Operational Semantics

Loop Fusion

Simplify

Deforest

Abstract Machine

Shallow patterns

Denotational style

Refunctionalize

CPS Transformation

Two Roads Through the Semantic Woods

And I had to take the one less traveled by

Search & Reduce

CPS

Defunctionalize

Simplify

Operational Semantics

Loop Fusion

Simplify

Deforest

Abstract Machine

Shallow patterns

Denotational style

Refunctionalize

Inline & fusion

Syntactic style

Defunctionalize

CPS Transformation

Two Roads Through the Semantic Woods

And I had to take the one less traveled by

Search & Reduce

CPS
Defunctionalize
Simplify

Operational Semantics

Loop Fusion
Simplify
Deforest

Loop fission
Expand
Reforest

Abstract Machine

Shallow patterns
Denotational style
Refunctionalize

Inline & fusion
Syntactic style
Defunctionalize

CPS Transformation

The Solution

A Round-Trip in the Semantic Park

Taking the easy path to retrace our steps

- “Reforestation” and loop “fission” are hard
- Why? They undo lossy transformations
  - Loop fusion destroys the inner loops
  - Deforestation destroys intermediate data structures
- It helps to see what was lost to restore it
- Trick: First take the easy “forward” path from a similar starting point to light the way for the trip back

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$(\beta) \quad C[\lambda\{L_i \to M_i{}^{1 \leq i \leq n}\}] = M_j[\overrightarrow{N/x}]$$

$$\begin{pmatrix} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j,\ \nexists \overrightarrow{N},\ & C = L_i[\overrightarrow{N/x}] \end{pmatrix}$$

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

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

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$(\beta) \quad C[\lambda\{L_i \to M_i \overset{1 \leq i \leq n}{\dots}\}] = M_j[\overrightarrow{N/x}]$$

$$\left(\begin{array}{lr} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j,\ \nexists \overrightarrow{N}, & C = L_i[\overrightarrow{N/x}] \end{array}\right)$$

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$
(\beta) \quad C[\lambda\{L_i \to M_i \overset{1\leq i\leq n}{\ldots}\}] = M_j[\overrightarrow{N/x}]
$$

$$
\left(\begin{array}{lr}
\text{if} & C = L_j[\overrightarrow{N/x}] \\
\text{and } \forall i < j,\ \nexists \overrightarrow{N}, & C = L_i[\overrightarrow{N/x}]
\end{array}\right)
$$

A Naïve Semantics for Monolithic Copatterns

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

$$
(\beta) \quad C[\lambda\{L_i \to M_i \overset{1\leq i\leq n}{\ldots}\}] = M_j[\overrightarrow{N/x}]
$$

$$
\begin{pmatrix}
\text{if} & C = L_j[\overrightarrow{N/x}] \\
\text{and } \forall i < j,\ \nexists \overrightarrow{N}, & C = L_i[\overrightarrow{N/x}]
\end{pmatrix}
$$

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$
(\beta) \quad C[\lambda\{L_i \to M_i \overset{1 \le i \le n}{\ldots}\}] = M_j[\overrightarrow{N/x}]
$$

$$
\begin{pmatrix}
\text{if} & C = L_j[\overrightarrow{N/x}] \\
\text{and } \forall i < j,\ \nexists \vec{N},\ & C = L_i[\overrightarrow{N/x}]
\end{pmatrix}
$$

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$
(\beta) \quad C[\lambda\{L_i \to M_i \overset{1 \leq i \leq n}{\ldots}\}] = M_j\overrightarrow{[N/x]}
$$

$$
\begin{pmatrix}
\text{if} & C = L_j\overrightarrow{[N/x]} \\
\text{and } \forall i < j,\ \nexists \vec{N},\ & C = L_i\overrightarrow{[N/x]}
\end{pmatrix}
$$

A Naïve Semantics for Monolithic Copatterns

Copatterns = equational reasoning on the context

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

$$(\beta) \quad C[\lambda\{L_i \to M_i \overset{1 \le i \le n}{\ldots}\}] = M_j[\overrightarrow{N/x}]$$

$$\left(\begin{array}{ll} \text{if} & C = L_j[\overrightarrow{N/x}] \\ \text{and } \forall i < j,\ \nexists \overrightarrow{N},\ C = L_i[\overrightarrow{N/x}] \end{array} \right)$$

Writing an Algorithm for Small Steps

```haskell
-- Syntax
data Term      i a -- `i` represents literal index, `a` represents variable
data Copattern i a
type Question  i a = Copattern i (Term i a) -- copattern-shaped contexts
```

Writing an Algorithm for Small Steps

```
-- Syntax
data Term      i a -- `i` represents literal index, `a` represents variable
data Copattern i a
type Question  i a = Copattern i (Term i a) -- copattern-shaped contexts

-- Reduction
data Redex    i a = Respond [Option i a] | FreeVar a
data Reduct   i a = Reduced (Term i a) | Unhandled | Unknown a
data Followup i a = Next (Reduct i a) (Question i a)
                  | More (Copattern i a) (Term i a)
                         [Option i a] (Question i a)

reduce :: (Eq i, Eq a) => Redex i a -> Question i a -> Followup i a
```

Writing an Algorithm for Small Steps

```haskell
-- Syntax
data Term      i a -- `i` represents literal index, `a` represents variable
data Copattern i a
type Question  i a = Copattern i (Term i a) -- copattern-shaped contexts

-- Reduction
data Redex    i a = Respond [Option i a] | FreeVar a
data Reduct   i a = Reduced (Term i a) | Unhandled | Unknown a
data Followup i a = Next (Reduct i a) (Question i a)
                  | More (Copattern i a) (Term i a)
                         [Option i a] (Question i a)

reduce :: (Eq i, Eq a) => Redex i a -> Question i a -> Followup i a

-- Search
data Found i a = Asked (Redex i a) (Question i a)

search :: Term i a -> Found i a
```

A few moments later …

The Derived CPS Transformation

Applicative forms look like normal CBN CPS:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![M\,N]\!] &= \lambda k.\ [\![M]\!]\ ([\![N]\!], k) \\
[\![M\,X]\!] &= \lambda k.\ [\![M]\!]\ (X\ k)
\end{aligned}
$$

λs begin copattern-matching various options against the continuation:

$$
\begin{aligned}
[\![\lambda\{\varepsilon\}]\!] &= \lambda k.\ k \\
[\![\lambda\{L \to M \mid L' \to M' ...\}]\!] &= \lambda k.\ [\![L \to M]\!]\ k\ [\![\lambda\{L' \to M' ...\}]\!]\ k
\end{aligned}
$$

The Derived CPS Transformation

Copatterns match on the given continuation:

$$
\begin{aligned}
[\![\varepsilon \to N]\!] &= \lambda q.\lambda f.\ [\![N]\!] \\
[\![x\ L \to N]\!] &= \mathbf{rec}\, r = \lambda q.\lambda f.\lambda k. \\
&\qquad \mathbf{case}\, k\, \mathbf{of} \begin{array}[t]{ll} (x, k') & \to [\![L \to N]\!]\ q\ f\ k' \\ () & \to r\ q\ f \\ k & \to f\ q \end{array} \\
[\![X\ L \to N]\!] &= \mathbf{rec}\, r = \lambda q.\lambda f.\lambda k. \\
&\qquad \mathbf{case}\, k\, \mathbf{of} \begin{array}[t]{ll} (X\ k') & \to [\![L \to N]\!]\ q\ f\ k' \\ () & \to r\ q\ f \\ k & \to f\ q \end{array}
\end{aligned}
$$

The Derived CPS Transformation

Copatterns match on the given continuation:

$$
\begin{aligned}
[\![\varepsilon \to N]\!] &= \lambda q.\lambda f.\ [\![N]\!] \\
[\![x\ L \to N]\!] &= \textbf{rec}\ r = \lambda q.\lambda f.\lambda k. \\
&\qquad \textbf{case}\ k\ \textbf{of}\ \begin{array}{ll} (x, k') & \to [\![L \to N]\!]\ q\ f\ k' \\ () & \to r\ q\ f \\ k & \to f\ q \end{array} \\
[\![X\ L \to N]\!] &= \textbf{rec}\ r = \lambda q.\lambda f.\lambda k. \\
&\qquad \textbf{case}\ k\ \textbf{of}\ \begin{array}{ll} (X\ k') & \to [\![L \to N]\!]\ q\ f\ k' \\ () & \to r\ q\ f \\ k & \to f\ q \end{array}
\end{aligned}
$$

- If the question is too short, check again when given more continuation

The Derived CPS Transformation

Copatterns match on the given continuation:

$$
\begin{aligned}
[\![\varepsilon \to N]\!] &= \lambda q.\lambda f.\, \lambda k.[\![N]\!]\; k \\
[\![x\; L \to N]\!] &= \mathbf{rec}\; r = \lambda q.\lambda f.\lambda k. \\
&\qquad \mathbf{case}\; k\; \mathbf{of}\; \begin{array}{ll} (x, k') & \to [\![L \to N]\!]\; q\; f\; k' \\ () & \to r\; q\; f \\ k & \to f\;\; q \end{array} \\
[\![X\; L \to N]\!] &= \mathbf{rec}\; r = \lambda q.\lambda f.\lambda k. \\
&\qquad \mathbf{case}\; k\; \mathbf{of}\; \begin{array}{ll} (X\; k') & \to [\![L \to N]\!]\; q\; f\; k' \\ () & \to r\; q\; f \\ k & \to f\;\; q \end{array}
\end{aligned}
$$

- If the question is too short, check again when given <u>more</u> continuation
- On successful match, keep checking the rest of the continuation
- On a mismatch failure, reset to original question and try next option

Bridging the Gap between Monolithic & Compositional

Equational reasoning to "clean up" the CPS to resemble compositional copattern macros:

- Delimiting the context
  - Add an explicit "end of message" marker ($M \,!\, R$) at the end of the question
  - Avoids confusion between "not enough context" versus "question too short"
- Nesting copatterns
  - Regroup copatterns ($L \to M$) to single steps that lean to the right (O)
  - Add an explicit "alternative" ($O \,?\, M$) for when matching fails
- Eliminate redundancy
  - Instead of passing failure+alternative+success continuations, fold failure into alternative
  - $[\![\lambda\{L \to M \mid L' \to M' \dots\}]\!] = \lambda k.\, [\![L \to M]\!]\; k\; [\![\lambda\{L' \to M' \dots\}]\!]\; k$
- Regain proper CPS through double-barrel continuations (like Shift+Reset)

Bridging the Gap between Monolithic & Compositional

Equational reasoning to "clean up" the CPS to resemble compositional copattern macros:

- Delimiting the context
  - Add an explicit "end of message" marker ($M \mathbin{!} R$) at the end of the question
  - Avoids confusion between "not enough context" versus "question too short"
- Nesting copatterns
  - Regroup copatterns ($L \to M$) to single steps that lean to the right (O)
  - Add an explicit "alternative" ($O \mathbin{?} M$) for when matching fails
- Eliminate redundancy
  - Instead of passing failure+alternative+success continuations, fold failure into alternative
  - $[\![\lambda\{L \to M \mid L' \to M' \ldots\}]\!] = \lambda k.\ [\![L \to M]\!] \quad [\![\lambda\{L' \to M' \ldots\}]\!]\ k$
- Regain proper CPS through double-barrel continuations (like Shift+Reset)

A Calculus for Compositional Copatterns with Control

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

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

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

A Calculus for Compositional Copatterns with Control

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

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

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

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

A few moments later …

The Derived Operational Semantics

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

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

The Denouement

Thank You, Olivier!

- We met when I was but a green Ph.D. student
- Your generosity has taught me much
- I'm happy the influence is still alive today!

If you want to play with these toys for yourself

Semantic derivations & examples

https://github.com/pdownen/derive-copat

Copatterns for Racket & R^6RS

https://github.com/pdownen/CoScheme