Controlling Copatterns: There and Back Again

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

Abstract

Copatterns give functional programs a flexible mechanism for responding to their context, and composition can greatly enhance their expressiveness. However, that same expressive power makes it harder to precisely specify the behavior of programs. Using Danvy's functional and syntactic correspondence between different semantic artifacts, we derive a full suite of semantics for copatterns, twice. First, a calculus of monolithic copatterns is taken on a journey from small-step operational semantics to abstract machine to continuation-passing style. Then within continuation-passing style, we refactor the semantics to derive a more general calculus of compositional copatterns, and take the return journey back to derive the other semantic artifacts in reverse order.

CCS Concepts: • **Software and its engineering** → **Patterns**; **Interpreters**; • **Theory of computation** → **Control primitives**; **Operational semantics**; **Abstract machines**.

Keywords: Copatterns, Delimited Control, Handlers, Functional Correspondence, Syntactic Correspondence

ACM Reference Format:
Paul Downen. 2025. Controlling Copatterns: There and Back Again. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3759427.3760362

1 Introduction

Pattern matching—a common feature among functional languages that expresses complex traversals over trees—can be made even more powerful through a modern extension: *copatterns* [1]. The dual to patterns, copatterns let multi-clause definitions match over more information in their calling context, reacting to the structure of projections in addition to the structure of arguments. In contrast to Haskell-style lazy data structures, copatterns are especially useful for modeling and manipulating infinite information like streams in settings that are more sensitive to termination like proof assistants and eager languages.

OLIVIERFEST '25, Singapore, Singapore

ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760362

Compositional copatterns—as recently implemented as a Scheme macro library [16]—extend copatterns with new ways to combine and apply partially-defined, context-sensitive code using a fusion of functional and object-oriented techniques. For example, a (lazy) pair can be represented in Scheme as procedure that takes `'fst` or `'snd` as an argument and returns the respective element. The two-by-two nested pair $((1,2),(3,4))$ can then be defined through multiple equations describing chains of projection like so:

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

Suppose we want to override the diagonal elements—the first of the first, and the second of the second—with new values. The function `(diag x y z)` below returns `x` as the very first, `y` as the very last, and is the same as `z` elsewhere

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

so that `(diag 50 60 quad)` represents $((50,2),(3,60))$. Notice the use of the third "fall-through" case taken whenever the first two cases don't apply. Crucially, the context this fall-through case matches is *less specified* than the earlier cases, so `(((diag 50 60 quad) 'snd) 'fst)` should simplify to `((quad 'snd) 'fst) = 3`. Effectively, the above 3-clause definition of `diag` on two-by-two pairs is short-hand for the following expansion that manually elaborates all four cases:

```
(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)])
```

As the notation suggests, we can understand this code through equational reasoning—replacing call sites matching the left-hand side with the right-hand side. However, it is not so obvious how to convert this into an operational semantics capable of directly calculating the result of any source-level program, without elaborations like the above. This becomes even more challenging when taking into account other forms of higher-order composition that can be performed at run-time: multiple clauses can be stitched together vertically to handle undefined cases, and individual clauses can be extended horizontally to consider more context or side conditions before committing to a right-hand side.

The implementation [16] gives a semantics based on macro expansion into a small subset of Scheme (*i.e.,* λ-calculus), but only describes the semantics via translation to the target language and not in terms of the source-level language itself.

Thankfully, there is a general-purpose technique for converting one form of semantics into another! Danvy's functional correspondence [2] and syntactic correspondence [4] show how to use off-the-shelf program transformations to derive semantic artifacts—operational semantics, abstract machines, and continuation-passing style transformations—from one another in a way that is correct by construction. In other words, we can (mostly mechanically) generate functional and correct operational semantics directly from the macro definition implementing copatterns [16]. Yet, this is not just an exercise of turning the crank. Doing so reveals connections between copattern matching with delimited control and a form of first-class handlers.

Our journey today is a round-trip hike in the semantic park. Starting from a simplified source copattern calculus with a straightforward operational semantics, we derive its corresponding continuation-passing style (CPS) transformation. Then after making a few adjustments generalizing it to match the macro definition found in [16], we turn around and walk backwards to derive a direct-style operational semantics for compositional copattern matching. More specifically, our technical contributions are as follows:

- (Section 3) We derive a trio of semantic artifacts for a monolithic copattern calculus—small-step operational semantics, abstract machine, and continuation-passing style (CPS) translation—through a mechanical derivation based on functional code for copattern matching on contexts and a search to find the next redex.
- (Section 4) We refactor the monolithic copattern calculus into more compositional primitives that give first-class control over (1) the (delimited) calling context and (2) the options for how to handle match failure. The CPS translation corresponds to the macro definition in [16], and shows that the compositional calculus is a conservative extension of the monolithic one.
- (Section 5) Going in reverse, we walk back from the CPS translation of compositional copatterns and derive the missing semantic artifacts—an abstract machine and small-step operational semantics—that correspond by construction to the CPS translation.

The main highlights of these derivations are shown here, illustrating the Haskell code that corresponds to the various semantic artifacts for copattern matching. To find the detailed step-by-step process, which serves as their proof of correctness, see https://github.com/pdownen/derive-copat.

Before diving into the semantics, we begin with an example that illustrates the added expressive power that compositionality gives to copatterns in functional code, by infusing it with some techniques from the object-oriented paradigm.

2 Expressiveness of Copatterns

To illustrate the expressiveness of copatterns, let's consider how they can be used to write an interpreter for arithmetic expressions in Scheme.[1] A simple way to represent expressions is as trees where the nodes start with the (symbolic) name of an operator followed by sub-expressions, and leaves are just number literals. For a simple language, the trees representing addition and multiplication follow the patterns `‘(add ,l ,r)` and `‘(mul ,l ,r)`.[2] The arithmetic interpreter `arith` can then be defined as an object [16] by (co)pattern matching on calls of the form `(arith ‘eval e)` where `‘eval` signals the request to evaluate an expression tree `e`:

```
(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))])

(define expr0 ‘(add 1 (mul 2 3)))
```

The first clause contains a *guard* `try-if` used to check that the variable `n` is bound to a numeric value; if not, the next two clauses are tried, matching the two operators. Following these equations derives `(arith ‘eval expr0) = 7`.

Now, what if we wanted to add a new expression node, such as (unary) negation `‘(neg ,e)`? We can't just define a new function which handles the `‘neg` operation and calls the old one for all other cases, like this *wrong* extension of `arith`:

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

(define expr1 ‘(add 1 (neg (mul 2 3))))
```

Certain special cases will work. Passing in an expression from the old syntax will produce the correct answer—like `(arith-wrong ‘eval expr0) = 7`—or limited uses of negation—like `(arith-wrong ‘eval ‘(neg 5)) = -5`. However, examples where old and new operations are nested within one another produce an error, like `(arith-wrong ‘eval expr1)`.

In a functional language with built-in support for algebraic data types—like Haskell and ML-family languages—we would have to copy-and-paste the old code and add one extra clause for the new case. Instead, with compositional copatterns, we can reuse the old code as-is by composing it with another object that defines the new behavior. A correct way to extend the arithmetic evaluator looks like:

[1] The library defining the Scheme and Racket macros used in these examples can be found at https://github.com/pdownen/CoScheme.

[2] This is *quasiquote* syntax, where the backquote `‘...` means to interpret the expression literally as a data structure—leaving names as quoted symbols—except for internal "unquoting" form `,l` which means to insert the value of `l` in place. When used as a pattern, the form `‘...` matches against a nested tree data structure of that shape, and `,l` inside of a backquote pattern is a wildcard variable that can match any value, which gets bound to `l`.

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

where the tag `compose denotes an implicit inherited method of objects that combines the clauses vertically in an either-or fashion. That way, it correctly calculates answers to both old expressions `(arith-ext `eval expr0) = 7` and new ones `(arith-ext `eval expr1) = -5`. Textually, it is as if the underlying runtime system did the copy-and-pasting for us, behaving exactly like the equivalent expanded definition:[3]

```
(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))])
```

So we can add as many operators as we want to the evaluator without modifying any old code. Fantastic!

However, what if we attempted a more radical change, such as extending arithmetic to algebraic expressions with variables in them? When evaluating a variable, we need access to a mapping from variables to numbers to look up its value. Unfortunately, just composing clauses together vertically will no longer suffice. Typically, we would need to thread this environment around in every other case where it's not needed, requiring careful surgery of the old code.

Instead, we can take a page out of the object-oriented book and think about objects that "encapsulate" additional information within them. For instance, an object might hold onto exactly the variable-to-number environment we need. Fortunately, we already have all the tools at our disposal to get the job done—without needing to introduce the baggage of a whole class system. Rather, we can model lightweight "classes" à la JavaScript as functions (representing the constructor) that return an **`object`**. For our needs, the simplest class of objects containing a given variable-to-number environment is defined by a single accessor method:

```
(define (with-env dict)
  (object [(_ `env) = dict]))
```

So that `((with-env e) `env) = e`. This internal functionality can be composed with our existing arithmetic evaluator and a new clause to `eval that says how to look up a variable (represented as a plain symbol) in an expression tree.

[3]We renamed the recursive evaluator to `arith-ext` in each equation for ease of reading, including the ones that were copied from the original `arith` definition. This is neither necessary nor changes the semantics, because **`define*`** and **`define-object`** are based on *open recursion*, so that the variables used internally for recursive references on the right-hand sides of equations are different from the variable bound for external use by outside code. In fact, the recursive name used to refer back to the object can be different between each line, and be different from the top-level name.

```
(define (alg dict)
  (arith-ext `compose
   (with-env dict)
   (object
    [(self `eval x) (try-if (symbol? x))
     = (dict-ref (self `env) x)])))

(define env-xy `((x . 10) (y . 20)))
(define expr2  `(add x (neg (mul 2 y))))
```

The algebraic evaluator gives the same answers on all the old examples without having to use its environment:

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

But now, it can handle new expressions that contain symbolic variables in them, like

```
((alg env-xy) `eval expr2) = -30
```

which is what we would get from performing the following manual textual revision and inlining:

```
(define (alg dict)
  (object
   [(self `eval n) (try-if (number? n))
    = n]
   [(self `eval `(add ,l ,r))
    = (+ (self `eval l) (self `eval r))]
   [(self `eval `(mul ,l ,r))
    = (* (self `eval l) (self `eval r))]
   [(self `eval `(neg ,e))
    = (- (self `eval e))]
   [(self `env)
    = dict]
   [(self `eval x) (try-if (symbol? x))
    = (dict-ref (self `env) x)]))
```

but all without touching any old code!

3 Deriving Copatterns: A Journey of Small Steps to the Land of Continuations

In order to derive a semantics that can handle the types of open recursion and composition of partial functions from section 2, let's consider a small core calculus of copattern matching in figure 1. This calculus has bound variables (x), applications of arguments (as $M\ N$), and projecting a specific index tagged X (as $M\ X$). The only two other features are:

- *Open recursion:* To simplify the formalization of open-ended, late-bound self-reference, we follow a model close to the Python programming language. The "self" pointer is given explicitly as the first argument, so calling a self-recursive function begins with an explicit punctuation (written $M.$) signaling that a copy of the same function should be passed first (as $M\ M$). The self-duplicating δ rule captures this step.
- *Copattern λs:* Instead of just taking a single parameter, λ-abstractions are built out of a sequence of *options* (O) for mapping *copatterns* (L) on the left-hand side to a new term on the right-hand side. Copatterns are a sequence of distinct parameter abstractions ($x\ L$) and checks against specific index projections ($X\ L$), until

$$
\begin{aligned}
\mathit{Variable} \ni\ & x, y, z ::= \ldots \\
\mathit{Index} \ni\ & X, Y, Z ::= \ldots \\
\mathit{Term} \ni\ & M, N ::= x \mid M\ N \mid M\ X \mid M.\ \mid \lambda\{O \ldots\} \\
\mathit{Option} \ni\ & O ::= L \to M \\
\mathit{Copat} \ni\ & L ::= \varepsilon \mid x\ L \mid X\ L
\end{aligned}
$$

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

Figure 1. Equational specification of monolithic copatterns.

the empty copattern (ε) signals that no more information is needed to decide to return the right-hand side. The resolution of a copattern λ is a complex process, modeled by the β rule, that checks each copattern in turn against the λ's context C. If somehow the λ appears in a context C, and one of its copatterns L_i can match C by substituting for its bound variables, then its right-hand side M_i should be returned under the same substitution, as long as L_i is the first such match.

Example 3.1. To see how this core calculus models recursion and matching, consider this infinite stream object (with a *Head* and *Tail*) that counts *From* an initial number:

$$
\begin{aligned}
\mathit{count} = \lambda\{&\mathit{self}\ \mathit{From}\ x\ \mathit{Head} \to x \\
&\mid \mathit{self}\ \mathit{From}\ x\ \mathit{Tail} \to \mathit{self}.\mathit{From}(\mathit{succ}\ x)\}
\end{aligned}
$$

Starting the *count* from 0 and accessing the third element (via two *Tail*s and a *Head*) shows how δ and β enable recursion:

$$
\begin{aligned}
& \underline{\mathit{count}}.\mathit{From}\ 0\ \mathit{Tail}\ \mathit{Tail}\ \mathit{Head} \\
&= \underline{\mathit{count}\ \mathit{count}\ \mathit{From}\ 0\ \mathit{Tail}}\ \mathit{Tail}\ \mathit{Head} && (\delta) \\
&= \underline{\mathit{count}.\mathit{From}(\mathit{succ}\ 0)}\ \mathit{Tail}\ \mathit{Head} && (\beta) \\
&= \underline{\mathit{count}\ \mathit{count}\ \mathit{From}\ (\mathit{succ}\ 0)\ \mathit{Tail}}\ \mathit{Head} && (\delta) \\
&= \underline{\mathit{count}.\mathit{From}(\mathit{succ}\ (\mathit{succ}\ 0))}\ \mathit{Head} && (\beta) \\
&= \underline{\mathit{count}\ \mathit{count}\ \mathit{From}\ (\mathit{succ}\ (\mathit{succ}\ 0))\ \mathit{Head}} && (\delta) \\
&= (\mathit{succ}\ (\mathit{succ}\ 0)) && (\beta)
\end{aligned}
$$

3.1 Small-step operational semantics

The equational axioms β and δ can specify *why* an answer is correct, but they don't give an algorithmic method showing *how* to get there. So let's write an algorithm!

The first step is to represent syntax trees as a concrete data structure, which we can do in Haskell as shown in figure 2. We use infix constructors `m:*:n` and `m:@:x` for function application ($M\ N$) and indexing ($M\ X$), respectively, and the other constructors are for variables (`Var "x"` as x)

```
data Term i a
  = Var a
  | Term i a :*: Term i a
  | Term i a :@: i
  | Dot (Term i a)
  | Obj [Option i a]

data Option i a = Copattern i a :-> Term i a

data Copattern i a
  = Nop
  | a :* Copattern i a
  | i :@ Copattern i a

instance Semigroup (Copattern i a) where
  Nop       <> q' = q'
  (x :* q) <> q' = x :* (q <> q')
  (i :@ q) <> q' = i :@ (q <> q')
instance Monoid (Copattern i a) where
  mempty = Nop

type Question i a = Copattern i (Term i a)

ask :: Term i a -> Question i a -> Term i a
ask m Nop       = m
ask m (n :* q) = ask (m :*: n) q
ask m (i :@ q) = ask (m :@: i) q

type Env a b = [(a, b)]
type TermEnv i a = Env a (Term i a)
(//)::Eq a=>Term i a->TermEnv i a->Term i a
```

Figure 2. Syntax trees as an algebraic data type.

the "dot" operator (`Dot m` as $M.$) and copattern λ-objects (`Obj [l :-> m, ...]` as $\lambda\{L \to M \mid \ldots\}$). Copatterns are built using similar infix constructors and end with no-op `Nop`, forming a stylized list that we can concatinate using `<>`. Of note, we abstract variable identifiers and projection indexes as generic types `a` and `i`, respectively, which will come in handy several times. A special case of `Copatterns` are `Questions`—contexts that might match copatterns—given by filling the variables (left of `:*`) in a copattern with a `Term i a`. Being contexts, we can plug a term into a question via `ask`.

Second, we have to implement a single step of reduction, which is shown in figure 3. The `reduce` function takes a (potential) `Redex` *and* a `Question` to produce some `Followup` result: either a `Reduct` is asked the next `Question`, or comatching needed more context. The `comatch` function compares the left-hand-side against a question, creating a substitution environment and saying if there is a full match (producing a followup question out of the remaining context), an incomplete match (producing the copattern that continues past the given question), or a mismatch. In the case of a full match, `reduce` will substitute (via `//`) the matching environment into the right-hand side and return the followup question. In the case of an incomplete match, reduction is blocked, and in the case of a mismatch, the next option is tried.

```
data Redex    i a
  = Introspect (Term 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
reduce (Introspect m) q
  = Next (Reduced $ m :*: m) q
reduce (FreeVar x) q
  = Next (Unknown x) q
reduce (Respond (lhs :-> rhs : ops)) q
  = case suffix match of
      Followup q'  -> Next (Reduced rhs') q'
      Unasked lhs' -> More lhs' rhs' ops q
      Mismatch _ _ -> reduce (Respond ops) q
  where match = comatch lhs q
        rhs'  = rhs // prefix match
reduce (Respond []) q
  = Next Unhandled q

data CoMatch   i a b
  = Comatch { prefix :: Env a b,
              suffix :: Remainder i a b }

data Remainder i a b
  = Followup (Copattern i b)
  | Unasked  (Copattern i a)
  | Mismatch (Copattern i a) (Copattern i b)

comatch :: Eq i
        => Copattern i a -> Copattern i b
        -> CoMatch i a b
comatch Nop        cxt
  = Comatch [] (Followup cxt)
comatch lhs        Nop
  = Comatch [] (Unasked lhs)
comatch (x :* lhs) (y :* cxt)
  = Comatch ((x, y) : prefix q) (suffix q)
  where q = comatch lhs cxt
comatch (i :@ lhs) (j :@ cxt)
  | i == j = comatch lhs cxt
comatch lhs        cxt
  = Comatch [] (Mismatch lhs cxt)
```

Figure 3. An implementation of small-step reduction.

Third, we must spell out how to find the next redex in a term. A direct-style `search` function is shown in figure 4, which identifies both a redex as well as the question asked of it. Following the syntactic correspondence methodology [4], we can derive a tail-recursive decomposition function from `search` using standard program transformations: CPS transformation, defunctionalization [26], and compressing corridor transitions (*i.e.,* inlining and simplifying known function calls to partially-known arguments). Along the way, it becomes clear that the evaluation contexts are isomorphic

```
data Found i a
  = Asked (Redex i a) (Question i a)

search :: Term i a -> Found i a
search (Var x)   = Asked (FreeVar x) Nop
search (Dot m)   = Asked (Introspect m) Nop
search (Obj ops) = Asked (Respond ops) Nop
search (m :*: n) = case search m of
  Asked r q -> Asked r $ q <> n :* Nop
search (m :@: i) = case search m of
  Asked r q -> Asked r $ q <> i :@ Nop
```

Figure 4. Searching for the next redex.

```
data Decomp i a = Asked (Redex i a) (Question i a)

recomp :: Term i a -> Question i a -> Term i a
recomp = ask

decomp :: Term i a -> Decomp i a
decomp m = refocus m Nop

refocus :: Term i a -> Question i a -> Decomp i a
refocus (Var x)   k = Asked (FreeVar x) k
refocus (Dot m)   k = Asked (Introspect m) k
refocus (Obj eqs) k = Asked (Respond eqs) k
refocus (m :*: n) k = refocus m $ n :* k
refocus (m :@: i) k = refocus m $ i :@ k
```

Figure 5. Decomposing a term into a redex and question.

```
data Answer i a
  = Under (Copattern i a) (Term i a)
          [Option i a]    (Question i a)
  | Raise (Question i a)
  | Stuck a (Question i a)

eval::(Eq a, Eq i) => Term i a -> Answer i a
eval m = iter $ decomp m

iter::(Eq a, Eq i) => Decomp i a -> Answer i a
iter (Asked r q) = case reduce r q of
  Next (Reduced m) k -> eval $ recomp m k
  Next (Unknown x) k -> Stuck x k
  Next Unhandled   k -> Raise k
  More lhs rhs eqs k -> Under lhs rhs eqs k
```

Figure 6. Functional small-step interpreter loop.

to the `Questions` we need for reduction, which comes from the choice of call-by-name evaluation order. Swapping to the existing representation and applying the monoid laws gives `decomp` in figure 5.

Finally, the small-step interpreter is given in figure 6, which repeatedly decomposes a term, reduces it, recomposes the result, and loops. This algorithm is equivalent to the following relational small-step semantics, presented in terms of *inside out* evaluation contexts and a *comatch* function:

$$\mathit{Cont} \ni K ::= \varepsilon \mid N\ K \mid X\ K$$

```
eval m = refocus m Nop

refocus (Var x)   k = Stuck x k
refocus (Dot m)   k = refocus m $ m :* k
refocus (Obj os)  k = case os of
  []               -> Raise k
  lhs :-> rhs : os -> comatch lhs k rhs os k
refocus (m :*: n) k = refocus m $ n :* k
refocus (m :@: i) k = refocus m $ i :@ k

comatch Nop          cxt       rhs _  _
  = refocus rhs cxt
comatch lhs          Nop       rhs os q
  = Under lhs rhs os q
comatch (x :* lhs) (y :* cxt) rhs os q
  = comatch lhs cxt (rhs // [(x,y)]) os q
comatch (i :@ lhs) (j :@ cxt) rhs os q
  | i == j
  = comatch lhs cxt rhs os q
comatch lhs          cxt       _   os q
  = refocus (Obj os) q
```

Figure 7. Tail-recursive abstract machine interpreter.

$$(\delta) \qquad K[M.] \mapsto K[M\ M]$$

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

3.2 Abstract machine

Continuing on, we can use the syntactic correspondence to transform the direct-style small-step interpreter into a tail-recursive abstract machine. First, we short-cut the recompose-decompose step and instead continue by refocusing in place.

Lemma 3.2. `decomp (recomp m k) = refocus m k`.

Proof. By induction on `k :: Question i a`. □

From there, it is a matter of applying standard program transformations: loop fusion, compressing corridor transitions, and deforesting intermediate data structures. In order to fuse `reduce` and `comatch` in with the small-step interpreter, we need to give reduction the same treatment as `search` to put it into a tail-recursive form: CPS transforming, defunctionalizing [26], and compressing corridor transitions. As a simplification, the contexts produced by defunctionalization are isomorphic to substitution environments, whose order is irrelevant (assuming distinct names). We also modify copattern matching to substitute immediately when available, since substitution reassociates.

Lemma 3.3. `m // (env ++ env') = (m // env) // env'`

Proof. By induction on `m :: Term i a`. □

The end result is the abstract machine interpreter shown in figure 7. We can rephrase this code into a traditional stepping relation using machine states of forms (1) a refocusing or reduction state $\langle M \parallel K \rangle$, or (2) a copattern-matching state $\langle L \parallel K \parallel M \parallel O... \parallel K \rangle$. The initial state for evaluating M is $\langle M \parallel \varepsilon \rangle$, and the final states are one of: (1) stuck on an unknown variable $\langle x \parallel K \rangle$, (2) an unhandled question $\langle \lambda\{\} \parallel K \rangle$, or (3) an underspecified question $\langle L \parallel \varepsilon \parallel M \parallel O... \parallel K \rangle$ where $L \neq \varepsilon$. This gives us the following stepping relation:

- Refocusing steps:

$$\langle M\ N \parallel K \rangle \mapsto \langle M \parallel N\ K \rangle \qquad \langle M\ X \parallel K \rangle \mapsto \langle M \parallel X\ K \rangle$$

- Reduction steps:

$$\langle M. \parallel K \rangle \mapsto \langle M \parallel M\ K \rangle$$
$$\langle \lambda\{L \to M \mid O...\} \parallel K \rangle \mapsto \langle L \parallel K \parallel M \parallel O... \parallel K \rangle$$

- Copattern-matching steps:

$$\langle x\ L \parallel N\ K' \parallel M \parallel O... \parallel K \rangle \mapsto \langle L \parallel K' \parallel M[N/x] \parallel O... \parallel K \rangle$$
$$\langle X\ L \parallel X\ K' \parallel M \parallel O... \parallel K \rangle \mapsto \langle L \parallel K' \parallel M \parallel O... \parallel K \rangle$$
$$\langle \varepsilon \parallel K' \parallel M \parallel O... \parallel K \rangle \mapsto \langle M \parallel K' \rangle$$
$$\langle L \parallel \varepsilon \parallel M \parallel O... \parallel K \rangle \not\mapsto \qquad (\text{if } L \neq \varepsilon)$$
$$\langle L \parallel K' \parallel M \parallel O... \parallel K \rangle \mapsto \langle \lambda\{O...\} \parallel K \rangle \qquad (\text{otherwise})$$

Remark 3.4. Note that this abstract machine inefficiently traverses terms many times to perform substitution. To derive an environment-based machine that more efficiently uses closures and lookup, see appendix section A.

3.3 Continuation-passing style transformation

At the end of our journey, we arrive at a continuation-passing style translation to native Haskell functions, as shown in figure 8. This translation is derived from figure 7 by (1) desugaring nested patterns into flat patterns on a single value, (2) η-reduction, and (3) applying the transition functions as soon as possible. Of note, the code elaborates a detail that is usually suppressed in CPS transformations: when going under a binder, we have to replace a *syntactic name* from the source program with a *semantic denotation* within the CPS. We accommodate this step in the middle of the CPS process by representing variables as either a plain `Name` or one that was `Substituted` by a `CPSTerm`, and should be translated as-is.

To better understand the code, we can elide some of these details in the corresponding CPS translation of terms $[\![M]\!]$, lists of options $[\![O...]\!]$, and individual copattern-matching options $[\![L \to M]\!]$ in a more traditional notation (where **rec** denotes a recursive fixed point to handle the non-structural recursion for under-application in `comatch`):

- Translating terms $[\![M]\!]$:

$$[\![x]\!] = x \qquad [\![M.]\!] = \lambda k.\ [\![M]\!]\ ([\![M]\!], k)$$
$$[\![M\ N]\!] = \lambda k.\ [\![M]\!]\ ([\![N]\!], k) \qquad [\![M\ X]\!] = \lambda k.\ [\![M]\!]\ (X\ k)$$
$$[\![\lambda\{O...\}]\!] = [\![O...]\!]$$

- Translating lists of options $[\![O...]\!]$:

$$[\![\varepsilon]\!] = \lambda k.\ k$$
$$[\![L \to M \mid O...]\!] = \lambda k.\ [\![L \to M]\!]\ k\ [\![O...]\!]\ k$$

```
data Answer i a
  = Under (CPSTerm i a)
  | Raise (CPSQuestion i a)
  | Stuck a (CPSQuestion i a)

type CPSQuestion i a = Copattern i (CPSArg i a)
type CPSTerm i a = CPSQuestion i a -> Answer i a
type CPSOption i a = CPSTerm i a -> CPSTerm i a
type CPSCopat i a = CPSQuestion i a -> CPSOption i a

newtype CPSArg i a = Arg { useArg :: CPSTerm i a }

data CPSVar i a = Name a | Subs (CPSTerm i a)

instance Eq a => Eq (CPSVar i a) where
  Name x == Name y = x == y
  _      == _      = False

eval :: (Eq i, Eq a) => Term i a -> Answer i a
eval m = (term (fmap Name m)) Nop

term :: (Eq i, Eq a) => Term i (CPSVar i a)
     -> CPSTerm i a
term (Var (Name x)) = Stuck x
term (Var (Subs m)) = m
term (Dot m)   = \k -> (term m) (Arg(term m) :* k)
term (Obj os)  = options os
term (m :*: n) = \k -> (term m) (Arg(term n) :* k)
term (m :@: i) = \k -> (term m) (i :@ k)

options :: (Eq i, Eq a) => [Option i (CPSVar i a)]
        -> CPSTerm i a
options [] = Raise
options (lhs :-> rhs : os)
  = \q -> (comatch lhs rhs) q (options os) q

comatch :: (Eq i, Eq a) => Copattern i (CPSVar i a)
        -> Term i (CPSVar i a) -> CPSCopat i a
comatch Nop        rhs = \_ _   -> (term rhs)
comatch (x :* lhs) rhs = \q os -> \case
  (y :* k) -> (comatch lhs (rhs // [(x, n)])) q os k
    where n = Var (Subs (useArg y))
  Nop      -> Under $ (comatch (x :* lhs) rhs) q os
  _        -> os q
comatch (i :@ lhs) rhs = \q os -> \case
  (j :@ k)
  | i == j -> (comatch lhs rhs) q os k
  Nop      -> Under $ (comatch (i :@ lhs) rhs) q os
  _        -> os q
```

Figure 8. Continuation-passing style translation to Haskell.

- Translating one copattern-matching option $[\![L \to M]\!]$:

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

Remark 3.5. Typically, we would keep going and refunctionalize continuations—in this case, `Questions`—into first-class functions. This step becomes difficult in cases like ours, compounded next in section 4, which is not in the image of ordinary defunctionalization. Refunctionalization can be made total by using copattern matching on codata [25]—but that verges on begging the question by defining copatterns in terms of copatterns. But not to worry! The CPS transformation given here corresponds to a well-known and well-behaved one for call-by-name λ-calculus, based on a concrete representation of continuations as pair and sum types [18, 20, 28, 30], so we can stop here.

Because all three semantics have been derived from a single origin using correct program transformations, we now get a theorem about their correspondence that is correct by construction. For simplicity, we single out raising an unanswered question as the main observation of programs.

Theorem 3.6. *The three* `eval` *functions are equal,* i.e., *the following relations between M and K are all equivalent:*

(a) $M \mapsto^* K[\lambda\{L_i \to M_i ...\}]$ *such that, for all i,* $comatch(L_i, K) = Mismatch$.
(b) $\langle M \parallel \varepsilon \rangle \mapsto^* \langle \lambda\{\} \parallel K \rangle$.
(c) $[\![M]\!]\,() \mapsto^* [\![K]\!]$, *where* $[\![K]\!]$ *is*

$$[\![\varepsilon]\!] = () \qquad [\![N\ K]\!] = ([\![N]\!], [\![K]\!]) \qquad [\![X\ K]\!] = X\ [\![K]\!]$$

4 Refactoring Syntax and Semantics: A Short Rest Among the Lambdas

The joy of working with a CPS transformation, like the one we now have, is that we can employ the high-powered theory of the λ-calculus to reason about the semantics of our programming language. The λ-calculus has a thoroughly developed suite of semantic tools to help us prove properties of the transformation, and the denotational style of CPS unlocks many out-of-order simplifications and rewrites for free. Let's now use this ability to refactor our language.

4.1 First refactor: Delimiting the context

One of the awkward aspects of the semantics so far has to do with unresolved matching, when a copattern is expecting more information than the context provides. Currently, there is no way to tell when a question is really "done," or if we are missing some part of the context. As a consequence, the CPS transformation has to handle cases of an empty stack by trying again to consume more continuation. This leads to the complex recursive structure of copatterns (seen in the **rec** fixed point) and confusion between different types of continuations (*e.g.*, $x\ L$ may get a pair or an empty unit).

To resolve these ambiguities, let's add a delimiter to the language, $M!$, that signals the definite end to a question. If copattern matching reaches the final punctuation (!), then it knows there will never be more context arriving and it can move on to the next option. But if $M!$ signals the end

of questioning, we can never interrogate the answer with another question, so where else can the delimiter appear except at the "top" of the whole program?

Giving the program some internal control over delimited questions amounts to a form of call-by-name delimited control. That is to say, a term can abstract over a given question by giving it a name q in $!q \to R$, where R is a *response* that explains how to continue, which could be asking the same question $M \mathbin{!} q$, a different one $M \mathbin{!} q'$, or a now explicitly empty $M \mathbin{!} \varepsilon$. Instead of asking, a response could also raise an unanswered question itself, to be handled at a higher level.

Extending the syntax we have so far, first-class delimited questions have the following grammatical structure:

$$\begin{aligned} \mathit{Response} \ni R &::= q \mid \varepsilon \mid M \mathbin{!} R \\ \mathit{Term} \ni M, N &::= \cdots \mid !q \to R \end{aligned}$$

Whose semantics is given by extending the CPS transform:

$$[\![M \mathbin{!} R]\!] = [\![M]\!]\ [\![R]\!] \qquad [\![\varepsilon]\!] = () \qquad [\![q]\!] = q$$

$$[\![!q \to R]\!] = \lambda q.\ [\![R]\!]$$

Notice how call-by-name delimited questions can compose, but the compositional structure is opposite to call-by-value delimited control like shift and reset [7, 8]. Rather than composing multiple continuations like functions from inputs to outputs, we can instead compose several terms—one after another—as functions from questions to answers that handle the unanswered questions raised by the next one in line:

$$[\![M \mathbin{!} (N \mathbin{!} R)]\!] = [\![M]\!]\ ([\![N]\!]\ [\![R]\!]) \quad [\![M \mathbin{!} \varepsilon]\!] = [\![M]\!]\ ()$$

The term that immediately raises its given question can be expressed $!q \to q$. Later in section 5, it will be useful to have a special form **raise** to denote when this has happened, without having to take any more steps to calculate the response:

$$[\![\mathbf{raise}]\!] = \lambda q.q = [\![!q \to q]\!]$$

4.2 Second refactor: Nesting copatterns

Another source of difficulty is the complex structure of copattern-matching λ-abstractions, which forces a monolithic matching algorithm. In the interest of factoring out individual aspects of copattern matching, we will decompose the copattern options into smaller parts. The first step is to reassociate copatterns to the right, taking them one step at a time in the style of currying, for example replacing $(x\ L) \to M$ with $x \to (L \to M)$. The second step is to get a handle on how a match failure should proceed, since every copattern option needs to do something if it can't respond to its context. We write $O \mathbin{?} M$ to mean that O is the first option to answer the context and, if it fails, the term proceeds as M in the *same* context. Dually, the option needs a way to signal success when the right-hand side is reached, which we write $?x \to N$ to mean that N is returned to this context, and the (now untaken) failure alternative is bound to x in case the program needs to invoke it manually.

Nested options have the following grammatical structure:

$$\begin{aligned} \mathit{Term} \ni M, N &::= \cdots \mid O \mathbin{?} M \\ \mathit{Option} \ni O &::= x \to O \mid X \to O \mid ?x \to M \end{aligned}$$

On the one hand, we can understand the new forms (besides $?x \to M$) in terms of the old ones:

$$\begin{aligned} O \mathbin{?} M &= \lambda\{O \mid \varepsilon \to M\} \\ x \to (L \to M) &= (x\ L) \to M \\ X \to (L \to M) &= (X\ L) \to M \end{aligned}$$

On the other hand, we can decompose the monolithic syntax into smaller, nested pieces:

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

More interestingly, we now have enough flexibility over options and their failure modes to encode dynamic composition of copattern matching, as used by the arithmetic evaluator in section 2. Importantly, we can express vertical composition of cases with a special method (here, *Open*) as:

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

so that *compose* $\texttt{object}\{O\}\ \texttt{object}\{O'\} = \texttt{object}\{O \mid O'\}$.

4.3 Third refactor: Eliminating redundancy

At this point, there is some redundancy in the way the CPS translation handles options. Deriving a CPS transformation (here named *Opt*) for the option handler $O \mathbin{?} M$ gives:

$$\mathit{Opt}[\![O \mathbin{?} M]\!] = \lambda k.\ \mathit{Opt}[\![O]\!]\ k\ [\![M]\!]\ k$$

The given continuation is used twice: once to be analyzed for copattern matching, and a copy to revert back on a failure so that M can start again from the original k. However, it would be cleaner to just pass the continuation once like so (naming the new transformation *Opt′* to disambiguate):

$$\mathit{Opt}'[\![O \mathbin{?} M]\!] = \lambda k.\ \mathit{Opt}'[\![O]\!]\ [\![M]\!]\ k$$

We can get away without the copy by modifying the failure term on each step of copattern matching, in a way that restores the original structure of the continuation as follows:

$$\begin{aligned} \mathit{Opt}'[\![x \to O]\!] &= \lambda f.\lambda k.\ \mathbf{case}\ k\ \mathbf{of} \\ &\qquad (x, k') \to \mathit{Opt}'[\![O]\!]\ (\lambda q.\ f\ (x, q))\ k' \\ &\qquad k \to f\ k \\ \mathit{Opt}'[\![X \to O]\!] &= \lambda f.\lambda k.\ \mathbf{case}\ k\ \mathbf{of} \\ &\qquad (X\ k') \to \mathit{Opt}'[\![O]\!]\ (\lambda q.\ f\ (X\ q))\ k' \\ &\qquad k \to f\ k \\ \mathit{Opt}'[\![?x \to M]\!] &= \lambda x.[\![M]\!] \end{aligned}$$

Even though the failure handler f is called with a different continuation, the result is the same as before.

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

Translation of results $[\![R]\!]$:

$$[\![M \mathbin{!} R]\!] = \lambda s.\, [\![R]\!]\ \lambda q.\, [\![M]\!]\ q\ s \qquad [\![q]\!] = \lambda s.\, s\ q \qquad [\![\varepsilon]\!] = \lambda s.\, s\ ()$$

Translation of terms $[\![M]\!]$:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![M\ X]\!] &= \lambda k.\, [\![M]\!]\ (X\ k) \\
[\![M\ N]\!] &= \lambda k.\, [\![M]\!]\ ([\![N]\!], k) \\
[\![M.]\!] &= \lambda k.\, [\![M]\!]\ ([\![M]\!], k) \\
[\![\mathbf{raise}]\!] &= \lambda k.\, \lambda s.s\ k \\
[\![O \mathbin{?} M]\!] &= \lambda k.\, [\![O]\!]\ [\![M]\!]\ k \\
[\![{!}q \to R]\!] &= \lambda q.\, [\![R]\!]
\end{aligned}
$$

Translation of options $[\![O]\!]$:

$$
\begin{aligned}
[\![x \to O]\!] &= \lambda f.\lambda k.\mathbf{case}\ k\ \mathbf{of}\ (x, k') \to [\![O]\!]\ (\lambda q.\, f\ (x, q))\ k' \\
&\qquad\qquad\quad k \qquad\quad \to f\ k \\
[\![X \to O]\!] &= \lambda f.\lambda k.\mathbf{case}\ k\ \mathbf{of}\ (X\ k') \to [\![O]\!]\ (\lambda q.\, f\ (X\ q))\ k' \\
&\qquad\qquad\quad k \qquad\quad \to f\ k \\
[\![{?}x \to M]\!] &= \lambda x.\, [\![M]\!]
\end{aligned}
$$

Figure 9. Refactored CPS and calculus of nested copatterns.

Lemma 4.1.

$$Opt[\![O]\!]\ (q \diamond k)\ f\ k = Opt'[\![O]\!]\ (\lambda k'.\, f\ (q \diamond k'))\ k$$

Proof. By induction on L and cases on k. □

Corollary 4.2. $Opt[\![O]\!]\ k\ f\ k = Opt'[\![O]\!]\ f\ k$.

4.4 Fourth refactor: Fully continuation-passing style

The last small detail revolves around the fact that the CPS transformation is no longer strictly in CPS form, due to responses like $M \mathbin{!} (N \mathbin{!} R)$, which gets transformed to an application of $[\![M]\!]$ to the non-value $[\![N]\!]\ [\![R]\!]$. But thankfully there is an easy fix to get back into strict CPS: iterate CPS again [8]! This gives another layer of continuation for responses to elaborate the evaluation order of $M \mathbin{!} R$ to say R is evaluated first, and on a return its answer is passed to M. The only affected cases of the transformation are:

$$[\![M \mathbin{!} R]\!] = \lambda s.[\![R]\!]\ \lambda k.[\![M]\!]\ k\ s \qquad [\![q]\!] = \lambda s.s\ q \qquad [\![\varepsilon]\!] = \lambda s.s\ ()$$

$$[\![\mathbf{raise}]\!] = \lambda k.\lambda s.s\ k$$

This last refactoring gives the final syntax and CPS transformation of the compositional copattern calculus, which is shown in its entirety in figure 9.

5 Controlling Copatterns: The Return Voyage Back to Direct Style

Having made the journey deriving semantics of monolithic copatterns from small-step operational semantics to abstract machine to continuation-passing style, we now aim to derive the semantics of compositional copatterns in reverse.

```
data Response i a
  = Splat a
  | End
  | Term i a :!: Response i a

data Term i a
  = Var a
  | Term i a :*: Term i a
  | Term i a :@: i
  | Dot (Term i a)
  | Option i a :?: Term i a
  | a :!-> Response i a
  | Raise

data Option i a
  = a :*-> Option i a
  | i :@-> Option i a
  | a :?-> Term i a
```

Figure 10. Data type representing refactored syntax trees.

5.1 Continuation-passing style

We begin with the continuation-passing style transformation from figure 9 that defines the semantics. The high-level specification of the transformation can be given a concrete Haskell implementation shown in figures 10 and 11.

5.2 Abstract machine

The continuation-passing style transformation can be modified in several standard steps: (1) defunctionalization, (2) waiting to apply the transformation function until the last moment, (3) syntactically inlining the semantics for substituted variables, (4) fusing substitution inlining with transformation, (5) η-expansion, and (6) rewriting chains of case-analysis as nested patterns. The result of these program transformations on the CPS implementation gives the tail-recursive abstract machine shown in figure 12. This implementation can be rephrased into a traditional presentation of a stepping relation on machine states:

- Delimiting steps:

$$\langle M \mathbin{!} R \parallel S\rangle \mapsto \langle R \parallel M; S\rangle \qquad \langle \varepsilon \parallel M; S\rangle \mapsto \langle M \parallel \varepsilon \parallel S\rangle$$

- Refocusing steps:

$$
\begin{aligned}
\langle M\ X \parallel K \parallel S\rangle &\mapsto \langle M \parallel X\ K \parallel S\rangle \\
\langle M\ N \parallel K \parallel S\rangle &\mapsto \langle M \parallel N\ K \parallel S\rangle \\
\langle O \mathbin{?} M \parallel K \parallel S\rangle &\mapsto \langle O \parallel M \parallel K \parallel S\rangle
\end{aligned}
$$

- Reduction steps:

$$
\begin{aligned}
\langle M. \parallel K \parallel S\rangle &\mapsto \langle M \parallel M\ K \parallel S\rangle \\
\langle \mathbf{raise} \parallel K \parallel M; S\rangle &\mapsto \langle M \parallel K \parallel S\rangle \\
\langle {!}q \to R \parallel K \parallel S\rangle &\mapsto \langle R[K[\mathbf{raise}] \mathbin{!} \varepsilon/q] \parallel S\rangle
\end{aligned}
$$

```
data Answer i a
  = Final   (CPSQuestion i a)
  | Stuck   [CPSTerm i a] a (CPSQuestion i a)
  | CoStuck [CPSTerm i a] a

run :: (Eq a, Eq i) => Response i a -> Answer i a
run r = (response (fmap Name r))

eval :: (Eq i, Eq a) => Term i a -> Answer i a
eval m = (term (fmap Name m)) Nop

try :: (Eq i, Eq a) => Option i a -> Answer i a
try o = (option (fmap Name o)) (term Raise) Nop

response :: (Eq i, Eq a) => Response i (CPSVar i a)
         -> Answer i a
response (Splat (Name k)) = CoStuck [] k
response (Splat (CPSQ q)) = Final q
response (Splat (CPST _)) = error "Illegal expr"
response (End)     = Final Nop
response (m :!: r) = (term m) <!> (response r)

(<!>) :: CPSTerm i a -> Answer i a -> Answer i a
f <!> Final q     = f q
f <!> Stuck gs x q = Stuck (f:gs) x q
f <!> CoStuck gs q = CoStuck (f:gs) q

term :: (Eq i, Eq a) => Term i (CPSVar i a)
     -> CPSTerm i a
term (Var (Name x)) = Stuck [] x
term (Var (CPST m)) = m
term (Var (CPSQ _)) = error "Illegal expr"
term (Dot m)    = \k -> (term m) (Arg(term m):*k)
term (m :*: n)  = \k -> (term m) (Arg(term n):*k)
term (m :@: i)  = \k -> (term m) (i :@ k)
term (Raise)    = \k -> Final k
term (q :!-> r) = \k -> (response r')
  where r' = r /!/ [(q, subQ k)]
term (o :?: m)  = \k -> (option o) (term m) k

option :: (Eq a, Eq i) => Option i (CPSVar i a)
       -> CPSOption i a
option (x :?-> m) = \f -> (term m')
  where m' = m // [(x, subT f)]
option (x :*-> o) = \f -> \case
  (y :* k) -> (option o') (f . (y :*)) k
  k        -> f k
  where o' = o /?/ [(x, subT $ useArg y)]
option (i :@-> o) = \f -> \case
  (j :@ k) | i == j -> (option o) (f . (i :@)) k
  k                 -> f k

subT m = TSub (Var (CPST m))
subQ k = RSub (Splat (CPSQ k))
```

Figure 11. Continuation-passing style translation of copatterns with nested options and control into Haskell.

- Copattern matching steps:

$$\langle x \to O \parallel M \parallel N\ K \parallel S \rangle \mapsto \langle O[N/x] \parallel M\ N \parallel K \parallel S \rangle$$
$$\langle X \to O \parallel M \parallel X\ K \parallel S \rangle \mapsto \langle O \parallel M\ X \parallel K \parallel S \rangle$$
$$\langle ?x \to N \parallel M \parallel K \parallel S \rangle \mapsto \langle N[M/x] \parallel K \parallel S \rangle$$
$$\langle O \parallel M \parallel K \parallel S \rangle \mapsto \langle M \parallel K \parallel S \rangle \quad \text{(otherwise)}$$

Remark 5.1. Similar to theorem 3.4, we can derive a more efficient environment-based machine by starting with a CPS

```
data Answer i a
  = Final   (Cont i a)
  | Stuck   (MetaCont i a) a (Cont i a)
  | CoStuck (MetaCont i a) a

type Cont     i a = Question i a
type MetaCont i a = [Term i a]

subQ :: Question i a -> TRSub i a
subQ k = RSub (Raise `ask` k :!: End)

run :: (Eq a, Eq i) => Response i a
    -> Answer i a
run r = delim r []

eval :: (Eq i, Eq a) => Term i a
     -> Answer i a
eval m = refocus m Nop []

try :: (Eq i, Eq a) => Option i a
    -> Answer i a
try o = comatch o Raise Nop []

delim :: (Eq a, Eq i)
      => Response i a -> MetaCont i a
      -> Answer i a
delim (Splat k) s     = CoStuck s k
delim (End)     []    = Final Nop
delim (End)     (m:s) = refocus m Nop s
delim (m :!: r) s     = delim r (m : s)

refocus :: (Eq i, Eq a)
        => Term i a -> Cont i a -> MetaCont i a
        -> Answer i a
refocus (Var x)    k s = Stuck s x k
refocus (Dot m)    k s = refocus m (m:*k) s
refocus (m :*: n)  k s = refocus m (n:*k) s
refocus (m :@: i)  k s = refocus m (i:@k)s
refocus (q :!-> r) k s = delim r' s
  where r' = r /!/ [(q, subQ k)]
refocus (o :?: m)  k s = comatch o m k s
refocus (Raise)    k (m:s) = refocus m k s
refocus (Raise)    k []    = Final k

comatch :: (Eq a, Eq i)
        => Option i a -> Term i a
        -> Cont i a -> MetaCont i a
        -> Answer i a
comatch (x :?-> n) m k      = refocus n' k
  where n' = n // [(x, TSub m)]
comatch (x :*-> o) m (n:*k) = comatch o' (m:*:n) k
  where o' = o /?/ [(x, TSub n)]
comatch (i :@-> o) m (j:@k)
  | i == j                  = comatch o (m:@:i) k
comatch o          m k      = refocus m k
```

Figure 12. Abstract machine for controlling copatterns.

that threads a substitution environment to lookup variables. This machine is discussed in appendix section A.

5.3 Small-step operational semantics

The hardest step of the journey is to derive a small-step operational semantics from the abstract machine. This requires undoing several steps (fusion, deforesting) which destroy information. However, having already completed the easier direction for a similar calculus, we have the advantage of

```
data CoFrame i a = Arg a | At i

data CoObject i a
  = CoO { coframe :: CoFrame i a,
          success :: Option i a,
          failure :: Term i a }

data RxTerm i a
  = FreeVar a
  | Introspect (Term i a)
  | Try a (Term i a) (Term i a)
  | Pop (CoObject i a) (Term i a)
  | Get (CoObject i a) i

data RdTerm i a
  = RdT (Term i a)
  | UnknownA a

reduce :: (Eq i, Eq a) => RxTerm i a
       -> RdTerm i a
reduce (Introspect m) = RdT $ m :*: m
reduce (Try x n m) = RdT $ n // [(x, TSub m)]
reduce (Pop (CoO (Arg x) o m) n)
                   = RdT $ o' :?: (m :*: n)
  where o' = o /?/ [(x, TSub m)]
reduce (Pop o n)   = RdT $ failure o :*: n
reduce (Get (CoO (At i) o m) j)
  | i == j         = RdT $ o :?: (m :@: i)
reduce (Get o i)   = RdT $ failure o :@: i
reduce (FreeVar x) = UnknownA x

data RxResponse i a
  = FreeCoVar a
  | Reset (Term i a) (Question i a)
  | Shift a (Response i a) (Question i a)
  | Under (CoObject i a)

data RdResponse i a
  = RdR (Response i a)
  | UnknownQ a

handle :: Eq a => RxResponse i a
       -> RdResponse i a
handle (FreeCoVar k) = UnknownQ k
handle (Reset m q)   = RdR $ m`ask`q:!:End
handle (Shift k r q) = RdR $ r/!/[(k, subQ q)]
handle (Under o)     = RdR $ failure o:!:End

subQ :: Question i a -> TRSub i a
subQ k = RSub (Raise `ask` k :!: End)
```

Figure 13. Functional small-step reduction of copatterns with delimited control.

knowing something about the overall structures we should be looking for. First, identifying which steps are associated with reduction (the ones that can delete or duplicate information), we can factor out a non-recursive `reduce` function that turns redexes into reducts on `Terms`, along with a similar `handle` function on `Responses`, shown in figure 13. Next, by identifying the remaining steps that are purely refocusing (the ones that are perfectly reversible), we can factor out a set of decomposition functions that work through delimiters as shown in figure 14. Finally, we can de-optimize refocusing in terms of decompose-recompose, and catching

```
data Delimit i a
  = Around (RxTerm i a) (Question i a) [Term i a]
  | Caught (RxResponse i a) [Term i a]
  | Uncaught (Question i a)

delimit :: Response i a -> Delimit i a
delimit r = delim r []

unwind :: [Term i a] -> Response i a
       -> Response i a
unwind []    r = r
unwind (m:s) r = m :!: r

delim :: Response i a -> [Term i a]
      -> Delimit i a
delim (m :!: r) s = delim r (m : s)
delim (End) (m:s) = catch (refocus m Nop) s
delim (End) []    = Uncaught Nop
delim (Splat k) s = Caught (FreeCoVar k) s

catch :: Decomp i a -> [Term i a] -> Delimit i a
catch (Internal r q) s = Around r q s
catch (External r)   s = Caught r s
catch (Raised q) (m:s) = Caught (Reset m q) s
catch (Raised q) []    = Uncaught q

data Decomp i a
  = Internal (RxTerm i a) (Question i a)
  | External (RxResponse i a)
  | Raised   (Question i a)

decomp :: Term i a -> Decomp i a
decomp m = refocus m Nop

recomp :: Question i a -> Term i a -> Term i a
recomp q m = m `ask` q

refocus :: Term i a -> Question i a -> Decomp i a
refocus (m :*: n)  k = refocus m (n :* k)
refocus (m :@: i)  k = refocus m (i :@ k)
refocus (o :?: m)  k = decide (consider o m) k
refocus (Dot m)    k = Internal (Introspect m) k
refocus (q :!-> r) k = External (Shift q r k)
refocus (Raise)    k = Raised k
refocus (Var x)    k = Internal (FreeVar x) k

data Consider i a
  = Inward a (Term i a) (Term i a)
  | Outward (CoObject i a)

only :: Option i a -> Consider i a
only o = consider o Raise

consider :: Option i a -> Term i a
         -> Consider i a
consider (x :?-> n) m = Inward x n m
consider (x :*-> o) m = Outward $ CoO (Arg x) o m
consider (i :@-> o) m = Outward $ CoO (At i) o m

decide :: Consider i a -> Question i a
       -> Decomp i a
decide (Outward o)    = comatch o
decide (Inward x n m) = Internal (Try x n m)

comatch :: CoObject i a -> Question i a
        -> Decomp i a
comatch o (n :* k) = Internal (Pop o n) k
comatch o (j :@ k) = Internal (Get o j) k
comatch o Nop      = External $ Under o
```

Figure 14. Decomposition of terms with delimited questions.

```
data Answer i a
  = Final   (Question i a)
  | Stuck   [Term i a] a (Question i a)
  | CoStuck [Term i a] a

try :: (Eq i, Eq a) => Option i a -> Answer i a
try o = eval $ o :?: Raise

eval :: (Eq i, Eq a) => Term i a -> Answer i a
eval m = run $ m :!: End

run :: (Eq a, Eq i) => Response i a -> Answer i a
run r = case delimit r of
  Around r q s -> case reduce r of
    UnknownA x -> Stuck s x q
    RdT m      -> run $ unwind s $
                  recomp q m :!: End
  Caught r s   -> case handle r of
    UnknownQ k -> CoStuck s k
    RdR r      -> run $ unwind s r
  Uncaught q   -> Final q
```

Figure 15. Direct-style, delimited small-step interpreter.

delimited responses in terms of delimit-unwind. This gives us the direct-style, small-step operational interpreter shown in figure 15. The Haskell implementation can be reinterpreted as the following corresponding small-step reduction relation:

$$\begin{aligned} \mathit{CoObj} \ni P &::= x \to O \mid X \to O \\ \mathit{DelimCxt} \ni D &::= \Box \mid M \mathbin{!} D \\ \mathit{EvalCxt} \ni E &::= \Box \mid E\ N \mid E\ X \end{aligned}$$

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

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

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

Again, we get an analogous correspondence from the semantic derivations as before in theorem 3.6. Delimited questions and **raise** make it more clear that the observable results of responses are unanswered questions. For simplicity, we focus on non-empty questions which must be explicitly **raise**d.

Theorem 5.2. *The three* `eval` *functions are equal,* i.e., *the following relations between R and $K \neq \varepsilon$ are all equivalent:*

(a) $R \mapsto^* K[\mathbf{raise}]!$.
(b) $\langle R \parallel \varepsilon \rangle \mapsto^* \langle \mathbf{raise} \parallel K \parallel \varepsilon \rangle$.
(c) $[\![R]\!]\,(\lambda k.k) \mapsto^* [\![K]\!]$.

6 Related Work

Copatterns. Our starting point comes from a macro implementation in Scheme and Racket [16], where we are primarily concerned with specifying the behavior of different dimensions of compositionality. Alternative macro implementations of copatterns have been given for OCaml [21, 22], which leverage different restrictions to aid code generation. Copatterns have also seen use in proof assistants like Agda [5] which use a type-driven approach to elaboration [27, 29].

Functional and syntactic correspondence. The functional and syntactic correspondence between semantic artifacts [2, 4, 6, 9–11] is based on the approach of definitional interpreters [26] and a long history of semantics-preserving program transformations. It has been especially useful for studying the semantics of call-by-need evaluation [12–14] and its connection to sequent [3] and process calculi [17].

Delimited control. The CPS semantics of delimited copattern matching is similar to delimited control, specifically shift and reset [7, 8]. We use a call-by-name semantics for a close connection between evaluation contexts and copatterns. A similar approach to call-by-name delimited control [19] is related to shift0 [15], a powerful variant of shift [23, 24].

7 Conclusion

Now at the end of our round-trip journey, the disciplined approach to deriving semantic artifacts has been a powerful methodology for understanding complex programming features. The ability to generate CPS transformations is especially useful to explore and refactor the language design space, and coming back gives tools to understand source programs directly. These artifacts still have untapped potential to explore for understanding copattern and program composition, such as extracting a type system from the CPS [7].

Acknowledgments

I want to thank Olivier Danvy for so generously devoting his time, encouragement, and teaching while I was still an early Ph.D. student. While on an extended visit to the University of Oregon, Olivier gave a week-long hands-on tutorial on his technique for inter-deriving semantics. Soon thereafter, I realized this was the perfect solution to a difficult problem on the semantics of the sequent calculus that had been lingering for over a year, which directly led to my second publication [3]. Now, many years later, I had been puzzling over a similar problem of trying to capture the direct-style operational semantics for composing copatterns, which eluded me for quite some time. Only after thinking of how I could contribute to OlivierFest, did I realize "Aha! This is the perfect problem to fix with the technique Olivier taught me!" This paper is a celebration and revival of that early influence.

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

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 27–38. doi:10.1145/2429069.2429075

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. ACM, New York, NY, USA, 8–19. doi:10.1145/888251.888254

[3] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. 2012. Classical Call-by-Need Sequent Calculi: The Unity of Semantic Artifacts. In *Functional and Logic Programming: 11th International Symposium*. Vol. 7294. Springer Berlin Heidelberg, Berlin, Heidelberg, 32–46. doi:10.1007/978-3-642-29822-6_6

[4] Magorzata Biernacka and Olivier Danvy. 2007. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science* 375, 1–3 (April 2007), 76–108. doi:10.1016/j.tcs.2006.12.028

[5] Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 75 (2018), 30 pages. doi:10.1145/3236770

[6] Olivier Danvy. 2008. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. ACM, New York, NY, USA, 131–142. doi:10.1145/1411204.1411206

[7] Olivier Danvy and Andrzej Filinski. 1989. *A Functional Abstraction of Typed Contexts*. Technical Report 89/12. DIKU, University of Copenhagen, Copenhagen, Denmark.

[8] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, New York, NY, USA, 151–160. doi:10.1145/91556.91622

[9] Olivier Danvy and Jacob Johannsen. 2010. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. System Sci.* 76, 5 (Aug. 2010), 302–323. doi:10.1016/j.jcss.2009.10.004

[10] Olivier Danvy, Jacob Johannsen, and Ian Zerny. 2011. A walk in the semantic park. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*. ACM, New York, NY, USA, 1–12. doi:10.1145/1929501.1929503

[11] Olivier Danvy and Kevin Millikin. 2008. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Inform. Process. Lett.* 106, 3 (April 2008), 100–109. doi:10.1016/j.ipl.2007.10.010

[12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. 2010. Defunctionalized interpreters for call-by-need evaluation. In *Proceedings of the 10th International Conference on Functional and Logic Programming* (Sendai, Japan) *(FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 240–256. doi:10.1007/978-3-642-12251-4_18

[13] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. 2012. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science* 435 (June 2012), 21–42. doi:10.1016/j.tcs.2012.02.023

[14] Olivier Danvy and Ian Zerny. 2013. A synthetic operational account of call-by-need evaluation. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming* (Madrid, Spain) *(PPDP '13)*. ACM, New York, NY, USA, 97–108. doi:10.1145/2505879.2505898

[15] Paul Downen and Zena M. Ariola. 2014. Compositional Semantics for Composable Continuations: From Abortive to Delimited Control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 109–122. doi:10.1145/2628136.2628147

[16] Paul Downen and Adriano Corbelino II. 2025. CoScheme: Compositional Copatterns in Scheme. In *International Symposium on Trends in Functional Programming*. Springer, 37 pages.

[17] Paul Downen, Luke Maurer, Zena M. Ariola, and Daniele Varacca. 2014. Continuations, Processes, and Sharing. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* (Canterbury, United Kingdom) *(PPDP '14)*. ACM, New York, NY, USA, 69–80. doi:10.1145/2643135.2643155

[18] Ken-Etsu Fujita. 2003. A sound and complete CPS-translation for $\lambda\mu$-calculus. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications* (Valencia, Spain) *(TLCA'03)*. Springer-Verlag, Berlin, Heidelberg, 120–134. doi:10.1007/3-540-44904-3_9

[19] Hugo Herbelin and Silvia Ghilezan. 2008. An Approach to Call-by-Name Delimited Continuations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 383–394. doi:10.1145/1328438.1328484

[20] Martin Hofmann and Thomas Streicher. 1997. Continuation models are universal for lambda-mu-calculus. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, USA, 387. doi:10.1109/LICS.1997.614964

[21] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae* 150, 3 (2017), 347–377. doi:10.3233/FI-2017-1473

[22] Paul Laforgue and Yann Régis-Gianas. 2017. Copattern matching and first-class observations in OCaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming* (Namur, Belgium) *(PPDP '17)*. ACM, New York, NY, USA, 97–108. doi:10.1145/3131851.3131869

[23] Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, New York, NY, USA, 81–93. doi:10.1145/2034773.2034786

[24] Marek Materzok and Dariusz Biernacki. 2012. A Dynamic Interpretation of the CPS Hierarchy. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 296–311. doi:10.1007/978-3-642-35182-2_21

[25] Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. Automatic refunctionalization to a language with copattern matching: with applications to the expression problem. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 269–279. doi:10.1145/2784731.2784763

[26] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. ACM, New York, NY, USA, 717–740. doi:10.1145/800194.805852

[27] Anton Setzer, Andreas Abel, Brigitte Pientka, and David Thibodeau. 2014. Unnesting of Copatterns. In *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, Vienna, Austria, July 14-17, 2014. Proceedings*, Vol. 8560. Springer, 31–45. doi:10.1007/978-3-319-08918-8_3

[28] Th. Streicher and B. Reus. 1998. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming* 8, 6 (Nov. 1998), 543–572. doi:10.1017/S0956796898003141

[29] David Thibodeau. 2015. *Programming Infinite Structures using Copatterns*. Master's thesis. School of Computer Science, Mcgill University, Montreal.

[30] Hayo Thielecke. 2004. Answer Type Polymorphism in Call-by-Name Continuation Passing. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 279–293. doi:10.1007/978-3-540-24725-8_20

```
type ClosEnv i a = Env a (Closure i a)
type ClosQuestion i a
  = Copattern i (Closure i a)
data Closure i a
  = (:/:) { openTerm  :: Term i a,
            staticEnv :: ClosEnv i a }

data Redex i a
  = Introspect (Term i a) (ClosEnv i a)
  | Respond [Option i a] (ClosEnv i a)
  | FreeVar a (ClosEnv i a)

data Reduct i a
  = Reduced (Closure i a)
  | Unhandled
  | Unknown a

reduce :: (Eq i, Eq a)
       => Redex i a -> ClosQuestion i a
       -> Followup i a
reduce (Introspect m env) q
  = Next (Reduced $ m :*: m :/: env) q
reduce (FreeVar x env)    q
  = case lookup x env of
  Nothing -> Next (Unknown x) q
  Just m  -> Next (Reduced m) q
reduce (Respond (lhs :-> rhs : ops) env) q
  = case suffix match of
      Followup q'  ->
        Next (Reduced $ rhs:/:env'++env) q'
      Unasked lhs' ->
        More lhs' (rhs:/:env') ops env q
      Mismatch _ _ ->
        reduce (Respond ops env) q
  where match = comatch lhs q
        env'  = prefix match
reduce (Respond [] env)   q
  = Next Unhandled q

data Decomp i a
  = Asked (Redex i a) (ClosQuestion i a)

decomp  :: Closure i a -> Decomp i a
recomp  :: Term i a -> Question i a -> Term i a
refocus :: Closure i a -> ClosQuestion i a
        -> Decomp i a

eval :: (Eq a, Eq i) => Term i a -> Answer i a
eval m = iter $ decomp (m :/: [])

iter :: (Eq a, Eq i) => Decomp i a -> Answer i a
iter (Asked r q) = case reduce r q of
  Next (Reduced m)     k -> iter $ refocus m k
  Next (Unknown x)     k -> Stuck x k
  Next Unhandled       k -> Raise k
  More lhs rhs ops env k -> Under lhs rhs ops env k
```

Figure 16. Small-step reduction with an environment

```
data Answer i a
  = Under (Copattern i a) (Closure i a)
          [Option i a] (ClosEnv i a)
          (ClosQuestion i a)
  | Raise (ClosQuestion i a)
  | Stuck a (ClosQuestion i a)

eval :: (Eq a, Eq i) => Term i a -> Answer i a
eval m = refocus m [] Nop

refocus :: (Eq a, Eq i) => Term i a
        -> ClosEnv i a -> ClosQuestion i a
        -> Answer i a
refocus (Var x)   env k = case lookup x env of
  Nothing -> Stuck x k
  Just (m :/: env)  -> refocus m env k
refocus (Dot m)   env k
  = refocus m env $ (m :/: env) :* k
refocus (Obj os)  env k = case os of
  lhs :-> rhs : os -> comatch lhs k [] rhs os env k
  []               -> Raise k
refocus (m :*: n) env k
  = refocus m env $ (n :/: env) :* k
refocus (m :@: i) env k
  = refocus m env $ i :@ k

comatch :: (Eq a, Eq i) => Copattern i a
        -> ClosQuestion i a -> ClosEnv i a
        -> Term i a -> [Option i a] -> ClosEnv i a
        -> ClosQuestion i a
        -> Answer i a
comatch Nop      cxt      env' rhs _   env _
  = refocus rhs (env' ++ env) cxt
comatch lhs      Nop      env' rhs os env q
  = Under lhs (rhs :/: env') os env q
comatch (x:*lhs) (y:*cxt) env' rhs os env q
  = comatch lhs cxt ((x,y):env') rhs os env q
comatch (i:@lhs) (j:@cxt) env' rhs os env q
  | i == j = comatch lhs cxt env' rhs os env q
comatch lhs      cxt      _    _   os env q
  = refocus (Obj os) env q
```

Figure 17. Environment-passing, tail-recursive abstract machine interpreter.

A Postscript: Efficiently Passing Environments

The abstract machines derived in sections 3 and 5 are based on substitution, which is a correct but notoriously slow implementation of static binding. A more efficient implementation technique is to explicitly thread environments through the machine states and form closures when necessary to correctly implement static scope. These kind of environment-based implementations are standard practice, but correctly managing static scope through closures can be tricky, its correctness is not as obvious.

In the context of the derivations we have done so far, we could treat addition of environments and closures to an abstract machine as a complex monolithic program transformation. Instead, here we stay within the incremental style, and perform a small, but obvious transformation at the right level of abstraction that makes environment-passing straightforward. Then, turning the crank in the same way as before will mechanically generate a more efficient abstract machine with more confidence that it is correct by construction.

A.1 Closing over monolithic copatterns

In order to thread environments efficiently, we start from the very beginning with the small-step semantics. The main

```
data Answer i a
  = Final   (CPSQuestion i a)
  | Stuck   [CPSTerm i a] a (CPSQuestion i a)
  | CoStuck [CPSTerm i a] a

type CPSQuestion i a = Copattern i (CPSArg i a)
type CPSResponse i a = Answer i a
type CPSTerm i a = CPSQuestion i a -> CPSResponse i a
type CPSOption i a = CPSTerm i a -> CPSTerm i a

newtype CPSArg i a
  = Arg { useArg :: CPSTerm i a }

data CPSSub i a = CPST (CPSTerm i a)
                | CPSQ (CPSQuestion i a)
type CPSEnv i a = Env a (CPSSub i a)

run :: (Eq a, Eq i) => Response i a -> Answer i a
run r = (response r [])

eval :: (Eq i, Eq a) => Term i a
     -> Answer i a
eval m = (term m []) Nop

try :: (Eq i, Eq a) => Option i a -> Answer i a
try o = (option o []) Nop (term Raise []) Nop

response :: (Eq a, Eq i) => Response i a
         -> CPSEnv i a -> Answer i a
response (Splat k) env = case lookup k env of
  Just (CPSQ q) -> Final q
  _             -> CoStuck [] k
response (End)     env = Final Nop
response (m :!: r) env
  = (term m env) <!> (response r env)

(<!>) :: CPSTerm i a -> Answer i a
      -> Answer i a
f <!> Final r      = f r
f <!> Stuck gs x q = Stuck (f : gs) x q
f <!> CoStuck gs q = CoStuck (f : gs) q

term :: (Eq a, Eq i) => Term i a -> CPSEnv i a
     -> CPSTerm i a
term (Var x)   env = case lookup x env of
  Just (CPST m) -> m
  _             -> Stuck [] x
term (Dot m)    env
  = \k -> (term m env) (Arg (term m env) :* k)
term (m :*: n)  env
  = \k -> (term m env) (Arg (term n env) :* k)
term (m :@: i)  env = \k -> (term m env) (i :@ k)
term (Raise)    env = \k -> Final k
term (q :!-> r) env
  = \k -> (response r ((q, CPSQ k) : env))
term (o :?: m)  env
  = \k -> (option o env) k (term m env) k

option :: (Eq i, Eq a) => Option i a -> CPSEnv i a
       -> CPSQuestion i a -> CPSOption i a
option (x :*-> o) env = \q f -> \case
  (y :* k) -> (option o env') q f k
    where env' = (x, CPST (useArg y)) : env
  _        -> f q
option (i :@-> o) env = \q f -> \case
  (j :@ k) | i == j -> (option o env) q f k
  _                 -> f q
option (x :?-> m) env = \_ f -> (term m env')
  where env' = (x, CPST f) : env
```

Figure 18. Environment and continuation-passing style translation for copatterns with nested options.

```
data Answer i a
  = Final   (ClosQuestion i a)
  | Stuck   (MetaCont i a) a (ClosQuestion i a)
  | CoStuck (MetaCont i a) a

type MetaCont i a = [Closure i a]

run :: (Eq a, Eq i) => Response i a -> Answer i a
run r = delim r [] []

eval :: (Eq i, Eq a) => Term i a -> Answer i a
eval m = refocus m [] Nop []

try :: (Eq i, Eq a) => Option i a -> Answer i a
try o = comatch o [] Nop (Raise :/: []) Nop []

delim :: (Eq a, Eq i)
      => Response i a -> ClosEnv i a
      -> MetaCont i a -> Answer i a
delim (Splat k) env (m:/:e:s)
  | Just (QSub q) <- lookup k env
  = refocus m e Nop s
delim (Splat k) env []
  | Just (QSub q) <- lookup k env
  = Final q
delim (Splat k) env s
  = CoStuck s k
delim (End)     env (m:/:e:s)
  = refocus m e Nop s
delim (End)     env []
  = Final Nop
delim (m :!: r) env s
  = delim r env $ (m :/: env) : s

refocus :: (Eq a, Eq i) => Term i a
        -> ClosEnv i a -> ClosQuestion i a
        -> MetaCont i a -> Answer i a
refocus (Var x)    env k s
  | Just (CSub (m:/:e)) <- lookup x env
  = refocus m e k s
refocus (Var x)    env k s
  = Stuck s x k
refocus (Dot m)    env k s
  = refocus m env ((m :/: env) :* k) s
refocus (m :*: n)  env k s
  = refocus m env ((n :/: env) :* k) s
refocus (m :@: i)  env k s
  = refocus m env (i :@ k) s
refocus (Raise)    env k (m:s)
  = refocus (openTerm m) (staticEnv m) k s
refocus (Raise)    env k []
  = Final k
refocus (q :!-> r) env k s
  = delim r ((q, QSub k) : env) s
refocus (o :?: m)  env k s
  = comatch o env k (m :/: env) k s

comatch :: (Eq i, Eq a) => Option i a
        -> ClosEnv i a -> ClosQuestion i a
        -> Closure i a -> ClosQuestion i a
        -> MetaCont i a -> Answer i a
comatch (x :*-> o) env (n:*k) m q s
  = comatch o ((x, CSub n) : env) q m k s
comatch (i :@-> o) env (j:@k) m q s
  | i == j = comatch o env q m k s
comatch (x :?-> n) env k      m _ s
  = refocus n ((x, CSub m) : env) k s
comatch _          env _      m q s
  = refocus (openTerm m) (staticEnv m) q s
```

Figure 19. Environment-passing, abstract machine interpreter for copatterns with control.

Refocusing / Reduction steps:

$$\langle M\ N \parallel \sigma \parallel K\rangle \mapsto \langle M \parallel \sigma \parallel N\{\sigma\}\ K\rangle \qquad \langle M. \parallel \sigma \parallel K\rangle \mapsto \langle M \parallel \sigma \parallel M\{\sigma\}\ K\rangle \qquad \langle x \parallel \sigma \parallel K\rangle \mapsto \langle M \parallel \sigma' \parallel K\rangle$$

$$\langle M\ X \parallel \sigma \parallel K\rangle \mapsto \langle M \parallel \sigma \parallel X\ K\rangle \qquad \langle \lambda\{L \to M; \vec{O}\} \parallel \sigma \parallel K\rangle \mapsto \langle L \parallel K \parallel \sigma \parallel M \parallel \vec{O} \parallel \sigma \parallel K\rangle \qquad (M\{\sigma'\}/x \in \sigma)$$

Copattern-matching steps:

$$\langle x\ L \parallel N\{\sigma'\}\ K' \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle \mapsto \langle L \parallel K' \parallel N\{\sigma'\}/x, \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle$$

$$\langle X\ L \parallel X\ K' \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle \mapsto \langle L \parallel K' \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle$$

$$\langle \varepsilon \parallel K' \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle \mapsto \langle M \parallel \sigma \parallel K'\rangle$$

$$\langle L \parallel \varepsilon \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle \not\mapsto \qquad (\text{if } L \neq \varepsilon)$$

$$\langle L \parallel K' \parallel \sigma \parallel M \parallel O\ldots \parallel \sigma_0 \parallel K\rangle \mapsto \langle \lambda\{O\ldots\} \parallel \sigma_0 \parallel K\rangle \qquad (\text{otherwise})$$

Figure 20. Environment-based abstract machine for calculating monolithic copatterns.

Meta-continuation steps:

$$\langle M\ !\ R \parallel \sigma \parallel S\rangle \mapsto \langle R \parallel \sigma \parallel M\{\sigma\}; S\rangle \quad \langle \varepsilon \parallel \sigma \parallel M\{\sigma'\}; S\rangle \mapsto \langle M \parallel \sigma' \parallel \varepsilon \parallel S\rangle \quad \langle q \parallel \sigma \parallel M\{\sigma'\}; S\rangle \mapsto \langle M \parallel \sigma' \parallel \sigma(q) \parallel S\rangle$$

Refocusing / reduction steps:

$$\langle M\ X \parallel \sigma \parallel K \parallel S\rangle \mapsto \langle M \parallel \sigma \parallel X\ K \parallel S\rangle$$

$$\langle M\ N \parallel \sigma \parallel K \parallel S\rangle \mapsto \langle M \parallel \sigma \parallel N\{\sigma\}\ K \parallel S\rangle \qquad \langle !q \to R \parallel \sigma \parallel K \parallel S\rangle \mapsto \langle R \parallel K/q, \sigma \parallel S\rangle$$

$$\langle M. \parallel \sigma \parallel K \parallel S\rangle \mapsto \langle M \parallel \sigma \parallel M\{\sigma\}\ K \parallel S\rangle \qquad \langle O\ ?\ M \parallel \sigma \parallel K \parallel S\rangle \mapsto \langle O \parallel \sigma \parallel K \parallel M\{\sigma\} \parallel K \parallel S\rangle$$

Copattern-matching steps:

$$\langle x \to O \parallel \sigma \parallel N\{\sigma'\}\ K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle \mapsto \langle O \parallel N\{\sigma'\}/x, \sigma \parallel K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle$$

$$\langle X \to O \parallel \sigma \parallel X\ K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle \mapsto \langle O \parallel \sigma \parallel K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle$$

$$\langle ?x \to N \parallel \sigma \parallel K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle \mapsto \langle N \parallel M\{\sigma_0\}/x, \sigma \parallel K \parallel S\rangle$$

$$\langle O \parallel \sigma \parallel K \parallel M\{\sigma_0\} \parallel K_0 \parallel S\rangle \mapsto \langle M \parallel \sigma_0 \parallel K_0 \parallel S\rangle \qquad (\text{otherwise})$$

Figure 21. Environment-based abstract machine for controlling compositional copatterns.

change takes place in the `reduce` function as shown in figure 16: the `Redex` it processes will now contain an explicit environment representing some delayed substitutions that haven't been finished yet, and its `Reduct` can now return a `Closure` (pair of an open term and static environment) with potentially more delayed substitutions.

The reasoning behind why this program transformation is correct with respect to figure 3 is that, if we eagerly perform all delayed substitutions before and after the environment-passing `reduce` step, it is the same as the substitution-based `reduce` step. Since `reduce` is a non-recursive stepping function, this property can be manually confirmed by manually checking each case.

Since the new `Redex` type now contains closures, we also have to update the decomposition functions `decomp` and `refocus`. These now start with explicit closures and search for the next redex—which follows exactly the same code structure before, since the search never goes under binders—which produces a redex with explicit substitution and a question containing closures in place of raw terms.

Putting this all together, we then get the environment-passing, small step evaluator `eval` and main driver loop `iter` shown in figure 16—already in the in-place refocusing form—which corresponds to the original small-step interpreter up to performing the delayed substitutions. The main correctness property about the top-level `eval` function can be derived from each step of `iter` by relating the above relationship of `reduce` and `refocus`.

From here on out, there is nothing new. Applying the same program transformations as before—CPS transformation, defunctionalization, loop fusion, compressing corridor transitions, deforesting, and other representational data structure changes—yields the environment-passing, tail-recursive interpreter in figure 17.

We can continue on to derive a continuation-passing style transformation like before as well, using the same transformation steps—desugaring pattern matching, η-reduction, and immediately applying transition functions to all subexpressions as soon as they are available. The resulting code corresponds to a form of CPS transformation that is parameterized by a static environment that gets used to interpret

both free and bound variables, in the style of many denotational semantics. Rephrased as a translation function into the λ-calculus, this CPS is as follows:

- Translating terms $[\![M]\!]_\sigma$:

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

- Translating lists of options $[\![O\ldots]\!]_\sigma$:

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

- Translating copattern-matching options $[\![L \to M]\!]_\sigma$:

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

As a convention, when bound names are introduced on the right-hand side of a defining equation, they are always chosen to be distinct from the free variables of σ to avoid accidental capture.

A.2 Closing over compositional copatterns

The refactorings used section 4 to generalize the calculus for delimited and compositional copattern matching were orthogonal to the question about substitution versus environments as the semantics for static variables. Therefore, we can replay the changes to the environment and continuation-passing transformation in section A.1 to derive a similar environment-based CPS translation of compositional copatterns:

- Translating responses $[\![R]\!]_\sigma$

$$\begin{aligned}
[\![M\ !\ R]\!]_\sigma &= \lambda s.\ [\![R]\!]_\sigma\ \lambda q.\ [\![M]\!]_\sigma\ q\ s \\
[\![q]\!]_\sigma &= \lambda s.\ s\ \sigma(q) \\
[\![\varepsilon]\!]_\sigma &= \lambda s.\ s\ ()
\end{aligned}$$

- Translating terms $[\![M]\!]_\sigma$

$$\begin{aligned}
[\![x]\!]_\sigma &= \sigma(x) \\
[\![M\ X]\!]_\sigma &= \lambda k.\ [\![M]\!]_\sigma\ (X\ k) \\
[\![M\ N]\!]_\sigma &= \lambda k.\ [\![M]\!]_\sigma\ ([\![N]\!]_\sigma, k) \\
[\![M.]\!]_\sigma &= \lambda k.\ [\![M]\!]_\sigma\ ([\![M]\!]_\sigma, k) \\
[\![\mathbf{raise}]\!]_\sigma &= \lambda k.\ \lambda s.s\ k \\
[\![O\ ?\ M]\!]_\sigma &= \lambda k.\ [\![O]\!]_\sigma\ [\![M]\!]_\sigma\ k \\
[\![!q \to R]\!]_\sigma &= \lambda q.\ [\![R]\!]_\sigma
\end{aligned}$$

- Translating options $[\![O]\!]_\sigma$

$$\begin{aligned}
[\![x \to O]\!]_\sigma &= \lambda f.\lambda k.\ \mathbf{case}\ k\ \mathbf{of} \\
&\qquad\qquad (x, k') \to [\![O]\!]_\sigma\ (\lambda q.\ f\ (x, q))\ k' \\
&\qquad\qquad k \to f\ k \\
[\![X \to O]\!]_\sigma &= \lambda f.\lambda k.\ \mathbf{case}\ k\ \mathbf{of} \\
&\qquad\qquad (X\ k') \to [\![O]\!]_\sigma\ (\lambda q.\ f\ (X\ q))\ k' \\
&\qquad\qquad k \to f\ k \\
[\![?x \to M]\!]_\sigma &= \lambda x.\ [\![M]\!]_\sigma
\end{aligned}$$

The corresponding Haskell embedding is shown in figure 18.

From here on out, we can turn the CPS transformation into an abstract machine using the same general derivation technique. Applying standard code transformations—defunctionalization, delaying the application of translation functions until the last moment of application, η-expansion, and the use of nested pattern matching—gives the environment-passing, tail-recursive interpreter shown in figure 19.

To compare the difference of the low-level execution of the two calculi—one for monolithic matching of complex copatterns, and the other for compositional matching of copatterns with control—we can put them in more common forms. Rephrasing the Haskell implementations as stepping relations on machine configurations for both calculi are shown in figures 20 and 21.

Received 2025-06-10; accepted 2025-07-31