CoScheme: Compositional Copatterns in Scheme

Paul Downen[1][0000−0003−0165−9387] and
Adriano Corbelino II[1][0000−0002−6014−6189]

University of Massachusetts Lowell, Lowell MA 01854, USA
Paul_Downen@uml.edu
Adriano_VilargaCorbelino@uml.edu

Abstract. Since their introduction, copatterns have promised to extend functional languages — with their familiar pattern matching facilities — to synthesize and work with infinite objects through a finite set of observations. Thus far, their adoption in practice has been limited and primarily associated with specific tools like proof assistants. With that in mind, we aim to make copattern matching usable for ordinary functional programmers by implementing them as macros in the Scheme and Racket programming languages. Our approach focuses on composable copatterns, which can be combined in multiple directions and offer a new solution to the expression problem through novel forms of extensibility. To check the correctness of the implementation and to reason equationally about copattern-matching code, we describe an equational theory for copatterns with a sound, selective translation into λ-calculus.

Keywords: Codata · Copatterns · Scheme · Macros · Composition · Expression Problem.

1 Introduction

Composition is one of the great promises of functional programming to combat complexity. As opposed to monolothic solutions, functional programming languages encourage us to decompose large problems into small, reusable, and reliable parts and then to recompose them back into a whole solution [13]. This practice is encouraged through tools like higher-order functions to abstract out common patterns and laziness to separate generation, selection, and consumption of information. Rather than implementing a complex algorithm as a single special-purpose loop, functional programming lets us express the same solution as the composition of simple domain-specific operations and generic combinators: maps, filters, folds, and unfolds.

However, the *expression problem* [29] is a familiar foe that still resists this (de)compositional approach. It captures the common problem that arises when we want to maintain code — such as an evaluator for the syntax trees of an expression language — by extending it in two different directions: adding new forms of data (*i.e.,* classes of objects) and new operations (*i.e.,* methods) on them. Traditionally, functional languages can easily add new operations over

any given data type, but adding a new constructor requires a major rewrite that can potentially alter the rest of the code. Conversely, object-oriented languages make it easy to add a new class of object, but extending a base class with a new method again requires major rewriting. Being a common obstacle in the way of maintaining, extending, and decomposing code, the expression problem has garnered many solutions in the object-oriented [19,30] and functional [24,16] worlds, and especially hybrid languages that mix both [3,11].

This work presents a novel solution to the expression problem: composable copatterns. Copatterns [2] are often associated with codata types for expressing infinite objects, but their use is not limited to just that. Their composition, in particular, allows us to define programs by performing equational reasoning in the evaluation context. Performing the "substitution of equals for equals" [28] enhances the predictability and composability of our programs since we can analyze our part code in isolation.

Previous implementations of copatterns can be found in strongly typed languages which impose prescribed restrictions on their use. For example, Agda gives the most full-fledged implementation of copatterns in a real system [6]. However, Agda is primarily a proof assistant rather than a general-purpose programming language, and as such, has different concerns than an ordinary functional programmer. There is also some support for copatterns in OCaml [18], but as an unofficial extension that has not been merged into the main compiler.

The copatterns implemented here are also implemented as macros like [18]; however, we present a different encoding that focuses on providing new methods of extensibility that were not available before, and can be desugared without any static typing information. To achieve that, and to fully integrate it into a practical general-purpose programming language, we choose a programmable programming language [10] and provide a new language extension as a library [27]. We focus, in particular, on Scheme and Racket, which offers a robust macro system for seamlessly implementing new language features.

Our extension presents three different composition flavors, allowing us to capture some "design patterns" used by functional programmers as first-class abstractions. First, we have *vertical* composition, which permits us to gather a collection of alternative options with failure handling. Second, we have *horizontal* composition, which permits us to compose a sequence of steps, parameters, matching, or guards. Third, we have *circular* composition, which allows us to recurse back on the entire composition itself.

Our primary contributions are organized as follows:

- Section 2 shows examples of programming with copattern equations in Scheme-like languages, including new forms of program composition — vertical and horizontal — that allows us to solve familiar examples of the expression problem [29] through a fusion of functional and object-oriented techniques.
- Section 3 exposes the challenges related to implementing copatterns in this scenario, introduces our library API, shows how we can desugar our abstractions into a set of primitives and how the implementations differ between Racket and a standard R^{6}RS-compliant Scheme.

– Section 4 presents a theory for how to translate copatterns into a small core target language — untyped λ-calculus with recursion and patterns — with a local double-barrel transformation reminiscent of selective continuation-passing style transformation. Importantly, only the new language constructs are transformed, while existing ones in the target language are unchanged.
– Section 4.2 demonstrates correctness in terms of an equational theory for reasoning about copattern-matching code in the source language, which is a conservative extension of the target language, and we prove that it is sound with respect to translation.

2 Programming with Composable Copatterns in Scheme

All examples shown below are executable Scheme and Racket code. You can follow along and interact with the code using the supporting library found online at https://github.com/pdownen/CoScheme.

2.1 Infinite streams

For decades, functional programmers have had a reliable and versatile method for representing tree-shaped structures: inductive data types. These can model data of any size — for example, lists of an arbitrary length — but each instance must be *finite*. But how does a program handle infinite amounts of information, that cannot possibly occupy a finite memory space?

One method of modeling infinite information is through laziness, as in the Haskell programming language. For example, consider the usual infinite list of Fibonacci numbers in Haskell:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

`fibs` cannot be fully evaluated because it has no base case — it would eventually expand out to `0 : 1 : 1 : 2 : 3 : 5 : 8 : ...` forever — but this is no problem in a non-strict language that only evaluates as much as needed. But what if we are working within a strict language without laziness built in? Must we give up on the approach entirely, or is there an alternate solution that works just as well with eager and lazy evaluation?

In contrast, *codata* describes types defined by primitive *destructors* that *use* values of the codata type — as opposed to the primitive constructors that define how to build values of a data type — and lets us easily model infinite data in eager languages, too. For example, the usual `Stream a` codata type of infinite `a`'s is defined by two destructors: `Head : Stream a -> a` extracts the first element and `Tail : Stream a -> Stream a` discards the first element and returns the rest. To define new streams, we can describe how they react to different combinations of `Head` and `Tail` destructors using *copatterns* [2]. Borrowing Agda's syntax, a possible copattern-based definition of the same `fibs` function above is:

```
fibs : Stream Nat
Head fibs = 0
Head (Tail fibs) = 1
Tail (Tail fibs) = zipWith _+_ fibs (Tail fibs)
```

However, at the moment, Agda currently does not understand if `fibs` is well-founded — it is — and so `fibs` is rejected. As a proof assistant, Agda has demanding requirements on all definitions to ensure well-foundedness: they must never have unproductive infinite loops, and they must cover every possible case (when matching on arguments or copatterns, as in `fibs`). But for a general-purpose programming language, we expect to be able to write arbitrary loops that may or may not terminate. Copattern-based definitions need to gracefully handle cases that fail to return — either due to an infinite loop or an exception, like an unhandled case, which are semantically similar [22] — and should generate code based on whatever is given. In this kind of setting, the language does not enforce — and indeed, our implementation does not check — coverage, which is instead up to the programmer to determine.

Let us now consider some examples of programming by equational reasoning to get familiar with copatterns and how we can use them in Scheme. For example, even in a dynamically-typed language like Scheme, linked lists can be thought of as an inductively-defined type combining two constructed forms: `List a = null | (cons a (List a))`. Likewise, infinite streams can be understood as the type of a procedure that exhibits two different behaviors at the same time: `Stream a = 'head -> a & 'tail -> Stream a`. In other words, any `Stream a` is a procedure that takes one argument, and its response depends on the exact value: given `'head` an `a` is returned, and given `'tail` another `Stream a` is returned.

In order to define new coinductive processes, one of the main entry points is the top-level, multi-line `define*` macro. This macro enables us to declare codata objects through a list of equations between a copattern on the left-hand side and an expression on the right-hand side. At the root of every copattern is a name for the object *itself*, which can be inside any number of applications — the applications may just list parameter names or more specific patterns, narrowing down the concrete arguments that match. Using `define*`, we can define the trivial `zeroes` stream — whose `'head` is `0` and whose `'tail` is more `zeroes` — as:

```
;; zeroes : Stream nat
(define* [(zeroes 'head) = 0]
         [(zeroes 'tail) = zeroes])
```

Streams like `zeroes` are black boxes that can only be observed by passing `'head` or `'tail` as arguments to get their response. Still, this is enough for many useful operations, like taking the first `n` elements, which can be `define*`d as:

```
;; takes : (Stream a, nat) -> List a
(define* [(takes s 0) = '()]
         [(takes s n) = (cons (s 'head) (takes (s 'tail) (- n 1)))])
```

A constant stream is not particularly useful; more interesting streams will change over time. For example, imagine a "stuttering" stream $(0, 0, 1, 1, 2, 2, 3, 3, \ldots)$

that repeats numbers twice before moving on. This stream can be defined by copattern matching equations:

```
;; stutter : nat -> Stream nat
(define* [ ((stutter n) 'head)        = n]
         [(((stutter n) 'tail) 'head) = n]
         [(((stutter n) 'tail) 'tail) = (stutter (+ n 1))])
```

So that `(takes (stutter 1) 10) = '(1 1 2 2 3 3 4 4 5 5)`,[1] because the first and second elements — `((stutter n) 'head)` and `(((stutter n) 'tail) 'head)` respectively — return the same `n` before incrementing.

But why is `stutter` well-defined, and how can we understand its meaning? As in many functional languages, the `=` in code really implies equality between the two sides, and this equality still holds when we plug in real values for placeholder variables like `n`. So to determine the first `'head` element, of `(stutter 1)`, we match the left-hand side and replace it with the right to get `((stutter 1) 'head) = 1`. Similarly, the second element is `(((stutter 1) 'tail) 'head) = 1` as well. The third element is accessed by two `'tail` projections and then a `'head` as the nested applications `((((stutter 1) 'tail) 'tail) 'head)`, which doesn't exactly match any left-hand side. However, equality holds in any context, and the inner application `(((stutter 1) 'tail) 'tail)` *does* match the third equation. Thus, we can apply a few steps of equational reasoning to derive the expected answer `2`:

```
((((stutter 1) 'tail) 'tail) 'head) = ((stutter (+ 1 1)) 'tail)  ; line 3
                                    = ((stutter 2) 'head)        ; +
                                    = 2                          ; line 1
```

So these three examples work, but is every case really covered? The `Stream Nat` interface that `stutter`'s output follows allows for any number of `'tail` projections followed by a final application to `'head` that returns a natural number. `stutter` works its way through these projections in groups of two, eliminating a pair of `'tail` projections at a time until it gets to the end case, which is either a `'head` (if the total number of `'tail`s is even) or a `'tail` followed by `'head` (if the total number of `'tail`s is odd). So, `stutter` behavior is defined no matter what is asked of it. Even with other observations like `takes`, which passes around partial applications of `stutter` as a first-class value, internally `stutter` only "sees" the `'head` and `'tail` applications from `takes`, and is dormant otherwise.

Reasoning about the coverage of our copatterns is important since our implementation does not provide coverage analysis. If we encounter an uncovered case, our implementation emits a runtime error, explaining that this is an uncovered copattern. Non-total configurations, akin to partial functions, are not always undesirable. They can simplify the development during a prototyping phase, and if the missing case does not make sense, they can be the most semantically meaningful.

With this practice under our belt, we can now directly translate the canonical Fibonacci example from Agda to Scheme like so:

[1] Try it! https://github.com/pdownen/CoScheme has implementations of `define*` and related macros used by these examples.

```
;; zips-with : ((a, b) -> c, Stream a, Stream b) -> Stream c
(define*
  [((zips-with f xs ys) 'head) = (f (xs 'head) (ys 'head))]
  [((zips-with f xs ys) 'tail) = (zips-with f (xs 'tail) (ys 'tail))])

;; fibs : Stream nat
(define*
  [ (fibs 'head)        = 0]
  [((fibs 'tail) 'head) = 1]
  [((fibs 'tail) 'tail) = (zips-with + fibs (fibs 'tail))])
```

so that `(takes fibs 10)` is `'(0 1 1 2 3 5 8 13 21 34)`.

2.2 Self-referential objects

Codata can also be used to implement an abstract interface over regular finite data. As an alternate syntax for `define*`, we can explicitly give a top-level name to bind the definition to for external use, and on each equation give a hidden internal for self-reference and recursion. To illustrate this, consider the following queue example, which internally refers to itself by the name `self` for an object-oriented feel:

```
(define* queue
  [ (self 'new)            = (self '() '())]
  [((self in  out) 'enq x) = (self (cons x in) out)]
  [((self '() '()) 'deq)   = (error "Invalid dequeue: empty queue")]
  [((self in  '()) 'deq)   = ((self '() (reverse in)) 'deq)]
  [((self in  out) 'deq)   = (cons (car out) (self in (cdr out)))])
```

This reflects the purely functional queue implementation in using two lists (an inbox and an outbox) as internal states. We externally bound this declaration to the name `queue`, but the internal recursion is through the name `self`. This `queue` object responds to three methods: `'new` returns a new empty queue, `'enq x` puts the `x` to the end of the queue (*i.e.,* the top of the inbox), and `'deq` returns the oldest enqueued element (from the top of the outbox or bottom of the inbox, as appropriate). Thus, `((((queue 'new) 'enq 1) 'enq 2) 'deq)` returns the oldest element `1` and a queue object containing only `2`.

Visualizing what we are defining through the lens of the object-oriented paradigm can give a new perspective here. With this metaphor, we can view our definitions as describing the protocols of objects, where the equations specify how an object should respond to a sequence of messages. Here, `queue` itself can only directly respond to one message — `'new` — that initializes the object with two empty lists for its private internal state. From there, the initialized `queue` object now only responds to the `'enq x` and `'deq` messages which can read and update the object's internal state. However, besides these two messages, there is no other way to manipulate the internal state of an initialized `queue` object; the `in` and `out` lists are completely hidden within an opaque procedural abstraction, enforcing an encapsulation of private state. Given an initialized `queue` object,

it would not be possible, for instance, to break the first-in-first-out ordering by taking an element from the `in` list or to put an element on the `out` list.

Since we can already use encapsulation in copattern-based definitions, can we also use a functional model [1] of inheritance and dynamic dispatch? Our implementation of copattern matching in Scheme includes new facilities for composing code snippets compared to current functional (or object-oriented) languages. However, to avoid unwanted surprises, the programmer does have to ask for them. This is a small request, and can be done by replacing `define*` with `define-object`, as in the following file system example:

```
(define-object
  [((fs-object p . _) 'path) = p])

(define-object (<: (import-object fs-object))
  [((file p txt) 'text) = txt]
  [((file p txt) 'size) = (string-length ((file p txt) 'text))])

(define-object (directory <: (import-object fs-object))
  [((apply dir p cts) 'contents) = cts]
  [((apply dir p cts) 'overhead) = 8]
  [((apply dir p cts) 'size)
   = (apply + ((apply dir p cts) 'overhead)
              (map (λ(o) (o 'size)) ((apply dir p cts) 'contents)))])
```

This example emulates some functionality of a filesystem, specifically calculating various sizes. Every filesystem object has a path, which is captured by the `fs-object` object that only knows how to calculate its `'path` by returning the first piece of information it was given and ignoring the rest of the object's internal data. Filesystem objects all also have a size, but calculating it requires more object-specific information. This additional functionality is spelled out by more specific filesystem objects:

- A `file` contains some `'text` and its `'size` is the length of that text.
- A `directory` contains any number of additional filesystem objects, stored as its `'contents`, and its `'size` is the sum of it's `'contents` size plus an additional `'overhead` that defaults to `8`.

Both `file` and `directory` objects inherit the code for calculating it's `'path` by importing `fs-object` with the extension clause `<: (import-object fs-object)`. Note that these three object definitions exercise three of the four possible definition forms, based on whether the external name is given explicitly or inferred, and on whether there is an extension clause is included:

- `fs-object`'s external name is inferred from the internal name in its copattern equation, and it has no listed extension clause,
- `file`'s external name is taken from its internal one and it extends `fs-object`,
- `directory`'s external name is given explicitly (and is different from its internal name) and it extends `fs-object`.

The last possibility is an explicit external name with no extensions, such as the following equivalent definition of `fs-object` that elaborates the naming inference:

```
(define-object fs-object
  [((fs-object p . _) 'path) = p])
```

While these definitions are functional, they contain some undesirable redundancy. In particular, we have to repeat the same initialization forms — `(file p txt)` and `(apply dir p cts)` — in front of every equational definition because the object must be initialized with parameters before it is used. What is worse, every time we want to ask a question about the object itself by recursively passing it a message, we have to repeat this same initialization again exactly as it occurred, leading to longer and more error-prone code. It would be better here to follow the common object-oriented factorization of steps: *first* the object is initialized with some internal data at the time of its construction, and *then* we get an object that can (recursively) respond to methods. This can be done by factoring out the common construction phase in the above definitions using a `construct` clause like so:

```
(define-object (<: (import-object fs-object))
  [(file p txt) (construct (list p txt))
   (object
    [(self 'text) = txt]
    [(self 'size) = (string-length (self 'text))])])

(define-object (directory* <: (import-object fs-object))
  [(apply dir p cts) (construct (cons p cts))
   (object
    [(self 'contents) = cts]
    [(self 'overhead) = 8]
    [(self 'size)
     = (apply + (self 'overhead)
               (map (λ(o) (o 'size)) (self 'contents)))])])
```

The `construct` operation lists its internal parameters given at the time of initialization. After this step, we describe a first-class anonymous `object` that knows how to refer to its fully-constructed form by an internal name (here we use the name `self`), so there is no need to re-construct `(file p txt)` or `(apply dir p cts)` to recursively call other methods. Factoring out the common copattern leads to shorter code, the definitions describe objects with exactly the same behavior as before.

So far, we have only shaved off small parts of shared code: the common `'path` method and the initial initialization copattern. Where this coding style starts to pay off is when we override some methods to automatically influence the result of others. For example, we might have a fancier type of directory structure that replicates the exact same behavior as a normal directory, but its `'overhead` is `128` instead of `8`. Or we might have static links that act like a normal `file`, except that they only contain a path to the real place its text is stored, so it always has a fixed size (`8`). These specialized revisions of directories and files can be

implemented by importing from the original definitions and modifying certain lines like so (where we omit the code for looking up the text for a static link):

```
(define-object (fancy-directory <: (import-object directory))
  [((fancy-dir p . cts) 'overhead) = 128])

(define-object (<: (import-object file))
  [(static-link p lnk) (construct (list p lnk))
   (object
    [(_ 'link) = lnk]
    [(_ 'text) = "..."]
    [(_ 'size) = 8])])
```

Although `fancy-directory` has only one defining equation about its `'overhead`, the implication is that its `'size` should be `120` larger than a regular `directory` due to the `'overhead` increase. We can test out this use of inheritance and dynamic dispatch by simulating a small directory structure:

```
(define ham (file "hamlet.md" "Words, words, words...."))
(define guide (file "Guide.md" "Don't Panic"))
(define books (directory* "Books" ham guide))
(define shortcut (static-link "Guide.md" "Books/Guide.md"))
(define docs (fancy-directory "Documents" shortcut books))
```

and calculating the sizes of each file system object:

```
(ham      'size) = 23  = (string-length "Words, words, words....")
(guide    'size) = 11  = (string-length "Don't Panic")
(books    'size) = 42  = (+ 8 23 11)
(shortcut 'size) = 8
(docs     'size) = 178 = (+ 128 8 42)
```

The question is: how were we able to inject new code in the middle of an object like `directory` to change its behavior? The key issue is that, within `directory`, recursive calls to `((apply dir p cts) 'overhead)` — or just simply `(self 'overhead)` in the second version — cannot be tied to *this* definition of `directory`. Instead, we employ *open recursion*: the internal references to the recursive object itself are left as unbound parameters that will only be bound to the full object value when the final definition is ready to use. This is why the internal and external names can be different — like in `queue` and `directory` — since the external name is bound to the final object value while the internal names are left (temporarily) open-ended and will be filled in later. In the same way, the internal names used in each clause are fully independent and can also differ from one another. This difference of using open recursion to leave internal names temporarily unbound also applies to definitions where the external name is inferred: although the external names `file` and `static-link` are inferred from the internal names in the copatterns, the internal variables are left unbound until the externally-visible name gets bound to the object value.

Thankfully, this whole framework is still built on purely functional idioms, which makes it easier to reason about code. How can we understand what inheritance should do? The answer should be familiar to functional programmers:

inheritance is composition and substitution! For example, the inheritance dependencies can be fully inlined as-is into `fancy-directory` to bring the full definition into one place by copying the inherited code into place like so:

```
(define-object fancy-directory
  [((fancy-dir p . cts) 'overhead) = 128]
  [((apply dir p cts)   'overhead) = 8]
  [((apply dir p cts)   'contents) = cts]
  [((apply dir p cts)   'size)
   = (apply + ((apply dir p cts) 'overhead)
              (map (λ(o) (o 'size)) ((apply dir p cts) 'contents)))]
  [((fs-object p . _)   'path)     = p])
```

After naïve inlining, there are some irrelevant differences: there are three different internal names (`fancy-dir`, `dir`, and `fs-object`) used in various clauses, and we use two equivalent syntaxes (`(fancy-dir p . cts)` and `(apply dir p cts)`) for copatterns that bind arbitrary-length argument sequences to `cts`. Cleaning up these differences by rewriting each initializing copattern to `(apply self p cts)` and renaming as necessary gives a more uniform code:

```
(define-object fancy-directory
  [((apply self p cts) 'overhead) = 128]
  [((apply self p cts) 'overhead) = 8]
  [((apply self p cts) 'contents) = cts]
  [((apply self p cts)   'size)
   = (apply + ((apply self p cts) 'overhead)
              (map (λ(o) (o 'size)) ((apply self p cts) 'contents)))]
  [((apply self p _)   'path)     = p])
```

From here, it becomes more obvious that the two different equations defining `(... 'overhead)` overlap, so the first one takes precedence and the second one is dead code that can be completely erased. Furthermore, when there are no more future extensions, we can inline the recursive calls at this point, to get the simpler closed definition that reveals exactly what each method will do:

```
(define* fancy-directory
  [((apply self p cts) 'overhead) = 128]
  [((apply self p cts) 'contents) = cts]
  [((apply self p cts) 'size)
   = (apply + 128 (map (λ(o) (o 'size)) cts))]
  [((apply self p _)   'path)     = p])
```

2.3 Decomposing the expression problem

Our notion of compositional copatterns can capture some object-oriented styles of code (de)composition with encapsulation, inheritance, and dynamic dispatch. How can this new capability for composition influence the kinds of functional programs we write? For example, consider the usual definition of a simple arithmetic expression evaluator in typed functional languages like Haskell and OCaml (we use Haskell syntax here):

```
data Expr = Num Int | Add Expr Expr

eval :: Expr -> Int
eval (Num n)   = n
eval (Add l r) = eval l + eval r
```

While Scheme does not have algebraic data types, we can encode complex constructor expressions as a list starting with the constructor name as a quoted symbol and the arguments as the remainder of the list, and when unambiguous, leave atomic data alone. So `Num 5` could just be represented as the plain number `5`, and `Add l r` would be represented as the *quasiquote* `` `(add ,l ,r) `` which plugs in the values bound to variables `l` and `r` as the second and third elements of the list (denoted by the "unquote" comma `,` before the variable names). We can then use the facilities of `define*` to write almost identical code in Scheme like so, using the *guard* `try-if` to test if the argument is a number:

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

Fantastic, it works! Both the Scheme and Haskell code have the same structure. And on the surface, they both share the same strengths and weaknesses. From the lens of the *expression problem* [29], it is easy to add new operations to existing expressions — such as listing the numeric literals in an expression

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

— but adding new classes of expressions is hard. For example, if we wanted to support multiplication, we could add a `Mult` constructor to the `Expr` data type, but this would require modifying *all* existing operations and case-splitting expressions over `Expr` values. Even worse, if we wanted to support both expression languages — with or without multiplication — we would have to copy the code and maintain both versions.

Thankfully, our implementation of copattern matching in Scheme includes new facilities for composing code snippets. As we previously saw with the object-oriented examples, we can turn ordinary functional code into a more extensional form by using `define-object` instead of `define*`.

```
;; list-nums* : Expr -> List num
(define-object
  [(list-nums* n)
   (try-if (number? n))     = (list n)]
  [(list-nums* `(add ,l ,r)) = (append (list-nums* l) (list-nums* r))])
```

The `list-nums*` object behaves exactly like `list-nums` in all the same contexts it works in, but in addition, it implicitly inherits additional functionality for composition defined elsewhere. This new composition lets us break existing multi-line

definitions into individual parts, and recompose them later. For example, the evaluator can be composed in terms of separate objects for each line like so:

```
(define-object
  [(eval-num n) (try-if (number? n)) = n])

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

;; eval* : Expr -> num
(define eval* (eval-num 'compose eval-add))
```

So `(eval expr)` is the same as `(eval* expr)` for any well-formed expression argument. Why program in this way? Now, if we want to extend the functionality of existing operations — like evaluation and listing literals — to support a new class of expression, we can define the new special cases separately as a patch and then *compose* them with the existing code as-is like so:

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

;; eval-arith : Expr+Mul -> num
(define eval-arith (eval* 'compose eval-mul))

;; eval-arith : Expr+Mul -> List num
(define list-nums-arith (list-nums* 'compose list-mul))
```

So for an expression `(define expr1 '(add (mul 2 3) 4))`, the extended code correctly yields `(eval-arith expr1) = 10` and `(list-nums-arith expr1) = '(2 3 4)` whereas the original code fails at the `'mul` case.[2] Note that this composition automatically generates *new* functions and leaves the original code intact, which can still be used for the smaller expression language with only numbers and addition.

This example emphasizes our guiding principle: *composition*. We call combinations like `(eval-num 'compose eval-add eval-mul)` *vertical composition* since they behave as if we simply stacked their internal cases vertically, like in the original definition of `eval`.

Not all types of language extensions are this simple, though. Consider what happens if we want to support algebraic expressions which might have variables in them. To evaluate a variable, we need a given environment — mapping names to numbers — which we can use to look up the variable's value.

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

[2] The astute reader might notice the open recursion at work here: the recursive calls to `eval-mul` cannot be specifically tied to this definition because it only says what to do with multiplication and fails to handle the other cases. Instead, recursive calls to `eval-mul` must *also* open to invoking the other code associated with `eval-num` and `eval-add` even though it is not known to be associated with them yet.

However, it is wrong to just vertically compose this variable evaluator with the previous code because the arithmetic evaluator only takes a single expression as an argument, whereas the variable evaluator needs *both* an environment and an expression. The manual way to perform this extension is routine for functional programmers: in addition to adding a new case, we have to add an extra parameter to each case, which gets passed along on all recursive calls.

It would be highly disappointing to have to rewrite our existing code in-place to do this extension. Fortunately, our copattern language allows for another type of composition — *horizontal composition* — which allows us to combine sequences of steps, one after another, and automatically fall through to the next case if something fails. For this example, we can define a general procedure `with-environment` to perform the above transformation, taking any extensible evaluator object expecting just an expression and threading an environment along each recursive call. This lets us patch our existing arithmetic evaluator with an environment and then compose it with variable evaluation like so:

```
(define (with-environment eval-ext)
  (object [(self env expr)
           (with-self (override-lambda* self
                        [(_ sub-expr) = (self env sub-expr)])
             (try-apply-forget eval-ext expr))]))

;; Env = List (Symbol . num)

;; eval-alg : (Env, Expr+Mul+Var) -> num
(define eval-alg
  ((with-environment (eval-arith 'unplug)) 'compose eval-var))
```

The `with-environment` function is the most complex code we have seen so far, but it just spells out the usual steps a functional programmer uses to modify existing code with an environment.

- Given the evaluator `eval-ext`, it returns a new first-class `object` (which is the same as `define-object` without assigning a name) that expects both an environment and expression to process.
- This new object then invokes `eval-ext` by passing just the expression, except that if `eval-ext` ever tries to recur with a sub-expression, the calls `(self sub-expr)` gets replaced with `(self env sub-expr)` just like the template transformation.
- This transformation of the evaluator's notion of self is done by the `with-self` operation, which can override the original recursive `self`.
- Finally, if none of the clauses of `eval-ext` succeed, then this updated evaluator also falls through as before, forgetting the application had ever happened via `try-apply-forget`.

The complete algebraic evaluator can then be made from an open-ended, extensible version of the arithmetic evaluator — retrieved from `(eval-arith 'unplug)` — horizontally composed to take an environment and vertically composed with the single-line `eval-var`. It can now successfully evaluate algebraic expressions,

such as `(define expr2 '(add (var x) (mul 3 (var y))))`, so that running the evaluation `(eval-arith '((x . 10) (y . 20)) expr2)` returns `70` because the environment maps `x` to `10` and `y` to `20`.

Another possible way to evaluate expressions with variables is *constant folding*, a common optimization where operations are simplified unless they are blocked by variables whose values are unknown. In other words, the evaluator might return a blocked expression if it cannot fully calculate the final number. Ideally, we would like to extend our existing evaluator as-is, with the additional cases when blocked expressions are encountered. However, as written, the equation handling `(eval `(add ,l ,r))` already commits to a real numeric addition, even if evaluating `l` or `r` does not give a numeric result.

To avoid over-committing before we know whether evaluation will successfully calculate a final number or not, we can — at first glance — rewrite the basic clauses of evaluation in a more defensive style. Essentially, this splits evaluation into two separate steps: (1) check which operation we are supposed to do and evaluate the two sub-expressions, (2) combine the two expressions according to that operation. For example, the steps for addition and multiplication look like:

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

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

Here, the evaluation step is explicated by a `'eval` tag, to help distinguish from the other operation `'add` for adding the left and right results. Note that in this code, the `'add` clause only performs a numeric addition `+` if it knows for sure that *both* of the arguments are actually numbers. We can now compose the original base-case for evaluating numbers with this "safer" version of addition that fails to match cases where sub-expressions don't evaluate to numbers (multiplication could be added as well in a similar style):

```
;; eval-arith-safe : ('eval, Expr+Mul) -> num
;;                 & ('add, num, num) -> num
;;                 & ('mul, num, num) -> num
(define eval-arith-safe (eval-num 'compose eval-add-safe eval-mul-safe))
```

So `(eval-arith-safe expr1)` still evaluates to `70`, but `(eval-arith-safe expr2)` fails when it finds a variable sub-expression.

If it finds a variable, constant folding will just leave it alone and return an unevaluated expression rather than a final number. Because the `'eval` operation might return a (partially) unevaluated expression, we now need to handle cases

where the left or right (or both) sub-expressions do not evaluate to numbers. In each of those cases, we must reform the addition expression out of what we find, converting numbers `n` into a syntax tree of the form `` `(num ,n) ``.

```
(define-object
  [(leave-variables 'eval ('var x)) = (list 'var x)])

(define-object reform-operations
  [(reform 'add l r) = (list 'add l r)]
  [(reform 'mul l r) = (list 'mul l r)])
```

The final constant-folding algorithm can be composed from this "safe" version of evaluation, along with the cases for leaving variables alone and reforming partially-evaluated additions and multiplications.

```
;; constant-fold : ('eval, Expr+Mul+Var) -> Expr+Mul+Var
;;               & ('add, Expr+Mul+Var, Expr+Mul+Var) -> Expr+Mul+Var
;;               & ('mul, Expr+Mul+Var, Expr+Mul+Var) -> Expr+Mul+Var
(define constant-fold
  (eval-arith-safe 'compose leave-variables reform-operations))
```

So now `(constant-fold 'eval expr2)` successfully returns `expr2` itself (because there are no operations to perform without knowing the values of variables `x` and `y`). And running `(constant-fold 'eval expr3)` on the expression

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

simplifies it down to `'(add 2 (mul (var x) 10))`. To add other operations, like subtraction, we can easily define similar `eval-sub-safe` and `reform-subtraction`, and `'compose` them with `constant-fold` without having to rewrite any code.

3 A Composable Copattern Macro Library

3.1 Challenges

Even though the behavior of small examples may be straightforward to understand, there are several challenges to correctly implementing copatterns in the general case. Some of these challenges are specific to Scheme — a dynamically-typed, call-by-value language — which forces us to carefully resolve the timing of when and which copatterns are matched. Other challenges are specific to our extensions to copatterns — the ability to compose copattern matching in two different directions — which also brings in the notion of the recursive "self."

Timing and the order of copattern matching Copatterns may have ambiguous cases where two different overlapping copattern equations match the same application. For example, this following function moves a number by `1` away from `0` — positives are incremented and negatives are decremented:

```
(define* [(away-from0 x) (try-if (>= x 0)) = (+ x 1)]
 [(away-from0 x) (try-if (<= x 0)) = (- x 1)])
```

Consequently, we must interpret the programmer's code as it is written since we cannot gain any information from a static type system. In the previous example, the two different equations overlap for `0` itself: either one matches the call `(away-from0 0)`. To disambiguate overlapping copatterns, the listed equations are always tried top-down, and the first full match "wins," as is typical in functional languages. In this case, the first line wins, so `(away-from0 0)` is `1`. Furthermore, guards like `try-if` and `try-match` are run left-to-right with short-circuiting — the moment a copattern or a guard fails, everything to the right is skipped. This makes it possible to protect potentially-erroneous guards with another safety guard to its left, such as `(try-if (not (= y 0)))` followed by `(try-if (> (/ x y) z))`.

However, there are more timing issues besides these usual choices for disambiguation and short-circuiting. First of all, since we are in a call-by-value language, we have to handle cases where an object is used in a context that doesn't fully match a copattern *yet*, but could in the future — and possibly multiple different times. This can happen for instances like curried functions that take arguments in multiple different calls. Just like with ordinary curried functions, using such an object in a calling context passing only the first list of arguments — but not the second — builds a *value* which closes over the parameters so far. For example, consider this simple counter object that can add or get its current internal state.

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

The call `((counter 4) 'get)` matches the second equation, returning the answer `4`, but `(counter 4)` on its own is not enough information to definitively match either copattern, so it is just a value remembering that `x = 4` and waiting for another call. Similarly, the call `(counter (+ x y))` on the right-hand side is *also* incomplete in the same sense, so it, too, is a value. This definition gives us an object with the following behavior:

```
> (define c0 (counter 4))
> (define c1 (c0 'add 1))
> ((c1 'add 2) 'get)
7
> (c1 'get)
5
```

So far, what we have seen so far seems similar to pattern-matching functions in languages that are curried-by-default. One way in which copatterns generalize curried functions is that each equation can take a *different* number of arguments. For example, consider this reordering of the `stutter` stream from section 2:

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

Since none of the copatterns overlap, its behavior is exactly the same as before. But notice the extra complication here: calling `((stutter 10) 'head)` with two arguments (`10` and `'head`) should immediately return `10`. However, the first equation is waiting for three arguments (an `n` and two `'tail`s passed separately). That means that the underlying code implementing `stutter` *cannot* ask for three arguments in three different calls and then checks that the last two are `'tail`. Instead, it has to eagerly match the arguments that it is given against the patterns and try each of the guards to see if the current line fails — and only *after* that all succeed, it may ask for more arguments and continue the copattern match.

Composition and the dimensions of extensibility The second set of challenges is due to the new notions of object composition that we develop here. In particular, we want to be able to combine objects in two different directions:

- *vertical composition* is an "either or" combination of two or more objects, such as `(o1 'compose o2 ...)` that acts like `o1` or `o2`, *etc,* depending on which one knows how to respond to the context. Textually, the vertical composition of `(object line-a1 ...)` and `(object line-b1 ...)` behaves as if we copied all each line of copattern-matching equations internally used to define the two objects and pasted them vertically into the newly-composed object as:

```
(object line-a1
        ...
        line-b1
        ...)
```

- *horizontal composition* is an "and then" object combination in a copattern-matching line, such as `[(self 'method1) (o1 'unplug)]` defining a `'method1` that continues to act like `o1` when `o1` knows how to respond to the surrounding context, and otherwise tries the next line. Textually, the horizontal composition of a `'method1` followed by trying another object with its own copattern-matching contexts `Q1 Q2 ...` acts as if the two copatterns are combined, and the inner object is inlined into the outer one like so:

```
(object [(self 'method1) ((object [Q1 = response1]
                                  [Q2 = response2]
                                  ...) 'unplug)]
        ...)
=
(object [(self 'method1) (comatch Q1) = response1]
        [(self 'method1) (comatch Q2) = response2]
        ...)
```

Even though we can visually understand the two directions of composition by the textual manipulations above, in reality, both of these compositions are done at run-time (*i.e.,* with arbitrary procedural values), as opposed to "compile-time" transformation (*i.e.,* macro-expansion time manipulations of code). This means we need an extensible representation of run-time object values that allows

for automatically switching from one object to another in the case of copattern-match failure, as well as correctly keeping track of what to try next.

The basic idea of this representation can be understood as an extension of an idiom in ordinary functional programming. In order to define an open-ended, pattern-matching function, we can give the cases we know how to handle now by matching on the arguments and include a default "catch-all" case at the end for the other behavior. In Haskell, this might look like

```
f next PatA1 PatA2 ... = expr1
f next PatB1 PatB2 ... = expr2
...
f next x1    x2    ... = next x1 x2 ...
```

For example, consider the single-line `eval-add` evaluator object from section 2. In order to compose `eval-add` with another evaluator handling a different case, like `eval-mul`, its internal extensible code takes an extra hidden argument saying what to try `next` if its line does not match, analogous to:

```
(define (eval-add-ext1 next)
  (lambda*
    [(self 'eval `(add ,l ,r)) = (+ (self 'eval l) (self 'eval r))]
    [ self                     = (next self)]))
```

Note that, unlike the Haskell code above, the hidden `next` parameter also takes *another* hidden parameter: `self`. Why? Because if the `next` set of equations needs to recurse, it cannot actually jump to itself directly — that would skip the `eval-add` code entirely — but needs to jump back to the very first equation to try. This `self` parameter holds the value of the *whole* object after all compositions have been done, as it appeared in the original call site. Thus, the internal extensible code `eval-add-ext` *also* takes this second `self` parameter for the same reason: it may be the second component of a composition, and it is similar to the following Racket definition:

```
(define ((eval-add-ext2 next) self)
  (match-lambda*
    [(list 'eval `(add ,l ,r)) (+ (self 'eval l) (self 'eval r))]
    [args                      ((next self) args)]))
```

One final detail to note: unless otherwise stated, the `self` parameter — which is visible as the root of any copattern — is *always* the same view of the entire object. That means nesting multiple copatterns in sequence might not give the expected result because the `self` parameter in the hole of every copattern context will be bound to the same value. If we instead want the parameter in the hole of every copattern context to reflect the object at that point in time — that is, be assigned the value given by the partial applications given by the preceding copatterns — we can use the `nest` operation. For example, nesting copatterns in a sequence gives us a shorthand for the common functional idiom of a "local" loop that closes over some parameters that never change, such as this definition of mapping a function over a list:

```
(define* [(map* f xs) = ((map* f) xs)]
         [(map* f) (nest)
           (extension
            [(go null)        = null]
            [(go (cons x xs)) = (cons (f x) (go xs))])])
```

The `map*` function supports both curried and uncurried applications, and they are defined to be equal. Its real code is given in the curried case, where the function parameter `f` is bound first and never changes. Then, in a second step, we have the internal looping function `go`, which matches over its list parameter and recurses with a new list. The definition of `go` is given directly in the form of an `extension` — the result of `'unplug`ging an object — which is composed with the curried application `(map* f)`. By using `nest`, we have `go = (map* f)`, so that `f` is visible from the closure but does not need to be passed again at every step of the loop.

3.2 The library API and macro desugaring

To meet all the implementation challenges listed above, the API of our compositional copattern library revolves around three groups of first-class values:

- *Objects* are ordinary values, typically functions, that can be used directly via application. Their exact interface varies based on their definition — such as a function from a list of numbers to a list of booleans, an evaluator from expression trees to numbers, or an object following the infinite stream interface — but they can be applied without any additional information.
- *Templates* represent openly-recursive, self-referential code without a fixed self. Instead, the "self" placeholder remains unbound for now, and it can be instantiated later via application to yield an *object* described above.
- *Extensions* represent extensible code that can be composed together both vertically and horizontally. Instead of failing on an unsuccessful match, will try an as-of-yet unspecified "next" option specified later via application to a template to produce a new template. This way, the "self" placeholder is also unbound for now — just like with templates — and can be bound later when the whole object is finished being composed.

Figure 1 describes a core API for forming or manipulating objects, templates, and extensions as first-class values. These individual operations can implement all of the examples seen thus far, which make heavy use of top-level definitional forms like `define*` and `define-object`. The grammar of syntax supported by these definitional forms, the operations which interpret copatterns, and multi-clause equations, is described in fig. 2. Let us now look at how each layer of the abstractions — from top-level definitions, multi-equation first-class values, nested copatterns, and the small terminal operations — can be desugared to the next one, all the way down to basic, purely-functional Scheme.

Object formation:

```
(λ* TemplateStx)                    : Object
(object ExtensionStx)               : Object
(object (<: Expr ...) ExtensionStx) : Object
(plug Extension)                    : Object
(introspect Template)               : Object
```

Template formation:

```
(template TemplateStx)   : Template
(continue x Expr)        : Template
(non-rec Expr)           : Template
(closed-cases Extension) : Template
```

Extension formation:

```
(extension ExtensionStx)           : Extension
(compose Extension ...)            : Extension
(comatch Copattern Extension)      : Extension
(chain ResponseStx)                : Extension
(always-is Expr)                   : Extension
(try x Template)                   : Extension
(try-if Expr Extension)            : Extension
(try-match Expr Pattern Extension) : Extension
(try-λ Params Extension)           : Extension
(nest Extension)                   : Extension
```

Fig. 1. Compositional copattern API.

Top-level definitions There are two main styles of top-level definitions. The simplest one is when an explicit name is given to the definition in question, which directly expands to an ordinary definition bound to a first class object — described via `λ*` or `object` — like so:

```
(define* x TemplateStx)        = (define x (λ* TemplateStx))
(define-object x ExtensionStx) = (define x (object ExtensionStx))
```

This style can be used to evoke an object-oriented flavor of code, in which an object is externally named `x` to the outside world, but internally refers to itself by some other name like `self` or `this` for recursive calls.

In a more functional style of code, recursive function definitions use the same name for both external callers as well as internal recursive calls, so there is no need to declare the name another time. We support this style of definition, too, by looking for the "root" of the initial copattern on the left-hand side, which gives a name to the function itself in the context of some application. Following the syntax of `Copattern`s in fig. 2, the root of the identifier `x` is just `x`, the anonymous wildcard `_` has no root, and the root of any other nested copattern is the root of its inner `Copattern`. Eliding the full details of digging into a copattern to reveal its root, if the name `x` is the root of first `Copattern` inside `TemplateStx`

```
TemplateStx  ::= ExtensionStx | ExtensionStx (else Expr)
               | ExtensionStx (continue x Expr)
ExtensionStx ::= (Copattern ResponseStx) ...
ResponseStx  ::= (op ...) ResponseStx | = Expr | try x Template

Params    ::= x | (Pattern ...) | (Pattern ... . x)
Copattern ::= x | _
            | (Copattern Pattern ...)
            | (Copattern Pattern ... . x)
            | (apply Copattern Pattern ... x)
Pattern   ::= x | _ | 'Expr | `QQPat | ...
QQPat     ::= ,Pattern | (QQPat ...) | Expr
```

Definitional forms:

```
(define* x TemplateStx)
(define* TemplateStx)
(define-object (x <: Expr ...) ExtensionStx)
(define-object x ExtensionStx)
(define-object (<: Expr ...) ExtensionStx)
(define-object ExtensionStx)
```

Fig. 2. Grammar of syntax used by multi-clause definitions and multi-step operations.

or `ExtensionStx`, then the implicitly-named definitional forms expand into the explicitly-named ones by copying that first recursive name:[3]

```
(define* TemplateStx)        = (define* x Template)
(define-object ExtensionStx) = (define-object x ExtensionStx)
```

Two more conspicuous cases introduce a *inheritance declaration* `<: Expr ...` to the definition. This implicitly incorporates additional code from a list of super-types in `Expr ...` to an object that is not explicitly written in the definition itself, allowing for a form of object-oriented-like inheritance evoked by the sub-typing `<:` syntax. The generalized definition is implemented directly in terms of the support for inheritance in anonymous `object`s. A `define-object` with an explicit name is expanded to just:

```
(define-object (x <: Expr ...) ExtensionStx)
= (define x (object <: Expr ...) ExtensionStx)
```

For an implicitly-named `define-object`, we must look for the name as before; assuming `x` is the first root inside `ExtensionStx`, the two forms produce the same definition:

[3] Note that we only extract the root name in the *first* clause of the definition to be used for the external name. In its full generality, a different internal name can be used in each clause. Only good taste dictates that the names in each clause should be the same to fit within a standard functional style.

```
(define-object (<: Expr ...) ExtensionStx)
= (define-object (x <: Expr ...) ExtensionStx)
```

First-class objects, templates, and extensions The most unassuming way to form a usable object — which does not introduce any implicit code not given in the object itself — is with the multi-clause `λ*` form. A `λ*` interprets the list of clauses as an openly-recursive `template`, and then closes off the recursive loop via `introspect` that binds the final object value in for the recursive parameter:

```
(λ* TemplateStx) = (introspect (template TemplateStx))
```

In contrast, an `object` is like a `λ*`, but enhanced with the ability to inherit code — given in a `(<: Expr ...)` constraint — which can extend or modify the clauses it contains. These modifiers are themselves transformations on extensions — first-class functions from old extensions to new extensions — which are applied to the given `extension` value before it is closed off with the `plug` operation on extensions analogous to `introspect` above:

```
(object ExtensionStx)
= (object (<:) ExtensionStx)
(object (<: Expr ...) ExtensionStx)
= (object (!<: meta Expr ...) ExtensionStx)
(object (!<: Expr ...) ExtensionStx)
= (plug ((compose Expr ...) (extension ExtensionStx)))
```

Note that the multiple extension modifiers are composed together using ordinary function composition, so the left-most one has precedence (being the final function in the chain). The empty inheritance `(<:)` is the same as if no inheritance constraint is given. Furthermore, the normal inheritance `(<: Expr ...)` is the same as a literal inheritance `(!<: meta Expr ...)` adding `meta` which is the implicit superclass of "all" objects.[4] Literal inheritance `(!<: Expr ...)` is instead interpreted exactly as-is, with no added defaults.

The default `meta` superclass provides the extra code for composition seen in the examples, including `'compose` and `'unplug`. How does it work? Since the superclasses used by the object inheritance mechanism are just extension transformers, we can implement `meta` directly as a function that composes the extension value making up an object with extra methods for manipulating it:

```
(define (meta ext)
  (compose ext
           (extension
             [(self 'unplug)    = ext]
             [(self 'adapt mod) = (plug (meta (mod (self 'unplug))))]
             [(self 'compose . os)
             = (plug (meta (apply compose
                                  (self 'unplug)
                                  (map (λ(o) (o 'unplug)) os))))])))
```

[4] The library implementation stores this choice of the default superclass in a mutable parameter `default-superclass` that can be changed by the programmer.

In other words, `(meta ext)` does everything that `ext` does, and in addition supports the methods:

- `(self 'unplug)` returns `ext` in its original state,
- `(self 'adapt mod)` is defined in terms of `(self 'unplug)` to retrieve the original extension value and apply some new transformation to it — plugging it back together with the same `meta` functionality — and
- `(self 'compose o ...)` will `'unplug` this object and each of `o ...` to get all the underlying extensions, composes them together vertically into one, and then plugs it back into a usable `meta` object again.

Now, what are these first-class extensions and templates, and how are they created? The `extension` macro — defined in terms of a list of clauses with an initial `Copattern` paired with some response (a combination of guards or other operations before a right-hand side) — interprets each line separately as its own horizontal composition `chain` before vertically composing them together with `compose` like so:

```
(extension [Copattern ResponseStx] ...)
= (compose [chain (comatch Copattern) ResponseStx] ...)
```

The main difference between a template and an extension is that a template is responsible for providing the final "catch-all" clause saying what to do if nothing matches. As such, the `template` macro first builds an `extension` out of the given syntax, and then applies it to one of three final clauses:

```
(template ExtensionStx [else Expr])
= ((extension ExtensionStx) (non-rec Expr))
(template ExtensionStx [continue x Expr])
= ((extension ExtensionStx) (continue x Expr))
(template ExtensionStx)
= (closed-cases (extension ExtensionStx))
```

The first case has an `[else Expr]` clause that evaluates some default expression if nothing else matches. The `[continue x Expr]` clause in the second case is similar but more general: `Expr` is evaluated if nothing else matches, and `x` is bound to the whole template itself, allowing for `Expr` to continue the recursive loop again if it uses `x`. The final case, providing no catch-all clause, uses `closed-cases` to instead raise an error if nothing matches.

Nested copattern matching and chains Nested copattern matching is now straightforward to desugar in terms of horizontally composing curried sequences of more basic, λ-like forms. More specifically, `try-`λ is a functional abstraction over extensions analogous to the normal functional abstraction over expressions, and mimics all three of Scheme's λ forms: `(try-`λ `(Pattern ...) Extension)` matches a sequence of arguments against the given sequence of parameter patterns, `(try-`λ `(Pattern ... . x) Extension)` does the same while binding any additional arguments as a list to `x`, and `(try-`λ `x Extension)` just binds any list of arguments to `x`. These three forms correspond to three application contexts in a copattern, which are desugared like so:

```
(comatch (Copattern . x) Extension)
= (comatch Copattern (try-λ x Extension))
(comatch (Copattern Pattern ...) Extension)
= (comatch Copattern (try-λ (Pattern ...) Extension))
(comatch (Copattern Pattern ... . x) Extension)
= (comatch Copattern (try-λ (Pattern ... . x) Extension))
```

Alternate copattern forms described using `apply`, like `(apply Copattern x y z)`, are defined similarly to the above.

An `extension` value is also created by chaining this initial `comatch` with some response. In practice, this chaining just involves reassociating the parentheses of intermediate operations and interpreting the right-hand side, so `chain` has the following behavior:

```
(chain (op ...) ResponseStx) = (op ... (chain ResponseStx))
(chain = Expr)               = (always-is Expr)
(chain try x Template)       = (try x Template)
```

Because reassociation does not depend on the operations `op` involved, extension chains automatically work with *any* new user-defined operations — macros or functions — which turn an extension (as their last argument) into another extension.

Terminal operations and combinators Single-step operations and combinators are terminal forms, defined directly in terms of plain Scheme expressions. The simplest ones of these are also the most general: the `try` and `continue` macros let programmers write arbitrary extensions and templates (respectively) by abstracting over the next template to try or the object to continue recursion (respectively). In reality, both of these forms desugar directly into plain λ-abstractions.

```
(try x Template)  = (λ(x) Template)
(continue x Expr) = (λ(x) Expr)
```

Common special cases include the self-contained forms: the non-recursive template `(non-rec Expr)` and the fully-committed extension `(always-is Expr)` that evaluate and run `Expr` without failing to the next case or looping back to the beginning. Both forms are shorthand for the above abstractions that just never use the bound variable.

```
(non-rec Expr)   = (continue _ Expr)
(always-is Expr) = (try _ (non-rec Expr)) = (try _ (continue _ Expr))
```

The remaining basic combinators for adding guards and functional abstractions to extensions can be defined in terms of the above. A convenient shorthand for manually writing new extension combinators is to abstract over both the "next" case to try and the "self" reference at the same time, which is just a shorter notation for a curried function:

```
(try next self Expr) = (try next (continue self Expr))
                     = (λ(next) (λ(self) Expr))
```

The boolean `try-if` guard will check some boolean expression first: if it is true it continues to try the underlying extension (which may succeed or fail on its own accord), and otherwise skips it entirely.

```
(try-if Expr Extension)
= (try next self
       (if Expr
           ((Extension next) self)
           (next self)))
```

The pattern-matching guard `try-if` operates similarly, but generalizes the simple boolean check to matching the value of an expression against a given pattern.

```
(try-match Expr Pattern Extension)
= (try next self
       (match Expr
           [Pattern ((Extension next) self)]
           [_ (next self)]))
```

The most complicated single combinator is the `try-λ` functional abstraction over an extension. This form presents a functional interface that can pattern-match against its arguments: if the match succeeds then the variables in the pattern are bound while trying to execute the underlying extension, but if the match fails the extension is never tried at all. The pattern-matching component can be factored out of `try-λ` via horizontal composition by first just binding the arguments as is, and then matching them against the given patterns in a second step like so:[5]

```
(try-λ (Pattern ...) Extension)
= (try-λ (x ...) (chain (try-match x Pattern) ... Extension))
```

Now we only have to handle the simpler job of accepting a given number of arguments. This operation can still fail if applied to a different number of arguments, or if the underlying extension fails. We can efficiently define different cases for these two cases using a `case-λ` which combines functions of different numbers of arguments like so: [6]

```
(try-λ (x ...) Extension)
= (try next self
       (case-λ
         [(x ...) ((Extension (continue s ((next s) x ...))) self)]
         [args    (apply (next self) args)]))
```

Notice the most subtle part of `try-λ`: we must ensure that when we try the next case to handle failure, they must be copied and passed again whenever `next` is

[5] This equation is more of a specification than an efficient implementation. In practice, the `try-match` operations are only generated if the given pattern is non-trivial.

[6] Alternatively, some implementations like Racket have a built-in `match-λ` which effectively extends `case-λ` to pattern-match directly on the arguments. In these systems, we can use `match-λ` here to handle both pattern-matching and argument passing here at the same time.

invoked. If we forgot to wrap both applications of `next` with the given arguments, they would disappear forever if we handle to fall through to the next case.

Object-manipulating functions The final operations remaining in the API are not even macros: they are just ordinary functions. For example, the `compose` that we have used twice now — to implement vertical composition of extensions as well as to combine multiple superclass constraints — is just ordinary function composition! Some of the other purely functional operations are similarly simple. For example, `closed-cases` closes off an extension by providing a final catch-all case that raises an error when run, and `plug`ging an extension to get an object is just composition of `introspect` and `closed-cases`.

```
(closed-cases Extension) = (Extension (non-rec (error "...")))
(plug Extension)         = (introspect (closed-cases Extension))
```

The `introspect` function is a little tricker, because it needs to plug in the final form of an object while creating that object. Attempting to recursively tie the knot directly as

```
(introspect Template)
=/= (letrec ([self (Template self)])
      self)
```

will cause a run-time error: since Scheme is a call-by-value language, it will try to evaluate `self` before it is actually bound to a value yet. So instead, we have to delay the self-reference just by one step by η-expanding, since λ-abstractions return values without evaluating their bodies.

```
(introspect Template)
= (letrec ([self (Template (λ args (apply self args)))])
    self)
```

The `nest` operation does the same knot-tying as `introspect`, but also returns a new extension, rather than an object. To do so, it will recursively instantiate the view of the extension at this point in time — after some applications and guards have been evaluated — to provide the new self-referencing value. If at any point this extension fails and falls past its last case, the updated self-reference will be undone, and reverts back to its old form.

```
(nest Extension)
= (try next there
     (letrec ([here ((Extension (non-rec (next there)))
                     (λ args (apply here args)))])
       here))
```

Use case: Defining new custom macros via the API To demonstrate the flexibility of the API, we show how some other, more niche, operations can be defined in terms of the ones we already saw. In particular, the case studies on the

object-oriented file system and the expression problem involved some complex meta-programming facilities.

First, we needed to override some of the behavior of multi-clause λ*s, which can be done by wrapping its value in a new λ* that handles the new behavior or else falls through to the original.

```
(override-λ* Expr ExtensionStx)
= (λ* ExtensionStx [else Expr])
```

Next, we needed to temporarily replace the self-reference in an extension with another, until we fall out of the scope of that extension. This can be done by capturing the old notion of the object itself — to be used if the extension ever falls through past its end — and passing a new one in its place like so:

```
(with-self new-self Extension)
= (try next old-self ((Extension (non-rec (next old-self))) new-self))
```

For the object-oriented examples, we needed a way to import the code of one object into another. We also used shorthand to construct an object in two steps (initialization and then recursive method definition). Both of these two operations are implemented as ordinary functions using the above library functionality that behave like so:

```
((import-object super-obj) sub-ext)
= (compose sub-ext (super-obj 'unplug))

(construct state obj)
= (extension
   [self try (with-self
               (λ args (apply (apply self state) args))
             (obj 'unplug))]))
```

Finally, to extend an evaluator with an environment, we needed a combinator to apply a functional extension to some argument(s) in the form of a new extension. The most direct way to do so is to apply those arguments after passing the `next` template and `self` object

```
(try-apply-remember Extension Expr ...)
= (try next self
       (((Extension next) self) Expr ...))
```

however, this does not give the desired behavior! This application looks "permanent" from the perspective of the next cases that follow this extension, which will be passed the arguments `Expr ...` as if they were in the original calling context. Instead, we want any alternatives that handle `Extension`'s failure to be evaluated in the same context, and so the extra arguments `Expr ...` need to be forgotten like so:

```
(try-apply-forget Extension Expr ...)
= (try next self
       (let ([forgetful (continue self (λ _ (next self)))])
         (((Extension forgetful) self) Expr ...)))
```

3.3 Practical details of macro definitions

One concern for a real implementation is to consider what kind of pattern-matching facilities the host language already provides. Unfortunately, the answer is not standard across different languages in the Scheme family. For example, the R^{6}RS standard does not require any built-in support for pattern matching to be fully compliant, but specific languages like Racket may include a library for pattern matching by default. Thus, we provide two different implementations to illustrate how copatterns may be implemented depending on their host language:

- A Racket implementation that uses its standard pattern-matching constructs `match` and `match-lambda*`. Thus, the **match** from the target language in fig. 3 is interpreted as Racket's `match`, and the translation of $E[\![\lambda P.O]\!]$ is implemented directly as `match-lambda*` instead of separating the λ from the pattern as in fig. 5. This choice ensures the pattern language implemented is exactly the same as the pattern language already used in Racket programs.
- A general implementation intended for any R^{6}RS-compliant Scheme,[7] which internally implements its own pattern-matching macro, `try-match`, by expanding into other primitives like `if` and comparison predicates. Of note, due to only having to handle a single line of pattern-matching at a time, this implementation is 75 lines of Scheme and supports quasiquoting forms of patterns. This gives a sufficiently expressive intersection between Racket's pattern-matching syntax and the manually implemented R^{6}RS version.

4 A Core Copattern Calculus

4.1 Double-barrel translation

To help study the behavior and correctness of composable copattern matching, we model a simplified version of the library API in the form of an extended λ-calculus, and give a high-level translation into a conventional λ-calculus with recursion and pattern matching (given in fig. 3). Our pattern language is modeled after a small common core found among various implementations of Scheme, which includes normal variable wildcards x that can match anything, quoted symbols $'x$, and lists of the form null or $(\text{cons}\, P\, P')$. Note that we assume all bound variables x in a pattern are distinct. As shorthand, we write a list of patterns $P_1\ P_2\ \ldots\ P_n$ for $(\text{cons}\, P_1\ (\text{cons}\, P_2\ \ldots (\text{cons}\, P_n\, \text{null})))$. To model the patterns found in typed functional languages like ML and Haskell, such as constructor applications $K\ P\ldots$, we can represent the constructor as a quoted symbol $'K$ and the application as a list $'K\ P\ldots$. The patterns' specifics are surprisingly not essential to the main copattern translation and could be extended with other features found in more specific implementations.

For simplicity, this translation begins from a smaller source language with copatterns (given in fig. 4) separated into three main syntactic categories that reflect the different groups of values from the macro library:

[7] We have explicitly tested this implementation against Chez Scheme.

$$\begin{array}{rl} \mathit{Term} \ni M, N & ::= x \mid M\ N \mid \lambda x.M \mid K \mid \mathbf{match}\, M\, \mathbf{with}\, \{\, P \to N \ldots \,\} \mid \mathbf{rec}\, x = M \\ \mathit{Pattern} \ni & \quad P ::= x \mid \text{'}x \mid \text{null} \mid \text{cons}\, P\, P' \end{array}$$

Fig. 3. Target language: pure λ-calculus with pattern-matching and recursion.

- (M, N) *Term* syntax represents all ordinary expressions of the host language as well as the new first-class *objects* of the library. The new forms of terms are $\lambda^* B$, which gives a self-referential copattern-matching object, along with **template** B and **extension** O which include the other two syntactic categories as first-class values that can be applied as functions to instantiate their open-ended recursion and composition.
- (B) *Template* syntax represents a simplified grammar supported by `template` and similar macros specified as `TemplateStx` in fig. 2. Including some extension cases in a template is written as $O; B$, the catch-all clause which may continue the loop again via a recursive object bound to x is written as **continue** $x \to M$, and the closed case where the catch-all clause raises an error is the empty string ε. Since the simpler final **else** clause is a special case of **continue**, we treat it as syntactic sugar.
- (O) *Extension* syntax represents a simplified grammar supported by `extension` and similar macros specified as `ExtensionStx` in fig. 2 with terser notation. Vertical composition is written as $O; O'$, similar to $O; B$, with the empty string ε as its neutral element. Copattern-matching is written as $Q[x]O$, where Q is a copattern context with x as the root identifier naming the recursive object itself. The more basic forms are written as $\lambda P.O$ for a functional abstraction over an extension, **match** $P \leftarrow M\ O$ for a pattern-matching guard, and **try** $x \to B$ for the statement which binds the following cases to x before running a template specified by B. We treat if-guards and the form $(= M)$ as syntactic sugar for special cases of the more general forms, and also sometimes use the alternative notation **do** M in place of $(= M)$ in contexts where the latter notation appears awkward.

The syntax in B and O directly reflects the core operations for forming and combining copattern-matching expressions of the library API. Here, the copattern syntax $Q[x]$ itself is expressed as a subset of contexts, Q, surrounding an object internally named x. Two lines separated by a semicolon $(O; O')$ represents a binary vertical composition `compose` that tries either O or O', and ε represents an empty extension (`extend`) with respect to vertical composition: it immediately refers to the next option. Prefixing with a copattern-matching expression $(Q[x]\ O)$ represents the (`comatch Q[x] O`) form that tries $Q[x]$ and then O. Smaller special cases of matching include pattern lambdas ($\lambda P.O$) for `try-`λ, and pattern guards (**match** $P \leftarrow M\ O$) for `try-match`. Other operations use the same names as in fig. 1.

This simplified grammar makes it easier to define the full macro expansion as a translation from the source (fig. 4) to target (fig. 3) as given in fig. 5. This

$$
\begin{aligned}
\mathit{Term} \ni M, N &::= \cdots \mid \lambda^* B \mid \mathbf{template}\, B \mid \mathbf{extension}\, O \\
\mathit{Template} \ni \quad B &::= \varepsilon \mid O; B \mid \mathbf{continue}\, x \to M \\
\mathit{Extension} \ni \quad O &::= \varepsilon \mid O; O' \mid Q[x]\ O \mid \lambda P.\ O \mid \mathbf{match}\, P \leftarrow M\ O \mid \mathbf{try}\, x \to B \\
\mathit{Copattern} \ni \quad Q &::= \square \mid Q\ P \\
\mathit{Pattern} \ni \quad P &::= x \mid \text{'}x \mid \mathrm{null} \mid \mathrm{cons}\, P\, P'
\end{aligned}
$$

Syntactic sugar:

$$
\begin{aligned}
\mathbf{else}\, M &= \mathbf{continue}\ \_ \to M & \qquad (= M) &= \mathbf{do}\, M = \mathbf{try}\ \_ \to \mathbf{else}\, M \\
\mathbf{if}\ M\ O &= \mathbf{match}\, \mathrm{True} \leftarrow M\ O & \qquad (\mathbf{let}\, x = M\ O) &= \mathbf{match}\, x \leftarrow M\ O
\end{aligned}
$$

Fig. 4. Source language: target extended with nested copatterns, self-referential objects, recursion templates, and composable extensions.

translation shares many similarities to continuation-passing style (CPS) translations. However, we explicitly avoid converting the entire program to CPS. Notably, every syntactic form for the source language is unchanged; for example, $[\![M\ N]\!] = [\![M]\!]\ [\![N]\!]$. Instead, the only time we need to introduce an extra parameter is for the two new syntactic categories. All templates are translated to functions that take a value for the whole object itself to a new version of that object. Similarly, all extensions are translated to functions that take both a template as the "base case" to try next and a value for the whole object itself. Even though this is dynamically-typed, we can view the type of templates as object transformers and extensions as template transformers:

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

The interesting cases for translating terms are the new forms. **template** B and **extension** O are just translated to their given forms as transformation functions. With $\lambda^* B$, we need to recursively plug its translation in for its self parameter. Note the one detail that the recursive *self* is η-expanded to in this application. This ensures that $\lambda x.\mathit{self}\ x$ is treated as a value in a real implementation, and is always safe assuming that B describes a function (non-functional cases of $\lambda^* B$ are undefined user error).

For templates and extensions, the terminators **continue** and **try** are translated to plain λ-abstractions that allow the programmer direct access to their implicit parameters. Complex copatterns $(x\ P_1 ... P_n\ O)$ are reduced down to a simpler sequence of pattern lambdas $(x\ \lambda P_1. \ldots \lambda P_n.\ O)$, and pattern lambdas $(\lambda P.O)$ are reduced down to a simpler non-matching lambda followed by an explicit match guard $(\lambda x.\, \mathbf{match}\, P \leftarrow x\ O)$.

This leaves just the base cases of simple extension forms. Each time an extension (of form $\lambda b.\lambda s. \ldots$) "fails," it calls the given next template with the given

Translating new terms:

$$\begin{aligned}
[\![\lambda^* B]\!] &= (\mathbf{rec}\ \mathit{self} = T[\![B]\!]\ (\lambda x.\mathit{self}\ x)) \\
[\![\mathbf{template}\, B]\!] &= T[\![B]\!] \\
[\![\mathbf{extension}\, O]\!] &= E[\![O]\!] \\
[\![M]\!] &= \text{by induction} && \text{(otherwise)}
\end{aligned}$$

Translating templates:

$$\begin{aligned}
T[\![\varepsilon]\!] &= \lambda s.\mathit{fail}\ s \\
T[\![O; B]\!] &= \lambda s.E[\![O]\!]\ T[\![B]\!]\ s \\
T[\![\mathbf{continue}\, x \to M]\!] &= \lambda x.[\![M]\!]
\end{aligned}$$

Translating copattern-matching and pattern-matching functions:

$$\begin{aligned}
E[\![(Q[x]\ P)\ O]\!] &= E[\![Q[x]\ (\lambda P.O)]\!] \\
E[\![x\ O]\!] &= \lambda b.\lambda x.E[\![O]\!]\ b\ x \\
E[\![\lambda P.O]\!] &= E[\![\lambda x.\, \mathbf{match}\, P \leftarrow x\ O]\!] && (\text{if } P \notin \mathit{Variable})
\end{aligned}$$

Translating other extensions:

$$\begin{aligned}
E[\![\varepsilon]\!] &= \lambda b.\lambda s.b\ s \\
E[\![O; O']\!] &= \lambda b.\lambda s.E[\![O]\!]\ (E[\![O']\!]\ b)\ s \\
E[\![\lambda x.O]\!] &= \lambda b.\lambda s.(\lambda x.E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s) \\
E[\![\mathbf{match}\, P \leftarrow M\ O]\!] &= \lambda b.\lambda s.\, \mathbf{match}\, [\![M]\!]\, \mathbf{with}\, \{\, P \to E[\![O]\!]\ b\ s; _ \to b\ s\,\} \\
E[\![\mathbf{try}\, x \to B]\!] &= \lambda x.T[\![B]\!]
\end{aligned}$$

Fig. 5. Translating copattern-based source code to the target language.

self object ($b\ s$). A match guard $[\![\mathbf{match}\, P \leftarrow M\ O]\!]$ will try to match the translation of M against the pattern P; the success case continues as $E[\![O]\!]$ with the same next template and self. A non-matching lambda $[\![\lambda x.O]\!]$ always succeeds (for now), but note that the next template to try on failure has to be changed to include the given argument. Why? Because the lambda has already consumed the next argument from its context, it would be gone if, later on, the following operations fail and move on to the next option. So instead of invoking the given b directly as $b\ s'$ (for a potentially different future s'), they need to invoke b applied to this argument x as $b\ s'\ x$.

In this translation, we also give the η-reduced forms on the right-hand side when available. This shows that the empty extension ε is just the identity function (given the next thing b to try, ε does nothing and immediately moves on to b), and horizontal composition $O; O'$ is just ordinary function composition.

$$\begin{aligned} \mathit{Value} \ni V, W &::= x \mid \lambda x.M \mid \mathrm{null} \mid \mathrm{cons}\, V\, W \mid \text{'}x \\ \mathit{EvalCxt} \ni E &::= \Box \mid E\ M \mid V\ E \mid \mathbf{match}\, E\, \mathbf{with}\, \{\, P \to N ... \,\} \mid \mathbf{rec}\, x = E \end{aligned}$$

$$(\beta) \qquad (\lambda x.M)\ V = M[V/x]$$

$$(\mathit{match}) \qquad \mathbf{match}\, V\, \mathbf{with}\, \{\ P \to N;\ P' \to N' ...\ \} = N[W.../x...] \qquad (\text{if } P[W.../x...] = V)$$

$$(\mathit{apart}) \qquad \mathbf{match}\, V\, \mathbf{with}\, \{\ P \to N;\ P' \to N' ...\ \} = \mathbf{match}\, V\, \mathbf{with}\ \{\, P' \to N' ...\,\} \qquad (\text{if } P \# V)$$

$$(\mathit{rec}) \qquad (\mathbf{rec}\, x = V) = V[(\mathbf{rec}\, x = V)/x]$$

Apartness between patterns and values ($P \# V$):

$$\frac{V \notin \mathit{Variable} \cup \{\, \text{'}x \,\}}{\text{'}x \# V} \qquad \frac{V \notin \mathit{Variable} \cup \{\, \mathrm{null} \,\}}{\mathrm{null} \# V}$$

$$\frac{V \notin \mathit{Variable} \cup \{\, \mathrm{cons}\, W\ W' \mid W, W' \in \mathit{Value} \,\}}{\mathrm{cons}\, P\ P' \# V}$$

$$\frac{P \# W}{\mathrm{cons}\, P\ P' \# \mathrm{cons}\, W\ W'} \qquad \frac{P' \# W'}{\mathrm{cons}\, P\ P' \# \mathrm{cons}\, W\ W'}$$

Fig. 6. Untyped equational axioms of the target language.

4.2 Correctness

We already used the translation to a core λ-calculus as a specification for implementing compositional copatterns, but the translation is also useful for another purpose: checking the expected meaning of copattern-matching code. With that in mind, we now look for some laws that let us equationally reason about some programs to make sure they behave as expected.

First, the core target language — a standard call-by-value λ-calculus extended with pattern-matching and recursion — has the equational theory shown in fig. 6, which is the *reflexive*, *symmetric*, *transitive*, and *compatible* (*i.e.,* equalities can be applied in *any* context) closure of the listed rules. It has the usual β axiom (restricted to substituting value arguments), two axioms for handling pattern-match success (*match*) and failure (*apart*), and an axiom for unrolling recursive values (*rec*). Values (V, W) include the usual ones for call-by-value λ-calculus (x and $\lambda x.M$) as well as lists (null and $\mathrm{cons}\, V\ W$) and symbolic literals ('x). Matching a value V against a pattern P will succeed if the variables ($x...$) in the pattern can be replaced by other values ($W...$) to generate exactly that V: $P[W.../x...] = V$. In contrast, matching fails if the two are known to be *apart*, written $P \# V$ and defined in fig. 6, which implies that all possible substitutions of P will *never* generate V. Note that while matching and apartness are mutually exclusive, there are some values that are neither matching nor apart from

$$\begin{aligned} \mathit{ExtensionFunc} \ni F &::= Q[x\ P]\ O \mid \lambda P.O \\ \mathit{Value} \ni V &::= \cdots \mid \lambda^*(F; B) \mid \mathbf{template}\, B \mid \mathbf{extension}\, O \end{aligned}$$

Identity, associativity, and annihilation laws of composition:

$$\begin{array}{lll} \varepsilon; O = O & (O_1; O_2); O_3 = O_1; (O_2; O_3) & \mathbf{do}\, M; O = \mathbf{do}\, M \\ \varepsilon; B = B & (O_1; O_2); B = O_1; (O_2; B) & \mathbf{do}\, M; B = \mathbf{else}\, M \end{array}$$

Pattern and copattern matching:

$$\begin{aligned} \mathbf{match}\, P \leftarrow V\ O &= O[W.../x...] && (\text{if } P[W.../x...] = V) \\ \mathbf{match}\, P \leftarrow V\ O &= \varepsilon && (\text{if } P \# V) \\ (\mathbf{template}\, (\lambda P.\, \mathbf{do}\, M); B)\ V'\ V &= M[W.../x...] && (\text{if } P[W.../x...] = V) \\ (\mathbf{template}\, (\lambda P.O); B)\ V'\ V &= (\mathbf{template}\, B)\ V'\ V && (\text{if } P \# V) \\ C[(\mathbf{template}\, (Q[y] = M); B)\ V] &= M[V/y][W.../x...] && (\text{if } Q[W.../x...] = C) \\ C[(\mathbf{template}\, (Q[y]\ O); B)\ V] &= C[(\mathbf{template}\, B)\ V] && (\text{if } Q \# C) \\ C[\lambda^*(Q[y] = M); B] &= M[(\lambda^*(Q[y] = M); B)/y] & & (\text{if } Q[W.../x...] = C) \\ & \quad [W.../x...] & & \\ C[\lambda^*(Q[y]\ O); \mathbf{else}\, M] &= C[M] && (\text{if } Q \# C) \end{aligned}$$

Apartness between copatterns and contexts ($Q \# C$):

$$\frac{Q[W.../x...] = C \quad P \# V}{Q\ P \# C\ V} \qquad \frac{Q \# C}{Q\ P \# C} \qquad \frac{Q \# C}{Q \# C\ V}$$

Fig. 7. Some equalities of copattern extensions.

some patterns. For example, compare the variable x against the pattern null; x may indeed stand for null or another value like $\lambda y.M$.

The first usual property is that the translation specified in fig. 5 is a *conservative extension*: any two terms that are equal by the target equational theory are still equal after translation. Because the translation is hygienic and compositional by definition, we can follow the proof strategy in [9].

Proposition 1 (Conservative Extension). *If $M = N$ in the equational theory of the target (fig. 6), then so too does $[\![M]\!] = [\![N]\!]$.*

To reason about the new features in the source language — introduced by λ^*, **template**, and **extension** — we introduce additional axioms given in fig. 7, so that the source equational theory is the *reflexive*, *symmetric*, *transitive*, and *compatible* closure of these rules in both figs. 6 and 7. The purpose of these new equalities is to perform some reasoning about programs using copatterns, and in particular, to check that the syntactic use of = really means equality. For example, a special case is $Q[\lambda^*(Q[y] = M); B] = M[\lambda^*(Q[y] = M); B/y]$, which says a λ^*

appearing in *exactly* the same context as the left-hand side of an equation will unroll (recursively) to the right-hand side. Other equations describe algebraic laws of copattern alternatives and how to fill in templates and extensions when applied. This source equational theory is *sound* with respect to translation.

Proposition 2 (Soundness). *The translation is* sound *w.r.t. the source and target equational theories (*e.g., $M = N$ *in fig. 7 implies* $[\![M]\!] = [\![N]\!]$ *in fig. 6).*

Proofs of these propositions are given in section A of the appendix.

5 Related Work

While we use the expression problem as a motivating application of compositional copatterns, it has multiple previous solutions using various features and spreading on multiple paradigms. The Visitor pattern [19] is a classic solution in the object-oriented world. However, we have other options such as [30], which brings Java interfaces closer to Haskell's type classes, and [21], which presents the abstraction of object algebras that are easily integrated into OO languages, since they do not depend upon fancy features. On the other hand, on the functional paradigm side, we have solutions like [24] and [16]. The former utilizes Haskell's type system, specifically type classes, to provide a modular way to construct data types and functions. However, this approach relies on a type system. The latter is a compiler framework that provides tools to specify different languages for the compiler passes. The framework achieves this by automatically handling a portion of the mapping between two given entities.

Previously, copatterns have been developed exclusively from the perspective of statically-typed languages. Much of the work has been for dependently typed languages like Agda [6], which use a type-driven approach to elaborate copatterns [23,25]. The closest related works are about implementing copatterns using macros in OCaml [18,15]. As with the above work in dependently typed languages, [18] is also concerned with type system ramifications, and CoCaml [15] supports only a subset of coinductive data types, called *regular*, with periodic repetitions that allow for finite representation in memory using cyclic structures. Here, we show how to implement copatterns with no typing information or restrictions on cyclic structures, and focus instead on composition and equational reasoning. The literature also presents other examples of open recursion and inheritance in a functional setting; for instance, [4] implements memorization for monadic computations using inheritance and mixins.

The translation in fig. 5 is reminiscent of "double-barreled CPS" [26] used to define control effects like delimited control [8] and exceptions [17]. In our case, rather than a "successful return path" continuation, there is a "resume recursion" continuation. Expressions that return successfully just return as normal, similar to a selective CPS [20], which makes it possible to implement as a macro expansion. A "next case" continuation — to handle copattern-matching failure — is introduced to make each line of a copattern-based definition a separate first-class

value. From that point, the "recursive self" must be a parameter because no one sliver of a definition suffices to describe the whole.

Theories of object-oriented languages [1,7] also model the "self" keyword as a parameter later instantiated by recursion; either as an explicit recursive binding, or encoded as self-application. This is done to handle the implicit composition of code from inheritance, whereas here, we need to handle explicit composition of first-class extensible objects. The full connection between copatterns — as we describe here — and object-oriented languages remains to be seen. In terms of the Lisp family of languages, the approach here seems closest to a first-class generalization of *mixins* [3,11] with a simple dispatch mechanism (matching), in contrast to class-based frameworks focused on complex dispatch [12,14,5].

6 Conclusion

We have shown here how to implement a more extensible, compositional version of copatterns as a macro in standard Scheme as well as Racket. Our major focus involves new ways to compose (co)pattern matching code in multiple directions — vertically and horizontally — which can be used to solve the expression problem since it can encode certain functional and object-oriented design patterns. Despite the more general forms of program composition, we still support straightforward equational reasoning to understand code behavior, even when that code is assembled from multiple parts of the program. This equational reasoning is formalized in terms of an extended λ-calculus, which is soundly translated into a common core calculus familiar to functional programmers; we leave the definition of a *complete* and minimal equational theory for copatterns as future work.

Our work here does not include static types, inherited from Scheme's nature as a dynamically typed language. As future work, we intend to develop a type system for the copattern language described here; specific challenges include correctly specifying type types of (de)composed code as well as coverage analysis that ensures every case is handled after the composition is finished. The second direction of future work is to incorporate effects into copattern definitions and their equational reasoning, for example, subsuming (delimited) control operators into the copattern language as a way of expressing compositional effect handlers.

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

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer (1996)
2. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 27–38. POPL '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2429069.2429075
3. Bracha, G., Cook, W.R.: Mixin-based inheritance. In: Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, OOPSLA/ECOOP 1990, Ottawa, Canada, October 21-25, 1990, Proceedings. pp. 303–311. ACM (1990). https://doi.org/10.1145/97945.97982
4. Brown, D.S., Cook, W.R.: Function inheritance : Monadic memoization mixins (2009)
5. Chambers, C.: Object-oriented multi-methods in Cecil. In: European Conference on Object-Oriented Programming. pp. 33–56. Springer (1992)
6. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proceedings of the ACM on Programming Languages **2**(ICFP) (2018). https://doi.org/10.1145/3236770
7. Cook, W.R., Palsberg, J.: A denotational semantics of inheritance and its correctness. Information and Computation **114**(2), 329–350 (1994). https://doi.org/10.1006/INCO.1994.1090
8. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990. pp. 151–160. ACM (1990). https://doi.org/10.1145/91556.91622
9. Downen, P., Ariola, Z.M.: Compositional semantics for composable continuations: From abortive to delimited control. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 109–122. ICFP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628147
10. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Communications of the ACM **61**(3), 62–71 (2018). https://doi.org/10.1145/3127323
11. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 171–183 (1998)
12. Gabriel, R., White, J., Bobrow, D.: Clos: Integrating object-oriented and functional programming. Communications of the ACM **34**, 28–38 (1991). https://doi.org/10.1145/114669.114671
13. Hughes, J.: Why functional programming matters. The Computer Journal **32**(2), 98–107 (1989). https://doi.org/10.1093/comjnl/32.2.98
14. Ingalls, D.H.H.: A simple technique for handling multiple polymorphism. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. p. 347–349. OOPSLA '86, Association for Computing Machinery, New York, NY, USA (1986). https://doi.org/10.1145/28697.28732
15. Jeannin, J.B., Kozen, D., Silva, A.: CoCaml: Functional programming with regular coinductive types. Fundamenta Informaticae **150**(3), 347–377 (2017). https://doi.org/10.3233/FI-2017-1473, https://journals.sagepub.com/doi/full/10.3233/FI-2017-1473

16. Keep, A.W., Dybvig, R.K.: A nanopass framework for commercial compiler development. In: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. pp. 343–350. ICFP '13, Association for Computing Machinery (2013). https://doi.org/10.1145/2500365.2500618
17. Kim, J., Yi, K., Danvy, O.: Assessing the overhead of ML exceptions by selective CPS transformation. BRICS Report Series **5** (1998)
18. Laforgue, P., Régis-Gianas, Y.: 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, October 09 - 11, 2017. pp. 97–108 (2017). https://doi.org/10.1145/3131851.3131869
19. Lasater, C.G.: Design Patterns. Wordware Publishing, Inc. (2006)
20. Nielsen, L.R.: A selective CPS transformation. In: Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001. Electronic Notes in Theoretical Computer Science, vol. 45, pp. 311–331. Elsevier (2001). https://doi.org/10.1016/S1571-0661(04)80969-1
21. Oliveira, B.C.D.S., Cook, W.R.: Extensibility for the masses. In: ECOOP 2012 – Object-Oriented Programming, vol. 7313, pp. 2–27. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_2
22. Peyton Jones, S., Reid, A., Henderson, F., Hoare, T., Marlow, S.: A semantics for imprecise exceptions. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. p. 25–36. PLDI '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/301618.301637
23. Setzer, A., Abel, A., Pientka, B., Thibodeau, D.: Unnesting of copatterns. In: Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8560, pp. 31–45. Springer (2014). https://doi.org/10.1007/978-3-319-08918-8_3
24. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4) (2008). https://doi.org/10.1017/S0956796808006758
25. Thibodeau, D.: Programming Infinite Structures using Copatterns. Master's thesis, School of Computer Science, Mcgill University, Montreal (2015)
26. Thielecke, H.: Comparing control constructs by double-barrelled CPS. Higher-Order and Symbolic Computation **15**, 141–160 (2002). https://doi.org/10.1023/A:1020887011500
27. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 132–141. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993514
28. Wadler, P.: A critique of Abelson and Sussman or why calculating is better than scheming. SIGPLAN Not. **22**(3), 83–94 (1987). https://doi.org/10.1145/24697.24706
29. Wadler, P., et al.: The expression problem. Posted on the Java Genericity mailing list (1998)
30. Wehr, S., Thiemann, P.: JavaGI: The interaction of type classes with interfaces and inheritance. ACM Transactions on Programming Languages and Systems **33**(4), 1–83 (2011). https://doi.org/10.1145/1985342.1985343

A Proofs

A.1 Conservative extension of the target

Lemma 1. *The following instances of translation are all values up to the equational theory of the target language in fig. 6:*

(a) $T[\![B]\!] = \lambda s.M$ *for some term* M*,*
(b) $E[\![O]\!] = \lambda b.\lambda s.M$ *for some term* M*,*
(c) $E[\![F]\!] = \lambda b.\lambda s.\lambda x.M$ *for some term* M*,*
(d) $[\![V]\!] = W$ *for some value* W*.*

Proof. By mutual induction on the syntax of templates B, extensions O, extension functions F, and values V.

(a) $T[\![B]\!] = \lambda s.M$ for some term M, as shown by the following cases:
- $(B = \varepsilon)$ $T[\![\varepsilon]\!] = \lambda s.\ \mathit{fail}\ s$, so $M = \mathit{fail}\ s$.
- $(B = O; B')$ $T[\![O; B']\!] = \lambda s.\ [\![O]\!]\ [\![B']\!]\ s$, so $M = [\![O]\!]\ [\![B']\!]\ s$.
- $(B = \mathbf{continue}\, x \to N)$ $T[\![\mathbf{continue}\, x \to N]\!] = \lambda x.[\![N]\!]$, so $M = [\![N]\!]$.

(b) $E[\![O]\!] = \lambda b.\lambda s.M$ for some term M, as shown by the following cases:
- $(O = \varepsilon)$ $E[\![\varepsilon]\!] = \lambda b.\lambda s.\ b\ s$, so $M = b\ s$.
- $(O = O_1; O_2)$ $E[\![O_1; O_2]\!] = \lambda b.\lambda s.\ [\![O_1]\!]\ ([\![O_2]\!]\ b)\ s$, so $M = [\![O_1]\!]\ ([\![O_2]\!]\ b)\ s$.
- $(O = Q[x]\ O)$ follows by induction on Q (generalizing O):
  - $(Q = \square)$ for all O,

$$\begin{aligned} E[\![x\ O]\!] &= \lambda b.\lambda x.E[\![O]\!]\ b\ x \\ &= \lambda b.\lambda s.E[\![O]\!][s/x]\ b\ s \qquad (\alpha) \end{aligned}$$

    so $M = E[\![O]\!][s/x]\ b\ s$ for the given O.
  - $(Q = Q'\ P)$ assuming the inductive hypothesis (IH) that, for all O, there is an N_O such that $E[\![Q[x]\ O]\!] = \lambda b.\lambda s.N_O$. For all O,

$$\begin{aligned} E[\![(Q'[x]\ P)O]\!] &= E[\![Q'[x]\ (\lambda P.O)]\!] \\ &= \lambda b.\lambda s.N_{(\lambda P.O)} \qquad (IH) \end{aligned}$$

    so $M = N_{(\lambda P.O)}$ given by the inductive hypothesis applied to $\lambda P.O$.
- $(O = \lambda P.\ O)$ follows by cases if P is a variable or another pattern:
  - $(P \in \mathit{Variable})$ $E[\![\lambda x.\ O]\!] = \lambda b.\lambda s.(\lambda x.E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s)$, so $M = \lambda x.E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s$.
  - $(P \notin \mathit{Variable})$ $E[\![\lambda P.\ O]\!] = E[\![\lambda x.\, \mathbf{match}\, P \leftarrow x\ O]\!]$, which follows by the above case.
- $(O = \mathbf{match}\, P \leftarrow N\ O)$

$$\begin{aligned} E[\![\mathbf{match}\, P \leftarrow N\ O]\!] = \lambda b.\lambda s.\, &\mathbf{match}\, [\![N]\!]\ \mathbf{with} \\ &\{\ P \to E[\![O]\!]\ b\ s; \\ &\ \ \_ \to b\ s\ \} \end{aligned}$$

  so $M = \mathbf{match}\, [\![N]\!]\ \mathbf{with}\, \{\ P \to E[\![O]\!]\ b\ s; \_ \to b\ s\ \}$.

- $(O = \mathbf{nest}\, O)$ $E[\![\mathbf{nest}\, O]\!] = \lambda b.\lambda s.(\mathbf{rec}\, s' = E[\![O]\!]\ (\lambda\_.b\ s)\ (\lambda x.s'\ x))$, so $M = (\mathbf{rec}\, s' = E[\![O]\!]\ (\lambda\_.b\ s)\ (\lambda x.s'\ x))$
- $(O = \mathbf{try}\, x \to B)$ assuming the inductive hypothesis (IH) from part (a) that $T[\![B]\!] = \lambda s.N$ for some N,

$$\begin{aligned} E[\![\mathbf{try}\, x \to B]\!] &= \lambda x.T[\![B]\!] \\ &= \lambda x.(\lambda s.\ N) && (IH) \\ &= \lambda b.(\lambda s.\ N)[b/x] && (\alpha) \end{aligned}$$

  so $M = N[b/x]$

(c) $E[\![F]\!] = \lambda b.\lambda s.\lambda x.M$ for some term M, as shown by the following cases:

- $(F = \lambda P.\ O)$ Following the same calculation in the matching special case of part (b) above, $E[\![\lambda P.O]\!] = \lambda b.\lambda s.(\lambda x.M)$ for some M.
- $(F = Q[x\ P]\ O)$ follows by induction on Q (generalizing O):
  - $(Q = \square)$ for all O,

$$E[\![(y\ P)\ O]\!] = E[\![y\ (\lambda P.O)]\!] = \lambda b.\lambda y.E[\![\lambda P.O]\!]\ b\ y$$

    Following the same calculation in the previous case $(F = \lambda P.O)$ gives us some N such that $E[\![\lambda P.O]\!] = \lambda b.\lambda s'.\lambda x.N$, so continuing we have

$$\begin{aligned} E[\![(y\ P)\ O]\!] &= \lambda b.\lambda y.(\lambda b.\lambda s'.\lambda x.N)\ b\ y \\ &= \lambda b.\lambda y.\lambda x.N[y/s'] && (\beta) \\ &= \lambda b.\lambda s.\lambda x.N[y/s'][s/y] && (\alpha) \end{aligned}$$

    so $M = N[y/s'][s/y]$.
  - $(Q = Q'\ P')$ assuming the inductive hypothesis that, for all O, there is an N_O such that $E[\![Q'[y\ P]\ O]\!] = \lambda b.\lambda s.\lambda x.N_O$. For all O,

$$\begin{aligned} E[\![(Q'[y\ P]\ P')\ O]\!] &= E[\![Q'[y\ P]\ (\lambda P'.O)]\!] \\ &= \lambda b.\lambda s.\lambda x.N_{(\lambda P'.O)} && (IH) \end{aligned}$$

    so $M = N_{(\lambda P'.O)}$ given by the inductive hypothesis applied to $\lambda P'.O$.

(d) $[\![V]\!] = W$ for some value W, as shown by the following cases:

- $(V = x)$ $[\![x]\!] = x$, so $W = x$.
- $(V = \lambda x.M)$ $[\![\lambda x.M]\!] = \lambda x.[\![M]\!]$, so $W = \lambda x.[\![M]\!]$.
- $(V = \text{null})$ $[\![\text{null}]\!] = \text{null}$, so $W = \text{null}$.
- $(V = \text{cons}\, V_1\ V_2)$ $[\![\text{cons}\, V_1\ V_2]\!] = \text{cons}\, [\![V_1]\!]\ [\![V_2]\!]$, where $[\![V_1]\!] = W_1$ and $[\![V_2]\!] = W_2$ by the inductive hypotheses, so $W = \text{cons}\, W_1\ W_2$.
- $(V = \mathbf{template}\, B)$ $[\![\mathbf{template}\, B]\!] = T[\![B]\!] = \lambda s.M$, for some M, by the inductive hypothesis part (a), so $W = \lambda s.M$.
- $(V = \mathbf{extension}\, O)$ $[\![\mathbf{extension}\, O]\!] = E[\![O]\!] = \lambda b.\lambda s.M$, for some M, by the inductive hypothesis part (b), so $W = \lambda b.\lambda s.M$.
- $(V = \lambda^*(F; B))$ assuming the inductive hypotheses that

  IH_1 there is some N_1 such that $E[\![F]\!] = \lambda b.\lambda s.\lambda x.N_1$, and

  IH_2 there is some N_2 such that $T[\![B]\!] = \lambda s.N_2$,

$$\begin{aligned}
[\![\lambda^*(F;B)]\!] &= (\mathbf{rec}\ self = T[\![F;B]\!]\ (\lambda x.self\ x)) \\
&= (\mathbf{rec}\ self = E[\![F]\!]\ T[\![B]\!]\ (\lambda x.self\ x)) && (\beta) \\
&= (\mathbf{rec}\ self = (\lambda b.\lambda s.\lambda x.N_1)\ T[\![B]\!]\ (\lambda x.self\ x)) && (IH_1) \\
&= (\mathbf{rec}\ self = (\lambda b.\lambda s.\lambda x.N_1)\ (\lambda s.N_2)\ (\lambda x.self\ x)) && (IH_2) \\
&= (\mathbf{rec}\ self = \lambda x.N_1[(\lambda s.N_2)/b][(\lambda x.self\ x)/s]) && (\beta) \\
&= \lambda x.N_1[(\lambda s.N_2)/b] && (rec) \\
&\qquad [(\lambda x.self\ x)/s] \\
&\qquad [(\mathbf{rec}\ self = \lambda x.N_1[(\lambda s.N_2)/b][(\lambda x.self\ x)/s])/self]
\end{aligned}$$

Lemma 2 (Pattern Translation).
$[\![P]\!] = P$.

Proof. By induction on the syntax of P. □

Lemma 3 (Apartness Translation).
If $P \# V$ then $P \# [\![V]\!]$.

Proof. By induction on the derivation of apartness $P \# V$. □

Lemma 4 (Compositionality).
There exists a translation of contexts $[\![C]\!]$ such that $[\![C[M]]\!] = [\![C]\!][[\![M]\!]]$, and similarly for contexts which surround templates B and extensions O. The same holds for $T[\![C]\!]$ and $E[\![C]\!]$ for contexts returning templates and extensions, respectively.

Proof. By induction on the possible contexts C and the definition of the translation. □

Lemma 5 (Hygiene).
If x is captured by C if and only if it is captured by $[\![C]\!]$ in $[\![C]\!][x]$. The same holds for $T[\![C]\!]$ and $E[\![C]\!]$ for contexts returning templates and extensions, respectively.

Proof. By induction on the possible contexts C and the implicit side-conditions on the definition of translation where new binding forms introduced on the right-hand side must not capture free variables of sub-terms. □

Lemma 6 (Substitution). *For all values V,*

(a) $[\![M[V/x]]\!] = [\![M]\!][[\![V]\!]/x]$,
(b) $T[\![B[V/x]]\!] = T[\![B]\!][[\![V]\!]/x]$,
(c) $E[\![O[V/x]]\!] = E[\![O]\!][[\![V]\!]/x]$.

Proof. The proof strategy follows from proposition 2 of [9].

Suppose we have a free occurrence of x in a term M which is decomposed as $C[x]$, where C does not capture x. Because the translation is compositional (lemma 4), we can decompose every occurrence of a variable from its surrounding

context as $[\![C[x]]\!] = [\![C]\!][x]$. Then, because the translation is hygienic (lemma 5), we know $[\![C]\!]$ does not capture x. Thus,

$$\begin{aligned}
[\![C[x]]\!][[\![V]\!]/x] &= [\![C]\!][x][[\![V]\!]/x] && (\textit{lemma } 4)\\
&= [\![C]\!][[\![V]\!]][[\![V]\!]/x]\\
&= [\![C[V]]\!][[\![V]\!]/x] && (\textit{lemma } 4)
\end{aligned}$$

This replacement can then be iterated for each free occurrence of x in the term, until x no longer appears free in the final term M', in which the substitution is $[\![M']\!][[\![V]\!]/x] = [\![M']\!]$.

The same procedure applies for free occurrences of variables in templates and extensions. □

Lemma 7 (Equivalence Relation).

The translation preserves the structure of an equivalence relation with respect to the equational theory of the target. Equalities between translated terms have the following properties:

(a) Reflexivity: $[\![M]\!] = [\![M]\!]$.
(b) Symmetry: *if* $[\![M]\!] = [\![N]\!]$ *then* $[\![N]\!] = [\![M]\!]$,
(c) Transitivity: *if* $[\![M_1]\!] = [\![M_2]\!]$ *and* $[\![M_2]\!] = [\![M_3]\!]$ *then* $[\![M_1]\!] = [\![M_3]\!]$.
(d) Congruence: *if* $[\![M]\!] = [\![M]\!]$ *then for all contexts* C, $[\![C[M]]\!] = [\![C[N]]\!]$.

and similarly for equalities between translated templates $T[\![B]\!]$ *and extensions* $E[\![O]\!]$.

Proof. The proof strategy follows from proposition 1 of [9].

Reflexivity, transitivity, and symmetry of the equational theory follows immediately.

Congruence—$M = N$ implies $C[M] = C[N]$ for all contexts C—follows from the fact that the translation is *compositional* (lemma 4). An equation $C[M] = C[N]$, derived from congruence of $M = N$ inside C, can be derived by distributing the translation across contexts to apply the underlying equality $[\![M]\!] = [\![N]\!]$ gotten from soundness of $M = N$:

$$\begin{aligned}
[\![C[M]]\!] &= [\![C]\!][[\![M]\!]] && (\textit{lemma } 4)\\
&= [\![C]\!][[\![N]\!]]\\
&= [\![C[N]]\!] && (\textit{lemma } 4)
\end{aligned}$$

Proposition 1 (Conservative Extension). *If* $M = N$ *in the equational theory of the target (fig. 6), then so too does* $[\![M]\!] = [\![N]\!]$.

Proof. The soundness of the inference rules defining the structure of the source equivalence relation follow from lemma 7.

It then remains to show that each individual axiom (β, *rec*, *etc.*) from the target language is still equal after translation. Each one holds by the inductive hypothesis—because the definition of translation for these cases does not

$$\varepsilon; O = O \qquad (O_1; O_2); O_3 = O_1; (O_2; O_3) \qquad \mathbf{do}\, M; O = \mathbf{do}\, M$$
$$\varepsilon; B = B \qquad (O_1; O_2); B = O_1; (O_2; B) \qquad \mathbf{do}\, M; B = \mathbf{else}\, M$$

$$\begin{aligned}
\lambda^*(F; B) &= (\mathbf{template}\, F; B)\ (\lambda^*(F; B)) \\
\lambda x.(\lambda^*(F; B))\ x &= \lambda^*(F; B) \\
(\mathbf{template}\, O; B)\ V &= (\mathbf{extension}\, O)\ (\mathbf{template}\, B)\ V \\
(\mathbf{template}\, \varepsilon)\ V &= \mathit{fail}\ V \\
(\mathbf{template}\, \mathbf{continue}\, x \to M)\ V &= M[V/x] \\
(\mathbf{extension}\, \mathbf{try}\, x \to B)\ V &= \mathbf{template}\, B[V/x] \\
(\mathbf{template}\, (x\ O); B)\ V &= (\mathbf{template}\, O[V/x]; B)\ V \\
(\mathbf{template}\, (\lambda x.O); B)\ V\ W &= \left(\begin{array}{l}\mathbf{template} \\ \quad O[W/x]; \\ \quad \mathbf{continue}\, s' \to \\ \qquad (\mathbf{template}\, B)\ s'\ W\end{array}\right) V
\end{aligned}$$

$$\begin{aligned}
\mathbf{match}\, P \leftarrow V\ O &= \varepsilon && (\text{if } P \# V) \\
\mathbf{match}\, P \leftarrow V\ O &= O[W.../x...] && (\text{if } P[W.../x...] = V) \\
\lambda P.O &= \lambda x.(\mathbf{match}\, P \leftarrow x)\ O \\
(Q[x]\ P)\ O &= Q[x]\ (\lambda P.O)
\end{aligned}$$

Fig. 8. Core axioms of the equational theory.

change syntax—along with lemmas 1 and 6 for cases which substitute values. For example, with the β axiom, we have

$$\begin{aligned}
[\![(\lambda x.M)\ V]\!] &= (\lambda x.[\![M]\!])\ [\![V]\!] \\
&= (\lambda x.[\![M]\!])\ W && (\mathit{lemma}\ 1) \\
&= [\![M]\!][W/x] && (\beta) \\
&= [\![M]\!][[\![V]\!]/x] && (\mathit{lemma}\ 1) \\
&= [\![M[V/x]]\!] && (\mathit{lemma}\ 6)
\end{aligned}$$

since values are only translated to values, $[\![V]\!] = W$ up to the target equational theory (lemma 1), which means that β-reduction still applies after translation. For the *match* and *apart* axioms, we need to know that patterns and apartness is not affected by translation, given by lemmas 2 and 3.

A.2 Soundness of the source

Proposition 2 (Soundness). *The translation is* sound *w.r.t. the source and target equational theories (*e.g., *$M = N$ in fig. 7 implies $[\![M]\!] = [\![N]\!]$ in fig. 6).*

$$(\mathbf{template}\,(\lambda P.O);B)\ V'\ V = \left(\begin{array}{l}\mathbf{template}\\ \quad O[W/x];\\ \quad \mathbf{continue}\,s' \to\\ \quad\quad (\mathbf{template}\,B)\ s'\ V\end{array}\right)\ V' \quad (\text{if } P[W.../x...] = V)$$

$$(\mathbf{template}\,(\lambda P.\,\mathbf{do}\,M);B)\ V'\ V = M[W.../x...] \quad (\text{if } P[W.../x...] = V)$$

$$(\mathbf{template}\,(\lambda P.O);B)\ V'\ V = (\mathbf{template}\,B)\ V'\ V \quad (\text{if } P \# V)$$

$$C[(\mathbf{template}\,(Q[y]\ O);B)\ V] = \left(\begin{array}{l}\mathbf{template}\\ \quad O[W/x];\\ \quad \mathbf{continue}\,s' \to\\ \quad\quad C[(\mathbf{template}\,B)\ s']\end{array}\right)\ V \quad (\text{if } Q[W.../x...] = C)$$

$$C[(\mathbf{template}\,(Q[y] = M);B)\ V] = M[V/y][W.../x...] \quad (\text{if } Q[W.../x...] = C)$$

$$C[(\mathbf{template}\,(Q[y]\ O);B)\ V] = C[(\mathbf{template}\,B)\ V] \quad (\text{if } Q \# C)$$

$$C[\lambda^*(Q[y] = M);B] = M[(\lambda^*(Q[y] = M);B)/y]\,[W.../x...] \quad (\text{if } Q[W.../x...] = C)$$

$$C[\lambda^*(Q[y]\ O);\mathbf{else}\,M] = C[M] \quad (\text{if } Q \# C)$$

Fig. 9. Other equalities derived from the core axioms.

Proof. The reflexive, symmetric, transitive, and congruent structure of the equational theory is sound by lemma 7, and the axioms of the target language are sound by proposition 1.

It remains to show that each equality in fig. 7 are sound as well. The proof of soundness is finished by the following lemmas, which are organized in two parts:

- The equalities in fig. 8 are *core axioms*, whose soundness is shown by translating both sides into the target λ-calculus and deriving an equality in the target equational theory.
- The remaining equalities in fig. 9 are *derived* only in terms of those core axioms directly in the source language, so their soundness follows from lemma 7 in addition to the specific lemmas proving soundness of the core axioms.

Note that every equation listed in fig. 7 can be found in either fig. 8 or fig. 9, however, there are some additional core axioms in fig. 8 that do not appear in fig. 7, but are useful for deriving other equations. □

Core axioms

Lemma 8 (Extension Composition Identity Left). *In the target,*

$$E[\![\varepsilon;O]\!] = E[\![O]\!]$$

Proof.

$$\begin{array}{lll}
& E[\![\varepsilon; O]\!] & \\
= & \lambda b.\lambda s.\ E[\![\varepsilon]\!]\ (E[\![O]\!]\ b)\ s & \\
= & \lambda b.\lambda s.\ E[\![\varepsilon]\!]\ ((\lambda b.\lambda s.\ M_0)\ b)\ s & (lemma\ 1) \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.\ b\ s)\ ((\lambda b.\lambda s.\ M_0)\ b)\ s & \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.\ b\ s)\ (\lambda s.\ M_0)\ s & (\beta) \\
= & \lambda b.\lambda s.\ (\lambda s.\ (\lambda s.\ M_0)\ s)\ s & (\beta) \\
= & \lambda b.\lambda s.\ (\lambda s.\ M_0)\ s & (\beta) \\
= & \lambda b.\lambda s.\ M_0 & (\beta) \\
= & E[\![O]\!] & (lemma\ 1)
\end{array}$$

□

Lemma 9 (Template Composition Identity Left). *In the target,*

$$T[\![\varepsilon; B]\!] = T[\![B]\!]$$

Proof.

$$\begin{array}{lll}
& T[\![\varepsilon; B]\!] & \\
= & \lambda s.\ E[\![\varepsilon]\!]\ T[\![B]\!]\ s & \\
= & \lambda s.\ (\lambda b.\lambda s.\ b\ s)\ T[\![B]\!]\ s & \\
= & \lambda s.\ T[\![B]\!]\ s & (\beta, lemma\ 1) \\
= & T[\![B]\!] & (\alpha, \beta, lemma\ 1)
\end{array}$$

□

Lemma 10 (Extension Composition Associativity). *In the target,*

$$E[\![(O_1; O_2); O_3]\!] = E[\![O_1; (O_2; O_3)]\!]$$

Proof. The left-hand side simplifies as follows:

$$\begin{array}{lll}
& E[\![(O_1; O_2); O_3]\!] & \\
= & \lambda b.\lambda s.\ E[\![O_1; O_2]\!]\ (E[\![O_3]\!]\ b)\ s & \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.\ E[\![O_1]\!]\ (E[\![O_2]\!]\ b)\ s)\ (E[\![O_3]\!]\ b)\ s & \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ ((\lambda b.\lambda s.M_2)\ b)\ s)\ ((\lambda b.\lambda s.M_3)\ b)\ s & (lemma\ 1) \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ ((\lambda b.\lambda s.M_2)\ b)\ s)\ (\lambda s.M_3)\ s & (\beta) \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ ((\lambda b.\lambda s.M_2)\ (\lambda s.M_3))\ s & (\beta) \\
= & \lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.M_2[(\lambda s.M_3)/b])\ s & (\beta) \\
\\
= & \lambda b.\lambda s.\ M_1[(\lambda s.M_2[(\lambda s.M_3)/b])/b] & (\beta)
\end{array}$$

The right-hand side simplifies to the same term as follows:

$$\begin{aligned}
& E[\![O_1; (O_2; O_3)]\!] \\
= \;& \lambda b.\lambda s.\ E[\![O_1]\!]\ ([\![O_2; O_3]\!]\ b)\ s \\
= \;& \lambda b.\lambda s.\ E[\![O_1]\!]\ ((\lambda b.\lambda s.\ E[\![O_2]\!]\ (E[\![O_3]\!]\ b)\ s)\ b)\ s \\
= \;& \lambda b.\lambda s.\ E[\![O_1]\!]\ (\lambda s.\ E[\![O_2]\!]\ (E[\![O_3]\!]\ b)\ s)\ s && (\beta) \\
= \;& \lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.\ (\lambda b.\lambda s.M_2)\ ((\lambda b.\lambda s.M_3)\ b)\ s)\ s && (lemma\ 1) \\
= \;& \lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.\ (\lambda b.\lambda s.M_2)\ (\lambda s.M_3)\ s)\ s && (\beta) \\
= \;& \lambda b.\lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.M_2[(\lambda s.M_3)/b])\ s && (\beta) \\
= \;& \lambda b.\lambda s.\ M_1[(\lambda s.M_2[(\lambda s.M_3)/b])/b] && (\beta)
\end{aligned}$$

□

Lemma 11 (Template Composition Associativity). *In the target,*

$$T[\![(O_1; O_2); B]\!] = T[\![O_1; (O_2; B)]\!]$$

Proof. The left-hand side simplifies as follows:

$$\begin{aligned}
& T[\![(O_1; O_2); B]\!] \\
= \;& \lambda s.\ E[\![O_1; O_2]\!]\ T[\![B]\!]\ s \\
= \;& \lambda s.\ (\lambda b.\lambda s.\ E[\![O_1]\!]\ (E[\![O_2]\!]\ b)\ s)\ T[\![B]\!]\ s \\
= \;& \lambda s.\ E[\![O_1]\!]\ (E[\![O_2]\!]\ T[\![B]\!])\ s && (\beta, lemma\ 1) \\
= \;& \lambda s.\ (\lambda b.\lambda s.M_1)\ ((\lambda b.\lambda s.M_2)\ T[\![B]\!])\ s && (lemma\ 1) \\
= \;& \lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.M_2[T[\![B]\!]/b])\ s && (\beta, lemma\ 1) \\
= \;& \lambda s.\ M_1[(\lambda s.M_2[T[\![B]\!]/b])/b] && (\beta)
\end{aligned}$$

The right-hand side simplifies to the same term as follows:

$$\begin{aligned}
& T[\![O_1; (O_2; B)]\!] \\
= \;& \lambda s.\ E[\![O_1]\!]\ T[\![O_2; B]\!]\ s \\
= \;& \lambda s.\ E[\![O_1]\!]\ (\lambda s.\ E[\![O_2]\!]\ T[\![B]\!]\ s)\ s \\
= \;& \lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.\ (\lambda b.\lambda s.\ M_2)\ T[\![B]\!]\ s)\ s && (lemma\ 1) \\
= \;& \lambda s.\ (\lambda b.\lambda s.M_1)\ (\lambda s.M_2[T[\![B]\!]/b])\ s && (\beta, lemma\ 1) \\
= \;& \lambda s.\ M_1[(\lambda s.M_2[T[\![B]\!]/b])/b] && (\beta)
\end{aligned}$$

□

Lemma 12 (Extension Commit). *In the target,*

$$E[\![\mathbf{do}\, M; O]\!] = E[\![\mathbf{do}\, M]\!]$$

Proof.

$$
\begin{array}{lll}
 & E[\![\mathbf{do}\, M; O]\!] & \\
= & \lambda b.\lambda s.\ E[\![\mathbf{do}\, M]\!]\ (E[\![O]\!]\ b)\ s & \\
= & \lambda b.\lambda s.\ E[\![\mathbf{do}\, M]\!]\ ((\lambda b.\lambda s.\ M_O)\ b)\ s & (lemma\ 1) \\
= & \lambda b.\lambda s.\ E[\![\mathbf{do}\, M]\!]\ (\lambda s.\ M_O)\ s & (\beta) \\
= & \lambda b.\lambda s.\ E[\![\mathbf{try}\ \_ \to \mathbf{continue}\ \_ \to M]\!]\ (\lambda s.\ M_O)\ s & \\
= & \lambda b.\lambda s.\ (\lambda \_.\ T[\![\mathbf{continue}\ \_ \to M]\!])\ (\lambda s.\ M_O)\ s & \\
= & \lambda \_.\lambda s.\ T[\![\mathbf{continue}\ \_ \to M]\!]\ s & (\beta, b \notin FV(M)) \\
= & \lambda \_.\lambda s.\ (\lambda \_.\ [\![M]\!])\ s & \\
= & \lambda \_.\lambda \_.\ [\![M]\!] & (\beta, s \notin FV(M)) \\
= & \lambda \_.\ T[\![\mathbf{continue}\ \_ \to M]\!] & \\
= & E[\![\mathbf{try}\ \_ \to \mathbf{continue}\ \_ \to M]\!] & \\
= & E[\![\mathbf{do}\, M]\!] &
\end{array}
$$

□

Lemma 13 (Template Commit). *In the target,*

$$T[\![\mathbf{do}\, M; B]\!] = T[\![\mathbf{else}\, M]\!]$$

Proof.

$$
\begin{array}{lll}
 & T[\![\mathbf{do}\, M; B]\!] & \\
= & \lambda s.\ E[\![\mathbf{do}\, M]\!]\ T[\![B]\!]\ s & \\
= & \lambda s.\ E[\![\mathbf{try}\ \_ \to \mathbf{continue}\ \_ \to M]\!]\ T[\![B]\!]\ s & \\
= & \lambda s.\ (\lambda \_.\ T[\![\mathbf{continue}\ \_ \to M]\!])\ T[\![B]\!]\ s & \\
= & \lambda s.\ (\lambda \_.\ \lambda \_.\ [\![M]\!])\ T[\![B]\!]\ s & \\
= & \lambda s.\ (\lambda \_.\ [\![M]\!])\ s & (\beta) \\
= & \lambda \_.\ [\![M]\!] & (\beta, s \notin FV(M)) \\
= & T[\![\mathbf{continue}\ \_ \to M]\!] & \\
= & T[\![\mathbf{else}\, M]\!] &
\end{array}
$$

□

Lemma 14 ($\eta\lambda^*$). *In the target,*

$$E[\![\lambda x.\ (\lambda^*(F; B))\ x]\!] = E[\![\lambda^*(F; B)]\!]$$

Proof. From lemma 1, we have $E[\![F]\!] = \lambda b.\lambda s.\lambda z.M$ for some term M. In the following, let $M' = M[T[\![B]\!]/b][(\lambda y.self\ y)/s]$,

$$E[\![\lambda x.\ (\lambda^*(F; B))\ x]\!]$$

$$
\begin{array}{lll}
= & \lambda x.E[\![(\lambda^*(F;B))]\!]\ x & \\
= & \lambda x.(\mathbf{rec}\ self = (\lambda s.E[\![F]\!]\ T[\![B]\!]\ s)\ (\lambda y.self\ y))\ x & \\
= & \lambda x.(\mathbf{rec}\ self = E[\![F]\!]\ T[\![B]\!]\ (\lambda y.self\ y))\ x & (\beta) \\
= & \lambda x.(\mathbf{rec}\ self = (\lambda b.\lambda s.\lambda z.M)\ T[\![B]\!]\ (\lambda y.self\ y))\ x & (lemma\ 1) \\
= & \lambda x.(\mathbf{rec}\ self = \lambda z.M[T[\![B]\!]/b][(\lambda y.self\ y)/s])\ x & \\
= & \lambda x.(\mathbf{rec}\ self = \lambda z.M')\ x & (\beta) \\
= & \lambda x.(\lambda z.M'[\mathbf{rec}\ self = \lambda z.M'/self])\ x & (rec) \\
= & \lambda x.M'[\mathbf{rec}\ self = \lambda z.M'/self][x/z] & (\beta) \\
= & \lambda z.M'[\mathbf{rec}\ self = \lambda z.M'/self] & (\alpha) \\
= & \mathbf{rec}\ self = \lambda z.M' & (rec) \\
= & \mathbf{rec}\ self = \lambda z.M[T[\![B]\!]/b][(\lambda y.self\ y)/s] & \\
= & \mathbf{rec}\ self = (\lambda b.\lambda s.\lambda z.M)\ T[\![B]\!]\ (\lambda x.self\ x) & (\beta) \\
= & \mathbf{rec}\ self = E[\![F]\!]\ T[\![B]\!]\ (\lambda x.self\ x) & (lemma\ 1) \\
= & \mathbf{rec}\ self = (\lambda s.E[\![F]\!]\ T[\![B]\!]\ s)\ (\lambda x.self\ x) & (\beta) \\
= & \mathbf{rec}\ self = T[\![F;B]\!]\ (\lambda x.self\ x) & \\
= & E[\![\lambda^*(F;B)]\!] &
\end{array}
$$

□

Lemma 15 (Unfold λ^*). *In the target,*

$$E[\![\lambda^*(F;B)]\!] = [\![(\mathbf{template}\,F;B)\ (\lambda^*(F;B))]\!]$$

Proof. Note, $T[\![F;B]\!]\ (\lambda x.self\ x) = E[\![F]\!]\ T[\![B]\!]\ (\lambda x.self\ x)$ is β-equal to some value of the form $\lambda z.M'$ because $T[\![F;B]\!] = \lambda b.\lambda s.\lambda z.M$ from lemma 1. So $\mathbf{rec}\ self = T[\![F;B]\!]\ (\lambda x.self\ x)$ unfolds via β and rec in the following,

$$
\begin{array}{lll}
 & E[\![\lambda^*(F;B)]\!] & \\
= & (\mathbf{rec}\ self = T[\![F;B]\!]\ (\lambda x.self\ x)) & \\
= & T[\![F;B]\!]\ (\lambda x.(\mathbf{rec}\ self = T[\![F;B]\!]\ (\lambda x.self\ x))\ x)) & (\beta, rec) \\
= & T[\![F;B]\!]\ (\lambda x.[\![\lambda^*(F;B)]\!]\ x) & \\
= & T[\![F;B]\!]\ [\![\lambda^*(F;B)]\!] & (lemma\ 14) \\
= & [\![\mathbf{template}(F;B)]\!]\ [\![\lambda^*(F;B)]\!] & \\
= & [\![(\mathbf{template}(F;B))\ (\lambda^*(F;B))]\!] &
\end{array}
$$

□

Lemma 16 (Template Extension). *In the target,*

$$[\![(\mathbf{template}\,O;B)\ V]\!] = [\![(\mathbf{extension}\,O)\ (\mathbf{template}\,B)\ V]\!]$$

Proof.

$$[\![(\mathbf{template}\,O;B)\ V]\!]$$

$$
\begin{array}{lll}
= & [\![(\mathbf{template}\, O; B)]\!]\ [\![V]\!] & \\
= & (\lambda s.\ E[\![O]\!]\ T[\![B]\!]\ s)\ [\![V]\!] & \\
= & E[\![O]\!]\ T[\![B]\!]\ [\![V]\!] & (\beta, lemma\ 1) \\
= & E[\![\mathbf{extension}\, O]\!]\ T[\![\mathbf{template}\, B]\!]\ [\![V]\!] & \\
= & E[\![(\mathbf{extension}\, O)\ (\mathbf{template}\, B)\ V]\!] &
\end{array}
$$

□

Lemma 17 (Template Failure). *In the target,*

$$[\![(\mathbf{template}\, \varepsilon)\ V]\!] = [\![fail\ V]\!]$$

Proof.

$$
\begin{array}{lll}
 & [\![(\mathbf{template}\, \varepsilon)\ V]\!] & \\
= & [\![(\mathbf{template}\, \varepsilon)]\!]\ [\![V]\!] & \\
= & T[\![\varepsilon]\!]\ [\![V]\!] & \\
= & (\lambda s. fail\ s)\ [\![V]\!] & \\
= & fail\ [\![V]\!] & (\beta, lemma\ 1) \\
= & [\![fail\ V]\!] &
\end{array}
$$

□

Lemma 18 (Template Continue). *In the target,*

$$[\![(\mathbf{template}\, \mathbf{continue}\, x \to M)\ V]\!] = [\![M[V/x]]\!]$$

Proof.

$$
\begin{array}{lll}
 & [\![(\mathbf{template}\, \mathbf{continue}\, x \to M)\ V]\!] & \\
= & T[\![(\mathbf{continue}\, x \to M)]\!]\ [\![V]\!] & \\
= & (\lambda x.\ [\![M]\!])\ [\![V]\!] & \\
= & [\![M]\!][[\![V]\!]/x] & (\beta, lemma\ 1) \\
= & [\![M[V/x]]\!] & (lemma\ 6)
\end{array}
$$

□

Lemma 19 (Extension Try). *In the target,*

$$[\![(\mathbf{extension}\, \mathbf{try}\, x \to B)\ V]\!] = \mathbf{template}\, B[V/x]$$

Proof.

$$
\begin{array}{lll}
 & [\![(\mathbf{extension}\, \mathbf{try}\, x \to B)\ V]\!] & \\
= & [\![\mathbf{extension}\, \mathbf{try}\, x \to B]\!]\ [\![V]\!] & \\
= & E[\![\mathbf{try}\, x \to B]\!]\ [\![V]\!] &
\end{array}
$$

$$
\begin{array}{lll}
= & (\lambda x.\ T[\![B]\!])\ [\![V]\!] & \\
= & T[\![B]\!][[\![V]\!]/x] & (\beta, lemma\ 1) \\
= & T[\![B[V/x]]\!] & (lemma\ 6) \\
= & [\![\textbf{template}\, B[V/x]]\!] &
\end{array}
$$

□

Lemma 20 (Template Self). *In the target,*

$$[\![(\textbf{template}\,(x\ O); B)\ V = (\textbf{template}\, O[V/x]; B)\ V]\!]$$

Proof.

$$
\begin{array}{lll}
 & [\![(\textbf{template}\,(x\ O); B)\ V]\!] & \\
= & E[\![x\ O]\!]\ T[\![B]\!]\ [\![V]\!] & (\beta, lemma\ 1) \\
= & (\lambda b.\lambda x.E[\![O]\!]\ b\ x)\ T[\![B]\!]\ [\![V]\!] & \\
= & E[\![O]\!][[\![V]\!]/x]\ T[\![B]\!]\ [\![V]\!] & (\beta, lemma\ 1) \\
= & E[\![O[V/x]]\!]\ T[\![B]\!]\ [\![V]\!] & (lemma\ 6) \\
= & [\![(\textbf{template}\, O[V/x]; B)\ V]\!] & (\beta, lemma\ 1)
\end{array}
$$

□

Lemma 21 (Template Lambda). *In the target,*

$$[\![\textbf{template}\,(\lambda x.O; B)\ V\ W]\!] = \left[\!\!\left[\left(\begin{array}{l} \textbf{template} \\ \quad O[W/x]; \\ \quad \textbf{continue}\, s' \to (\textbf{template}\, B)\ s'\ W \end{array} \right)\ V \right]\!\!\right]$$

Proof.

$$
\begin{array}{lll}
 & [\![(\textbf{template}\,(\lambda x.O); B)\ V\ W]\!] & \\
= & (\lambda s.E[\![\lambda x.O]\!]\ T[\![B]\!]\ s)\ [\![V]\!]\ [\![W]\!] & \\
= & E[\![\lambda x.O]\!]\ T[\![B]\!]\ [\![V]\!]\ [\![W]\!] & (\beta, lemma\ 1) \\
= & (\lambda b.\lambda s.\lambda x.E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s)\ T[\![B]\!]\ [\![V]\!]\ [\![W]\!] & \\
= & E[\![O]\!][[\![W]\!]/x]\ (\lambda s'.T[\![B]\!]\ s'\ [\![W]\!])\ [\![V]\!] & (\beta, lemma\ 1) \\
= & E[\![O[W/x]]\!]\ (\lambda s'.T[\![B]\!]\ s'\ [\![W]\!])\ [\![V]\!] & (lemma\ 6) \\
= & \left[\!\!\left[\left(\begin{array}{l} \textbf{template} \\ \quad O[W/x]; \\ \quad \textbf{continue}\, s' \to (\textbf{template}\, B)\ s'\ W \end{array} \right)\ V \right]\!\!\right] & (\beta, lemma\ 1)
\end{array}
$$

□

Lemma 22 (Try Match). *In the target,*

$$[\![\textbf{match}\, P \leftarrow V\ O]\!] = [\![O[W.../x...]]\!] \qquad (\textit{if}\ P[W.../x...] = V)$$

Proof. By lemmas 2 and 6, $[\![V]\!] = [\![P[W.../x...]]\!] = [\![P]\!][[\![W]\!].../x...] = P[[\![W]\!].../x...]$ in the following:

$$\begin{array}{lll} & E[\![\mathbf{match}\, P \leftarrow V\ O]\!] & \\ = & \lambda b.\lambda s.\, \mathbf{match}\, [\![V]\!]\, \mathbf{with}\, \{\, P \to E[\![O]\!]\ b\ s; \_ \to b\ s\,\} & \\ = & \lambda b.\lambda s.\ E[\![O]\!][[\![W]\!].../x...]\ b\ s & (match, lemmas\ 2 and\ 6) \\ = & \lambda b.\lambda s.\ E[\![O[W.../x...]]\!]\ b\ s & (lemma\ 6) \\ = & E[\![O[W.../x...]]\!] & (\alpha, \beta, lemma\ 1) \end{array}$$

□

Lemma 23 (Try Match Apart). *In the target,*

$$[\![\mathbf{match}\, P \leftarrow V\ O]\!] = [\![\varepsilon]\!] \qquad (if\ P\ \#\ V)$$

Proof. By lemma 3, $P\ \#\ [\![V]\!]$ in the following:

$$\begin{array}{lll} & E[\![\mathbf{match}\, P \leftarrow V\ O]\!] & \\ = & \lambda b.\lambda s.\, \mathbf{match}\, [\![V]\!]\, \mathbf{with}\, \{\, P \to E[\![O]\!]\ b\ s; \_ \to b\ s\,\} & \\ = & \lambda b.\lambda s.\ b\ s & (apart, lemma\ 3) \\ = & E[\![\varepsilon]\!] & \end{array}$$

□

Lemma 24 (Pattern Lambda). *In the target,*

$$E[\![\lambda P.O]\!] = E[\![\lambda x.(\mathbf{match}\, P \leftarrow x)\ O]\!]$$

Proof. By cases on whether P is a variable or another pattern:

– If $P = y$, then

$$\begin{array}{lll} & E[\![\lambda x.(\mathbf{match}\, y \leftarrow x)\ O]\!] & \\ = & \lambda b.\lambda s.\lambda x.\ E[\![\mathbf{match}\, y \leftarrow x\ O]\!]\ (\lambda s'.b\ s'\ x)\ s & \\ = & \lambda b.\lambda s.\lambda x.\, \mathbf{match}\, x\, \mathbf{with}\, \{ & (\beta) \\ & \qquad y \to E[\![O]\!]\ (\lambda s'.b\ s'\ x)\ s; & \\ & \qquad \_ \to (\lambda s'.b\ s'\ x)\ s\ \} & \\ = & \lambda b.\lambda s.\lambda x.E[\![O]\!][x/y]\ (\lambda s'.b\ s'\ x)\ s & (match) \\ = & \lambda b.\lambda s.\lambda y.E[\![O]\!]\ (\lambda s'.b\ s'\ y)\ s & (\alpha) \\ = & E[\![\lambda y.O]\!] & \end{array}$$

– Otherwise, when $P \notin Variable$, $E[\![\lambda P.O]\!] = E[\![\lambda x.(\mathbf{match}\, P \leftarrow x)\ O]\!]$ directly by definition of translation. □

Lemma 25 (Copattern Abstraction). *In the target,*

$$E[\![(Q[x]\ P)\ O]\!] = E[\![Q[x]\ (\lambda P.O)]\!]$$

Proof. $E[\![(Q[x]\ P)\ O]\!] = E[\![Q[x]\ (\lambda P.O)]\!]$ directly by definition of translation. □

Derived equations

Lemma 26 (Template Match). *In the source,*

$$\textbf{template}\,(\lambda P.O; B)\ V'\ V = \left(\begin{array}{l}\textbf{template}\\ \quad O[W.../x...];\\ \quad \textbf{continue}\,s' \to (\textbf{template}\,B)\ s'\ V\end{array}\right)\ V'$$
$$(\textit{if}\ P[W.../x...] = V)$$

Proof.

$$\begin{array}{lll}
& \textbf{template}\,(\lambda P.O; B)\ V'\ V & \\
= & \textbf{template}\,(\lambda x.(\textbf{match}\,P \leftarrow x)\ O; B)\ V'\ V & (\textit{lemma}\ 24)\\
= & \left(\begin{array}{l}\textbf{template}\\ \quad (\textbf{match}\,P \leftarrow V)\ O;\\ \quad \textbf{continue}\,s' \to (\textbf{template}\,B)\ s'\ V\end{array}\right)\ V' & (\textit{lemma}\ 21)\\
= & \left(\begin{array}{l}\textbf{template}\\ \quad O[W.../x...];\\ \quad \textbf{continue}\,s' \to (\textbf{template}\,B)\ s'\ V\end{array}\right)\ V' & (\textit{lemma}\ 22)
\end{array}$$

□

Lemma 27 (Template Do Match). *In the source,*

$$(\textbf{template}\,(\lambda P.\,\textbf{do}\,M); B)\ V'\ V = M[W.../x...] \qquad (\textit{if}\ P[W.../x...] = V)$$

Proof.

$$\begin{array}{lll}
& (\textbf{template}\,(\lambda P.\,\textbf{do}\,M); B)\ V'\ V & \\
= & \left(\begin{array}{l}\textbf{template}\\ \quad \textbf{do}\,M[W.../x...];\\ \quad \textbf{continue}\,s' \to (\textbf{template}\,B)\ s'\ V\end{array}\right)\ V' & (\textit{lemma}\ 26)\\
= & (\textbf{template}\,\textbf{else}\,M[W.../x...])\ V' & (\textit{lemma}\ 13)\\
= & (\textbf{template}\,\textbf{continue}\ _ \to M[W.../x...])\ V' & \\
= & M[W.../x...] & (\textit{lemma}\ 18)
\end{array}$$

□

Lemma 28 (Template Apart). *In the source,*

$$(\textbf{template}\,(\lambda P.O); B)\ V'\ V = (\textbf{template}\,B)\ V'\ V \qquad (\textit{if}\ P\ \#\ V) \qquad (1)$$

Proof.

$$(\textbf{template}\,(\lambda P.O); B)\ V'\ V$$

$$
\begin{array}{lll}
= & (\mathbf{template}\,(\lambda x.(\mathbf{match}\,P \leftarrow x)\ O);B)\ V'\ V & (lemma\ 24) \\
= & \left(\begin{array}{l}\mathbf{template} \\ \quad (\mathbf{match}\,P \leftarrow V)\ O; \\ \quad \mathbf{continue}\,s' \to (\mathbf{template}\,B)\ s'\ V\end{array}\right)\ V' & (lemma\ 21) \\
= & \left(\begin{array}{l}\mathbf{template} \\ \quad \varepsilon; \\ \quad \mathbf{continue}\,s' \to (\mathbf{template}\,B)\ s'\ V\end{array}\right)\ V' & (lemma\ 23) \\
= & (\mathbf{template}\,\mathbf{continue}\,s' \to (\mathbf{template}\,B)\ s'\ V)\ V' & (lemma\ 9) \\
= & (\mathbf{template}\,B)\ V'\ V & (lemma\ 18)
\end{array}
$$

□

Lemma 29 (Context Match). *In the source,*

$$
C[(\mathbf{template}\,(Q[y]\ O);B)\ V] = \begin{array}{l}\mathbf{template} \\ \quad O[V/y][W.../x...]; \\ \quad \mathbf{continue}\,s' \to C[(\mathbf{template}\,B)\ s']\end{array}
$$
$$
(\mathit{if}\ Q[W.../x...] = C \neq \Box)
$$

Proof. By induction on the syntax of Q.

- $Q = \Box$ is impossible due to the assumption $Q[W.../x...] = C \neq \Box$.
- If $Q = \Box\ P$ then $C = \Box\ V'$ such that $P[W.../x...] = V$, and so

$$
\begin{array}{lll}
 & C[(\mathbf{template}\,(Q[y]\ O);B)\ V] & \\
= & (\mathbf{template}\,((y\ P)\ O);B)\ V\ V' & \\
= & (\mathbf{template}\,(y\ (\lambda P.O));B)\ V\ V' & (lemma\ 25) \\
= & (\mathbf{template}\,(\lambda P.O[V/y]);B)\ V\ V' & (lemma\ 20) \\
= & \left(\begin{array}{l}\mathbf{template} \\ \quad O[V/y][W.../x...]; \\ \quad \mathbf{continue}\,s' \to (\mathbf{template}\,B)\ s'\ V'\end{array}\right)\ V & (lemma\ 26)
\end{array}
$$

- If $Q = Q'\ P$ then $C = C'\ V'$ where $[W.../x...] = [W_1..., W_2.../x_1..., x_2...]$ such that $Q'[W_2.../x_1...] = C$ and $P[W_2.../x_2...] = V'$. Assume the inductive hypothesis

$$
C'[(\mathbf{template}\,(Q'[y]\ O);B)\ V] = \begin{array}{l}\mathbf{template} \\ \quad O[V/y][W.../x...]; \\ \quad \mathbf{continue}\,s' \to C'[(\mathbf{template}\,B)\ s']\end{array}
$$

  The equality holds by this inductive hypothesis and the following lemmas,

$$
C[(\mathbf{template}\,(Q[y]\ O);B)\ V]
$$

$$
\begin{array}{lll}
= & C'[(\textbf{template}\,((Q'[y]\ P)\ O); B)\ V]\ V' & \\
= & C'[(\textbf{template}\,(Q'[y]\ (\lambda P.O)); B)\ V]\ V' & (lemma\ 25) \\
= & \left(\begin{array}{l}\textbf{template} \\ \quad \lambda P.O[V/y][W_1.../x_1...]; \\ \quad \textbf{continue}\, s' \to C'[(\textbf{template}\, B)\ s']\end{array}\right)\ V\ V' & (IH) \\
= & \left(\begin{array}{l}\textbf{template} \\ \quad O[V/y][W_1.../x_1...][W_2.../x_2...]; \\ \quad \textbf{continue}\, s'' \to \\ \qquad \left(\begin{array}{l}\textbf{template} \\ \quad \textbf{continue}\, s' \to \\ \qquad C'[(\textbf{template}\, B)\ s']\end{array}\right)\ s''\ V'\end{array}\right)\ V & (lemma\ 26) \\
= & \left(\begin{array}{l}\textbf{template} \\ \quad O[V/y][W_1.../x_1...][W_2.../x_2...]; \\ \quad \textbf{continue}\, s'' \to (C'[(\textbf{template}\, B)\ s'])\ V'\end{array}\right)\ V & (lemma\ 18) \\
= & \left(\begin{array}{l}\textbf{template} \\ \quad O[V/y][W.../x...]; \\ \quad \textbf{continue}\, s' \to C[(\textbf{template}\, B)\ s']\end{array}\right)\ V & (\alpha)
\end{array}
$$

Lemma 30 (Context Exact Match). *In the source,*

$$C[(\textbf{template}\,(Q[y] = M); B)\ V] = M[V/y][W.../x...] \quad (\textit{if } Q[W.../x...] = C)$$

Proof.

$$
\begin{array}{lll}
 & C[(\textbf{template}\,(Q[y] = M); B)\ V] & \\
= & C[(\textbf{template}\,(Q[y]\,\textbf{do}\,M); B)\ V] & \\
= & \left(\begin{array}{l}\textbf{template} \\ \quad \textbf{do}\, M[V/y][W.../x...]; \\ \quad \textbf{continue}\, s' \to C[(\textbf{template}\, B)\ s']\end{array}\right)\ V & (lemma\ 29) \\
= & (\textbf{template}\,\textbf{else}\,M[V/y][W.../x...])\ V & (lemma\ 13) \\
= & (\textbf{template}\,\textbf{continue}\ \_ \to M[V/y][W.../x...])\ V & \\
= & M[V/y][W.../x...] &
\end{array}
$$

□

Lemma 31 (Context Apart). *In the source,*

$$[\![C[(\textbf{template}\,(Q[y]\ O); B)\ V]]\!] = [\![C[(\textbf{template}\, B)\ V]]\!] \quad (\textit{if } Q \,\#\, C)$$

Proof. By induction on the derivation of apartness $Q \,\#\, C$:

– Suppose $Q \# C$ because $Q = Q'\ P$ and $C = C'\ W'$ such that $P \# W$ and $Q'[W.../x...] = C'$.

$$\begin{array}{lll}
& C[(\mathbf{template}\,(Q'[y]\ P)\ O; B)\ V] & \\
= & C'[(\mathbf{template}\,(Q'[y]\ P)\ O; B)\ V]\ W' & \\
= & C'[(\mathbf{template}\,Q'[y]\ (\lambda P.O); B)\ V]\ W' & (\textit{lemma}\ 25) \\
= & \left(\begin{array}{l}\mathbf{template} \\ \quad \lambda P.O[W.../x...]; \\ \quad \mathbf{continue}\,s' \to C'[(\mathbf{template}\,B)\ s']\end{array}\right)\ V\ W' & (\textit{lemma}\ 29) \\
= & (\mathbf{template}\,\mathbf{continue}\,s' \to C'[(\mathbf{template}\,B)\ s'])\ V\ W' & (\textit{lemma}\ 28) \\
= & C'[(\mathbf{template}\,B)\ V]\ W' & (\textit{lemma}\ 18) \\
= & C[(\mathbf{template}\,B)\ V] &
\end{array}$$

– Suppose $Q \# C$ because $Q = Q'\ P$ and $Q' \# C$, and assume the inductive hypothesis $C[(\mathbf{template}\,(Q'[y]\ O); B)\ V] = C[(\mathbf{template}\,B)\ V]$.

$$\begin{array}{lll}
& C[(\mathbf{template}\,(Q[y]\ O); B)\ V] & \\
= & C[(\mathbf{template}\,((Q'[y]\ P)\ O); B)\ V] & \\
= & C[(\mathbf{template}\,(Q'[y]\ (\lambda P.\ O)); B)\ V] & (\textit{lemma}\ 25) \\
= & C[(\mathbf{template}\,B)\ V] & (IH)
\end{array}$$

– Suppose $Q \# C$ because $C = C'\ W$ and $Q \# C'$, and assume the inductive hypothesis $C'[(\mathbf{template}\,(Q[y]\ O); B)\ V] = C'[(\mathbf{template}\,B)\ V]$.

$$\begin{array}{lll}
& C[(\mathbf{template}\,(Q[y]\ O); B)\ V] & \\
= & C'[(\mathbf{template}\,(Q[y]\ O); B)\ V]\ W & \\
= & C'[(\mathbf{template}\,B)\ V]\ W & (IH) \\
= & C[(\mathbf{template}\,B)\ V] &
\end{array}$$

□

Lemma 32 (Context λ^* Match). *In the source,*

$$C[\lambda^*(Q[y] = M); B] = M[(\lambda^*(Q[y] = M); B)/y][W.../x...] \quad (\textit{if}\ Q[W.../x...] = C)$$

Proof.

$$\begin{array}{lll}
& C[\lambda^*(Q[y] = M); B] & \\
= & C[(\mathbf{template}\,(Q[y] = M); B)\ (\lambda^*(Q[y] = M); B)] & (\textit{lemma}\ 15) \\
= & M[(\lambda^*(Q[y] = M); B)/y][W.../x...] & (\textit{lemma}\ 30)
\end{array}$$

□

Lemma 33 (Context λ^* Apart). *In the source,*

$$C[\lambda^*(Q[y]\ O); \mathbf{else}\,M] = C[M] \quad (\textit{if}\ Q \# C)$$

Proof.

$$
\begin{aligned}
& C[\lambda^*(Q[y]\ O); \mathbf{else}\, M] \\
= \ & C[(\mathbf{template}\,(Q[y]\ O); \mathbf{else}\, M)\ (\lambda^*(Q[y]\ O); \mathbf{else}\, M)] && (\textit{lemma } 15) \\
= \ & C[(\mathbf{template}\,\mathbf{else}\, M)\ (\lambda^*(Q[y]\ O); \mathbf{else}\, M)] && (\textit{lemma } 31) \\
= \ & C[(\mathbf{template}\,\mathbf{continue}\ \_ \to M)\ (\lambda^*(Q[y]\ O); \mathbf{else}\, M)] \\
= \ & C[M] && (\textit{lemma } 18)
\end{aligned}
$$

□