**Classical (Co)Recursion: Programming**

PAUL DOWNEN and ZENA M. ARIOLA

*University of Oregon*  
*(e-mail: {pdownen,ariola}@cs.uoregon.edu)*

**Abstract**

Structural recursion is a widespread technique. It is suitable to be used by programmers in all modern programming languages, and even taught to beginning computer science students. What, then, of its dual: structural corecursion? For years, structural corecursion has proved to be an elegant programming technique made possible by languages like Haskell. There, its benefits are to enable compositional algorithm design by decoupling the generation and consumption of (potentially) infinite or large collections of data. However, it is usually thought of as a more advanced topic than structural recursion not suitable for beginners, and not easily applicable outside of the relatively narrow context of lazy, pure functional programming.

Our aim here is to illustrate how the benefits of structural corecursion can be found in a broader swath of the programming landscape than previously thought. Beginning from a tutorial on structural corecursion in the total, pure functional language Agda, we show how these same ideas are mapped to familiar concepts in a variety of different languages. We show how corecursion can be done in strict functional languages like Scheme, and even escapes the functional paradigm entirely, showing up in the natural expression of common object-oriented features found in languages like Python and Java. Opening up structural corecursion to a much wider selection of languages and paradigms—and therefore, also to a much larger audience of programmers—lets us also ask how corecursion interacts with computational effects. Of note, we demonstrate that combining structural corecursion with effects can increase its expressive power. We show a classical version of corecursion—using first-class control made possible by Scheme’s classical call/cc—that enables us to write some new stream-processing algorithms that aren’t possible in effect-free languages.

1 Introduction

In a sense, recursive equations are the ‘assembly language’ of functional programming, and direct recursion the goto.

(Gibbons, 2003)

Recursion on the structure of recursive data types is a common principle for designing practical programs. This notion—based on the idea of induction on the natural numbers in mathematics—has been fruitfully applied to programming for decades, and is so well-understood that every year it is taught to swaths of beginning computer science students as a general-purpose technique of algorithm design (Felleisen et al., 2018). The advantage of
structural recursion is to provide a template to processing input data of any size by combining results given for smaller sub-parts of that input, but with the guarantee that this process will always finish with a final answer. For example, the most basic form of structural recursion is “primitive recursion” on numbers—corresponding to “primitive induction” and which we refer to as just “recursion” for short—stipulates that recursive calls are only allowed on the immediate predecessor of the input. This differs from “general recursion,” which imposes no restrictions at all on the recursive calls, and as such does not come with any guidance for program design or guarantees of termination. Moreover, the technique of structural recursion extends far beyond just numbers, and can capture algorithms over any inductively-defined data type, from lists to finite trees of nearly any shape imaginable.

What, then, of the natural dual of structural recursion: structural corecursion? Corecursion has been used in several applications of coalgebras (Jacobs & Rutten, 1997; Rutten, 2019). But this is quite different from the way structural recursion is presented, in its own right, as an independent technique for practical program design. And it is certainly not a topic that is readily taught to beginning computer science students in this form. We believe this is a sadly missed opportunity, and what’s missing is a purely computational point of view for corecursion. Whereas the structure of recursion directly follows the structure of a program’s inputs, the structure of corecursion directly follows the structure of a program’s outputs (Gibbons, 2021). With this point of view, the practice of corecursion can and should be taught to first-year students, right alongside other fundamental methods of designing and structuring programs. Here, we aim to further this goal by demystifying structural corecursion, making it more suitable for widespread use in programming environments (Gordon, 2017) that share a common hidden notion of codata (Downen et al., 2019).

The primary goal of this paper is to introduce the basic principles of primitive, structural corecursion in a variety of real programming languages. The secondary goal of this paper is to point out the expressive power of different notions of structural corecursion. There are many different formulations of structural recursion that have their own trade-offs, creating an impact on issues like ease of use, expressive power, and computational complexity. For example, the recursion scheme corresponding to primitive recursion is sometimes called a “paramorphism” (Meertens, 1992), with “catamorphism” (Hinze et al., 2013) sometimes used for plain iteration. Identifying and studying these various recursion schemes opens a world of laws and theorems which can be applied by a compiler, such as the catamorphism fusion laws which eliminate intermediate structures (Malcom, 1990). Likewise, there are different formulations of structural corecursions with similar tradeoffs. Primitive coiteration, which unfolds from a given seed, is called “anamorphism” (Meijer et al., 1991); primitive corecursion which can stop the computation at any time is called “apomorphism” (Vene & Uustalu, 1998). These notions of corecursion are all usually studied in the context of pure functional languages like Haskell (Gibbons & Jones, 1998; Hutton, 1998; Gibbons & Hutton, 2005), where the types of infinite versus finite objects are conflated. What happens when we need to distinguish infinite streams from finite lists? Or when computational effects are added? Or when we move away from the functional paradigm entirely? Is there any difference to the expressive power of the various corecursion schemes, and if so, what new algorithms do they enable?
To illustrate the diversity and expressiveness of corecursion in practice, we present a series of examples in a variety of programming languages, which feature different levels of built-in support for corecursion, codata, computational effects, and programming paradigms. We consider four different programming languages: Agda, Scheme, Python, and Java. These four languages let us illustrate the impact of how corecursion appears depending on combinations of these different choices:

- **Paradigm**: either functional (Agda and Scheme) or object-oriented (Python and Java).
- **Typing discipline**: either static (Agda and Java) or dynamic (Scheme and Python)
- **Computational Effects**: which could include either first-class control (Scheme) or exceptions and handlers (Python and Java), or nothing at all (Agda).

We begin in Section 2 with a review of primitive, structural recursion as found in Agda—a dependently-typed, total, functional programming language—where we isolate some common recursive patterns that can be abstracted out in terms of concrete combinators. Following in Section 3, we dualize these recursive patterns in Agda to find corecursive patterns. Rather than inspecting the structure of input via pattern matching, we can inspect the structure of output via copattern matching (Abel et al., 2013). Next, we consider in Section 4 how corecursion can be expressed in Scheme without built-in support for copatterns. Despite the lack of copatterns, Scheme introduces a distinct advantage: programmable, first-class continuations made available by the call/cc control operator. First-class control lets us capture a notion of classical corecursion which is more expressive than before. For example, in Section 4.1 we show a stream algorithm that cannot be expressed with the pure corecursive combinators like anamorphisms and apomorphisms.

From there, we shift our attention away from the functional programming paradigm over to the object-oriented one. In Section 5, we illustrate how the same patterns of structural corecursion can be expressed in Python—a dynamically typed, object-oriented language. This involves the use of persistent objects whose behavior does not change over time, in contrast to the commonly-used ephemeral objects where each method call might affect its future behavior. There, we analyze the use of exceptions to model different forms of streams, adding new features like the ability to end early or skip intermediate elements.

We conclude, in Section 6, by discussing how static types in an object-oriented language like Java interact with structural corecursion and its combinators. In particular, subtyping between static interfaces formalizes the hierarchy streaming types based on their features, and allows us to distinguish between safe operations—in the sense that they will always produce some productive result in finite time—versus unsafe ones that might cause the program to hang. To conclude (Section 7), we hope this paper demonstrates how the general design patterns for corecursion—describing how boundless data should be generated, to be used elsewhere—are applicable to different programming paradigms, too.
Like most statically typed functional languages, Agda lets programmers define new types through data type declarations. For example, a representation of the Peano numbers are captured by this data type definition:

```agda
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

Nat is a type (i.e., a Set) defined inductively over the listed constructors: zero represents the number 0 and succ returns the successor of any number. The inductive property of Nat means all its values are uniquely built up from finite applications of zero and succ.\(^1\)

The inductive nature of Nat is an essential tool for defining operations over natural numbers. First, the fact that the constructors zero and succ create distinct values means that we can pattern-match over them, to inspect values and learn how they were constructed. Second, the fact that these values are finitely constructed means it is well-founded to define functions by structural recursion, by referring to solutions to that same function on “smaller” arguments, where “smaller” for Nat means the argument of succ, i.e., its predecessor. For example, the function for addition (plus m n) can be written by pattern-matching on the first argument in the following structurally recursive way:

```agda
plus : Nat → Nat → Nat
plus zero n = n
plus (succ m) n = succ (plus m n)
```

Since Agda is a total programming language, it checks that plus terminates for any two Nat arguments, and finds that it does because the recursive call to plus m n in the second clause is “smaller” than the original call plus (succ m) n, since m is smaller than succ m.

Since this pattern of well-founded structural recursion over Nat is so common, we could name it and use it many times. If we abstract over the particulars of addition, we arrive at a pattern of recursion that iterates over a single Natural number, providing only the recursive result in the case of a successor. This is also known as a catamorphism (Hinze et al., 2013), and is defined as follows (the parameter in \{ \} is an implicit type parameter):

```agda
iter : {A : Set} → Nat → A → (A → A) → A
iter zero z s = z
iter (succ n) z s = s (iter n z s)
```

In addition to the main number being recursed over, iter takes one argument (z) as the value to give in the base case and a function (s) to apply in the recursive case. Because this iteration might be used to produce any type of result, the return type is an arbitrary type A, so that the base case is z : A and the recursive step is s : A → A. With this abstraction in hand, we can give an equivalent definition for addition in one line by instantiating iter:

```agda
plus' : Nat → Nat → Nat
plus' m n = iter m n succ
```

Likewise, we can define other operations like multiplication both manually (by structural recursion) or as an application of iter as follows:

\(^1\) Any natural number \(n\) is represented as \((\text{succ}^n \text{ zero})\), e.g., 3 is \((\text{succ}(\text{succ}(\text{succ} \text{ zero})))\).
\[
times : \text{Nat} \to \text{Nat} \\
\times \text{zero \ n} = \text{zero} \\
\times (\text{succ \ m}) \ n = \text{plus \ n} (\times \ m \ n)
\]

\[
\times' : \text{Nat} \to \text{Nat} \\
\times' \ m \ n = \text{iter} \ m \ \text{zero} (\text{plus} \ n)
\]

However, some functions resist fitting into the \textit{iter} mold. For example, the predecessor function is the inverse of \textit{succ}. It can be easily defined by pattern-matching as:

\[
pred : \text{Nat} \to \text{Nat} \\
pred \ \text{zero} = \text{zero} \\
pred (\text{succ \ n}) = n
\]

where we take the predecessor of \text{zero} to be again \text{zero}. There does not seem to be a direct way to recover this definition by applying \textit{iter}. Instead, we can define another abstraction which captures this form of shallow \textit{case analysis} that does not recurse, and use it to define the predecessor function, like so:

\[
case : \{A : \text{Set} \} \to \text{Nat} \to A \to (\text{Nat} \to A) \to A \\
case \ \text{zero} \ z s = z \\
case (\text{succ \ n}) z s = s \ n
\]

\[
pred' : \text{Nat} \to \text{Nat} \\
pred' \ n = \text{case} \ n \ \text{zero} (\lambda \ n \to n)
\]

But what happens if we need to do both recursion and shallow pattern matching at the same time? For example, consider the well-known factorial function, defined as:

\[
fact : \text{Nat} \to \text{Nat} \\
fact \ \text{zero} = \text{succ \ zero} \\
fact (\text{succ \ n}) = \times (\text{succ \ n}) (\text{fact} \ n)
\]

The definition of \textit{fact} (\text{succ \ n}) refers to the recursive call (\text{fact} \ n) \textit{and also} to the number \text{n} itself. This pattern of structural recursion, also known as \textit{paramorphism} (Meertens, 1992), does not seem to directly fit either \textit{iter} or \textit{case}. Instead, here is an even more general form of primitive recursion which performs both tasks simultaneously:

\[
\text{rec} : \{A : \text{Set} \} \to \text{Nat} \to A \to (\text{Nat} \to A) \to A \\
\text{rec} \ \text{zero} \ z s = z \\
\text{rec} (\text{succ \ n}) z s = s \ n (\text{rec} \ n \ z \ s)
\]

In \textit{rec}, the function \textit{s} applied in the recursive step for \text{succ \ n} is provided \textit{both} the predecessor \text{n} as well as the recursive result (\text{rec} \ n \ z \ s). Supplying both pieces of information makes it straightforward to redefine \textit{fact} in terms of \textit{rec} like so:

\[
fact' : \text{Nat} \to \text{Nat} \\
fact' \ m = \text{rec} \ m (\text{succ \ zero}) (\lambda \ n \ x \to \times (\text{succ \ n}) x)
\]

We have covered recursion over one \text{Nat}, but what about two? For example, consider this definition for calculating the maximum of two natural numbers:

\[
\max : \text{Nat} \to \text{Nat} \to \text{Nat} \\
\max \ \text{zero \ n} = n \\
\max (\text{succ \ m}) \ \text{zero} = \text{succ \ m} \\
\max (\text{succ \ m}) (\text{succ \ n}) = \text{succ \ (max \ m \ n)}
\]
max pattern-matches against both of its arguments in the latter two clauses, and in the case where they are both successors, it recurses on the predecessor of the two. Is this definition well-founded? Yes, it is still an instance of structural recursion because (both) arguments to the recursive call of max are smaller. But how can max fit into the pattern of rec, which only iterates on a single number? We could select one of the arguments as the “main” number for recursion—in max, the first argument seems to fit well since every clause matches on it. Then the other argument can be inspected by shallow case analysis, as necessary.

However, there is still an issue: even if the first argument of max is the main one for purposes of recursion, both arguments still change in the recursive call. Thankfully, there is another trick we can use: rec can compute any type of result. Previously, all our applications of rec, iter, and case (used to define plus', times', pred', and fact') calculated just a single Nat number. Here, we can instead use rec to calculate a function of type Nat → Nat which corresponds to the partial application max m. In other words, we can use rec to perform higher-order recursion (Harper, 2016; Turner, 2004). This function, in turn, accepts the second argument to max and may inspect that number to respond appropriately.

The alternative definition of max can be given in terms of rec as follows, where we annotate the return type of rec and case by providing the implicit type parameter (in braces { }):

```agda
max' : Nat → Nat → Nat
max' m = rec {Nat → Nat} m
  (λ n → n)
  (λ m' f n → case {Nat} n
    (succ m')
    (λ n' → succ (f n')))
```

In the latter two arguments of rec, the λ-bound n corresponds to the second argument to max'. Additionally, the λ-bound m' stands for the predecessor of the first argument m, and f stands for the partial application of max' m'. Likewise, the λ-bound n' in the last argument of case stands for the predecessor of the second argument n. Notice how the three possible responses—n, (succ m'), and succ (f n')—correspond one-for-one with the right-hand sides to the three clauses defining max above. In particular, succ (f n') is equivalent to succ (max' m' n'), which is the same as the third clause defining max.

3 Agda: Typed Coinductive Copatterns

As a prototypical example of a coinductive type, consider the type of infinite streams, which is defined like so in Agda:

```agda
record Stream A : Set where
  coinductive
  field head : A
  tail : Stream A
```

Instead of a data type with multiple constructors, Stream A is defined as a record: a form of collection which contains a number of fields. In this case, the two fields of a Stream A are a head of type A and a tail of type Stream A. Because of the recursion in the type (the

---

2 This record can be seen as a recursively defined codata type (Downen et al., 2019), where the field accessors are the destructors. All operations on Stream A objects boil down to a combination of head and tail.
tail of a stream is another stream), we have to declare that this type is coinductive to say Stream A objects are infinite (whereas inductive only allows for finite objects).

Since head and tail only provide ways of using streams, how do we create them? Agda uses copatterns (Abel et al., 2013) as a particularly useful way for creating values of coinductive types like Stream. For example, when defining the plus function in Section 2, we gave different answers based on the shape of its arguments. So rather than just giving one definition for a generic call plus m n, we answered two different questions: what is plus zero n equal to? (it’s n) and what is plus (succ m) n equal to? (it’s succ (plus m n)). These two equalities hold by definition, and it is up to the implementation to figure out the details of branching checks and jumps that satisfy them.

Copatterns use this same idea of giving multiple equations, but also allow the programmer to match on the shape of the projections in the context of a definition, not just the arguments in the calling context. In other words, copatterns are a method of designing a program based on the shape of its output (Gibbons, 2021), complementing patterns that consider the shape of input. Consider how to describe a stream built by adding a new element on the front. Let’s write this stream as x ,, xs, where the single element x is followed by the infinite stream xs. The two primary questions to answer about this stream correspond exactly to the field projections: what is head (x ,, xs) equal to? (it’s x) and what is tail (x ,, xs) equal to? (it’s xs). This information is captured by the following definition by copatterns:

\[
_\_,_ : \{A : \text{Set}\} \rightarrow A \rightarrow \text{Stream A} \rightarrow \text{Stream A}
\]

\[
\begin{align*}
\text{head} (x ,, xs) &= x \\
\text{tail} (x ,, xs) &= xs
\end{align*}
\]

We can write corecursive definitions using copatterns, too. For example, the stream that is always the same value x (i.e., x ,, x ,, x ,, ...) and the stream built by repeatedly applying some function f to a starting value x (i.e., x ,, f x ,, f (f x) ,, ...) are:

\[
\text{always} : \{A : \text{Set}\} \rightarrow A \rightarrow \text{Stream A}
\]

\[
\begin{align*}
\text{head} (\text{always} x) &= x \\
\text{tail} (\text{always} x) &= \text{always} x
\end{align*}
\]

\[
\text{repeat} : \{A : \text{Set}\} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow \text{Stream A}
\]

\[
\begin{align*}
\text{head} (\text{repeat} f x) &= x \\
\text{tail} (\text{repeat} f x) &= \text{repeat} f (f x)
\end{align*}
\]

These basic building blocks already let us generate some different streams. For example, consider the stream of all zeroes (0 ,, 0 ,, 0 ,, ...), of all the nats (0 ,, 1 ,, 2 ,, ...), and the alternation back and forth between true and false (true ,, false ,, true ,, false ,, ...). Each is an instance of always and repeat:

\[
\text{zeroes} : \text{Stream Nat}
\]

\[
\text{zeroes} = \text{always} \text{zero}
\]

\[
\text{nats} : \text{Stream Nat}
\]

\[
\text{nats} = \text{repeat} \text{succ} \text{zero}
\]

\[3\] The _,,_ operation is roughly dual to case: whereas case shallowly inspects a natural number, _,,_ shallowly builds a stream. This is why we refer to this operation as cocase later in Section 4.
alt : Stream Bool
alt = repeat not true

But not every infinite stream fits within these simple patterns. For example, there is the stream that maps a function f over every element of another stream xs, as described by:

maps : {A B : Set} → (A → B) → Stream A → Stream B
head (maps f xs) = f (head xs)
tail (maps f xs) = maps f (tail xs)

maps is clearly different from always and repeat. Instead, we need a more general way of generating streams. We the dual of iteration, which is also called an anamorphism (Meijer et al., 1991), is defined by copatterns as:

coiter : {A B : Set} → (B → A) → (B → B) → B → Stream A
head (coiter f g x) = f x
tail (coiter f g x) = coiter f g (g x)

Like repeat g x, the tail of the stream is obtained by updating the seed x to (g x). Unlike repeat g x, each element is calculated on the fly by applying f to the seed x. Because of this additional step, the externally observable elements of the stream can be completely different from the internally private seed used to generate the stream. Intuitively, the type B of the seed is usually “larger” than the type A of the elements, because each element may be just a small part of the internal state of the stream. For example, maps can be expressed as an instance of coiter, where the internal seed is the entire stream of xs that have yet to be transformed, each element is obtained by applying f to just the head of the seed, and the seed is updated with its tail at every step:

maps' : {A B : Set} → (A → B) → Stream A → Stream B
maps' f xs = coiter (λ xs → f (head xs)) (tail xs)

Similar to maps is zipsWith, which combines the elements of two streams (pointwise) using some binary function f. The copattern-based definition of zipsWith is:

zipsWith : {A B C : Set} → (A → B → C) → Stream A → Stream B → Stream C
head (zipsWith f xs ys) = f (head xs) (head ys)
tail (zipsWith f xs ys) = zipsWith f (tail xs) (tail ys)

Is zipsWith also an instance of coiter? Yes, but the encoding is a little more complex. Notice that in the tail case, the corecursive call to zipsWith is given the same function f, but both the two stream arguments xs and ys are changed to (tail xs) and (tail ys). To capture this behavior, the coiteration seed has to contain both streams, which are modified after every step. This can be done by pairing the two together in a product type Stream A × Stream B as in the following definition:

record _×_ A B : Set where
    constructor _,_
    field proj₁ : A
    proj₂ : B

zipsWith' : {A B C : Set} → (A → B → C) → Stream A → Stream B → Stream C
zipsWith' {A} {B} {C} f xs ys = coiter {C} {Stream A × Stream B}
  (λ{(xs , ys) → f (head xs) (head ys)})
  (λ{(xs , ys) → tail xs , tail ys})
  (xs , ys)
Notice that, while `zipsWith'` returns a stream whose elements have type `C`, the internal state used to generate those elements has the type `Stream A × Stream B`, as noted by the explicitly-given implicit arguments (surrounded in braces `{ }`) to `coiter`.

Now consider the stream that counts down from a given number to zero, and then stays at zero from then on. For example, `countDown 3 = 3 ,, 2 ,, 1 ,, 0 ,, 0 ,, ...`. This stream can be defined like so, with three main cases to consider:

```
countDown : Nat → Stream Nat
head (countDown n) = n
tail (countDown zero) = zeroes
tail (countDown (succ n)) = countDown n
```

The head of `(countdown n)` is always the current number `n`, but the tail depends on the value of `n`. If it’s non-zero, as in `tail (countDown (succ n))`, then the stream proceeds to count down from the predecessor `n`. Otherwise it’s zero, and in this case the rest of the stream is always zero. Note that in this zero case, there is no need to continue counting, because the remaining elements are already known in advance: the rest of the stream is exactly the same as `zeroes` anyway, so `zeroes` itself is returned instead of referring to `countDown` again as in the successor case. This is more efficient, by avoiding some unnecessary checks against the argument of `countDown`. Is `countDown` an instance of `coiter`? Yes, this stream contains the same elements:

```
countDown' : Nat → Stream Nat
countDown' = coiter
  (λ n → n)
  (λ { zero → zero ; (succ n) → n})
```

But this equivalence is extensional (only considering what external observers see), not intensional (counting other factors used internally in the implementation). In particular, `countDown'` will keep checking the value of the internal seed at every step; even after the seed becomes 0 and stops changing further, `countDown'` will check it at every tail projection to find it still the same.

To more accurately reflect the cost of `countDown`, we need a more general way of generating streams. This is where the corecursor (also known as an apomorphism (Vene & Uustalu, 1998)) comes into play: it provides a path for ending corecursion once the rest of the stream becomes fully known in advance. We can allow for an early end to corecursion by having the coinductive type return a sum (`a.k.a` disjoint union `B ⊎ Stream A`) of the two possible options: either return a new seed (of type `B`) to continue corecursion, or return a previously-defined stream (of type `Stream A`) to serve as the remainder.

```
data _⊎_ A B : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B

corec : {A B : Set} → (B → A) → (B → Stream A ⊎ B) → B → Stream A
head (corec f g x) = f x
tail (corec f g x) with g x
tail (corec f g x) | inj₁ ys = ys
tail (corec f g x) | inj₂ x’ = corec f g x’
```
Note that the \texttt{with} allows us to pattern match on the result of the update $g \ x$: if it is a stream $ys$ (the \texttt{left} option) then $ys$ will be returned and corecursion stops, but if it is a new seed $x'$ (the \texttt{right} option), then corecursion continues with $x'$. \texttt{corec} lets us more accurately capture the intensional details of \texttt{countDown}, which stops once 0 is reached. This definition of \texttt{countDown} in terms of \texttt{corec} is:

\begin{verbatim}
\texttt{countDown'' : Nat \rightarrow Stream Nat}
\texttt{countDown'' = corec}
\hspace{1cm} (\lambda n \rightarrow n)
\hspace{1cm} (\lambda\{ \, zero \rightarrow \text{inj}_1 \text{ zeroes} \\
\hspace{1cm} , \, (\text{succ} \ n) \rightarrow \text{inj}_2 \ n\})
\end{verbatim}

Another example that shows the need to stop the corecursion is the function that appends a finite \texttt{List} in front of an infinite stream:

\begin{verbatim}
data \texttt{List A : Set where}
\hspace{1cm} [] : \texttt{List A}
\hspace{1cm} _::_ : A \rightarrow \texttt{List A} \rightarrow \texttt{List A}

\texttt{append : \{A : Set\} \rightarrow List A \rightarrow Stream A \rightarrow Stream A}
\texttt{head (append [] ys) = head ys}
\texttt{head (append (x :: xs) ys) = x}
\texttt{tail (append [] ys) = tail ys}
\texttt{tail (append (x :: xs) ys) = append xs ys}
\end{verbatim}

Note that \texttt{append} stops corecursing when the prefix list is empty. This behavior—including the end of \texttt{append}'s corecursion—is correctly expressed by this invocation of \texttt{corec}:

\begin{verbatim}
\texttt{append' : \{A : Set\} \rightarrow List A \rightarrow Stream A \rightarrow Stream A}
\texttt{append' xs ys = corec}
\hspace{1cm} (\lambda\{ [] \rightarrow \text{head} \ ys \\
\hspace{1cm} , \, (x :: xs) \rightarrow x\})
\hspace{1cm} (\lambda\{ [] \rightarrow \text{left} (\text{tail} \ ys) \\
\hspace{1cm} , \, (x :: xs) \rightarrow \text{right} \ xs\})
\hspace{1cm} xs
\end{verbatim}

4 Scheme: Resumable Corecursive Control

Unlike Agda, Scheme does not have built-in support for coinductive types or copatterns. However, we can still represent coinductive types as a form of first-class function which returns a different result depending on the question it receives (following the technique from (Regis-Gianas & Laforgue, 2017)).\footnote{Note that a function is another example of a codata type (Downen et al., 2019).} For example, streams can be modeled as a function $s$ whose single argument can be one of two options:

- If $s$ is passed $\text{'}\text{head}$, then the first element is returned.
- If $s$ is passed $\text{'}\text{tail1}$, then the rest of the stream is returned.

Informally, the type of these functions can be described by the following specification, which is an \textit{intersection} (written as $\&$) between two more specific function types:
The function type 'head -> a denotes functions which can only be applied to exactly the symbol 'head, and returns some a result. Likewise, a function 'tail -> Stream a can only be applied to 'tail. In total, both of these requirements together say that a Stream a is any function that returns a value of type a when applied to 'head, and returns another Stream a when applied to 'tail.

The Stream a interface lets us define a number of functions that operate over any generic Stream. For example, an operations that takes a number of elements from a stream, drops a number of elements from the start of a stream, and fetches the element at an index can each be defined like so:

;; takes : (Stream a, Nat) -> List a
(define (takes s n)
  (cond [(= n 0) '()
          [(= n 1) (list (s 'head))]
          [else (cons (s 'head) (takes (s 'tail) (- n 1)))]))

;; drops : (Stream a, Nat) -> Stream a
(define (drops s n)
  (cond [(= n 0) s
          [else (drops (s 'tail) (- n 1))])))

;; index : (Stream a, Nat) -> a
(define (index s n) ((drops s n) 'head))

But each of these functions assumes we have a stream already. How can we create a new stream from scratch in Scheme? Streams can be created by just returning any first class function matching the Stream interface. For example, the infinite stream of zeros, fitting this type specification, is:

(define zeroes
  (lambda (question)
    (cond [(equal? question 'head) 0]
          [(equal? question 'tail) zeroes])))

So that no matter how deep you go, (((((zeroes 'tail) 'tail) 'tail) ... ) 'head) will always return 0.

The definition of zeroes uses a structure (a lambda taking the caller's question, then testing if that question is equal to 'head or 'tail) that is extremely common for creating streams. Thankfully, we can write a macro which abstracts over this structure to make streams easier to write in Scheme. The cocase macro introduces syntactic sugar for creating a stream by cases on the observation:

(define-syntax-rule (cocase [destructor result] ...)
  (lambda (question)
    (cond [(equal? question destructor) result]
          [else (cocase [destructor result] ...)]))

So that this alternative definition of zeroes expands exactly into the longer one above.

5 Using (cocase ['head x] ['tail xs]) to generate a stream is equivalent to x ,, xs from Section 3.
We can also generalize zeroes to any stream that always returns the same value x:

(define (always x) (cocase ['head x] ['tail (always x)]))

As with zeroes, we can use cocase and self-reference to create a wide variety of infinite streams. For example, the streams which count up starting from some given number n (n (+ n 1) (+ n 2) ...), or down (n (- n 1) ... 1 0 0 ...; staying at 0) are:

(define (count-up n)  
 (cocase ['head n]  
     ['tail (countUp (+ n 1))])))

(define (count-down n)  
 (cocase ['head n]  
     ['tail (cond [(= n 0) zeroes]  
                    [else (countDown (- n 1))])))))

Notice how the three clauses defining the analogous countDown via copatterns in Agda (Section 3) can be found in the definition count-down here.

Just like before, we can abstract out a common pattern of generating streams by coiteration, which uses an internal state to make the 'head on the fly. To generate the 'tail, we need to update the state and continue coiterating.

;; coiter : (b -> a, b -> b , b) -> Stream a  
(define (coiter make update state)  
 (cocase  
     ['head (make state)]  
     ['tail (coiter make update (update state))])))

As we know, the operation which maps a function f over all the elements of a stream is an instance of coiteration in Scheme, too.

;; maps* : (a -> b, Stream a) -> Stream b  
(define (maps* f xs)  
 (coiter  
     (lambda (xs) (f (xs 'head)))  
     (lambda (xs) (xs 'tail))  
     xs))

But not every stream can be generated from just coiteration. We have seen in the previous section that even simple functions like count-down are not faithfully captured by coiteration. In Agda, we used a sum type that lets the programmer control when corecursion ends early (possibly continuing forever). But once a corec loop is ended in this way, it is done for good. In Scheme, we can do something more: using the first-class control provided by call/cc, we can provide two continuations in the 'tail step of the corecursive loop. The first (implicit) continuation lets the 'tail step update the state of the loop and continue corecursing, while the second (explicit) continuation captures the caller who requested the 'tail of the stream. As such, we define the classical corec as the following generalization of coiter:
 Classical (Co)Recursion: Programming

13

;; corec : (b -> a, (Cont (Stream a), b) -> b, b) -> Stream a
(define (corec make update state)
  (cocase
    ['head (make state)])
  ['tail (call/cc
    (lambda (finish)
      (corec make update (update finish state))))])

Unlike a sum type which definitively ends the loop once and for all, continuations can be invoked multiple times, which give the ability to “pause” and “resume” the loop at will. Yet, the only difference from coiter above is that in the coinductive step, corec captures the ‘tail caller in the continuation finish, and provides it to the update function along with the current state. Other than this difference, corec and coiter are the same.

Analogous to appending a list in front of a stream in Agda (Section 3), the direct definition of append in Scheme is:

;; append : (List a, Stream a) -> Stream a
(define (append xs ys)
  (cocase ['head (cond [(null? xs) (ys 'head)]
                        [else (car xs)]])
       ['tail (cond [(null? xs) (ys 'tail)]
                    [else (append (cdr xs) ys)]))])

This append function can alternatively be defined as an instance of corec:

(define (append* xs ys)
  (corec xs
    (lambda (xs)
      (cond [(null? xs) (ys 'head)]
            [else (car xs)]))
    (lambda (finish xs)
      (cond [(null? xs) (finish (ys 'tail))]  
            [else (cdr xs)]))))

The state of this corecursion is the finite prefix list xs. Notice that the bodies of the two functions in append* are almost identical to the ‘head and ‘tail branches used in append. The difference is that the coinductive step, instead of returning directly when xs is empty, must invoke the continuation finish to end the loop. And instead of continuing the loop by calling itself when xs is non-empty, we simply return the updated state (cdr xs).

4.1 The power of classical corecursion

The use of classical corec to encode append amounts to the same as the non-classical version in Agda (Section 3). There is an improved performance from ending the loop early, but performance aside, the same result could be calculated via coiter instead.

However, sometimes the difference between coiteration and corecursion goes beyond just mere performance. Some streams truly need the full generality of a classical corec to be defined at all. In here, we exhibit the expressive power of classical corecursion over
coiteration in a language like Scheme with control operators. In particular, the access to the continuation pointing to the \texttt{tail} caller gives more flexibility than merely stopping the iteration. For example, consider this fact about infinite bit streams made up of just two different element values \texttt{#t} (the 1 bit) and \texttt{#f} (the 0 bit):

**Fact 4.1.** For every infinite stream $s$ over a finite alphabet $A$, there is an element $x \in A$ such that $x$ occurs infinitely often in $s$. As a consequence, every infinite bit stream (over the alphabet \texttt{#t} and \texttt{#f}) contains either an infinite number of \texttt{#t}s or an infinite number of \texttt{#f}s.

This is an application of the pigeon-hole property. If there are only two holes (\texttt{#t} or \texttt{#f}) that together have to house an infinite number of occupants, than at least one of the holes (maybe both, but not necessarily) must have an infinite number of occupants. And this fact generalizes to any finite alphabet, because that still leads to a finite number of holes being filled with an infinite number of occupants.

Given a bit stream, can we calculate outright which value occurs infinitely often? Unfortunately, no. However, we can weaken fact 4.1 slightly: instead of an infinite number of occurrences of $x$ given all at once, we can promise only finite occurrences but in any amount desired.

**Fact 4.2.** For every infinite stream $s$ over a finite alphabet $A$ (such as $A = \{\texttt{#t}, \texttt{#f}\}$), and for any $n$, there is an element $x \in A$ such that there are $n$ different occurrences of $x$ in $s$.

This fact is much more tractable. Up front, we are only asked for a specific amount $n$ of occurrences, so we know when we have gathered enough evidence to say which element occurs that many times. Concretely, we can represent this evidence as indexes $i_0, i_1, \ldots$ into the original stream $s$, such that the value of $s$ at each such $i$ is the same $x$. Now, the main challenge is that the observer can ask for many different number of occurrences, and so our options for which $x$ is chosen might have to change as that count increases. For example, consider the stream $s$ made up of 100 \texttt{#f}s, followed by 1 million \texttt{#t}s, and then infinite \texttt{#f}s. If we are asked for an element that appears 10 times, then we might say \texttt{#f}, because it is very common at the start of the list. But then if we are asked for 1000 occurrences, we might want to say \texttt{#t} because many more of them will be found before we see the 101\textsuperscript{st} \texttt{#f}. Yet, if we are asked for 1 billion occurrences, we have no choice but to say \texttt{#f}; there simply aren’t enough \texttt{#t} occurrences in $s$ to satisfy the request.

We can mediate between fact 4.1 and fact 4.2 using first-class control in Scheme. Effectively, we can provide a stream that appears to implement fact 4.1 to the programmer. Yet, at the end of the day, only fact 4.2 need be implemented, because every terminating program can only inspect a finite number of elements in a stream before it ends. The first-class control present in classical corecursion lets us automatically infer this finite number while the program runs, without the programmer’s explicit knowledge or intervention.

\footnote{In a purely functional language, classical corec is clearly more expressive than coiter; it captures and provides a first-class continuation to the \texttt{tail} branch, which is otherwise not possible in a pure language.}

\footnote{We could be given the stream that is always \texttt{#f}, so the answer is obviously \texttt{#f}, but we could be given the stream of 100 \texttt{#f}s before it is always \texttt{#t}, and we would need to know the answer is \texttt{#t} even though there are many \texttt{#f}s at the start. There is no way to know, \textit{a priori}, how deep into a stream we need to check to be confident which element appears infinitely often, and we cannot exhaustively check all of them.}
Fig. 1: Search for infinite repetitions of a bit in a bit stream.

The call/cc operator creates a check-point by capturing our observer in a continuation that we can invoke several times, back-tracking to the point in time when call/cc was called so we can provide several different answers to that same observer. In this application, we can start to look for the infinite common occurrences by first creating a check-point with call/cc, and then guessing that the head element of the stream is the bit that might occur infinitely often. As long as we keep finding more repetitions of that head bit, then our guess appears correct, and we can keep providing more indexes to repetitions of that bit. However, if we find the other bit in the stream, then our guess might be wrong. In this case, we can back-track to the start and change our answer to the other bit, continuing to search into the remainder of the stream. If we find a repetition of the first bit again, we can back-track yet again to where we left off originally, rather than starting over entirely.

This algorithm can be implemented in Scheme as shown in Fig. 1. The top-level infinite-bits function takes any stream s made up of only two different elements (like #t and #f or 'a and 'b). For the sake of distinguishing these two options, the first element encountered at the 'head of s is considered the 0 bit; the other one not seen yet is the 1 bit. The task of infinite-bits is to return a stream of natural number indices into s all pointing to the same bit. To begin, infinite-bits invokes call/cc to save a check-point for when it was first called, making it possible to completely restart the stream over from the very
beginning, if needed. Then, infinite-bits guesses that there will be enough repetitions of the 0 bit and attempts to find them using the first helper function infinite-bit0:

- The first argument \((s \ '\text{head})\) is the value of the 0 bit.
- The second argument \((s \ '\text{tail})\) is the stream we are searching.
- The third argument keeps track of the depth we have descended into \(s\), used for calculating the indexes. Since we begin at the start of the stream, the initial value is 0.
- The last argument \(\text{restart}\) is a continuation that lets us replace our initial guess with the other possibility: that there are infinite repetitions of the 1 bit in the stream.

The \('\text{head}\) of infinite-bit0 is the current depth that we have searched into the original \(s\). The \('\text{tail}\) is computed on demand, and it depends on the \('\text{head}\) element of the \(\text{rest}\) of \(s\):

- If the 0 bit we’re looking for, then we just continue looking for more occurrences of it in the \('\text{tail}\) of \(\text{rest}\), making sure to increment our \text{depth}.
- Otherwise, it is the 1 bit. In this case, we pause our current search for 0 bits, change our guess to say that the 1 bit occurs infinitely often instead, and begin looking for 1s by switching to the other helper function infinite-bit1. In order to perform the switch, we have to create another check-point saving our current progress in the search for 0 bits. This check-point is another continuation captured by \(\text{call/cc}\), which is passed to infinite-bit1 so that it might switch back and resume the search for 0 bits from where it was paused.

The second helper function infinite-bit1 is defined almost identically to infinite-bit0. The only difference is that when checking each \('\text{head}\) element of the \(\text{rest}\) of the stream, infinite-bit1 instead checks for elements that are not equal to the 0 bit; by the process of elimination, these must be the 1 bit.

The main property of infinite-bits follows the logic of fact 4.1: For any stream \(s\), the lookup \((\text{index } s \ i)\) always returns the same bit \(x\) for every index \(i\) in \((\text{infinite-bits } s)\).

**Property 4.3** (Infinite Bits — Binary). Under the pre-condition that infinite-bits is given an infinite bit stream \(s\), its post-condition is that there is a bit value \(x\) such that

\[
(\text{maps } (\lambda (i) (\text{index } s \ i)) (\text{infinite-bits } s))
\]

is observationally equivalent to \((\text{always } x)\).

For example, we can test out infinite-bits by appending some irregular variations on top of a stream that is always some constant value. If we ask for only 3 repeated occurrences in the stream \('#t \ #f \ #f \ #t \ #f \ #t \ #t \ ...\) then infinite-bits will first find 3 occurrences of \(#f\) before anything else, pointing out their indexes at 1, 2, and 4:

\[
(\text{takes } (\text{infinite-bits } (\text{append } '(#t \ #f \ #f \ #t)) (\text{always } #t))) \ 3) = ' (1 2 4)
\]

However, if we ask for 5 repetitions in that very same stream, there are simply not enough \(#fs\) to be found. Thus, infinite-repetitions will point out 5 different indexes to \(#ts\):
(define (infinite-bits* s)
  (call/cc
   (lambda (restart)
     (infinite-bit0* (s 'head) (s 'tail) 0 restart)))))

(define (infinite-bit0* bit0 rest depth switch)
  (coiter
   (list rest depth switch)
   (lambda (state) (match state [(list _ depth _) depth]))
   (lambda (return state)
     (match state
       [(list rest depth switch)
         (cond
          [(equal? bit0 (rest 'head))
           (list (rest 'tail) (+ 1 depth) switch)]
          [else (call/cc
           (lambda (resume)
             (switch
              (infinite-bit1* bit0 (rest 'tail) (+ 1 depth) resume))))]))))))

(define (infinite-bit1* bit0 rest depth switch)
  (corec
   (list rest depth switch)
   (lambda (state) (match state [(list _ depth _) depth]))
   (lambda (return state)
     (match state
       [(list rest depth switch)
         (cond
          [(not (equal? bit0 (rest 'head)))
           (list (rest 'tail) (+ 1 depth) switch)]
          [else (switch (list (rest 'tail) (+ 1 depth) return))]))))))

(takes (infinite-bits (append '(#t #f #f #t #f) (always #t))) 5) = '(0 3 5 6 7)

Note that, despite the fact that the answer might depend on the observation, a single call to infinite-bits will always give consistent answers throughout its lifetime no matter how many times the result is inspected. For example, if we apply both of the above tests to the same stream returned by infinite-bits, we get consistent approximations each time.

(let [[(ix (infinite-bits (append '(#t #f #f #t #f) (always #t))))]
       (list (takes ix 3) (takes ix 5))]
  = '(((0 3 5) (0 3 5 6 7)))

Even though the first test (asking for only 3 indexes) could initially produce the different approximate result ’(1 2 4) shown above, it is automatically updated with ’(0 3 5) to be consistent with the second, more strenuous, test (asking for 5 indexes).

As we alluded to, identifying infinite repetitions cannot be an instance of coiter. That’s because the crucial infinite-bit0 and infinite-bit1 helper functions need to capture
the caller of each 'tail request (with call/cc), in order to save new check-points during the search. coiter—even when combined with call/cc—cannot generate infinite-bits because it hides the 'tail caller from the update function used in the 'tail step. Notice that if we invoke coiter as

\[
\text{(coiter make (lambda (new-state) (call/cc (k) ...)) state)}
\]

then the continuation k captures the context which updates coiter’s state, not the 'tail caller of coiter. Instead, corec provides exactly this extra information to update. Because of this, we can implement fact 4.1 using two nested corecursive loops, as shown in Fig. 2. The two top-level functions infinite-bits and infinite-bits* are identical. The outer loop of infinite-bits* looks for occurrences of bit 0 (the one we found first at the 'head of the stream). The inner loop(s) looks for occurrences of bit 1 (those not equal to bit 0).

The outer loop infinite-bit0 can be implemented as an instance of coiter, where the state is: 1. the rest of the stream to search through, 2. the current depth into the original stream, and 3. a continuation to switch to the other searching mode. The value of what we are searching for—the bit 0 that appeared at the 'head of the original stream—is not part of the state, because it doesn’t change. As before, the 'head element of this stream is the current depth (found in the state), and the 'tail is computed on-demand. If the next element is equal to 0 bit, then we just continue coiterating with an updated state with 1 more depth and with the 'tail of the 'rest of the stream. Otherwise, we found a 1 bit (because it is different from the 0 bit), and we have to switch our searching mode to look for more 1 bits. To do so, we save our place in this outer loop (with a call to call/cc) so that we can resume it later, and invoke the switch to search for 1 bits with a new inner loop.

The inner loop infinite-bit1 looks for 1 bits (that are not equal to the 0 bit), and is an instance of corec. The state of this corecursion is similar to that of infinite-bit0 except that the type of the switching continuation is different: rather than a fully-formed stream of indexes, it expects a new state that can be used to update and continue the outer-loop. So long as infinite-bit1 finds more 1 bits, it will continue its loop by updating its state in the coinductive step as above. But once infinite-bit1 finds a 0 bit, it has to switch back to the outer-loop, kick-starting it back up. To do so, it passes an updated state including the new continuation that returns to this 'tail caller.

Thus far, we have only considered bit streams made up of only two different values. Yet, facts 4.1 and 4.2 both promise to work with any finite alphabet, not just binary ones. What happens if we try to apply infinite-bits—or equivalently infinite-bits*—to some other non-bit stream? This does not satisfy the pre-condition of the main defining Property 4.3 for this algorithm. Instead, we can only ensure this weakened version:

**Property 4.4 (Infinite Bits — n-ary).** Given any infinite stream \(s\),

\[
(\text{maps (lambda (i) (index s i)) (infinite-bits s)})
\]

is observationally equivalent to one of the two following streams:

1. (always (s 'head)), or
2. some stream of elements \(x1 x2 x3 \ldots\) all non-equal to \((s 'head)\).
So in the general case, `infinite-bits` might not deliver what we were expecting. Given the stream `'(a #t #f #t #f ...)`, `infinite-bits` returns the indexes `1 2 3 4 ...` pointing out the alternating sequence of `#t` and `#f` values—all different from the first element `'(a`. If we want to generalize `infinite-bits` to search streams for infinite repetitions among any number of different elements, we need to move beyond the binary distinction made between its two helper functions. This generalization is made in Fig. 3. The main difference can be seen in the continuation that is used by the single helper function `infinite-of` that searches for repetitions of any given value `x`. In addition to a new stream of indexes, this continuation also expects the value `y` that each index points to. The extra information associates each element value `y` with the current progress in the search for `y` occurrences. In effect, this builds an association map between the different element values that make up the stream and the different continuations waiting for the stream of indexes to those values. The helper function `infinite-of` can then `switch` which element it is searching for by invoking the continuation with the next element it found. In order to return back to this place in the paused search, `call/cc` is used to save the `return` continuation to the current `’tail` caller, and the `switching` continuation is updated by associating the current value `x` with this `return` (with all other values `y` remaining unchanged).

We can almost write `infinite-repetitions` in terms of `corec`. There is just one problem: `infinite-repetitions` might not terminate! If `infinite-repetitions` is given a stream over an infinite alphabet (like, say, `Stream Nat`), then there might not be any infinite repetitions. For example, we cannot point to two occurrences of the same element in `(count-up 0)`, and `infinite-repetitions` will loop forever if it tries. This illustrates one utility of `corec` and `coiter`: they cannot cause infinite loops. In that way, we know that `infinite-bits` terminates because its manual recursion it can be encoded away as...
class Stream:
    # head : Stream -> elem
    # tail : Stream -> Stream

def __iter__(self):
curr = self
while True:
yield curr.head()
curr = curr.tail()

def drops(self, n):
curr = self
for i in range(n):
curr = curr.tail()
return curr

def takes(self, n):
    return [elem for (elem, i) in zip(self, range(n))]

Fig. 4: An abstract class of infinite Streams in Python.

infinite-bits* in terms of only coiter and corec. Though the reason for termination may not be entirely obvious on the surface, these combinators bake it in syntactically. And this is why we can’t encode infinite-repetitions in the same way. The pre-condition to infinite-repetitions—that the stream be built from a finite alphabet—is implicit, so the termination condition cannot be easily expressed in terms of corec.

5 Python: Corecursive Objects and Exceptions

So far, we’ve seen corecursion over streams in two different languages—Agda and Scheme—which are both functional. Does that mean that corecursion only makes sense in the context of the functional paradigm? No! In fact, the idea of corecursion over co-inductive types maps closely to familiar concepts in the object-oriented paradigm. In particular, coinductive types correspond to an interface, and the model of corecursion we have seen thus far corresponds to a form of immutable objects. For example, consider how streams—objects defined in terms of their head and tail projections—can be defined as the abstract Python class shown in Fig. 4. The actual response of the fundamental head and tail depend on the specific stream in question, and cannot be defined in the general case. Thus, they are left abstract for now. However, some helpful derived operations can be given for any stream—so long as they can be defined only in terms of head and tail. For example, we can give a method that drops a given number of elements from the front of the stream, and one that takes the first n elements and returns them in a list. Other special methods that are expected in Pythonic style—such as the __iter__ method for iterating sequentially through the elements of the stream—can also be given a generic definition via head and tail.8

8 The abstract Stream class can be seen as another instance of a codata type (Downen et al., 2019), where the interface corresponds to a codata type definition. Default methods of the Stream interface in Fig. 4 correspond
If the abstract class Stream defines the type of streams, then how do we define actual stream objects? These are given through subclasses of Stream that give real implementations of the head and tail methods. For example, the class of streams generated by repeating the same function on an initial value is:

```python
class Repeat(Stream):
    def __init__(self, state, update=lambda x: x):
        self.state = state
        self.update = update

    def head(self):
        return self.state

    def tail(self):
        return Repeat(self.update(self.state), self.update)
```

The head of Repeat(x,f) is just x. The tail is calculated by applying the update function to the state. That way Repeat(x,f).tail() returns Repeat(f(x),f), so Repeat(x,f) simulates the infinite stream x, f(x), f(f(x)), f(f(f(x))), .... By default, the update function returns its input unchanged, so Repeat(x) will just repeat x forever. Given:

```python
zeroes = Repeat(0)
nats = Repeat(0, lambda x: x+1)
```

then zeroes simulates the stream 0, 0, 0, 0 ... while nats simulates 0, 1, 2, 3, ....

The class of CoIterative objects generalizes Repeat to somehow make each element from the current value of the state, rather than requiring those elements are just the state exactly as-is. This generalization is given as this alternate subclass of Stream:

```python
class CoIter(Stream):
    def __init__(self, make, update, state):
        self.make = make
        self.update = update
        self.state = state

    def head(self):
        return self.make(self.state)

    def tail(self):
        return CoIter(self.make, self.update, self.update(self.state))
```

For an example use of CoIter, consider this class which Maps some transformation function over an existing stream:

```python
class Maps(Stream):
    def __init__(self, stream, trans):
        self.stream = stream
        self.trans = trans

    def head(self):
        return self.trans(self.stream.head())

    def tail(self):
        return Maps(self.stream.tail(), self.trans)
```

Each head element is given by applying the transformation to the head of the current stream, while the tail is computed by taking the tail of the underlying stream. The underlying stream is changed on the recursive call to Maps, so it must be part of its evolving to operations on objects of the codata type. And the following subclasses of Stream that define the head and tail methods are all functions which create objects of the Stream codata type.
internal state, but the transformation stays the same. Thus, the Maps constructor of the above class definition is equivalent to this application of Coiter:

```python
def maps(s, f):
    return CoIter(
        lambda s: f(s.head()),
        lambda s: s.tail(),
        s)
```

For example, we can use the stream of all nats above to simulate the stream of square numbers—0, 1, 4, 9, 16, ...—like so:

```python
squares = maps(nats, lambda x: x*x)
```

As another example, here is the function which zips together the values of two streams:

```python
def zips(left, right, combine = lambda x, y: (x, y)):
    return CoIter(
        lambda state: combine(state[0].head(), state[1].head()),
        lambda state: (state[0].tail(), state[1].tail()),
        (left, right))
```

By default, elements of the two streams x1, x2, x3, ... and y1, y2, y3, ... are just combined as pairs: (x1, y1), (x2, y2), (x3, y3), ....

This can be used to pair up the elements of a stream that appear sequentially next to one another, allowing us to view them together by twos:

```python
def by_twos(stream, f= lambda x, y : (x, y)):
    return zips(stream, stream.tail(), f)
```

For example, by_twos(nats) simulates the stream (0,1), (1,2), (2,3), (3,4), ... of all natural numbers paired with their successor.

**Remark 5.1.** An experienced Python programmer might think "these streams all look awfully similar to iterators, why to just use them instead?" The reason is that all the Stream objects above are persistent: an object's head and tail never changes, no matter how many times these methods are called. In contrast, iterators are ephemeral: each time the next element of an iterator object is requested, the response is different.

Fundamentally, the Iterator interface—which only include the single method next—must execute it's task imperatively. The next method returns the next element as stated, but it also implicitly mutates the iterator itself behind the scenes. That way, the next call to next will return a new element, and not the current one. Instead, the Streams above separate this task into two independent methods: head only returns the first element and tail returns a new Stream with the updated state. As such, head can return the same element every time, and there is no need to modify a Stream object to behave like its tail on the next call.

The persistence allowed by the Stream interface opens up new possibilities. For example, the by_twos function above takes a single stream and copies it, traversing the same stream twice simultaneously. This operation does not make sense for an Iterator, which we can only traverse once. Instead, the user of the Iterator must be responsible for remembering enough of the previous elements it returned in order to simulate the persistent behavior required by by_twos. This could be done by embedding the logic of zips directly into
by twos and explicitly keeping the next two elements at a time. Or it could be done gener-
ically like Python’s standard tee function, which explicitly memoizes the elements across
duplicate references to one Iterator. Persistent Streams eliminate this complication.

We now know to coiterate in an object-oriented style, but what of corecursion? Previ-
ously we saw the functional version of corecursion that could end the corecursive
loop early (in Agda, Section 3) and even provide a continuation that could be resumed
several times (in Scheme, Section 4). In either case, we needed to provide an alterna-
tive exit path to provide the rest of the stream, rather than updating the internal state. A suit-
ably object-oriented way to provide multiple exit paths is with exceptions. For example, a
Python class for CoRecursion can be defined in terms of a StopCoRec exception as follows:

class StopCoRec (Exception):
    def __init__ (self, s):
        self.remainder = s

class CoRec (CoIter):
    def tail (self):
        try :
            return CoRec (self.make, self.update, self.update (self.state))
        except StopCoRec as done :
            return done.remainder

CoRec is defined as a subclass of CoIter because they share many similarities. Both contain
three parts: 1. a way to make an element from the current state, 2. a way to update the
state, and 3. an initial value for the state. And in fact, the head of both CoRecursion and
CoIteration is identical, so it does not need to be specified again here. However, the method
for computing the tail is different. CoRec expects that the update function might raise an
exception, in which case it responds differently:

- If self.update (self.state) returns normally, then the value returned is used as the
  new state to continue corecursion.
- Otherwise, self.update (self.state) might raise a StopCoRec exception. In this
case, corecursion ends and the remainder of the stream (contained within the
exception) is returned as the tail of the CoRecursor.

For example, the scons function can be defined via CoRec in Python as:

def scons (hd, tl):
    def head (_): return hd
    def tail (_): raise StopCoRec (tl)
    return CoRec (head, tail, None)

Note that the tail of this CoRecursor will always stop immediately, and just return the
given stream tl.

Using exceptions to model alternate exits opens up a world of possibilities for general-
izing streams. For example, we can define a new class for streams which might eventually
come to an end when their tail raises an exception, as shown in Fig. 5. With Ending
streams, we need to expect that any call to tail might raise the Ended exception, and han-
dle it accordingly. For example, the method of iterating through the elements of the stream
(.__iter__) needs to be updated to account for this additional case: if the stream has Ended,
class Ended(Exception): pass

class Ending(Stream):
    # head : self -> elem
    # tail : self -> Stream throws Ended

def __iter__(self):
    curr = self
    while True:
        try:
            yield curr.head()
            curr = curr.tail()
        except Ended:
            break

Fig. 5: An abstract class of Ending streams in Python.

then the iteration should stop. The smallest Ending stream is the one that contains only a Single element:

class Single(Ending):
    def __init__(self, only):
        self.value = only

    def head(self): return self.value
    def tail(self): raise Ended

So that Single(1) represents the stream which contains only the value 1. We can also append one Ending stream onto another stream (which may or may not end) in terms of CoRec as:

def append(prefix, suffix):
    def head(pre):
        return pre.head()
    def tail(pre):
        try:
            return pre.tail()
        except Ended:
            raise StopCoRec(suffix)

    return CoRec(head, tail, prefix)

Similar to append modeled previously in Agda and Scheme, the state of the CoRecursion is the prefix. However, since the prefix is now an Ending stream containing at least one element, there are fewer cases to consider: the head is always the head of the remaining prefix, and the tail depends on whether or not that prefix has Ended. If the prefix has more elements, then the state is updated with its tail. But if the prefix has Ended, then the tail of the append is just the suffix (dropping the single element remaining in the prefix).

Coiteration is defined for Ending streams the same as it is for infinite ones, noting that Ended exceptions are propagated implicitly. So we can convert any finite list into an Ending stream like so:
class Skipped(Exception):
    def __init__(self, skip=None):
        self.value = skip

class Skipping(Stream):
    # head: self -> elem throws Skipped
    # tail: self -> Stream

def __iter__(self):
    curr = self
    while True:
        try:
            yield curr.head()
        except Skipped:
            pass
        curr = curr.tail()

Fig. 6: An abstract class of Skipping streams in Python.

class CoIterEnds(CoIter, Ending): pass

def stream_list(items):
    def increment(i):
        i += 1
        if i < len(items):
            return i
        else:
            raise Ended

    return CoIterEnds(lambda i: items[i], increment, 0)

For example, we can represent the stream that counts down from 3, i.e.,
3, 2, 1, 0, 0, 0, ..., as the following application of append, stream_list, and
zeroes:

count_down = append(stream_list(range(3,0,-1)), zeroes)

In contrast to Ending streams, we can also use exceptions to represent streams
that can skip certain elements, as shown in Fig. 6. With Skipping streams,
we need to expect that any request for the head element might raise a Skipped
exception, and handle it accordingly. For example, the method of iterating through
elements of a Skipping stream (__iter__) needs to be updated to account for this
additional case: if the current head element is Skipped, then iteration must continue
on through the remaining elements without yielding anything. Why might we want
to skip elements explicitly? Consider the task of filtering a stream: removing the
elements that do not pass some predicate. Normally, filter is not
an instance of CoIter for infinite streams because the result might not be another
infinite stream. For example, the predicate lambda x: false will reject every single
element from a stream, so filter(stream, lambda x: false).head() cannot return anything.
However, filtering is an instance of CoIter for Skipping streams:
```python
class CoIterSkips(CoIter, Skipping): pass

def filters(stream, check):
    def head(s):
        x = s.head()
        if check(x):
            return x
        else:
            raise Skipped(x)

    return CoIterSkips(head, lambda s: s.tail(), stream)
```

If the current element passes the check, then it is returned normally. Otherwise, it is just Skipped. For example, we can capture only the even square numbers as:

```python
even_squares = filters(squares, lambda x: x % 2 == 0)
```

In `even_squares`, every other number will be Skipped. We never have to worry about what the next element of the filtered stream is, or if it will ever come. So `filter(stream, lambda x: false)` is well-defined: it is just the infinite stream where every single element is Skipped.

### 6 Java: Typed Interfaces for Corecursive Methods

While Python makes it easy to program with objects and exceptions, our understanding of which methods might return certain exceptions, and which ones do not, is completely informal. This can be an issue for understanding code, because the meaning of the different stream types (ones that might end or skip elements) depends crucially on this implicit contract on when exceptions are expected.

If we want a more formal description of the different stream interfaces, we can instead look to a statically typed language like Java. In particular, Java has checked exceptions which keep track of the exceptions that might be thrown by a method in its type. That way, we will statically know when calling a tail method might end, or when it is guaranteed to go on forever, and the compiler keeps track of the difference for us.

However, before we delve into the different interfaces for streams, lets review how exceptions interact with subtyping. First, consider this functional interface for a basic unary function from type A to B:

```java
public interface Function<A, B> {
    B apply(A arg);
}
```

For example, Java 8 lets us write the lambda expression `x -> x + x`, which can be an object of type `Function<Integer, Integer>`: its apply method will be defined as:

```java
Integer apply(Integer x) { return x + x; }
```

However, an expression like `x -> { throw new Exception(); }` cannot have the same type `Function<Integer, Integer>`. Why not? Because the corresponding method definition

```java
Integer apply(Integer x) { throw new Exception(); }
```
is not well-typed. This method throws an Exception, and Java requires this fact be included in the type of the method.

To allow for checked exceptions in functions, we have to include them explicitly in the interface. For example, we can generalize the Function interface above to this one, whose apply method is stated to throw an exception E:

```java
public interface FunctionThrows<A, B, E extends Exception> {
    B apply(A arg) throws E;
}
```

Now, both \( x \to x + x \) and \( x \to \{ \text{throw new Exception();} \} \) can be given the same type `FunctionThrows<Integer,Integer,Exception>`. That's because

```java
Integer apply(Integer x) throws Exception {
    return x + x;
}
```

```java
Integer apply(Integer x) throws Exception {
    throw new Exception();
}
```

are both well-typed. Checked exceptions note which exceptions might be thrown by a method, but does not require them to.

Now that we have two similar functional interfaces, we can ask when can we use objects of one type in place of the other. Or in other words, which of `Function` or `FunctionThrows` is a subtype of the other? First, consider this method for explicitly converting a `Function` into a `FunctionThrows`:

```java
public static <A, B, E extends Exception>
    FunctionThrows<A, B, E> neverThrows(Function<A, B> f) {
        return x -> f.apply(x);
    }
```

Is this well-typed? Yes. \( f.\text{apply}(x) \) will never throw a checked exception, but it doesn’t matter that the new object \( x \to f.\text{apply}(x) \) doesn’t throw an exception \( E \); checked exceptions are an allowance, not a mandate. In contrast, consider the reverse conversion:

```java
public static <A, B, E extends Exception>
    Function<A, B> mightThrow(FunctionThrows<A, B, E> f) {
        return x -> f.apply(x);
    }
```

Is this well-typed? No. Here, \( f.\text{apply}(x) \) might throw an exception \( E \). This is forbidden by the `Function` interface, whose apply method cannot throw any checked exception. Thus, `Function<A, B>` can be seen as a subtype of `FunctionThrows<A, B, E>` (for any \( E \)), but not vice versa.

With this in mind, let’s now consider the different interfaces for (possibly) infinite streams. Previously in Section 5, we saw streams that expected to be infinite, with head and tail methods that always return. That was generalized to subclasses of streams that might come to an end (when tail raises an exception) or that might skip certain elements (when head raises an exception). But, if we look at the above subtyping relationship between methods that might raise exceptions versus ones that don’t, we see that the Section 5’s subclasses of streams are backwards! The interface of truly infinite streams—whose methods never
public interface Stream<A> {
    public A head() throws Skipped;
    public Stream<A> tail() throws Ended;
}

public class Ended extends Exception { public Ended() {} }
public class Skipped extends Exception { public Skipped() {} }

(a) General Streams that might skip or end.

public interface Ending<A> extends Stream<A> {
    public A head();
    public Ending<A> tail() throws Ended;
}

(b) Ending streams, that never skip elements.

public interface Skipping<A> extends Stream<A> {
    public A head() throws Skipped;
    public Skipping<A> tail();
}

(c) Skipping streams, that never end.

public interface Infinite<A> extends Ending<A>, Skipping<A> {
    public A head();
    public Infinite<A> tail();
}

(d) Infinite streams that never skip or end.

Fig. 7: Four different Stream interfaces in Java.

raise an exception—should be a subtype of both ending and skipping streams—whose tail might have Ended or head might be Skipped—not the other way around.

The four different interfaces of streams given in Fig. 7 present all these possible combinations of checked exceptions for ending and skipping, and take advantage of the subtyping relationship between them. The largest super-interface of Streams (Fig. 7a) that could either skip or end is given first. Then two sub-interfaces refine this one: Ending streams (Fig. 7b) never skip elements and Skipping streams (Fig. 7c) never end. Finally, truly Infinite streams (Fig. 7d) follow a sub-interface combining both of these refinements together. Take note that the types ensure that refinements made by each sub-interface are hereditary among different stream interfaces. For example, an Infinite stream is guaranteed to have a head and tail now, and also all of its tails share this promise because the tail of an Infinite stream is another Infinite stream. Similar heredity follows for the weaker promise of Ending and Skipping streams. Alternatively, we could have said that the tail method of each interface returns any Stream, but this would not express the intent
of the hereditary promise. If the tail of an Infinite stream were just any Stream, then its tail would loose the promise that future head or tail methods definitely return.

Unlike an ending stream—which must contain at least one element—a general Stream may truly be empty, as per the following class:

```java
public class Empty<A> implements Stream<A> {
    public Empty() { }
    public A head() throws Skipped { throw new Skipped(); }
    public Stream<A> tail() throws Ended { throw new Ended(); }
}
```

Like Single (from Section 5) and Empty object has no tail, but unlike Single(x) it also has no head. A skipping stream can simulate emptiness by always skipping its head element:

```java
public class AlwaysSkips<A> implements Skipping<A> {
    public AlwaysSkips() { }
    public A head() throws Skipped { throw new Skipped(); }
    public Skipping<A> tail() { return this; }
}
```

In contrast with Empty—whose observer can discover quite quickly that it will never produce anything—an outside observer will not be able to tell (in any finite amount of time) whether or not AlwaysSkips is effectively empty. The observer will find that every tail will skip, but it can never be sure whether or not an element will eventually be found further in. This issue hints at a problem with skipping streams (and Streams in general).

For example, consider the take method from Section 5, which returns a finite list of the first n elements of a stream. What would new AlwaysSkips<A>().take(1) do? It would loop forever, forever looking for that first non-skipped element of the stream, that will never come. In other words, operations like take, drop, or sequential iteration through the elements of a stream are unsafe for skipping streams (let alone general Streams), because they might loop forever. The proper place to introduce them in the interface hierarchy of Fig. 7 is in ending streams, which can be given as these default implementations in terms of head and tail that could be added to the interface in Fig. 7b:

```java
default public Ending<A> drop(int n) throws Ended {
    Ending<A> dropped = this;
    for (int i = 0; i < n; i++) {
        dropped = dropped.tail();
    }
    return dropped;
}

default public Vector<A> take(int n) {
    Vector<A> taken = new Vector<A>(n);
    Ending<A> dropped = this;
    for (int i = 0; i < n; i++) {
        taken.add(dropped.head());
        try {
            dropped = dropped.tail();
        } catch (Ended e) {
            break;
        }
    }
    return taken;
}
Through inheritance, we can take and drop elements from Infinite streams, too.\footnote{Although, the Infinite interface can override the drop method to provide a tighter type, since dropping elements from an Infinite stream will never be Ended prematurely, and will return another Infinite stream.} Attempting to take or drop some elements from an Ending stream might come up short, but they will never loop forever looking for more.

Instead, if we really insist on taking or iterating through the elements of a Skipping stream, the common, risky component is fast forwarding past all the skipped elements. This is the fundamentally unsafe operation, which could easily loop forever, and is performed by the following class for generating Infinite streams from Skipping ones:

```java
public class FastForward<A> implements Infinite<A> {
    private boolean compressed;
    private Skipping<A> skips;

    public FastForward(Skipping<A> s) {
        this.skips = s;
        this.compressed = false;
    }

    public A head() {
        while (true) {
            try {
                A hd = this.skips.head();
                this.compressed = true;
                return hd;
            } catch (Skipped e) {
                this.skips = this.skips.tail();
            }
        }
    }

    public FastForward<A> tail() {
        if (!this.compressed) {
            this.head();
        }
        return new FastForward<A>(this.skips.tail());
    }
}
```

The first time the `head` element is requested, then the underlying Skipping stream is queried until it finally returns a `head` element (or loops forever trying). That point in the stream is remembered for future calls to `head`, to find it in constant time. Computing the `tail` requires that we have at least found the next `head` element, so that we can move past it. In a stream like `Skipped, Skipped, 0, Skipped, 1, 2, Skipped, 3, ...` we do not want to count the `Skipped` elements, so its tail should at least be `Skipped, 1, 2, Skipped, 3, ...`

So what are the pattern for the well-founded ways of generating streams, and how do they differ between the four interfaces? The definition of coiteration for Infinite streams in Java resembles all the previous coiters, and especially the one in Python:
But this only lets us generate Infinite streams. What if we want to generate a general Stream that might skip or end? Perhaps surprisingly, the only difference is a change to the types of functional parameters that are allowed. In particular, the function that describes the base case—that makes the current element from the state—is allowed to throw a Skipped exception, effectively skipping that state. The function that describes the coinductive case—that updates the state—is allowed to throw the Ended exception, to end coiteration and the stream. Other than this change of types, the code is identical (modulo corecursively calling the StreamCoIter constructor rather than InfiniteCoIter):\(^\text{10}\)

```
public class StreamCoIter<B,A> implements Stream<A> {
    private final B state;
    private final FunctionThrows<B,A,Skipped> make;
    private final FunctionThrows<B,B,Ended> update;

    public StreamCoIter(B x, FunctionThrows<B,A,Skipped> f, FunctionThrows<B,B,Ended> g) {
        this.state = x;
        this.make = f;
        this.update = g;
    }

    public A head() throws Skipped {
        ...}
    public StreamCoIter<B,A> tail() throws Ended {
        ...}
}
```

For example, we know that mapping over an Infinite stream is an instance of coiteration; in Java the instantiation looks like this:

```
public static<A,B> Infinite<B> map(Infinite<A> stream, Function<A,B> trans) {
    return new InfiniteCoIter<Infinite<A>, B>(
        stream,
        s -> trans.apply(s.head()),
        s -> s.tail());
}
```

\(^{10}\) The in-between versions of using coiteration to generate Ending and Skipping streams are defined by allowing only update to throw Ended or allowing only make to throw Skipped, respectively.
Instead, when given a general Stream that might skip elements, the transformation function might want to skip elements, too. This gives us the following more general operation that sometimes maps over the elements of a Stream, and sometimes skips them:

```java
public static <A, B>
    Stream<B> mapSometimes(Stream<A> stream,
    FunctionThrows<A, B, Skipped> trans) {
    return new StreamCoIter<>(stream,
                          s -> trans.apply(s.head()),
                          s -> s.tail());
}
```

Again, note that the body of this function is nearly identical (up to a change of the co-iteration constructor). What's different is that either applying the transformation function or asking for the head of `s` might implicitly throw a `Skipped` exception. Likewise, `s.tail()` might implicitly come to an `Ended` stream. Because `mapSometimes` can accept more function parameters, it encompasses the usual filter function. In particular, `filter` just uses a boolean predicate to check each element, and the ones that fail are `Skipped`:

```java
public static <A>
    Stream<A> filter(Stream<A> stream, Function<A, Boolean> check) {
    return mapSomeTimes(stream,
                         x -> {
                            if (check.apply(x)) {
                                return x;
                            } else {
                                throw new Skipped();
                            }
                        });
}
```

This covers coiteration, but what of corecursion, which is able to more efficiently perform operations like appending a prefix on top of a stream? Here we finally come to the point where Java's type system restricts us. In Python, we used an exception, `HaltCoRec` to capture and effectively return the remainder of the stream. In Java, we cannot catch generic exceptions, and so we have to pick up-front a concrete type of streams—say, streams of integers—that will be given if corecursion halts early. Still, we can give the following class for corecursively generating a `Stream<Integer>`:

```java
public class HaltCoRec extends Exception {
    private final Stream<Integer> rest;
    public HaltCoRec(Stream<Integer> s) { this.rest = s; }
    public Stream<Integer> remainder() { return this.rest; }
}
```

```java
public class CoRec<B> implements Stream<Integer> {
    private final B state;
    private final FunctionThrows<B, Integer, Skipped> make;
    private final FunctionThrows<B, B, HaltCoRec> update;
    public CoRec(B x, FunctionThrows<B, Integer, Skipped> f, FunctionThrows<B, B, HaltCoRec> g) {
        this.state = x;
        this.make = f;
        this.update = g;
    }
```
public Integer head() throws Skipped {
    return this.make.apply(this.state);
}

public Stream<Integer> tail() {
    try {
        B next = this.update.apply(this.state);
        return new CoRec<B>(next, this.make, this.update);
    } catch (HaltCoRec halt) {
        return halt.remainder();
    }
}

Note that here, the make operation might skip the current element, like in the coiterator for Streams. However, instead of ending the stream entirely, the update operation might throw the HaltCoRec exception which contains the remainder of the Stream. Coreursors for the other types of streams—Infinite, Ending, and Skipping ones—can be obtained by changing the type of the remainder contained in the HaltCoRec exception, and by preventing or allowing Skipped exceptions in make as appropriate.

7 Conclusion

Surely, structural corecursion is especially elegant in purely functional languages for unfolding lazy data structures. Yet, we have found that this same core idea can also be found in many other contexts, too. Rather than laziness and purity, we base our shared notion of structural corecursion on codata (Downen et al., 2019), which appears in various guises in a wider variety of programming languages. This way, we are able to show how write programs with structural corecursion in strict languages. This new way of thinking empowers us to combine structural corecursion with computational effects—like first-class control—which increases the combined expressive power and allows us brand new algorithms that aren’t possible in a pure language. We can even leave the functional paradigm entirely to rephrase structural corecursion in terms of concepts familiar in common object-oriented languages.

Allowing the paradigm shift suggests that some of the iconic techniques that are unique to functional languages may be applicable within object-oriented programs, too. For example, we are in the process of translating all the applications of “why functional programming matters” (Hughes, 1989) to commonly-used object-oriented languages. We believe that the ideas of functional programming are truly universal, and can and should be employed by the broader audience of all programmers, even the non-functional ones.

Acknowledgments

This work is supported by the National Science Foundation under Grant No. 1719158.
References

Abel, A., Pientka, B., Thibodeau, D. and Setzer, A. (2013) Copatterns: Programming infinite structures by observations. Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’13, pp. 27–38. ACM.

Downen, P., Sullivan, Z., Ariola, Z. M. and Peyton Jones, S. (2019) Codata in action. Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019. ESOP ’19, pp. 119–146. Springer International Publishing.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2018) How to Design Programs: An Introduction to Programming and Computing. The MIT Press.

Gibbons, J. (2003) The Fun of Programming. Chap. Origami programming, pp. 41–60.

Gibbons, J. (2021) Functional pearl: How to design co-programs. Journal of Functional Programming e11.

Gibbons, J. and Hutton, G. (2005) Proof methods for corecursive programs. Fundamenta Informaticae 66(04):353–366.

Gibbons, J. and Jones, G. (1998) The under-appreciated unfold. In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming pp. 273–279. ACM Press.

Gordon, M. (2017) Corecursion and coinduction: what they are and how they relate to recursion and induction. https://www.cl.cam.ac.uk/archive/mjcg/Blog/WhatToDo/Coinduction.pdf.

Harper, R. (2016) Practical Foundations for Programming Languages. 2nd edn. Cambridge University Press.

Hinze, R., Wu, N. and Gibbons, J. (2013) Unifying structured recursion schemes. Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP ’13, p. 209–220. Association for Computing Machinery.

Hughes, J. (1989) Why functional programming matters. Computer Journal 32(2):98–107.

Hutton, G. (1998) Fold and unfold for program semantics. Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. ICFP ’98, p. 280–288. Association for Computing Machinery.

Jacobs, B. and Rutten, J. (1997) A tutorial on (co)algebras and (co)induction. EATCS Bulletin 62:62–222.

Malcom, G. (1990) Data structures and program transformation. Science of computer programming 14(2):255–279.

Meertens, L. (1992) Paramorphisms. Formal Aspects of Computing 4(09).

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture p. 124–144. Springer-Verlag.

Regis-Gianas, Y. and Laforgue, P. (2017) Copattern-matchings and first-class observations in OCaml, with a macro. Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming. PPDP ’17.

Rutten, J. (2019) The Method of Coalgebra: Exercises in coinduction. CWI, Amsterdam, The Netherlands.

Turner, D. A. (2004) Total functional programming. Journal of Universal Computer Science 10(7):751–768.

Vene, V. and Uustalu, T. (1998) Functional programming with apomorphisms (corecursion). Proceedings of the Estonian Academy of Sciences: Physics, Mathematics pp. 147–161.