A Computational Understanding of Classical (Co)Recursion

Paul Downen
University of Oregon
Eugene, Oregon
pdownen@cs.uoregon.edu

Zena M. Ariola
University of Oregon
Eugene, Oregon
ariola@cs.uoregon.edu

Abstract

Recursion and induction are mature, well-understood topics in programming. Yet their duals, corecursion and coinduction, are still exotic and underdeveloped programming features. We aim to put them on equal footing by giving a foundation for corecursion based on computation, analogous to the original computational foundation of recursion. At the lower level, we show how the connection between the two can be strengthened through their implementation details in an abstract machine. At the higher level, we develop a syntactic equational theory for inductive and coinductive reasoning based on control flow. We also observe the impact of evaluation strategy: call-by-name has efficient recursion and strong coinductive reasoning, but call-by-value has efficient corecursion and strong inductive reasoning.

CCS Concepts: • **Theory of computation → Program schemes**.

Keywords: Corecursion, Coinduction, Control, Duality

ACM Reference Format:
Paul Downen and Zena M. Ariola. 2020. A Computational Understanding of Classical (Co)Recursion. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3414080.3414086

1 Introduction

Induction is a familiar and commonplace notion with many firm foundations: logical, algebraic, and computational. Coinduction—the dual to induction—instead looks unfamiliar, and is usually relegated to coalgebras [36], since traditionally only the categorical setting speaks clearly about this duality.

PPDP '20, September 8–10, 2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09…$15.00
https://doi.org/10.1145/3414080.3414086

The goal of this paper is to introduce the basic principles of coinductive and inductive structures from a computational point of view, by presenting a language that captures known intuitions about them. For example, Harper [23] explains that an element in a infinite stream of natural numbers can only be generated after computing all the proceeding elements. Moreover, the introduction of stream objects is sometimes described as the "dual" to the elimination of natural numbers [35, 37], but how is this so? Especially since, in the λ-calculus, stream objects are introduced with an internal "seed" used to generate the elements in order, whereas the elimination of natural numbers has no such internal state.

To make these intuitions precise, we express the (co)-inductive principles with symmetric forms of *recursion* and *corecursion* in a programming language. This symmetry is encapsulated by the duality [9, 22] between *data types*—defined by the *structure* of objects—and *codata types*—defined by the *behavior* of objects. We aim to demystify coinductive codata types, and to make them more suitable for widespread use in programming environments [15, 21].

Our contributions (•) and observations (–) are:

- (Section 2) We point out the impact of evaluation strategy on recursion combinators for inductive types:
  - In call-by-value, recursion is eager, and is just as inefficient as its encoding in terms of the iterator.
  - In call-by-name, recursion may end early; an asymptotic complexity improvement.
- (Section 3) In an abstract machine, the recursor for inductive types like numbers accumulates a continuation during evaluation, maintaining the progress of recursion that is implicit in the λ-calculus.
- (Section 4) Applying duality in an abstract machine based on the sequent calculus [4, 11], we derive the corresponding corecursors for infinite streams with a value accumulator dual to the recursor's continuation.
- (Section 4.2) Like recursion, corecursion can be more efficient than coiteration by letting corecursive processes stop early. Dual to recursion, this improvement in algorithmic complexity is only seen in call-by-value.
- (Section 5) We translate the fully-general corecursor from the abstract machine back into direct style in terms of a $\lambda\mu$-calculus using first-class control effects.
- (Section 5.1) (Co)recursive introduction rules require an internal state to express non-trivial computations.

$$
\begin{array}{rl}
A, B & ::= A \to B \mid \mathrm{nat} \\
M, N & ::= x \mid \lambda x.M \mid M\ N \mid \mathrm{zero} \mid \mathrm{succ}\, M \mid \mathbf{rec}\, M\, \mathbf{as}\, \mathit{ralt} \\
\mathit{ralt} & ::= \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.M\}
\end{array}
$$

Figure 1. System T: λ-calculus with numbers and recursion.

In contrast, recursive elimination rules (as in System T) do not. We define the corresponding stateless corecursor as *coelimination*.

- (Section 6) We present a direct-style equational theory of natural numbers and streams, corresponding to weak induction and coinduction, based on *control flow*. The theory does not resort to bisimulation [18]; it uses syntactic, locally criteria for valid applications of the coinductive hypothesis.
- (Section 6.3) We identify the impact of evaluation strategy on equational reasoning in order to generalize the weak (co)inductive principles to *strong* (co)induction:
  - In call-by-value, strong induction is sound.
  - In call-by-name, strong coinduction is sound, which subsumes the traditional notion of bisimulation.

2 Recursion in the Lambda Calculus

The prototypical example of computational recursion is Gödel's System T [20], whose syntax is given in fig. 1. System T extends the simply-typed λ-calculus, whose focus is on functions of type $A \to B$, with ways to construct and use natural numbers of type nat. Values of nat are built with the familiar constructors zero (denoting the number 0) and succ M (denoting the successor of the number represented by M). This way, the number n can be written as succ^n zero. To use these numbers, System T includes a recursor of the form $\mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\}$. This recursor is similar to pattern-matching in functional programming: the term M is analyzed to determine if it has the shape zero or succ x, and the matching branch is returned. But in addition to binding the predecessor of M to x in the succ x branch, this recursor also binds y to the *recursive result* that is returned if M is replaced with its predecessor.

Notice how the recursor performs two jobs at the same time: finding the predecessor of a natural numbers as well as calculating the recursive result given for the predecessor. These two functionalities are captured separately by a conventional **case**-expression and the *iterator*, respectively. Both can be expressed in terms of macro-expansions into the recursor by ignoring the unneeded parameter:

$$
\begin{array}{rcl}
\mathbf{case}\, M\, \mathbf{of}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to N'\} & := & \mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to \_.\, N'\} \\
\mathbf{iter}\, M\, \mathbf{of}\, \{\mathrm{zero} \to N \mid \mathrm{succ} \to y.\, N'\} & := & \mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, \_ \to y.\, N'\}
\end{array}
$$

$$
\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var}
\qquad
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ {\to}I
\qquad
\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ {\to}E
$$

$$
\frac{}{\Gamma \vdash \mathrm{zero} : \mathrm{nat}}\ \mathrm{nat}I_{\mathrm{zero}}
\qquad
\frac{\Gamma \vdash M : \mathrm{nat}}{\Gamma \vdash \mathrm{succ}\, M : \mathrm{nat}}\ \mathrm{nat}I_{\mathrm{succ}}
$$

$$
\frac{\Gamma \vdash M : \mathrm{nat} \quad \Gamma \vdash N : A \quad \Gamma, x : \mathrm{nat}, y : A \vdash N' : A}{\Gamma \vdash \mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\} : A}\ \mathrm{nat}E
$$

Figure 2. Type system of System T.

Call-by-name values (V) and evaluation contexts (E):

$$V, W ::= M \qquad E ::= \Box \mid E\ N \mid \mathbf{rec}\, E\, \mathbf{as}\, \mathit{ralt}$$

Call-by-value values (V) and evaluation contexts (E):

$$
\begin{array}{rl}
V, W & ::= x \mid \lambda x.M \mid \mathrm{zero} \mid \mathrm{succ}\, V \\
E & ::= \Box \mid E\ N \mid V\ E \mid \mathrm{succ}\, E \mid \mathbf{rec}\, E\, \mathbf{as}\, \mathit{ralt}
\end{array}
$$

Operational rules, where $\mathit{ralt} = \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\}$:

$$
\begin{array}{ll}
(\beta_{\to}) & (\lambda x.M)\ V \mapsto M[V/x] \\
(\beta_{\mathrm{zero}}) & \mathbf{rec}\, \mathrm{zero}\, \mathbf{as}\, \mathit{ralt} \mapsto N \\
(\beta_{\mathrm{succ}}) & \mathbf{rec}\, \mathrm{succ}\, V\, \mathbf{as}\, \mathit{ralt} \mapsto (\lambda y.N'[V/x])\ (\mathbf{rec}\, V\, \mathbf{as}\, \mathit{ralt})
\end{array}
$$

Figure 3. Operational semantics of System T.

These macro-expansions are analogous to the common syntactic sugar of **let**-bindings in terms of functions:

$$\mathbf{let}\, x = M\, \mathbf{in}\, N := (\lambda x.N)\ M$$

The type system of System T is given in fig. 2. The *Var*, $\to I$ and $\to E$ typing rules are from the simply typed λ-calculus. The two nat$I$ introduction rules give the types of the constructors of nat, and the nat$E$ elimination rule types the nat recursor. If **rec** is seen instead as a primitive function, rather than a syntactic construct, it would have the type

$$\mathbf{rec}^{A}_{\mathrm{nat}} : \mathrm{nat} \to A \to (\mathrm{nat} \to A \to A) \to A$$

whereas the primitive functions corresponding to **case** and **iter** on nat would have the type

$$
\begin{array}{c}
\mathbf{case}^{A}_{\mathrm{nat}} : \mathrm{nat} \to A \to (\mathrm{nat} \to A) \to A \\
\mathbf{iter}^{A}_{\mathrm{nat}} : \mathrm{nat} \to A \to (A \to A) \to A
\end{array}
$$

System T's call-by-name and -value operational semantics are given in fig. 3. Both of these evaluation strategies share operational rules of the same form, with $\beta_{\to}$ being the well-known β rule of the λ-calculus and β_{zero} and β_{succ} defining recursion on the two nat constructors. The only difference between call-by-value and -name evaluation lies in their notion of *values* V (*i.e.,* those terms which can be substituted for variables) and *evaluation contexts* (*i.e.,* the location of the next reduction step to perform). Note that we take this notion seriously, and *never* substitute a non-value for a variable. As such, the β_{succ} rule does not substitute the recursive computation $\mathbf{rec}\, V\, \mathbf{as}\, \mathit{ralt}$ for y, since it might not

be a value (in call-by-value). Instead, the recursive computation is merely bound to y, so that the correct next reduction step can be performed in either evaluation strategy.

In call-by-name, this next step is indeed to substitute **rec** V **as** *ralt* for y, and so we have:

$$\textbf{rec}\ \mathrm{succ}\, M\ \textbf{as}\ \mathit{ralt} \mapsto\!\!\!\!\mapsto N'[M/x, \textbf{rec}\ M\ \textbf{as}\ \mathit{ralt}/y]$$

So call-by-name recursion is computed starting with the current (largest) number first and ending with the smallest number needed (possibly the base case for zero). If a recursive result is not needed (such as with the encoding of **case**) then it is not computed at all, allowing for an early end of the recursion. In contrast, call-by-value must evaluate the recursive result first before it can be substituted for y. As such, call-by-value recursion *always* starts by computing the base case for zero (whether or not it is needed), and the intermediate results are propagated backwards until the case for the initial number is reached. So call-by-value allows for no opportunity to end the computation of **rec** early.

Both the type system and operational semantics of System T work together, ensuring that well-typed programs (*i.e.,* closed terms of type nat) always finish and return a value built by the nat constructors.

Theorem 1 (Type safety & Termination)**.** *If* $\bullet \vdash M : \mathrm{nat}$ *in System T then* $M \mapsto\!\!\!\!\mapsto \mathrm{zero}$ *or* $M \mapsto\!\!\!\!\mapsto \mathrm{succ}\, V$ *for some* V*.*

As example uses of the recursor, we can encode the following definitions by pattern-matching

$$\begin{array}{rlrl} \mathit{plus}\,\mathrm{zero}\ y &= y & \mathit{plus}\ (\mathrm{succ}\, x)\ y &= \mathrm{succ}\ (\mathit{plus}\ x\ y) \\ \mathit{pred}\ \mathrm{zero} &= \mathrm{zero} & \mathit{pred}\ (\mathrm{succ}\, x) &= x \\ \mathit{minus}\ x\ \mathrm{zero} &= x & \mathit{minus}\ x\ (\mathrm{succ}\, y) &= \mathit{minus}\ (\mathit{pred}\ x)\ y \end{array}$$

into System T's **rec** as follows:

$$\begin{array}{rl} \mathit{plus} &= \lambda x.\lambda y.\ \textbf{iter}\ x\ \textbf{as}\{\mathrm{zero} \to y \mid \mathrm{succ} \to z.\, \mathrm{succ}\, z\} \\ \mathit{pred} &= \lambda x.\ \textbf{case}\ x\ \textbf{of}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\, y \to y\} \\ \mathit{minus} &= \lambda x.\lambda y.\ \textbf{iter}\ y\ \textbf{as}\{\mathrm{zero} \to x \mid \mathrm{succ} \to z.\ \mathit{pred}\ z\} \end{array}$$

Note that *plus* can be defined using only **iter**, however *pred* (and therefore *minus*) uses **case**, which is not immediately available from **iter**.

2.1 Recursion vs iteration

We saw how iteration can be encoded in terms of recursion by just ignoring the predecessor. The only cost of this encoding is an unused variable binding, which is easily optimized away. In practice, the encoding of iteration will perform exactly the same as if we had taken **iter**ation as a primitive.

Going the other way, it is also known that recursion can be encoded in terms of iteration by using pairs as follows:

$$\begin{array}{l} \textbf{rec}\ M\ \textbf{as} \\ \quad \{\ \mathrm{zero} \quad \to N \\ \quad \mid \mathrm{succ}\, x \to y.N'\} \end{array} \quad := \quad \begin{array}{l} \mathrm{snd}(\textbf{iter}\ M\ \textbf{as} \\ \quad \{\ \mathrm{zero} \to (\mathrm{zero}, N) \\ \quad \mid \mathrm{succ} \to (x, y).(\mathrm{succ}\, x, N')\}) \end{array}$$

Unfortunately, this encoding of recursion is not always as efficient as the original. If the recursive parameter y is never used (such as in the *pred* function), then **rec** can provide an answer without computing the recursive result. However, when encoding **rec** with **iter**, the result of the recursive value must always be computed before an answer is seen, regardless of whether or not y is needed.

Notice that this difference in cost is only apparent in call-by-name, which can be asymptotically more efficient when the recursive y is not needed to compute N', as in *pred*. In call-by-value, the recursor must descend to the base case anyway before the incremental recursive steps are propagated backward. That is to say, the call-by-value **rec** has the same asymptotic complexity as its encoding via **iter**.

In contrast, inductive reasoning is more powerful in call-by-value, because every (closed) value of type nat will have the form $\mathrm{succ}^n\,\mathrm{zero}$ for some number $n \geq 0$. We will explore the differences in inductive reasoning later in section 6. Therefore, when choosing between a call-by-value or call-by-name semantics for a language, there is a tension between reasoning power and asymptotic efficiency for recursion.

3 Recursion in an Abstract Machine

An abstract machine is a useful tool for explicating lower-level performance details of recursion. Unlike the operational semantics given in fig. 3, which has to search for the next redex at every step, an abstract machine explicitly includes this search in the computation itself. As such, every step of the machine can apply by matching only on the top-level form of the machine state.

We will now examine an abstract machine for System T, which spells out how to find and perform each redex in a computation.[1] First, consider this abstract machine for call-by-name System T based on the Krivine machine [28]:

$$\begin{array}{rl} \langle M\ N \| E\rangle & \mapsto \langle M \| N \cdot E\rangle \\ \langle \textbf{rec}\ M\ \textbf{as}\ \mathit{ralt} \| E\rangle & \mapsto \langle M \| \ \textbf{rec}\ \mathit{ralt}\ \textbf{with}\ E\rangle \\ \\ \langle \lambda x.M \| N \cdot E\rangle & \mapsto \langle M[N/x] \| E\rangle \\ \langle \mathrm{zero} \parallel \textbf{rec}\ \mathit{ralt}\ \textbf{with}\ E\rangle & \mapsto \langle N \| E\rangle \\ \langle \mathrm{succ}\, M \| \ \textbf{rec}\ \mathit{ralt}\ \textbf{with}\ E\rangle & \mapsto \langle N'[M/x, \textbf{rec}\ M\ \textbf{as}\ \mathit{ralt}/y] \| E\rangle \end{array}$$

where $\mathit{ralt} := \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\}$ above. The first two rules are *refocusing* rules that move the attention of the machine closer to the next reduction building a larger continuation: for function application ($N \cdot E$ corresponding to $E[\square\ N]$) and recursion (**rec** *ralt* **with** E corresponding to $E[\textbf{rec}\ \square\ \textbf{as}\ \mathit{ralt}]$). The latter three rules are *reduction* rules which correspond to steps of the operational semantics in fig. 3. Note that in the machine, the recursor must explicitly accumulate and build upon a continuation, "adding to" the place it returns to with every recursive call.

[1] Our primary interest in abstract machines is in the accumulation and use of these continuations. For simplicity, we leave out other common details sometimes specified by abstract machines, such as modeling a concrete representation of substitution and environments.

Commands (c), general terms (v), and general coterms (e):

$$c ::= \langle v \| e \rangle \quad v, w ::= \mu\alpha.c \mid V \quad e, f ::= \tilde{\mu}x.c \mid E$$
$$ralt ::= \{\text{zero} \to v \mid \text{succ}\, x \to y.w\}$$

Call-by-name values (V) and evaluation contexts (E):

$$V, W ::= \mu\alpha.c \mid x \mid \lambda x.v \mid \text{zero} \mid \text{succ}\, v$$
$$E, F ::= \alpha \mid v \cdot E \mid \mathbf{rec}\ ralt\ \mathbf{with}\ E$$

Call-by-value values (V) and evaluation contexts (E):

$$V, W ::= x \mid \lambda x.v \mid \text{zero} \mid \text{succ}\, V$$
$$E, F ::= \tilde{\mu}x.c \mid \alpha \mid V \cdot e \mid \mathbf{rec}\ ralt\ \mathbf{with}\ e$$

Operational reduction rules:

$$\begin{array}{rr}
(\mu) & \langle \mu\alpha.c \| E \rangle \mapsto c[E/\alpha] \\
(\tilde{\mu}) & \langle V \| \tilde{\mu}x.c \rangle \mapsto c[V/x] \\
(\beta_{\to}) & \langle \lambda x.v \| V \cdot E \rangle \mapsto \langle v[V/x] \| E \rangle \\
(\beta_{\text{zero}}) & \langle \text{zero} \| \mathbf{rec}\ ralt\ \mathbf{with}\ E \rangle \mapsto \langle v \| E \rangle \\
(\beta_{\text{succ}}) & \langle \text{succ}\, V \| \mathbf{rec}\ ralt\ \mathbf{with}\ E \rangle \mapsto \langle \mu\alpha.\langle V \| \mathbf{rec}\ ralt\ \mathbf{with}\ \alpha \rangle \\
 & \qquad \| \tilde{\mu}y.\langle w[V/x] \| E \rangle \rangle
\end{array}$$

where $ralt = \{\text{zero} \to v \mid \text{succ}\, x \to y.w\}$

Figure 4. Uniform abstract machine for System T.

This abstract machine is specialized for the call-by-name semantics. How should we approach the call-by-value version? We could define an entirely separate abstract machine with many more refocusing rules and continuations (corresponding to the additional cases of evaluation contexts in call-by-value). Instead, we will unite both evaluation strategies with a common abstract machine, shown in fig. 4, based on the $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus [7]. Machine states are commands (c) of the form $\langle v \| e \rangle$ which puts together a general term (v) which produces an output and a general coterm (e, read as "continuation terms" as well as the dual to terms) which consumes an input.

As with System T, the operational semantics depends on the notion of value (V) and covalue (E, analogous to evaluation contexts and read as "continuation values" as well as the dual to values) specialized to either evaluation strategy. However, now the syntax of general terms and coterms also depend on the definition of values and covalues. General terms include μ-bindings (corresponding to Scheme's `call/cc`) and values, and general coterms include $\tilde{\mu}$-bindings (corresponding to contexts $\mathbf{let}\, x = \square\, \mathbf{in}\, M$) and covalues. Note that in call-by-name, values are the same as general terms, but covalues are more restrictive than coterms. Dually, call-by-value covalues are the same as general coterms, but values are more restrictive than terms.

The semantics in fig. 4 no longer has any of the old refocusing rules; instead the generic μ and $\tilde{\mu}$ reductions perform any needed refocusing by substituting a value for the bound variable x, or a covalue for the bound covariable α (read as "continuation variable" as well as the dual to a variable). For example, the refocusing rule $\langle M\ N \| E \rangle \mapsto \langle M \| N \cdot E \rangle$ is encoded as $M\ N = \mu\alpha.\langle M \| N \cdot \alpha \rangle$, where α stands for the generic E in the refocusing step. More formally, all System T terms

$$\frac{\Gamma \vdash v : A \quad \Gamma \vdash e \div A}{\Gamma \vdash \langle v \| e \rangle}\ Cut$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ VarR \qquad \frac{}{\Gamma, \alpha \div A \vdash \alpha \div A}\ VarL$$

$$\frac{\Gamma, \alpha \div A \vdash c}{\Gamma \vdash \mu\alpha.c : A}\ ActR \qquad \frac{\Gamma, x : A \vdash c}{\Gamma \vdash \tilde{\mu}x.c \div A}\ ActL$$

$$\frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x.v : A \to B} \to R \qquad \frac{\Gamma \vdash v : A \quad \Gamma \vdash e \div B}{\Gamma \vdash v \cdot e \div A \to B} \to L$$

$$\frac{}{\Gamma \vdash \text{zero} : \text{nat}}\ \text{nat}R_{\text{zero}} \qquad \frac{\Gamma \vdash V : \text{nat}}{\Gamma \vdash \text{succ}\, V : \text{nat}}\ \text{nat}R_{\text{succ}}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : \text{nat}, y : A \vdash w : A \quad \Gamma \vdash E \div A}{\Gamma \vdash \mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E \div \text{nat}}\ \text{nat}L$$

Figure 5. Type system for the System T machine.

can be translated to the language of the abstract machine using μ- and $\tilde{\mu}$-bindings as follows:

$$\begin{aligned}
[\![x]\!] &:= x \\
[\![\lambda x.M]\!] &:= \lambda x.[\![M]\!] \\
[\![\text{zero}]\!] &:= \text{zero} \\
[\![\text{succ}\, M]\!] &:= \mu\alpha.\langle [\![M]\!] \| \tilde{\mu}x.\langle \text{succ}\, x \| \alpha \rangle \rangle \\
[\![M\ N]\!] &:= \mu\alpha.\langle [\![M]\!] \| \tilde{\mu}x.\langle [\![N]\!] \| \tilde{\mu}y.\langle x \| y \cdot \alpha \rangle \rangle \rangle \\
[\![\mathbf{rec}\, M\, \mathbf{as}\, ralt]\!] &:= \mu\alpha.\langle [\![M]\!] \| \mathbf{rec}\ [\![ralt]\!]\ \mathbf{with}\ \alpha \rangle \\
[\![\{\text{zero} \to M \mid \text{succ}\, x \to y.N\}]\!] &:= \{\text{zero} \to [\![M]\!] \mid \text{succ}\, x \to y.[\![N]\!]\}
\end{aligned}$$

We can also give a type system directly for the abstract machine, as shown in fig. 5. This system has judgments for assigning types to terms as usual: $\Gamma \vdash v : A$ states that v produces an output of type A. In addition, there are also judgments Γ for assigning types to coterms ($\Gamma \vdash e \div A$ states that e consumes an input of type A) and commands ($\Gamma \vdash c$ states that c is safe to compute, and does not produce or consume anything). With this type system, we can capture the fact that compilation preserves types as well as the operational semantics of System T terms.

Theorem 2 (Type Preservation). *$\Gamma \vdash M : A$ iff $\Gamma \vdash [\![M]\!] : A$.*

Theorem 3 (Operational Correspondence). *For any $\bullet \vdash M : \text{nat}$ in System T:*

1. *$M \mapsto\!\!\!\!\to \text{zero}$ if and only if $\langle [\![M]\!] \| \alpha \rangle \mapsto\!\!\!\!\to \langle \text{zero} \| \alpha \rangle$, and*
2. *$M \mapsto\!\!\!\!\to \text{succ}\, V$ if and only if $\langle [\![M]\!] \| \alpha \rangle \mapsto\!\!\!\!\to \langle \text{succ}\, V' \| \alpha \rangle$,*

under both call-by-name and -value evaluation.

Remark 1. Since the $\overline{\lambda}\mu\tilde{\mu}$-calculus is based on the logic of Gentzen's sequent calculus [16], the type system in fig. 5 too can be viewed as a term assignment for a particular sequent calculus. In particular, the statement $v : A$ corresponds to a proof that A is true. Dually $e \div A$ corresponds to a proof that A is false, and hence the notation, which can be understood as a built-in negation $-$ in $e : -A$. As such, the built-in negation in every $e \div A$ (or $\alpha \div A$) can be removed by swapping between the left- and right-hand sides of the turnstyle ($\vdash$), so that $e \div A$ on the right becomes $e : A$ on the left, and $\alpha \div A$ on the left

Types and corecursive alternatives for streams:

$$A, B ::= \cdots \mid \mathrm{strm}\, A \qquad \mathit{calt} ::= \{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}$$

Extension of call-by-name (co)values:

$$V, W ::= \cdots \mid \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ v \qquad E, F ::= \cdots \mid \mathrm{head}\, E \mid \mathrm{tail}\, E$$

Extension of call-by-value (co)values:

$$V, W ::= \cdots \mid \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V \qquad E, F ::= \cdots \mid \mathrm{head}\, e \mid \mathrm{tail}\, e$$

Operational reduction rules:

$$(\beta_{\mathrm{head}}) \quad \langle \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V \| \mathrm{head}\, E\rangle \mapsto \langle V \| e[E/\alpha]\rangle$$

$$(\beta_{\mathrm{tail}}) \quad \langle \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V \| \mathrm{tail}\, E\rangle \mapsto \langle \mu\gamma.\langle V \| f[E/\beta]\rangle \| \tilde{\mu}x.\langle \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ x \| E\rangle\rangle$$

where $\mathit{calt} := \{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}$.

Figure 6. Stream corecursion in the abstract machine.

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash \mathrm{head}\, E \div \mathrm{strm}\, A}\ \mathrm{strm}L_{\mathrm{head}} \qquad \frac{\Gamma \vdash E \div \mathrm{strm}\, A}{\Gamma \vdash \mathrm{tail}\, E \div \mathrm{strm}\, A}\ \mathrm{strm}L_{\mathrm{head}}$$

$$\frac{\Gamma, \alpha \div A \vdash e \div B \quad \Gamma, \beta \div \mathrm{strm}\, A, \gamma \div B \vdash f \div B \quad \Gamma \vdash V : B}{\Gamma \vdash \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V : \mathrm{strm}\, A}\ \mathrm{strm}R$$

Figure 7. Typing rules for streams in the abstract machine.

becomes $\alpha : A$ on the right. Doing so gives a conventional two-sided sequent calculus as in [4, 11], where the rules labeled L with conclusions of the form $x_i : B_i, \alpha_j \div C_j \vdash e \div A$ correspond to left rules of the form $x_i : B_i \mid e : A \vdash \alpha_j : C_j$ in the sequent calculus.

4 Corecursion in an Abstract Machine

Our abstract machine is based on the $\overline{\lambda}\mu\tilde{\mu}$-calculus, a symmetric language reflecting many dualities of classical logic. Producers (terms) are dual to consumers (coterms), call-by-value is dual to call-by-name [7, 38], and so on. This symmetry can help us answer the question: what is the dual to recursion, *i.e.,* what is *corecursion*?

As a prototypical example of a coinductive type, consider infinite streams of values, chosen for their familiarity (other coinductive types work just as well), which we will represent by the type $\mathrm{strm}\, A$ as given in fig. 6. The intention is that $\mathrm{strm}\, A$ is roughly dual to nat, and so we will flip the roles of terms and coterms belonging to streams. In contrast with nat, which has constructors for building values, $\mathrm{strm}\, A$ has two *destructors* for building covalues. First, the covalue $\mathrm{head}\, E$ projects out the first element of its given stream and passes its value to E. Second, the covalue $\mathrm{tail}\, E$ discards the first element of the stream and passes the remainder of the stream to E. Next, in order to define stream values, we have a *corecursor* of the form $\mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V$ where *calt* matches the head and tail destructors above.

The corecursor generates (on the fly) the values of the stream using V as an incremental accumulator or seed, saving the progress made through the stream so far. In particular, the base case $\mathrm{head}\,\alpha \to e$ matching the head projection just passes the accumulator to e, which (may) return the current element to the continuation α. The corecursive case $\mathrm{tail}\,\beta \to \gamma.f$ also passes the accumulator to f, which may return an updated accumulator (through γ) *or* circumvent further corecursion by returning another stream directly to the remaining projection (via β).

As with the syntax, the operational semantics in fig. 6 and typing rules in fig. 7 are roughly symmetric to the rules for natural numbers, where roles of terms and coterms have been flipped. As with System T, these typing and operational rules work together to ensure type safety and termination when extended with infinite streams.

Theorem 4 (Type safety & Termination)**.** *If* $\alpha \div \mathrm{nat} \vdash c$ *in the (co)recursive abstract machine then* $c \mapsto\!\!\!\!\to \langle \mathrm{zero} \| \alpha\rangle$ *or* $c \mapsto\!\!\!\!\to \langle \mathrm{succ}\, V \| \alpha\rangle$ *for some* V.

Similar to recursion, we can define two special cases of **corec** which only use part of its functionality by just ignoring a parameter in the corecursive branch:

$$\begin{array}{lcl} \mathbf{cocase}\ \{\ \mathrm{head}\,\alpha \to e & & \mathbf{corec}\ \{\ \mathrm{head}\,\alpha \to e \\ \quad \mid \mathrm{tail}\,\beta \to f\} & := & \quad \mid \mathrm{tail}\,\beta \to \_.f\} \\ \quad \mathbf{with}\ V & & \quad \mathbf{with}\ V \\ \mathbf{coiter}\ \{\ \mathrm{head}\,\alpha \to e & & \mathbf{corec}\ \{\ \mathrm{head}\,\alpha \to e \\ \quad \mid \mathrm{tail} \to \gamma.f\} & := & \quad \mid \mathrm{tail}\,\_ \to \gamma.f\} \\ \quad \mathbf{with}\ V & & \quad \mathbf{with}\ V \end{array}$$

Specifically, **cocase** simply pattern-matches on the projection of the form $\mathrm{head}\,\alpha$ or $\mathrm{tail}\,\beta$ without corecursing at all, whereas **coiter** *always* corecurses by providing an updated accumulator in the $\mathrm{tail}\,\beta$ case without referring to β.

As examples, consider the following two streams:

$$\begin{aligned} \mathit{scons}\ x\ (y_1, y_2, \ldots) &= x, y_1, y_2, \ldots \\ \mathit{count}\ x &= x, \mathrm{succ}\, x, \mathrm{succ}(\mathrm{succ}\, x), \ldots \end{aligned}$$

scons x s appends a new element x on top of the stream s, and *count x* counts all the successive natural numbers starting with x. These two definitions can be made formal in terms of a language with *copattern matching* [2] like so:

$$\begin{array}{ll} \mathrm{head}(\mathit{scons}\ x\ s) = x & \mathrm{head}(\mathit{count}\ x) = x \\ \mathrm{tail}(\mathit{scons}\ x\ s) = s & \mathrm{tail}(\mathit{count}\ x) = \mathit{count}\ (\mathrm{succ}\, x) \end{array}$$

which correspond to the following uses of corecursion:

$$\mathit{scons} := \lambda x.\lambda s.\ \mathbf{cocase}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail}\,\beta \to \tilde{\mu}\_.\langle s \| \beta\rangle\}\ \mathbf{with}\ x$$

$$\mathit{count} := \lambda x.\ \mathbf{coiter}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail} \to \gamma.\tilde{\mu}x.\langle \mathrm{succ}\, x \| \gamma\rangle\}\ \mathbf{with}\ x$$

To see the full generality of corecursion, consider *app* that appends a list (serving as the prefix) to a stream (serving as the suffix):

$$\mathit{app}\ [x_1, x_2, \ldots, x_n]\ (y_1, y_2, \ldots) = x_1, x_2, \ldots, x_n, y_1, y_2, \ldots$$

This function can be formally defined in terms of copatterns and **corec**, respectively, like so:

$$\begin{array}{ll} \mathrm{head}(\mathit{app}\ \mathrm{nil}\ \mathit{ys}) = \mathrm{head}\, \mathit{ys} & \mathrm{head}(\mathit{app}\ (\mathrm{cons}\, x\ \_)\ \mathit{ys}) = x \\ \mathrm{tail}(\mathit{app}\ \mathrm{nil}\ \mathit{ys}) = \mathrm{tail}\, \mathit{ys} & \mathrm{tail}(\mathit{app}\ (\mathrm{cons}\, x\ \mathit{xs})\ \mathit{ys}) = \mathit{app}\ \mathit{xs}\ \mathit{ys} \end{array}$$

$$app := \lambda xs.\lambda ys.\, \mathbf{corec}$$
$$\{\mathrm{head}\,\alpha \to \mathbf{case}\,\{\mathrm{nil} \to \langle ys \| \mathrm{head}\,\alpha\rangle \mid \mathrm{cons}\,x\,\_ \to \langle x \| \alpha\rangle\}$$
$$\mid \mathrm{tail}\,\beta \to \gamma.\,\mathbf{case}\,\{\mathrm{nil} \to \langle ys \| \mathrm{tail}\,\beta\rangle \mid \mathrm{cons}\,x\,xs' \to \langle xs' \| \gamma\rangle\}\}$$
$$\mathbf{with}\,xs$$

Above, the continuation $\mathbf{case}\{\mathrm{nil} \to c \mid \mathrm{cons}\,x\,xs \to c'\}$ corresponds to the $\mathbf{case}\,\square\,\mathbf{of}\,\{\mathrm{nil} \to c \mid \mathrm{cons}\,x\,xs \to c'\}$ evaluation context for pattern-matching on lists. Note that, the behavior of the corecursive case for $\mathrm{tail}\,\beta$ depends on the accumulator: the tail xs' of a non-empty $\mathrm{cons}\,x\,xs'$ is passed to γ which updates the corecursive seed for the next stream destructor, whereas the empty nil triggers the end of corecursion because γ is ignored and the suffix stream ys is passed directly to the original continuation $\mathrm{tail}\,\beta$.

4.1 Properly dual (co)recursive types

Although we derived corecursion from recursion using duality, our prototypical examples of natural numbers and streams were not perfectly dual to one another. While the (co)recursive case of $\mathrm{tail}\,E$ looks similar to $\mathrm{succ}\,V$, the base cases of $\mathrm{head}\,E$ and zero don't exactly line up, because head takes a parameter but zero does not.

One way to perfect the duality is to generalize the nat type to numbered A: this type represents values of A labeled with a natural number. We can define $\mathrm{numbered}\,A$ by the following generalized constructors:

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathrm{zero}\,V : \mathrm{numbered}\,A} \qquad \frac{\Gamma \vdash V : \mathrm{numbered}\,A}{\Gamma \vdash \mathrm{succ}\,V : \mathrm{numbered}\,A}$$

$\mathrm{zero}\,V$ labels the value V with 0, and $\mathrm{succ}\,V$ increments the numeric label of V. With the generalized zero constructor, we must generalize recursion with an extra parameter:

$$ralt ::= \{\mathrm{zero}\,x \to v \mid \mathrm{succ}\,y \to z.w\}$$

It turns out that $\mathrm{numbered}\,A$ *is* the proper dual to $\mathrm{strm}\,A$. In more detail, we can express the duality relation (which we write as $\eqsim$) between values and covalues of these two types. Assuming that $V \eqsim E$, we have the following duality between the constructors and destructors:

$$\mathrm{zero}\,V \eqsim \mathrm{head}\,E \qquad \mathrm{succ}\,V \eqsim \mathrm{tail}\,E$$

For (co)recursion, we have the following dualities, assuming $v \eqsim e$ (under $x \eqsim \alpha$) and $w \eqsim f$ (under $y \eqsim \beta$ and $z \eqsim \gamma$):

$$\mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}$$
$$\eqsim \mathbf{rec}\{\mathrm{zero}\,x \to v \mid \mathrm{succ}\,y \to z.w\}$$

We could also express the proper duality by restricting streams instead of generalizing numbers. In terms of the type defined above, nat is isomorphic to $\mathrm{numbered}\,\top$, where $\top$ represents the usual unit type with a single value (often written as ()). Since $\top$ corresponds to logical truth, its dual is the $\bot$ type corresponding to logical falsehood with a single covalue that represents an empty continuation. With this in mind, the type nat is properly dual to $\mathrm{strm}\,\bot$, *i.e.,* an infinite stream of computations that do not return.

From the point of view of polarity in programming languages [31, 39], the $\top$ type for truth we use in "numbered $\top$" should be interpreted as a positive type (written as 1 in linear logic [19]). Dually, the $\bot$ type for falsehood in "strm $\bot$" is a negative type (also called $\bot$ in linear logic).

4.2 Corecursion vs coiteration

Recall from section 2 that recursion in call-by-name versus call-by-value have different algorithmic complexities. The same holds for corecursion, with the benefit going instead to call-by-value. For example, consider the *scons* function for appending a new element to a stream. Ideally, *scons* should not leave a lingering effect on the underlying stream. That is to say, the tail of *scons x s* should just be *s*.

This happens directly in call-by-value. Consider indexing the $n+1^{th}$ element of *scons* in call-by-value:

$$\langle scons \| x \cdot s \cdot \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \mapsto\!\!\!\!\to \langle scons_{x,s} \| \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \quad (\beta_{\to})$$
$$\mapsto\!\!\!\!\to \langle s \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle \quad (\beta_{\mathrm{tail}}\mu\tilde{\mu})$$

Notice how, after the first tail is resolved, the computation incurred by *scons* has completely vanished. In contrast, the computation of *scons* continues to linger in call-by-name:

$$\langle scons \| x \cdot s \cdot \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \mapsto\!\!\!\!\to \langle scons_{x,s} \| \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \quad (\beta_{\to})$$
$$\mapsto\!\!\!\!\to \langle scons_{v_1,s} \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle \quad (\beta_{\mathrm{tail}}\tilde{\mu})$$
$$\text{where } v_1 = \tilde{\mu}\_.\langle x \| \tilde{\mu}\_.\langle s \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle\rangle$$

Here, we will spend time over the next n tail projections to build up an ever larger accumulator until the head is reached, even though the result will inevitably just be asking *s* directly. In this way, the efficiency of corecursion is better in call-by-value than in call-by-name.

Also recall from section 2.1 that we were able to encode the recursor in terms of the (apparently) weaker iterator using pairs. We can do a similar encoding of corecursion in terms of coiteration using the dual of pairs: sum types. Sum types in the sequent calculus look like [38]:

$$\langle \mathrm{left}\,V \| [e_1, e_2]\rangle \mapsto \langle V \| e_1\rangle \quad \langle \mathrm{right}\,V \| [e_1, e_2]\rangle \mapsto \langle V \| e_2\rangle$$

We can then write the following encoding of corecursion:

$$\mathbf{corec}\,\{\,\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\,V \quad := \quad \mathbf{coiter}\,\{\,\mathrm{head}\,\alpha \to [\mathrm{head}\,\alpha, e] \mid \mathrm{tail} \to [\beta, \gamma].[\mathrm{tail}\,\beta, f]\}\ \mathbf{with}\,\mathrm{right}\,V$$

As with recursion, this encoding forces a performance penalty for functions like *scons* which can return a stream directly in the corecursive case instead of updating the accumulator.

5 Functional, Effectful (Co)recursion

We saw how corecursion, and not just coiteration, can be expressed inside an abstract machine. How can corecursion be seen in a more familiar language similar to System T?

Our key insight is that the generality of corecursor is made possible by *control effects* equivalent to Scheme's `call/cc`

$$
\begin{aligned}
A, B &::= A \to B \mid \mathrm{nat} \mid \mathrm{strm}\, A \\
J, L &::= \langle M \| Q \rangle \\
M, N &::= x \mid \mu\alpha.J \mid \lambda x.M \mid M\ N \\
&\quad \mid \mathrm{zero} \mid \mathrm{succ}\, M \mid \mathbf{rec}\, M\ \mathbf{as}\ \mathit{ralt} \\
&\quad \mid \mathrm{head}\, M \mid \mathrm{tail}\, M \mid \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ M \\
Q, R &::= \alpha \mid \tilde{\mu}x.J \\
\mathit{ralt} &::= \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.M\} \\
\mathit{calt} &::= \{\mathrm{head} \to x.N \mid \mathrm{tail}\, \beta \to y.M\}
\end{aligned}
$$

Figure 8. Syntax of $\lambda\mu\tilde{\mu}$ extended with (co)recursion.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash Q \div A}{\Gamma \vdash \langle M \| Q \rangle}\ \mathit{Cut}$$

$$\frac{\Gamma, \alpha \div A \vdash J}{\Gamma \vdash \mu\alpha.J : A}\ \mathit{Act} \qquad \frac{\Gamma, x : A \vdash J}{\Gamma \vdash \tilde{\mu}x.J \div A}\ \mathit{CoAct}$$

$$\frac{\Gamma \vdash M : \mathrm{strm}\, A}{\Gamma \vdash \mathrm{head}\, M : A}\ \mathrm{strm}I_{\mathrm{head}} \qquad \frac{\Gamma \vdash M : \mathrm{strm}\, A}{\Gamma \vdash \mathrm{tail}\, M : \mathrm{strm}\, A}\ \mathrm{strm}I_{\mathrm{tail}}$$

$$\frac{\Gamma, x : B \vdash N : A \quad \Gamma, \beta \div \mathrm{strm}\, A, y : B \vdash N' : B \quad \Gamma \vdash M : B}{\Gamma \vdash \mathbf{corec}\{\mathrm{head} \to x.N \mid \mathrm{tail}\, \beta \to y.N'\}\ \mathbf{with}\ M : \mathrm{strm}\, A}\ \mathrm{strm}E$$

Figure 9. Type system for corecursion in $\lambda\mu\tilde{\mu}$.

Call-by-name values (V), covalues (K), and evaluation contexts (E):

$$
\begin{aligned}
V, W &::= M \qquad K, H ::= \alpha \\
E, F &::= \Box \mid E\ N \mid \mathbf{rec}\, E\ \mathbf{as}\ \mathit{ralt} \mid \mathrm{head}\, E \mid \mathrm{tail}\, E \mid \langle E \| K \rangle
\end{aligned}
$$

Call-by-value values, covalues, and evaluation contexts:

$$
\begin{aligned}
V, W &::= x \mid \lambda x.M \mid \mathrm{zero} \mid \mathrm{succ}\, V \mid \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V \\
K, H &::= Q \\
E, F &::= \Box \mid E\ N \mid V\ E \mid \mathrm{succ}\, E \mid \mathbf{rec}\, E\ \mathbf{as}\ \mathit{ralt} \\
&\quad \mid \mathrm{head}\, E \mid \mathrm{tail}\, E \mid \mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ E \mid \langle E \| K \rangle
\end{aligned}
$$

Operational reduction rules:

$$
\begin{aligned}
(\mu) \quad & \langle E[\mu\alpha.J] \| K \rangle \mapsto J[\langle E[M] \| K \rangle / \langle M \| \alpha \rangle] \\
(\tilde{\mu}) \quad & \langle V \| \tilde{\mu}x.J \rangle \mapsto J[V/x] \\
(\beta_{\to}) \quad & (\lambda x.M)\ V \mapsto M[V/x] \\
(\beta_{\mathrm{zero}}) \quad & \mathbf{rec}\ \mathrm{zero}\ \mathbf{as}\ \mathit{ralt} \mapsto N \\
(\beta_{\mathrm{succ}}) \quad & \mathbf{rec}\ \mathrm{succ}\, V\ \mathbf{as}\ \mathit{ralt} \mapsto \mu\alpha.\langle \mathbf{rec}\, V\ \mathbf{as}\ \mathit{ralt} \| \tilde{\mu}y.\langle N[V/x] \| \alpha \rangle \rangle \\
(\beta_{\mathrm{head}}) \quad & \mathrm{head}(\mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V) \mapsto N[V/x] \\
(\beta_{\mathrm{tail}}) \quad & \langle E[\mathrm{tail}(\mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ V)] \| K \rangle \\
& \mapsto \langle E[\mathbf{corec}\ \mathit{calt}\ \mathbf{with}\ N'[V/y, \langle E[M] \| K \rangle / \langle M \| \beta \rangle]] \| K \rangle
\end{aligned}
$$

Where $\mathit{ralt} := \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\}$ and $\mathit{calt} := \{\mathrm{head} \to x.N \mid \mathrm{tail}\, \beta \to y.N'\}$ above.

Figure 10. Operational semantics for (co)recursion in $\lambda\mu\tilde{\mu}$.

operator. The corecursive branch is given access to two continuations: one to update the accumulator and continue the corecursion, and the other to end corecursion early by providing another stream in its entirety.

Our functional language for expressing corecursion is based on the $\lambda\mu\tilde{\mu}$-calculus [8]—an extension of Parigot's $\lambda\mu$-calculus [33] with $\tilde{\mu}$-abstractions—as shown in fig. 8. In addition to everything from System T, this language has μ-abstractions ($\mu\alpha.J$) and jumps ($\langle M \| Q \rangle$) from the $\lambda\mu$-calculus, along with the dual $\tilde{\mu}$-abstractions ($\tilde{\mu}x.J$). Intuitively, the $\mu\tilde{\mu}$–abstractions serve the same role as in the abstract machine—assigning a name to the continuation of a term or the value given to a continuation, respectively—but now appear in the context of a direct-style, functional language.

The corecursor **corec** *calt* **with** M appears almost the same as it did in the abstract machine (albeit slightly generalized so that the accumulator M does not need to be a value), but the stream alternatives *calt* are written in a different from, $\{\mathrm{head} \to x.N \mid \mathrm{tail}\, \beta \to y.M\}$, so that corecursion can be written entirely with terms. Similarly, the stream projections now appear as terms rather than coterms. head M returns the first element of a stream M, and tail M returns the remaining elements of M. In total, the syntax of this $\lambda\mu\tilde{\mu}$-calculus is biased toward terms, where the only coterms are labels (*i.e.,* covariables α) or abstractions ($\tilde{\mu}x.J$).

As before, we can macro-define the special cases for **cocase** and **coiter** as ignoring the unneeded parameters of **corec** (and the unneeded accumulator for **cocase**):

$$
\begin{aligned}
&\mathbf{cocase}\ \{\mathrm{head} \to N \mid \mathrm{tail} \to M\} := \\
&\qquad \mathbf{corec}\ \{\mathrm{head} \to \_.N \mid \mathrm{tail}\, \alpha \to \_.\mu\_.\langle M \| \alpha \rangle\}\ \mathbf{with}\ \mathrm{zero} \\
&\mathbf{coiter}\ \{\mathrm{head} \to x.N \mid \mathrm{tail} \to y.N'\}\ \mathbf{with}\ M := \\
&\qquad \mathbf{corec}\ \{\mathrm{head} \to x.N \mid \mathrm{tail}\, \_ \to y.N'\}\ \mathbf{with}\ M
\end{aligned}
$$

The intuitive understanding is that **cocase** introduces a stream whose first element is exactly N and remainder is M, analogous to *scons* previously in section 4, so that:

$$
\begin{aligned}
\mathrm{head}(\mathbf{cocase}\{\mathrm{head} \to N \mid \mathrm{tail} \to M\}) &= N \\
\mathrm{tail}(\mathbf{cocase}\{\mathrm{head} \to N \mid \mathrm{tail} \to M\}) &= M
\end{aligned}
$$

As we saw with *scons*, call-by-value evaluation reaches these results directly but it may take (arbitrarily many) steps with call-by-name evaluation. **coiter** can be read as an endless loop that either returns the current element (given by binding the value of M to x in N) or calculates the remaining stream by updating its internal state (given by binding M to y in N). However, there is no opportunity for **coiter** to avoid updating the internal state in the tail case, and so the coiteration will continue no matter how deeply the stream is inspected.

The type system for the (co)recursive $\lambda\mu\tilde{\mu}$-calculus extends the one for System T with additional typing rules given in fig. 9. As with the abstract machine, we have judgments for checking that a continuation Q accepts an input of type A (written $\Gamma \vdash Q \div A$) and that a jump J is well-typed (written $\Gamma \vdash J$). Writing the type for a continuation of A as the negation $-A$ (recall remark 1), we can view the stream operations

as primitive constants with the following types:

$$\text{head} : \text{strm}\, A \to A \qquad \text{tail} : \text{strm}\, A \to \text{strm}\, A$$
$$\mathbf{cocase}_{\text{strm}\, A} : A \to \text{strm}\, A \to \text{strm}\, A$$
$$\mathbf{coiter}^{B}_{\text{strm}\, A} : (B \to A) \to (B \to B) \to B \to \text{strm}\, A$$
$$\mathbf{corec}^{B}_{\text{strm}\, A} : (B \to A) \to (-(\text{strm}\, A) \to B \to B) \to B \to \text{strm}\, A$$

The operational semantics for (co)recursion is given in fig. 10. The rules for functions ($\beta_{\to}$) and recursion (β_{zero}, β_{succ}) are the same as from fig. 3. The μ rule states how the evaluation context is captured by $\mu\alpha.J$ and the $\tilde{\mu}$ rule states how values are bound by $\tilde{\mu}x.J$, after which the jump J is performed in either case. The μ rule makes use of *structural substitution* to substitute an evaluation context (E) for a covariable (α), written as $J[E[M]/\langle\alpha\|M\rangle]$ in general, meaning that every jump of the form $\langle\alpha\|M\rangle$ inside J (where α appears free) is replaced by $E[M]$. Note that to be syntactically well-formed, the left- and right-hand-sides of the μ rule are both jumps. The β_{head} rule projects out the first element of a corecursive stream by substituting the accumulator for the variable bound by the head case. The β_{tail} rule projects out the remainder of a corecursive stream by updating the accumulator. This update performs two important bindings required by the tail case: (a) the current value of the accumulator is substituted for the bound variable, and (b) the current evaluation context is substituted for the covariable bound by the copattern tail β. As such, the β_{tail} reduction may result in a new accumulator if this computation returns normally, or it may jump to β and provide an altogether different stream.

The reduction theory of the (co)recursive $\lambda\mu\tilde{\mu}$-calculus is written as $J \to J'$ (and similar for (co)terms) and includes the compatible closure of the operational rules (that is, they may be applied in *any* context, not just evaluation contexts). We also include the following additional rule in the reduction theory, which accounts for administrative reductions that bind arbitrary terms inside of an evaluation context:

$$(\varsigma) \qquad \langle M\|\tilde{\mu}x.E[x]\rangle \to E[M]$$

ς reduction completes the correspondence with the abstract machine, given by this extension of the System T translation:

$$\begin{aligned}
[\![\langle M\|Q\rangle]\!] &:= \langle [\![M]\!] \| [\![Q]\!] \rangle \\
[\![\alpha]\!] &:= \alpha \\
[\![\tilde{\mu}x.J]\!] &:= \tilde{\mu}x.[\![J]\!] \\
[\![\text{head}\, M]\!] &:= \mu\alpha.\langle [\![M]\!] \| \text{head}\, \alpha\rangle \\
[\![\text{tail}\, M]\!] &:= \mu\alpha.\langle [\![M]\!] \| \text{tail}\, \alpha\rangle \\
[\![\mathbf{corec}\, calt\, \mathbf{with}\, M]\!] &:= \mu\alpha.\langle [\![M]\!] \| \tilde{\mu}x.\langle \mathbf{corec}\, [\![calt]\!]\, \mathbf{with}\, x \| \alpha\rangle\rangle
\end{aligned}$$

$$\begin{aligned}
&[\![\{\text{head} \to x.N \mid \text{tail}\, \beta \to y.M\}]\!] \\
&:= \{\text{head}\, \alpha \to \tilde{\mu}x.\langle [\![N]\!] \| \alpha\rangle \mid \text{tail}\, \beta \to \gamma.\tilde{\mu}y.\langle [\![M]\!] \| \gamma\rangle\}
\end{aligned}$$

This translation preserves both the types and operations of the (co)recursive $\lambda\mu\tilde{\mu}$-calculus in the abstract machine.

Theorem 5 (Type Preservation). *$\Gamma \vdash J$ in (co)recursive $\lambda\mu$ if and only if $\Gamma \vdash [\![J]\!]$, and analogously for (co)terms.*

Theorem 6 (Operational correspondence). *For any jump $\alpha \div \text{nat} \vdash J$ in (co)recursive $\lambda\mu$, under both call-by-name and -value evaluation:*

1. *$J \mapsto\!\!\!\!\to \langle \text{zero}\, \|\alpha\rangle$ if and only if $[\![J]\!] \mapsto\!\!\!\!\to \langle \text{zero}\, \|\alpha\rangle$, and*
2. *$J \mapsto\!\!\!\!\to \langle \text{succ}\, V\|\alpha\rangle$ if and only if $[\![J]\!] \mapsto\!\!\!\!\to \langle \text{succ}\, V'\|\alpha\rangle$.*

The example streams *repeat* $x = x, x, x, \ldots$ and *alt* $= 0, 1, 0, 1, \ldots$ can be written in terms of corecursion like so:

$$\begin{aligned}
\mathit{repeat} &:= \lambda x.\, \mathbf{coiter}\{\text{head} \to y.y \mid \text{tail} \to z.z\}\, \mathbf{with}\, x \\
\mathit{flip} &:= \lambda x.\, \mathbf{case}\, x\, \mathbf{of}\{\text{zero} \to \text{succ zero} \mid \text{succ}\, _ \to \text{zero}\} \\
\mathit{alt} &:= \mathbf{coiter}\{\text{head} \to x.x \mid \text{tail} \to x.\mathit{flip}\, x\}\, \mathbf{with}\, \text{zero}
\end{aligned}$$

Notice that the operational semantics of fig. 10 gives alternating outputs for the elements of *alt*:

$$\begin{aligned}
\langle \text{head}\, \mathit{alt}\|\alpha\rangle &\mapsto \langle \text{zero}\, \|\alpha\rangle \\
\langle \text{head}(\text{tail}\, \mathit{alt})\|\alpha\rangle &\mapsto\!\!\!\!\to \langle \text{head}\, \mathit{alt}'\|\alpha\rangle \mapsto \langle \text{succ zero}\, \|\alpha\rangle \\
\langle \text{head}(\text{tail}(\text{tail}\, \mathit{alt}))\|\alpha\rangle &\mapsto\!\!\!\!\to \langle \text{head}\, \mathit{alt}\|\alpha\rangle \mapsto \langle \text{zero}\|\alpha\rangle
\end{aligned}$$

and so on, where the intermediate state of the stream is $\mathit{alt}' := \mathbf{coiter}\{\text{head} \to x.x \mid \text{tail} \to x.\mathit{flip}\, x\}\, \mathbf{with}\, \text{succ zero}$.

5.1 A corecursive coeliminator

Recall from section 4.1 that in the abstract machine, recursion and cocorecursion are perfectly dual to one another for dual types. The recursor has an internal state in the form of a covalue (*i.e.,* continuation) keeping track of the call stack built up by each step of recursion, whereas the corecursor has an internal state in the form of a value (*i.e.,* accumulator) that keeping of the progress made from one element to the next as projections are made deeper into the stream.

Yet, in the $\lambda\mu\tilde{\mu}$-calculus, the recursor and corecursor no longer appear dual at all: the corecursor still has its internal state but the recursor just returns a result, seemingly statelessly. How can the duality between these two forms be restored? The mismatch is born from the fact that introductions and eliminations of terms are *not* dual do each other in the sense we need. Instead, an elimination of terms is symmetric to an *elimination of coterms*, *i.e.,* a *coelimination*. Coeliminators for functions have appeared before [6, 32], which has been used to study the issues of confluence [27] and head normalization [26] with control effects. We can derive a similar form of coelimination for streams here, using $\tilde{\mu}$-abstractions. First, the head and tail projections can be defined via macro-expansion as coterms like so:

$$\text{head}\, Q := \tilde{\mu}x.\langle \text{head}\, x\|Q\rangle \qquad \text{tail}\, Q := \tilde{\mu}x.\langle \text{tail}\, x\|Q\rangle$$

From there, we can give the following syntactic sugar for corecursion defined by eliminating these two coterms:

$$\begin{aligned}
&\mathbf{corec}\, Q\, \mathbf{as} \\
&\quad \{\ \text{head}\, \alpha \to R \\
&\quad \mid \text{tail}\, \beta \to \gamma.R'\}
\end{aligned} \quad := \quad
\begin{aligned}
&\tilde{\mu}x.\langle \mathbf{corec}\, \{\ \text{head} \to z.\mu\alpha.\langle z\|R\rangle \\
&\qquad\qquad \mid \text{tail}\, \beta \to y.\mu\gamma.\langle y\|R'\rangle\} \\
&\qquad\qquad \mathbf{with}\, x \\
&\qquad \|\ Q\ \rangle
\end{aligned}$$

The new definitions allow us to derive these reductions:

$$\mathbf{corec}\,\mathrm{head}\,K\,\mathbf{as}\,\mathit{calt} \twoheadrightarrow R[K/\alpha]$$
$$\mathbf{corec}\,\mathrm{tail}\,K\,\mathbf{as}\,\mathit{calt} \twoheadrightarrow \tilde{\mu}x.\langle \mu\gamma.\langle x \| R'[K/\beta]\rangle \| \mathbf{corec}\,K\,\mathbf{as}\,\mathit{calt}\rangle$$

This provides us an alternative understanding of corecursion: rather than focusing on generating some infinite object, we can instead focus on processing a finite observation. In other words, we can write corecursive definitions in the same style as recursion on the natural numbers, with the twist that we are calculating a question rather than an answer.

As an example, we can write definitions for the streams

$$\mathit{evens}\ (x_0, x_1, \dots) = x_0, x_2, x_4, \dots \quad \mathit{odds}\ (x_0, x_1, \dots) = x_1, x_3, x_5, \dots$$
$$\mathit{merge}\ (x_0, x_1, \dots)\ (y_0, y_1, \dots) = x_0, y_0, x_1, y_1, \dots$$

using pairs and the coeliminator above as follows:

$$\begin{aligned}
\mathit{evens} &:= \lambda s.\mu\gamma.\langle s \| \mathbf{corec}\,\gamma\,\mathbf{as}\,\{\,\mathrm{head}\,\alpha \to \mathrm{head}\,\alpha \\
&\qquad\qquad \mid \mathrm{tail}\,_ \to \beta.\,\mathrm{tail}(\mathrm{tail}\,\beta)\}\rangle \\
\mathit{odds} &:= \lambda s.\mu\gamma.\langle s \| \mathbf{corec}\,\gamma\,\mathbf{as}\,\{\,\mathrm{head}\,\alpha \to \mathrm{tail}(\mathrm{head}\,\alpha) \\
&\qquad\qquad \mid \mathrm{tail}\,_ \to \beta.\,\mathrm{tail}(\mathrm{tail}\,\beta)\}\rangle \\
\mathit{merge} &:= \lambda s.\lambda s'.\mu\gamma.\langle (s,s') \| \mathbf{corec}\,\gamma\{\,\mathrm{head}\,\alpha \to \tilde{\mu}(s,_).\langle s \| \mathrm{head}\,\alpha\rangle \\
&\qquad\qquad \mid \mathrm{tail}\,_ \to \gamma.\tilde{\mu}(s,s').\langle (s', \mathrm{tail}\,s) \| \gamma\rangle\}\rangle
\end{aligned}$$

6 Syntactic (Co)Inductive Reasoning

We have examined how to define (co)recursive programs and compute their results, but how can we reason about when expressions are equivalent? It's unsatisfactory to only consider two expressions equal when they reduce to some common reduct; that misses out on far too many equalities. For example, it is easy to show by reduction that $\mathit{plus}\,\mathrm{zero}\,x \mapsto\!\!\!\!\twoheadrightarrow x$ because *plus* was defined by recursion on its first argument, but $\mathit{plus}\,x\,\mathrm{zero}$ doesn't reduce at all, even though it is nonetheless equivalent to zero in any context. Likewise, *evens alt* and *repeat* zero are both equivalent to the same stream $(0, 0, 0, \dots)$, but how do we prove it? To answer these questions, we develop a syntactic equational theory with notions of induction and coinduction dual to one another.

6.1 Equational theory of the abstract machine

The equational theory for the abstract machine is given in fig. 11. All judgments have the form $\Gamma \vdash \Phi$, where Γ is the assumptions and Φ is the property being proved. The main properties are the base equalities for commands ($c = c'$) and (co)terms of some type A ($v = v' : A$ and $e = e' \div A$). In addition, properties may universally quantify over (co)variables ($\forall x{:}A.\Phi$ and $\forall \alpha \div A.\Phi$), or depend on the truth of another property ($\Phi' \Rightarrow \Phi$). These latter three forms of properties will turn out to be essential for proving equalities that require a more general (co)inductive hypothesis. (Co)-inductive hypotheses can be stored in the environment as Γ, Φ. Additionally, we have two special cases for properties. A property $\Psi(x)$ is *strict on* x because it uses x directly with some covalue on both sides of its equality. Dually, a property $\Psi(\alpha)$ is *productive on* α because it immediately returns a value to α on both side of its equality.

Properties (Φ), strict ($\Psi(x)$) and productive ($\Psi(\alpha)$) properties, hypothesis (θ), environments (Γ), and judgments:

$$\begin{aligned}
\Phi &::= c = c' \mid v = v' : A \mid e = e' \div A \mid \forall x{:}A.\Phi \mid \forall \alpha \div A.\Phi \mid \Phi' \Rightarrow \Phi \\
\Psi(x) &::= \langle x \| E\rangle = \langle x \| E'\rangle \mid \forall y{:}A.\Psi(x) \mid \forall \alpha \div A.\Psi(x) \qquad (y \neq x) \\
\Psi(\alpha) &::= \langle V \| \alpha\rangle = \langle V' \| \alpha\rangle \mid \forall x{:}A.\Psi(\alpha) \mid \forall \beta \div A.\Psi(\alpha) \qquad (\beta \neq \alpha) \\
\Gamma &::= \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A \mid \Gamma, \Phi \qquad \mathit{Judge} ::= \Gamma \vdash \Phi
\end{aligned}$$

Equality of computation (plus rules for reflexivity, symmetry, transitivity, and compatibility of equality):

$$\frac{\Gamma \vdash c \quad c \mapsto c'}{\Gamma \vdash c = c'}\ \mathit{Red}$$

Induction, coinduction, and extensionality principles:

$$\frac{\Gamma, x : A \vdash \langle x \| e\rangle = \langle x \| e'\rangle}{\Gamma \vdash e = e' \div A}\ \sigma\tilde{\mu} \qquad \frac{\Gamma, \alpha \div A \vdash \langle v \| \alpha\rangle = \langle v' \| \alpha\rangle}{\Gamma \vdash v = v' : A}\ \sigma\mu$$

$$\frac{\Gamma, x : A, \beta \div B \vdash \Psi(x \cdot \beta/\alpha)}{\Gamma, \alpha \div A \to B \vdash \Psi(\alpha)}\ \omega{\to}$$

$$\frac{\Gamma \vdash \Psi(\mathrm{zero}/x) \quad \Gamma, x : \mathrm{nat}, \Psi(x) \vdash \Psi(\mathrm{succ}\,x/x)}{\Gamma, x : \mathrm{nat} \vdash \Psi(x)}\ \omega\mathrm{nat}$$

$$\frac{\Gamma, \beta \div A \vdash \Psi(\mathrm{head}\,\beta/\alpha) \quad \Gamma, \alpha \div \mathrm{strm}\,A, \Psi(\alpha) \vdash \Psi(\mathrm{tail}\,\alpha/\alpha)}{\Gamma, \alpha \div \mathrm{strm}\,A \vdash \Psi(\alpha)}\ \omega\mathrm{strm}$$

Rules of properties:

$$\frac{}{\Gamma, \Phi \vdash \Phi}\ \mathit{Ax} \qquad \frac{}{\Gamma, \mathrm{succ}\,V = \mathrm{zero} : \mathrm{nat} \vdash \Phi}\ \mathit{Coher}$$

$$\frac{\Gamma, \Phi' \vdash \Phi}{\Gamma \vdash \Phi' \Rightarrow \Phi}\ \mathit{IntroH} \qquad \frac{\Gamma \vdash \Phi' \Rightarrow \Phi \quad \Gamma \vdash \Phi'}{\Gamma \vdash \Phi}\ \mathit{Lemm}$$

$$\frac{\Gamma, x : A \vdash \Phi}{\Gamma \vdash \forall x{:}A.\Phi}\ \mathit{IntroL} \qquad \frac{\Gamma \vdash \forall x{:}A.\Phi \quad \Gamma \vdash V = V' : A}{\Gamma \vdash \Phi[V/x = V'/x]}\ \mathit{SubstL}$$

$$\frac{\Gamma, \alpha \div A \vdash \Phi}{\Gamma \vdash \forall \alpha \div A.\Phi}\ \mathit{IntroR} \qquad \frac{\Gamma \vdash \forall \alpha \div A.\Phi \quad \Gamma \vdash E = E' \div A}{\Gamma \vdash \Phi[E/\alpha = E'/\alpha]}\ \mathit{SubstR}$$

Figure 11. Equational theory of (co)induction in the abstract machine.

The general rules for properties formalize their intuitive meaning above in the equational theory. *Ax* says we can conclude assumptions are true, while *Coher* says $\mathrm{zero} = \mathrm{succ}\,V$ is an incoherent assumption. *IntroH* and *Lemm* introduce an assumption or eliminate an assumption by proving the required property as a side lemma. Quantifiers can be introduced when their property holds on a generic (co)variable (*IntroL* and *IntroR*), and eliminated by substituting any equal (co)value for the bound (co)variable (*SubstL* and *SubstR*). The notation $\Phi[V/x = V'/x]$ (and likewise $\Phi[E/\alpha = E'/\alpha]$) means to perform the substitution $[V/x]$ on the left-hand side of all equations in Φ and $[V'/x]$ on the right-hand sides.

The *Red* rule states that reduction steps of the operational semantics count as equalities, and the reflexivity, symmetry, transitivity, and compatibility rules are standard for an equational theory. The $\sigma\tilde{\mu}$ and $\sigma\mu$ rules establish a logical equivalence between equality of commands versus equality of (co)terms. They say that any two terms (dually coterms)

are equal when they form equal commands with a generic covariable (dually variable). These inference rules allow us to derive the extensionality η axioms for μ- and $\tilde{\mu}$-abstractions in the $\overline{\lambda}\mu\tilde{\mu}$-calculus for any term $\Gamma \vdash v : A$ or coterm $\Gamma \vdash e \div A$:

$$(\eta_\mu) \quad \Gamma \vdash \mu\alpha.\langle v \| \alpha \rangle = v : A \qquad (\eta_{\tilde{\mu}}) \quad \Gamma \vdash \tilde{\mu}x.\langle x \| e \rangle = e \div A$$

Similarly, the $\omega\!\rightarrow$ rule expresses a form of extensionality for functions in terms of call stacks—it is sufficient to test a productive property on a generic call stack—that can derive the more usual η axiom for functions in $\overline{\lambda}\mu\tilde{\mu}$:

$$(\eta_\rightarrow) \qquad \Gamma \vdash \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle = V : A \rightarrow B$$

The restriction of $\omega\!\rightarrow$ to productive properties corresponds to the usual restriction of the η axiom in the call-by-value λ-calculus: $\lambda x.M\ x \neq M$ when M is not equal to a value. This prevents equalities like $\lambda\_.\mu\_.\langle \text{zero} \| \alpha \rangle = \mu\_.\langle \text{zero} \| \alpha \rangle$, that are incoherent under call-by-value evaluation. For example, with $e_1 := \tilde{\mu}\_.\langle \text{succ zero} \| \alpha \rangle$, we have on the one hand $\langle \lambda\_.\mu\_.\langle \text{zero} \| \alpha \rangle \| e_1 \rangle \mapsto\!\!\!\!\rightarrow \langle \text{succ zero} \| \alpha \rangle$, whereas on the other hand $\langle \mu\_.\langle \text{zero} \| \alpha \rangle \| e_1 \rangle \mapsto\!\!\!\!\rightarrow \langle \text{zero} \| \alpha \rangle$.

The most interesting rules of the equation theory are the ones for (co)induction. The ωnat rule corresponds to the following weak induction property on the natural numbers, and ωstrm corresponds to the dual weak coinduction property:

$$(\omega\text{nat}) \quad \Psi(\text{zero}) \Rightarrow (\forall x{:}\text{nat}.\Psi(x) \Rightarrow \Psi(\text{succ}\, x)) \Rightarrow (\forall x{:}\text{nat}.\Psi(x))$$

$$(\omega\text{strm}) \quad (\forall \beta \div A.\Psi(\text{head}\, \beta)) \Rightarrow (\forall \alpha \div \text{strm}\, A.\Psi(\alpha) \Rightarrow \Psi(\text{tail}\, \alpha)) \Rightarrow (\forall \alpha \div \text{strm}\, A.\Psi(\alpha))$$

Notice that both of these rules for induction and coinduction are restricted to strict and productive properties, respectively. Reminiscent of [34], this restriction prevents the same kinds of incoherence issues as the above one for functions, which we will return to in more detail in section 6.3. For now, the primary fact about this (co)inductive equational theory is that it is *coherent* for either evaluation strategy.

Definition 1 (Coherence)**.** An equational theory for the (co)recursive abstract machine is *coherent* if $\alpha \div \text{nat} \vdash c_1 = c_2$ implies either:

1. $c_i \mapsto\!\!\!\!\rightarrow \langle \text{zero} \| \alpha \rangle$ for both $i = 1, 2$, or
2. $c_i \mapsto\!\!\!\!\rightarrow \langle \text{succ}\, V_i \| \alpha \rangle$ for some V_i and both $i = 1, 2$.

Theorem 7. *The equational theory in fig. 11 is coherent for both call-by-value and call-by-name evaluation.*

To better understand (co)induction, ωnat and ωstrm prove the following properties about (co)iteration:

$$(\delta_{\text{nat}}) \qquad \forall \alpha \div \text{nat}.\ \mathbf{iter} \begin{Bmatrix} \text{zero} \rightarrow \text{zero} \\ \text{succ} \rightarrow y.\ \text{succ}\, y \end{Bmatrix} \mathbf{with}\ \alpha = \alpha \div \text{nat}$$

$$(\delta_{\text{strm}}) \quad \forall x{:}\text{strm}\, A.\ \mathbf{coiter} \begin{Bmatrix} \text{head}\, \alpha \rightarrow \text{head}\, \alpha \\ \text{tail} \rightarrow \beta.\ \text{tail}\, \beta \end{Bmatrix} \mathbf{with}\ x = x : \text{strm}\, A$$

Intuitively, these are *deep* extensionality axioms for the recursor and corecursor. Any generic observer α cannot tell the difference if a natural number is first broken down and rebuilt from scratch from the base case (zero) up. Likewise, any generic stream x gives the same response when its projection is broken down and rebuilt from scratch from the base case (head α) up. Now, consider how to prove δstrm. After supposing a generic observation $\alpha \div \text{strm}\, A\ (\sigma\mu)$ inside the quantifier for x (*IntroL*, *SubstL*, *ReflR*), it suffices to show:

$$\alpha \div \text{strm}\, A \vdash \forall x{:}\text{strm}\, A.$$
$$\langle \mathbf{coiter} \{\text{head}\, \alpha \rightarrow \text{head}\, \alpha \mid \text{tail} \rightarrow \beta.\ \text{tail}\, \beta\}\ \mathbf{with}\ x \| \alpha \rangle = \langle x \| \alpha \rangle$$

Notice that this property is productive on α, so we may apply ωstrm. The remaining calculations for the two premises of ωstrm continue as follows, where we make use of the shorthand $cdeep := \{\text{head}\, \alpha \rightarrow \text{head}\, \alpha \mid \text{tail} \rightarrow \beta.\ \text{tail}\, \beta\}$:

- For head β/α, we have the calculation:

$$\langle \mathbf{coiter}\ cdeep\ \mathbf{with}\ x \| \text{head}\, \beta \rangle \mapsto \langle x \| \text{head}\, \beta \rangle \qquad (\beta_{\text{head}})$$

- For tail β/α, assume the coinductive hypothesis (*CIH*) $\forall x{:}\text{strm}\, A.\langle \mathbf{coiter}\ cdeep\ \mathbf{with}\ x \| \beta \rangle = \langle x \| \beta \rangle$, so we have the following calculation in call-by-value and -name:

$$\begin{aligned} &\langle \mathbf{coiter}\ cdeep\ \mathbf{with}\ x \| \text{tail}\, \beta \rangle \\ &\mapsto \langle \mu\beta.\langle x \| \text{tail}\, \beta \rangle \| \tilde{\mu}y.\langle \mathbf{coiter}\ cdeep\ \mathbf{with}\ y \| \beta \rangle \rangle && (\beta_{\text{tail}}) \\ &= \langle \mu\beta.\langle x \| \text{tail}\, \beta \rangle \| \tilde{\mu}y.\langle y \| \beta \rangle \rangle && (CIH[y/x]) \\ &= \langle \mu\beta.\langle x \| \text{tail}\, \beta \rangle \| \beta \rangle && (\eta_{\tilde{\mu}}) \\ &\mapsto \langle x \| \text{tail}\, \beta \rangle && (\mu) \end{aligned}$$

  Note that the generalization over x in the coinductive hypothesis is essential for instantiating x (via *SubstL*) with the bound y newly introduced by β_{tail} reduction.

Analogously to the "deep" extensionality properties δnat and δstrm of (co)iteration, we can also derive the following "shallow" extensionality properties ηnat and ηstrm for performing (co)case analysis on the structure of a numeric value or a stream projection:

$$(\eta_{\text{nat}}) \qquad \forall \alpha \div \text{nat}.\ \mathbf{case} \begin{Bmatrix} \text{zero} \rightarrow \text{zero} \\ \text{succ}\, y \rightarrow \text{succ}\, y \end{Bmatrix} \mathbf{with}\ \alpha = \alpha \div \text{nat}$$

$$(\eta_{\text{strm}}) \quad \forall x{:}\text{strm}\, A.\ \mathbf{cocase} \begin{Bmatrix} \text{head}\, \alpha \rightarrow \text{head}\, \alpha \\ \text{tail}\, \beta \rightarrow \text{tail}\, \beta \end{Bmatrix} \mathbf{with}\ x = x : \text{strm}\, A$$

6.2 Equational theory of the $\lambda\mu\tilde{\mu}$-calculus

Instead of reasoning in the lower-level language of the abstract machine, we can also reason directly in the higher-level language of $\lambda\mu\tilde{\mu}$ using the equational theory given in fig. 12. This theory is closer to the λ-calculus. For example the familiar η axiom can be proved from $\omega\!\rightarrow$:

$$(\eta_\rightarrow) \qquad \Gamma \vdash \lambda x.V\ x = V : A \rightarrow B$$

The two theories correspond to one another, by pointwise extending the translation of $\lambda\mu\tilde{\mu}$ over Φ and Γ.

Theorem 8 (Soundness & Completeness)**.** $\Gamma \vdash \Phi$ *iff* $[\![\Gamma]\!] \vdash [\![\Phi]\!]$.

For example, we can prove $evens\ alt = repeat\ \text{zero} : \text{strm}\, A$. By applying $\sigma\mu$, it suffices to show that

$$\alpha \div \text{strm nat} \vdash \langle evens\ alt \| \alpha \rangle = \langle repeat\ \text{zero} \| \alpha \rangle$$

Both sides can reduce to a productive property on α (that is to say, $\langle evens\ alt \| \alpha \rangle \mapsto\!\!\!\!\rightarrow \langle evens_{alt} \| \alpha \rangle$ for a closed value

Properties (Φ), strict ($\Psi(x)$) and productive ($\Psi(\alpha)$) properties, hypothesis (θ), environments (Γ), and judgments:

$$\Phi ::= J = J' \mid M = M' : A \mid Q = Q' \div A \mid \forall x{:}A.\Phi \mid \forall \alpha \div A.\Phi \mid \Phi' \Rightarrow \Phi$$
$$\Psi(x) ::= E[x] = E'[x] \mid E[x] = E'[x] : A \mid \forall y{:}A.\Psi(x) \mid \forall \alpha \div A.\Psi(x)$$
$$\Psi(\alpha) ::= \langle V \| \alpha \rangle = \langle V' \| \alpha \rangle \mid \forall x{:}A.\Psi(\alpha) \mid \forall \beta \div A.\Psi(\alpha)$$
$$\Gamma ::= \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A \mid \Gamma, \Phi \qquad \mathit{Judge} ::= \Gamma \vdash \Phi$$

Equality of computation (plus rules for reflexivity, symmetry, transitivity, and compatibility of equality):

$$\frac{\Gamma \vdash J \;\; J \to J'}{\Gamma \vdash J = J'} Red \qquad \frac{\Gamma \vdash M : A \;\; M \to M'}{\Gamma \vdash M = M' : A} Red \qquad \frac{\Gamma \vdash Q \div A \;\; Q \to Q'}{\Gamma \vdash Q = Q' \div A} Red$$

Induction, coinduction, and extensionality principles:

$$\frac{\Gamma, \alpha \div A \vdash \langle M \| \alpha \rangle = \langle M' \| \alpha \rangle}{\Gamma \vdash M = M' : A}\ \sigma\mu \qquad \frac{\Gamma, x{:}A \vdash \langle x \| Q \rangle = \langle x \| Q' \rangle}{\Gamma \vdash Q = Q' \div A}\ \sigma\tilde{\mu}$$

$$\frac{\Gamma, x : A \vdash V\ x = V'\ x : B}{\Gamma \vdash V = V' : A \to B}\ \omega{\to}$$

$$\frac{\Gamma \vdash \Psi(\mathrm{zero}/x) \quad \Gamma, x : \mathrm{nat}, \Psi(x) \vdash \Psi(\mathrm{succ}\, x/x)}{\Gamma, x : \mathrm{nat} \vdash \Psi(x)}\ \omega\mathrm{nat}$$

$$\frac{\Gamma, \beta \div A \vdash \Psi(\langle \mathrm{head}\, M \| \beta \rangle / \langle M \| \alpha \rangle) \quad \Gamma, \alpha \div \mathrm{strm}\, A, \Psi(\alpha) \vdash \Psi(\langle \mathrm{tail}\, M \| \alpha \rangle / \langle M \| \alpha \rangle)}{\Gamma, \alpha \div \mathrm{strm}\, A \vdash \Psi(\alpha)}\ \omega\mathrm{strm}$$

Plus the same rules of properties as fig. 11.

Figure 12. Equational theory of (co)induction in $\lambda\mu\tilde{\mu}$.

$$\frac{\Gamma \vdash \Phi[\mathrm{zero}/x] \quad \Gamma, x : \mathrm{nat}, \Phi \vdash \Phi[\mathrm{succ}\, x/x]}{\Gamma, x : \mathrm{nat} \vdash \Phi}\ \sigma\mathrm{nat}$$

$$\frac{\Gamma, \beta \div A \vdash \Phi[\mathrm{head}\, \beta/\alpha] \quad \Gamma, \alpha \div \mathrm{strm}\, A, \Phi \vdash \Phi[\mathrm{tail}\, \alpha/\alpha]}{\Gamma, \alpha \div \mathrm{strm}\, A \vdash \Phi}\ \sigma\mathrm{strm}$$

Figure 13. Strong (co)induction in the abstract machine.

evens_{alt} and likewise $\langle \mathit{repeat}\ \ \mathrm{zero} \| \alpha \rangle \mapsto \langle \mathit{repeat}_{\mathrm{zero}} \| \alpha \rangle$), so we can apply ωnat to this reduced equality. The remaining calculations for the two premises of ωnat are as follows:

- For head β/α, we have the calculation:

$$\begin{aligned} \langle \mathit{evens}_{alt} \| \mathrm{head}\, \beta \rangle &\mapsto\!\!\!\!\to \langle \mathit{alt} \| \mathrm{head}\, \beta \rangle \\ &\mapsto \langle \mathrm{zero} \| \beta \rangle \leftarrow \langle \mathit{repeat}_{\mathrm{zero}} \| \mathrm{head}\, \beta \rangle \end{aligned}$$

- For tail β/α, assume the coinductive hypothesis (CIH) $\langle \mathit{evens}_{alt} \| \beta \rangle = \langle \mathit{repeat}_{\mathrm{zero}} \| \beta \rangle$, so we have the following calculation in both call-by-value and call-by-name:

$$\begin{aligned} \langle \mathit{evens}_{alt} \| \mathrm{tail}\, \beta \rangle &\mapsto\!\!\!\!\to \langle \mathit{evens}_{\mathrm{tail}(\mathrm{tail}\ alt)} \| \beta \rangle && (\mathit{evens}) \\ &= \langle \mathit{evens}_{alt} \| \beta \rangle && (\mathit{alt}) \\ &= \langle \mathit{repeat}_{\mathrm{zero}} \| \beta \rangle && (CIH) \\ &\leftarrow\!\!\!\!\leftarrow \langle \mathit{repeat}_{\mathrm{zero}} \| \mathrm{tail}\, \beta \rangle && (\mathit{repeat}) \end{aligned}$$

6.3 Strong (co)induction

The form of (co)induction we have considered so far is weak, and necessarily so since in some cases a stronger rule would be incoherent. But this raises the question: when is strong induction and coinduction safe? The rules for strong (co)induction (σnat and σstrm) are given for the abstract machine (fig. 13) and the $\lambda\mu\tilde{\mu}$-calculus (fig. 14), which generalize the weaker rules (ωnat and ωstrm) to any property, not just strict or productive ones. As it turns out, unlike the weaker forms, strong induction and coinduction are only coherent under certain evaluation strategies.

$$\frac{\Gamma \vdash \Phi[\mathrm{zero}/x] \quad \Gamma, x : \mathrm{nat}, \Phi \vdash \Phi[\mathrm{succ}\, x/x]}{\Gamma, x : \mathrm{nat} \vdash \Phi}\ \sigma\mathrm{nat}$$

$$\frac{\Gamma, \beta \div A \vdash \Phi[\langle \mathrm{head}\, M \| \beta \rangle / \langle M \| \alpha \rangle]}{\Gamma, \alpha \div \mathrm{strm}\, A, \Phi \vdash \Phi[\langle \mathrm{tail}\, M \| \alpha \rangle / \langle M \| \alpha \rangle]}\ \sigma\mathrm{strm}$$

Figure 14. Strong (co)induction in $\lambda\mu\tilde{\mu}$.

Theorem 9. *1. The equational theory in fig. 11 extended with* σnat *is coherent for call-by-value evaluation.*
2. The equational theory in fig. 11 extended with σstrm *is coherent for call-by-name evaluation.*

To see where the extensions could be incoherent, notice that the σnat rule proves this (non-x-strict) property:

$$\alpha \div \mathrm{nat} \vdash \forall x{:}\mathrm{nat}.\ \langle x \| \mathbf{case}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\ \_ \to \mathrm{zero}\}\ \mathbf{with}\ \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle$$

Call-by-name allows for the value $\mu\_.\langle \mathrm{succ}\ \mathrm{zero} \| \alpha \rangle$ to be substituted for x in this equation, leading to the incoherent equality $\alpha \div \mathrm{nat} \vdash \langle \mathrm{succ}\ \mathrm{zero} \| \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle$ contradicting theorem 7. Dually, σstrm proves (the non-α-productive):

$$\alpha \div \mathrm{nat} \vdash \forall \beta \div \mathrm{strm}\, A.\ \langle \mathbf{cocase}\{\mathrm{head}\ \_ \to \alpha \mid \mathrm{tail}\ \_ \to \alpha\}\ \mathbf{with}\ \mathrm{zero} \| \beta \rangle = \langle \mathrm{zero} \| \alpha \rangle$$

and yet call-by-value allows the covalue $\tilde{\mu}\_.\langle \mathrm{succ}\ \mathrm{zero} \| \alpha \rangle$ to be substituted for β, leading to the same incoherent equality contradicting theorem 7.

In contrast to ωnat, σnat is general enough to let us derive the traditional notion of strong induction on the natural numbers. First, define the ordering relation on numbers in terms of the following equality:

$$M \le N : \mathrm{nat} := \mathit{minus}\ M\ N = \mathrm{zero} : \mathrm{nat}$$

We then write $\forall x{\le}M{:}\mathrm{nat}.\Phi$ as shorthand for the property $\forall x{:}\mathrm{nat}.\ x \le M : \mathrm{nat} \Rightarrow \Phi$. Applying σnat to this gives:

$$\frac{\Gamma \vdash \forall y{\le}\mathrm{zero}{:}\mathrm{nat}.\Phi \quad \Gamma, x{:}\,\mathrm{nat}, \forall y{\le}x{:}\mathrm{nat}.\Phi \vdash \forall y{\le}\,\mathrm{succ}\, x{:}\mathrm{nat}.\Phi}{\Gamma, x{:}\,\mathrm{nat} \vdash \forall y{\le}x{:}\mathrm{nat}.\Phi}\ \sigma\mathrm{nat}$$

Since $\forall y{\le}\mathrm{zero}{:}\mathrm{nat}.\ y = \mathrm{zero} : \mathrm{nat}$ is derivable (by definition of $\le$) as well as $\forall x{:}\mathrm{nat}.\ x \le x : \mathrm{nat}$ (by induction with σnat), we can specialize the above application to derive the following simpler statement of strong induction on the naturals:

$$\frac{\Gamma \vdash \Phi[\mathrm{zero}/x] \quad \Gamma, x{:}\,\mathrm{nat}, \forall y{\le}x{:}\mathrm{nat}.\Phi \vdash \Phi[\mathrm{succ}\, x/x]}{\Gamma, x{:}\,\mathrm{nat} \vdash \Phi}$$

As with induction on the natural numbers, we can derive the dual notion of strong coinduction on infinite streams.

First, define the ordering relation on stream *projections* as:

$$Q \le R \div \mathrm{strm}\, A := \mathit{depth}\, Q \le \mathit{depth}\, R : \mathrm{nat}$$

where $\mathit{depth}\, Q$ computes the depth of any stream projection Q, effectively converting $\mathrm{tail}^n(\mathrm{head}\, \alpha)$ to succ^n zero:

$$\mathit{depth}\, Q := \mu\alpha.\langle \mathit{count} \| \mathbf{coiter}\, Q\, \mathbf{as} \{\mathrm{head}\, \_ \longrightarrow \mathrm{head}\, \alpha \mid \mathrm{tail} \longrightarrow \beta.\, \mathrm{tail}\, \beta\}\rangle$$

As before, we write the quantification $\forall \alpha \le Q \div \mathrm{strm}\, A.\Phi$ as shorthand for $\forall \alpha \div \mathrm{strm}\, A.\ \alpha \le Q \div \mathrm{strm}\, A \Rightarrow \Phi$. Applying σstrm to this property gives:

$$\frac{\Gamma, \delta \div A \vdash \forall \beta \le \mathrm{head}\, \delta \div \mathrm{strm}\, A.\Phi \qquad \Gamma, \alpha \div \mathrm{strm}\, A, \forall \beta \le \alpha \div \mathrm{strm}\, A.\Phi \vdash \forall \beta \le \mathrm{tail}\, \alpha \div \mathrm{strm}\, A.\Phi}{\Gamma, \alpha \div \mathrm{strm}\, A \vdash \forall \beta \le \alpha \div \mathrm{strm}\, A.\Phi}$$

Analogous to strong induction on the naturals, we can use this application to derive the following simpler statement of strong coinduction on streams:

$$\frac{\Gamma, \delta \div A \vdash \Phi[\mathrm{head}\, \delta/\alpha] \qquad \Gamma, \alpha \div \mathrm{strm}\, A, \forall \beta \le \alpha \div \mathrm{strm}\, A.\Phi[\beta/\alpha] \vdash \Phi[\mathrm{tail}\, \alpha/\alpha]}{\Gamma, \alpha \div \mathrm{strm}\, A \vdash \Phi}$$

From this, We can derive the following special case of strong coinduction, where we must show the first $n+1$ base cases (for $\mathrm{head}\, \beta$, $\mathrm{tail}(\mathrm{head}\, \beta)$, $\ldots \mathrm{tail}^n(\mathrm{head}\, \beta)$) directly, and then take the $n+1^{th}$ tail projection in the coinductive case:

$$\frac{\Gamma, \beta \div A \vdash \Phi[\mathrm{head}\, \beta/\alpha] \ldots \Gamma, \beta \div A \vdash \Phi[\mathrm{tail}^n(\mathrm{head}\, \beta)/\alpha] \qquad \Gamma, \alpha \div \mathrm{strm}\, A, \Phi \vdash \Phi[\mathrm{tail}^{n+1}\, \alpha/\alpha]}{\Gamma, \alpha \div \mathrm{strm}\, A \vdash \Phi}$$

For example, the corresponding principle in $\lambda\mu\tilde{\mu}$ can prove that $\forall s{:}\mathrm{strm}\, A.\ \mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s) = s : \mathrm{strm}\, A$ by stepping by 2. It suffices to show (via *IntroR*, *SubstR*, $\sigma\mu$) that

$$\alpha \div \mathrm{strm}\, A \vdash \forall s{:}\mathrm{strm}\, A.\langle \mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s) \| \alpha\rangle = \langle s \| \alpha\rangle$$

which follows from the strong coinduction principle above, derived from σstrm:

- For the first base case with $\langle \mathrm{head}\, M \| \beta\rangle / \langle M \| \alpha\rangle$, we have the following calculation from the definition of *merge* and *evens* for a generic s (*IntroL*):

$$\begin{aligned} \langle \mathrm{head}(\mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s)) \| \beta\rangle &= \langle \mathrm{head}(\mathit{evens}\, s) \| \beta\rangle && (\mathit{merge}) \\ &= \langle \mathrm{head}\, s \| \beta\rangle && (\mathit{evens}) \end{aligned}$$

- For the second base case where $\langle \mathrm{head}(\mathrm{tail}\, M) \| \beta\rangle / \langle M \| \alpha\rangle$, we have the following calculation for a generic s (*IntroL*):

$$\begin{aligned} &\langle \mathrm{head}(\mathrm{tail}(\mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s))) \| \beta\rangle \\ &= \langle \mathrm{head}(\mathit{merge}\, (\mathit{odds}\, s)\, (\mathrm{tail}(\mathit{evens}\, s))) \| \beta\rangle && (\mathit{merge}) \\ &= \langle \mathrm{head}(\mathit{odds}\, s) \| \beta\rangle && (\mathit{merge}) \\ &= \langle \mathrm{head}(\mathrm{tail}\, s) \| \beta\rangle && (\mathit{odds}) \end{aligned}$$

- For the coinductive step, we may use the hypothesis

  $\forall s{:}\mathrm{strm}\, A.\langle \mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s) \| \alpha\rangle = \langle s \| \alpha\rangle$

  to show the same property after 2 tail projections on a generic $s : \mathrm{strm}\, A$,

$$\langle \mathrm{tail}(\mathrm{tail}(\mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s))) \| \alpha\rangle = \langle \mathrm{tail}(\mathrm{tail}\, s) \| \alpha\rangle$$

  which proceeds by the following calculation:

$$\begin{aligned} &\langle \mathrm{tail}(\mathrm{tail}(\mathit{merge}\, (\mathit{evens}\, s)\, (\mathit{odds}\, s))) \| \alpha\rangle \\ &= \langle \mathrm{tail}(\mathit{merge}\, (\mathit{odds}\, s)\, (\mathrm{tail}(\mathit{evens}\, s))) \| \alpha\rangle && (\mathit{merge}) \\ &= \langle \mathit{merge}\, (\mathrm{tail}(\mathit{evens}\, s))\, (\mathrm{tail}(\mathit{odds}\, s)) \| \alpha\rangle && (\mathit{merge}) \\ &= \langle \mathit{merge}\, (\mathit{evens}(\mathrm{tail}(\mathrm{tail}\, s)))\, (\mathit{odds}(\mathrm{tail}(\mathrm{tail}\, s))) \| \alpha\rangle && (\mathit{evens}, \mathit{odds}) \\ &= \langle \mathrm{tail}(\mathrm{tail}\, s) \| \alpha\rangle && (\mathit{CIH}[\mathrm{tail}(\mathrm{tail}\, s)/s]) \end{aligned}$$

The traditional principle of bisimulation on streams is also subsumed by the strong coinduction rule σstrm, written as the following property Bisim^A_Φ, where the simulation relation is represented by Φ referring to two streams named s and s':

$$\begin{aligned} \mathit{Bisim}^A_\Phi := &(\forall s, s' {:} \mathrm{strm}\, A.\Phi \Rightarrow \mathrm{head}\, s = \mathrm{head}\, s' : A) \\ &\Rightarrow (\forall s, s' {:} \mathrm{strm}\, A.\Phi \Rightarrow \Phi[\mathrm{tail}\, s/s, \mathrm{tail}\, s'/s']) \\ &\Rightarrow (\forall s, s' {:} \mathrm{strm}\, A.\Phi \Rightarrow s = s' : \mathrm{strm}\, A) \end{aligned}$$

Where we use the shorthand $\forall s, s' {:} \mathrm{strm}\, A.\Phi$ to stand for multiple quantifications of the same type $\forall s {:} \mathrm{strm}\, A.\forall s' {:} \mathrm{strm}\, A.\Phi$. The two assumptions confirm that Φ is a valid bisimulation relation: Φ only relates streams with equal heads, and is closed under tail projection. Bisim^A_Φ is derivable from strong coinduction in the $\lambda\mu\tilde{\mu}$ equational theory.

7 Related Work

Our focus here primarily on theoretical foundations, rather than applications, of corecursion and coinduction. In particular, this work serves as a foundation for coinduction based on codata as found in practical languages like Agda, which supports copatterns [2] on coinductively-defined types. For an introduction to related ideas on computational corecursion based on more practical examples in programming, see [15].

The corecursor presented here is a computational interpretation of the categorical model of corecursion in a coalgebra [17]. A (weak) coalgebra for a functor F is defined by a morphism $\alpha : A \longrightarrow F(A)$ for some A. A *corecursive coalgebra* extends this idea, by commuting with other morphisms of the form $X \longrightarrow F(A+X)$. Intuitively, the option $A+X$ in the result corresponds to the multiple outputs in the corecursor presented here. Note that we interpret the corecursive $A + X$ as multiple continuations, rather than a sum type, which maps more closely to lower-level implementations. This gives improved efficiency of corecursion over coiteration with sum types in call-by-value (recall section 4.2).

The coiterator, which we define as the restriction of the corecursor to never short-cut corecursion, corresponds exactly to the Harper's [23] `strgen`. In this sense, the corecursor is a conservative extension of the purely functional coiterator. Coiteration with control operators is considered in [5], which gives a call-by-name CPS translation for a stream coiterator and primitive operation for *scons*. Here, the use of an abstract machine serves a similar role as CPS—making explicit information- and control-flow—but allows us to use the same translation for both call-by-value and -name. An alternative approach to (co)recursive combinators is *sized types* [1, 24],

which give the programmer control over recursion while still ensuring termination, and have been used for both purely functional [3] and classical [13] coinductive types.

Our investigation on evaluation strategy showed the (dual) impact of call-by-value versus call-by-name evaluation [7, 38] on the efficiency and strength of (co)induction. We show how call-by-name evaluation is necessary for strong coinduction in the presence of effects, dual to the fact that call-by-value evaluation is necessary for strong induction. We believe that this connection also holds for efficiency and optimization, as well. For example, previous work has showed that for function types—which are a form of non-corecursive codata type—optimizations of function arity are improved by call-by-name evaluation [12, 14]. Though counterintuitive, the improvements to arity are owed to the stronger equational theory (specifically, the η law of the λ-calculus) which is possible by call-by-name evaluation. We speculate there may be similar optimizations for corecursion on coinductive types made possible by a strong call-by-name theory.

In contrast to having a monolithic evaluation strategy, another approach is use a hybrid evaluation strategy as done by call-by-push-value [29] or polarized languages [31, 39]. With a hybrid approach, we could define one language which has both efficient (co)recursors and strong (co)inductive principles. Polarity also allows for incorporating other evaluation strategies, such as call-by-need which shares the work of computations, while keeping the strong equational theory [10, 30]. We leave to future work an investigation on the polarized version of the (co)inductive calculus and abstract machine presented here.

Our notions of strong (co)induction—which are founded on dual forms of data flow and control flow—allow for local reasoning about valid applications of the (co)inductive hypothesis, which leads to a compositional development of (co)inductive proofs. Similarly, Paco [25] aims to aid the development of coinductive proofs through both compositionality (local, not global, correctness criteria) and incrementality (new knowledge may be accumulated as the proof is developed). We showed how the strong version our equational theory encompasses well-known principles of strong induction and bisimulation of corecursive processes. We conjecture that Paco's coinductive principles can also be encoded as an application of strong coinduction—giving a computational model for its proofs—where accumulated knowledge may be represented as the accumulator of a corecursive process.

8 Conclusion

This paper defines a language for providing a computational foundation of (co)recursive programming and (co)inductive reasoning. The impact of evaluation strategy is also illustrated, where call-by-value and -name have (opposite) advantages for the efficiency of (co)recursion and strength of (co)induction. These (co)recursion schemes are captured by (co)data types whose duality is made apparent in their implementation in an abstract machine. The (co)inductive principles are derived from the definition of types in terms of *construction* or *destruction*, using *control flow* instead of bisimulation to guide the coinductive hypothesis. In the end, the logical dualities in computation—between data and codata; information flow and control flow—provide a unified framework for using and reasoning with (co)inductive types.

Acknowledgments

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

References

[1] Andreas Abel. 2006. *A Polymorphic Lambda Calculus with Sized Higher-Order Types.* Ph.D. Thesis. Ludwig-Maximilians-Universität München.

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

[3] Andreas M. Abel and Brigitte Pientka. 2013. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 185–196. https://doi.org/10.1145/2500365.2500591

[4] Zena M. Ariola, Aaron Bohannon, and Amr Sabry. 2009. Sequent Calculi and Abstract Machines. *ACM Transactions on Programming Languages and Systems* 31, 4, Article 13 (May 2009), 48 pages. https://doi.org/10.1145/1516507.1516508

[5] Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Portland, Oregon) *(PEPM '02)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/503032.503043

[6] Alberto Carraro, Thomas Ehrhard, and Antonino Salibra. 2012. The Stack Calculus. In *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications* (Rio de Janeiro, Brazil) *(LSFA 2012)*. 93–108. https://doi.org/10.4204/EPTCS.113.10

[7] Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

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

[9] Paul Downen and Zena M. Ariola. 2014. The Duality of Construction. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014.* Lecture Notes in Computer Science, Vol. 8410. Springer Berlin Heidelberg, 249–269. https://doi.org/10.1007/978-3-642-54833-8_14

[10] Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK (LIPIcs, Vol. 119)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:23. https://doi.org/10.4230/LIPIcs.CSL.2018.21

[11] Paul Downen and Zena M. Ariola. 2018. A Tutorial on Computational Classical Logic and the Sequent Calculus. *Journal of Functional Programming* 28 (2018), e3. https://doi.org/10.1017/S0956796818000023

[12] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds Are Calling Conventions. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 104 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408986

[13] Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. 2015. Structures for Structural Recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP '15)*. ACM, New York, NY, USA, 127–139. https://doi.org/10.1145/2784731.2784762

[14] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Making a faster Curry with extensional types. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. ACM, 58–70. https://doi.org/10.1145/3331545.3342594

[15] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *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* (Prague, Czech Republic) *(ESOP '19)*. Springer International Publishing, Cham, 119–146. https://doi.org/10.1007/978-3-030-17184-1_5

[16] Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39, 1 (1935), 176–210. https://doi.org/10.1007/BF01201353

[17] Herman Geuvers. 1992. Inductive and Coinductive types with Iteration and Recursion. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Bastad*. 193–217.

[18] Jeremy Gibbons and Graham Hutton. 2005. Proof Methods for Corecursive Programs. *Fundamenta Informaticae* 66 (04 2005), 353–366.

[19] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

[20] Kurt Gödel. 1980. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic* 9, 2 (1980), 133–142. https://doi.org/10.1007/BF00247744

[21] Mike Gordon. 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

[22] Tatsuya Hagino. 1987. A Typed Lambda Calculus With Categorical Type Constructors. In *Category Theory and Computer Science* (Edinburgh, U.K.). Springer Berlin Heidelberg, Berlin, Heidelberg, 140–157. https://doi.org/10.1007/3-540-18508-9_24

[23] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, USA.

[24] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

[25] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

[26] Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. 2016. First Class Call Stacks: Exploring Head Reduction. In *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015 (EPTCS, Vol. 212)*. 18–35. https://doi.org/10.4204/EPTCS.212

[27] Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. 2017. Call-by-name Extensionality and Confluence. *Journal of Functional Programming* 27 (2017), e12. https://doi.org/10.1017/S095679681700003X

[28] Jean-Louis Krivine. 2007. A Call-By-Name Lambda-Calculus Machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

[29] Paul Blain Levy. 2001. *Call-By-Push-Value*. Ph.D. Dissertation. Queen Mary and Westfield College, University of London.

[30] Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *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, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 235–262. https://doi.org/10.1007/978-3-030-17184-1_9

[31] Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. Ph.D. Dissertation. Université Paris Diderot.

[32] Koji Nakazawa and Tomoharu Nagai. 2014. Reduction System for Extensional Lambda-mu Calculus.. In *Rewriting and Typed Lambda Calculi (LNCS)*. 349–363.

[33] Michel Parigot. 1992. $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *Logic Programming and Automated Reasoning: International Conference* (St. Petersburg, Russia) *(LPAR '92)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 190–201. https://doi.org/10.1007/BFb0013061

[34] Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*. Reykjavik, Iceland, 12. https://doi.org/10.1109/LICS.2017.8005113

[35] Crole Roy. 2003. Coinduction and bisimilarity. In *Oregon Programming Languages Summer School (OPLSS)*.

[36] Jan Rutten. 2019. The Method of Coalgebra: Exercises in coinduction. (Feb. 2019), 1–261.

[37] Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA.

[38] Philip Wadler. 2003. Call-By-Value is Dual to Call-By-Name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden). ACM, New York, NY, USA, 189–201. https://doi.org/10.1145/944705.944723

[39] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph.D. Dissertation. Carnegie Mellon University.