*JFP* **33**, e4, 63 pages, 2023. 

doi:10.1017/S0956796822000168

# *Classical (co)recursion: Mechanics*

PAUL DOWNEN

*Department of Computer and Information Science, University of Oregon, Eugene, OR, USA*
*Department of Computer Science, University of Massachusetts, Lowell, MA, USA*
(*e-mails:* pdownen@cs.uoregon.edu, paul_downen@uml.edu)

ZENA M. ARIOLA

*Department of Computer and Information Science, University of Oregon, Eugene, OR, USA*
(*e-mail:* ariola@cs.uoregon.edu)

## Abstract

Recursion is a mature, well-understood topic in the theory and practice of programming. Yet its dual, corecursion is underappreciated and still seen as exotic. We aim to put them both on equal footing by giving a foundation for primitive corecursion based on computation, giving a terminating calculus analogous to the original computational foundation of recursion. We show how the implementation details in an abstract machine strengthens their connection, syntactically deriving corecursion from recursion via logical duality. We also observe the impact of evaluation strategy on the computational complexity of primitive (co)recursive combinators: call-by-name allows for more efficient recursion, but call-by-value allows for more efficient corecursion.

## 1 Introduction

Primitive recursion has a solid foundation in a variety of different fields. In the categorical setting, it can be seen in the structures of algebras. In the logical setting, it corresponds to proofs by induction. And in the computational setting, it can be phrased in terms of languages and type theories with terminating loops, like Gödel's System T (Gödel, 1980). The latter viewpoint of computation reveals a fine-grained lens with which we can study the subtle impact of the primitive combinators that capture different forms of recursion. For example, the recursive combinators given by Mendler (1987, 1988) yield a computational complexity for certain programs when compared to encodings in System F (Böhm & Berarducci, 1985; Girard *et al.*, 1989). Recursive combinators have desirable properties—like the fact that they always terminate—which make them useful for the design of well-behaved programs (Meijer *et al.*, 1991; Gibbons, 2003), also for optimizations made possible by applying those properties and theorems (Malcolm, 1990).

The current treatment of the dual of primitive recursion—*primitive corecursion*—is not so fortunate. Being the much less understood of the two, corecursion is usually only viewed in light of this duality. Consequently, corecursion tends to be relegated to a notion of coalgebras (Rutten, 2019), because only the language of category theory speaks clearly

enough about their duality. This can be seen in the study of corecursion schemes, where coalgebraic "anamorphisms" (Meijer *et al.*, 1991) and "apomorphisms" (Vos, 1995; Vene & Uustalu, 1998) are the dual counterparts to algebraic "catamorphisms" (Meertens, 1987; Meijer *et al.*, 1991; Hinze *et al.*, 2013) and "paramorphisms" (Meertens, 1992). Yet the logical and computational status of corecursion is not so clear. For example, the introduction of stream objects is sometimes described as the "dual" to the elimination of natural numbers (Crole, 2003; Sangiorgi, 2011), but how is this so?

The goal of this paper is to provide a purely computational and logical foundation for primitive corecursion based on classical logic. Specifically, we will express different principles of corecursion in a small core calculus, analogous to the canonical computational presentation of recursion (Gödel, 1980). Much of the early pioneering work in this area was inspired directly by the duality inherent in (co)algebras and category theory (Hagino, 1987; Cockett & Spencer, 1995). In contrast, we derive the symmetry between recursion and corecursion through the mechanics of programming language implementations, formalized in terms of an abstract machine. This symmetry is encapsulated by the syntactic and semantic duality (Downen *et al.*, 2015) between *data types*—defined by the *structure* of objects—and *codata types*—defined by the *behavior* of objects.

We begin in Section 2 with a review of the formalization of primitive recursion in terms of a foundational calculus: System T (Gödel, 1980). We point out the impact of evaluation strategy on different primitive recursion combinators, namely the *recursor* and the *iterator*:

- In call-by-value, the recursor is just as (in)efficient as the iterator.
- In call-by-name, the recursor may end early; an asymptotic complexity improvement.

Section 3 presents an abstract machine for both call-by-value and call-by-name evaluation and unifies both into a single presentation (Ariola *et al.*, 2009; Downen & Ariola, 2018*b*). The lower-level nature of the abstract machine explicitly expresses how the recursor of inductive types, like numbers, accumulates a continuation during evaluation, maintaining the progress of recursion. This is implicit in the operational model of System T. The machine is shown correct, in the sense that a well-typed program will always terminate and produce an observable value (Theorem 3.2), which in our case is a number.

Section 4 continues by extending the abstract machine with the primitive corecursor for streams. The novelty is that this machine is derived by applying syntactic duality, corresponding to de Morgan duality, to the machine with recursion on natural numbers, leading us to a classical corecursive combinator with multiple outputs modeled as multiple continuations. This corecursor is *classical* in the sense that it abstracts binds first-class continuations using control effects in a way that corresponds to multiple conclusions in classical logic (à la $\lambda\mu$-calculus (Parigot, 1992) and the classical sequent calculus (Curien & Herbelin, 2000)). From de Morgan duality in the machine, we can see that the corecursor relies on a value accumulator; this is logically dual to the recursor's return continuation. Like recursion versus iteration, in Section 5 we compare corecursion versus coiteration: corecursion can be more efficient than coiteration by letting corecursive processes stop early. Since corecursion is dual to recursion, and call-by-value is dual to call-by-name (Curien & Herbelin, 2000; Wadler, 2003), this improvement in algorithmic complexity is only seen in call-by-value corecursion. Namely:

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

- In call-by-name, the corecursor is just as (in)efficient as the coiterator.
- In call-by-value, the corecursor may end early; an asymptotic complexity improvement.

Yet, even though we have added infinite streams, we don't want to ruin System T's desirable properties. So in Section 6, we give an interpretation of the type system which extends previous models of finite types (Downen *et al.*, 2020, 2019) with the (co)recursive types of numbers and streams. The novel key step in reasoning about (co)recursive types is in reconciling two well-known fixed point constructions—Kleene's and Knaster-Tarski's—which is non-trivial for classical programs with control effects. This lets us show that, even with infinite streams, our abstract machine is terminating and type safe (Theorem 6.14).

In summary, we make the following contributions to the understanding of (co)recursion in programs:

- We present a typed *uniform abstract machine*, with both call-by-name and call-by-value instances, that can represent functional programs that operate on both an inductive type (Figs. 4 and 6) and a coinductive type (Figs. 7 to 8).
- In the context of the abstract machine (Section 4.2), we show how the primitive corecursion combinator for a coinductive type can be formally derived from the usual recursor of an inductive type using the notion of de Morgan duality in classical logic. This derivation is made possible by a computational interpretation of involutive duality inherent to classical logic (Fig. 9 and Theorem 4.2).
- We informally analyze the impact of call-by-value versus call-by-name evaluation on the algorithmic complexity of different primitive combinators for (co)inductive types. Dual to the fact that primitive recursion is more efficient than iteration (only) in call-name (Section 3.5), we show that classical corecursion is more efficient than intuitionistic coiteration (only) in call-by-value (Section 5).
- The combination of primitive recursion and corecursion is shown to be type safe and terminating in the abstract machine (Theorem 6.14), even though it can express infinite streams that do not end. This is proved using an extension of bi-orthogonal logical relations built on a semantic notion of subtyping (Section 6).

Proofs to all theorems that follow are given in the appendix. For further examples of how to apply classical corecursion in real-world programming languages, and for an illustration of how classicality adds expressive power to corecursion, see the companion paper (Downen & Ariola, 2021).

## 2 Recursion on natural numbers: System T

We start with Gödel's System T (Gödel, 1980), a core calculus which allows us to define functions by structural recursion. Its syntax is given in Fig. 1. It is a canonical extension of the simply typed $\lambda$-calculus, whose focus is on functions of type $A \to B$, with ways to construct natural numbers of type Nat. The Nat type comes equipped with two constructors zero and succ, and a built-in recursor, which we write as $\mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero} \to N \mid \mathrm{succ}\, x \to y.N'\}$. This **rec**-expression analyzes $M$ to determine if it has the shape zero or succ $x$, and

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$\mathit{Type} \ni A, B ::= A \to B \mid \mathrm{Nat}$$
$$\mathit{Term} \ni M, N ::= x \mid \lambda x.M \mid M\ N \mid \mathrm{zero} \mid \mathrm{succ}\ M \mid \mathbf{rec}\ M\ \mathbf{as}\ \{\mathrm{zero} \to N \mid \mathrm{succ}\ x \to y.M\}$$

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

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var}$$
$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \to I \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \to E$$
$$\frac{}{\Gamma \vdash \mathrm{zero} : \mathrm{Nat}}\ \mathrm{Nat}I_{\mathrm{zero}} \qquad \frac{\Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash \mathrm{succ}\ M : \mathrm{Nat}}\ \mathrm{Nat}I_{\mathrm{succ}}$$
$$\frac{\Gamma \vdash M : \mathrm{Nat} \quad \Gamma \vdash N : A \quad \Gamma, x : \mathrm{Nat}, y : A \vdash N' : A}{\Gamma \vdash \mathbf{rec}\ M\ \mathbf{as}\ \{\mathrm{zero} \to N \mid \mathrm{succ}\ x \to y.N'\} : A}\ \mathrm{Nat}E$$

Fig. 2. Type system of System T.

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

$$\mathit{Value} \ni V, W ::= M \qquad \mathit{EvalCxt} \ni E ::= \Box \mid E\ N \mid \mathbf{rec}\ E\ \mathbf{as}\ \{\mathrm{zero} \to N \mid \mathrm{succ}\ x \to y.N'\}$$

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

$$\mathit{Value} \ni V, W ::= x \mid \lambda x.M \mid \mathrm{zero} \mid \mathrm{succ}\ V$$
$$\mathit{EvalCxt} \ni E ::= \Box \mid E\ N \mid V\ E \mid \mathrm{succ}\ E \mid \mathbf{rec}\ E\ \mathbf{as}\ \{\mathrm{zero} \to N \mid \mathrm{succ}\ x \to y.N'\}$$

*Operational rules*

$$(\beta_{\to}) \qquad (\lambda x.M)\ V \mapsto M[V/x]$$
$$(\beta_{\mathrm{zero}}) \qquad \begin{array}{l}\mathbf{rec}\ \mathrm{zero}\ \mathbf{as} \\ \quad \{\ \mathrm{zero} \to N \\ \quad \mid \mathrm{succ}\ x \to y.N'\}\end{array} \mapsto N$$
$$(\beta_{\mathrm{succ}}) \qquad \begin{array}{l}\mathbf{rec}\ \mathrm{succ}\ V\ \mathbf{as} \\ \quad \{\ \mathrm{zero} \to N \\ \quad \mid \mathrm{succ}\ x \to y.N'\}\end{array} \mapsto (\lambda y.N'[V/x])\ \left(\begin{array}{l}\mathbf{rec}\ V\ \mathbf{as} \\ \quad \{\ \mathrm{zero} \to N \\ \quad \mid \mathrm{succ}\ x \to y.N'\}\end{array}\right)$$

Fig. 3. Call-by-name and Call-by-value Operational semantics of System T.

the matching branch is returned. In addition to binding the predecessor of $M$ to $x$ in the succ $x$ branch, the *recursive result*—calculated by replacing $M$ with its predecessor $x$—is bound to $y$.

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.

System T's call-by-name and call-by-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 $\beta$ rule of the λ-calculus, and $\beta_{\mathrm{zero}}$ and $\beta_{\mathrm{succ}}$ defining recursion on the two Nat constructors. The only difference between call-by-value and call-by-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

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

take this notion seriously and *never* substitute a non-value for a variable. As such, the $\beta_{\text{succ}}$ rule does not substitute the recursive computation **rec** $V$ **as** $\{\text{zero} \to N \mid \text{succ}\, x \to y.N'\}$ for $y$, since it might not be a value (in call-by-value). The next reduction step depends on the evaluation strategy. In call-by-name, this next step is indeed to substitute **rec** $V$ **as** $\{\text{zero} \to N \mid \text{succ}\, x \to y.N'\}$ for $y$, and so we have:

$$\begin{array}{l}\textbf{rec}\ \text{succ}\ M\ \textbf{as} \\ \quad \{\ \text{zero} \to N \\ \quad \mid \text{succ}\, x \to y.N'\}\end{array} \quad \mapsto\!\!\!\to N' \left[ M/x, \left( \begin{array}{l}\textbf{rec}\ M\ \textbf{as} \\ \quad \{\ \text{zero} \to N \\ \quad \mid \text{succ}\, x \to y.N'\}\end{array} \right) \Big/ y \right]$$

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, 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.

*Example* 2.1. The common arithmetic functions *plus*, *times*, *pred*, and *fact* can be written in System T as follows:

$$\begin{aligned} \mathit{plus} &= \lambda x.\lambda y.\ \textbf{rec}\ x\ \textbf{as}\ \{\text{zero} \to y \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\} \\ \mathit{times} &= \lambda x.\lambda y.\ \textbf{rec}\ x\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\ \_ \to z.\mathit{plus}\ y\ z\} \\ \mathit{pred} &= \lambda x.\ \textbf{rec}\ x\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\} \\ \mathit{fact} &= \lambda x.\ \textbf{rec}\ x\ \textbf{as}\ \{\text{zero} \to \text{succ zero} \mid \text{succ}\, y \to z.\mathit{times}\ (\text{succ}\, y)\ z\} \end{aligned}$$

Executing *pred* (succ(succ zero)) in call-by-name proceeds like so:

$$\begin{array}{ll} \mathit{pred}\ (\text{succ}(\text{succ zero})) & \\ \mapsto \textbf{rec}\ \text{succ}(\text{succ zero})\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\} & (\beta_{\to}) \\ \mapsto (\lambda z.\ \text{succ zero})\ (\textbf{rec}\ \text{succ zero}\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}) & (\beta_{\text{succ}}) \\ \mapsto \text{succ zero} & (\beta_{\to}) \end{array}$$

However, in call-by-value, the predecessor of both succ zero and zero is computed even though these intermediate results are not needed in the end:

$$\begin{array}{ll} \mathit{pred}\ (\text{succ}(\text{succ zero})) & \\ \mapsto \textbf{rec}\ \text{succ}(\text{succ zero})\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\} & (\beta_{\to}) \\ \mapsto (\lambda z.\ \text{succ zero})\ (\textbf{rec}\ \text{succ zero}\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}) & (\beta_{\text{succ}}) \\ \mapsto (\lambda z.\ \text{succ zero})\ ((\lambda z.\ \text{zero})\ (\textbf{rec}\ \text{zero}\ \textbf{as}\ \{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\})) & (\beta_{\text{succ}}) \\ \mapsto (\lambda z.\ \text{succ zero})\ ((\lambda z.\ \text{zero})\ \ \text{zero}) & (\beta_{\text{zero}}) \\ \mapsto (\lambda z.\ \text{succ zero})\ \ \text{zero} & (\beta_{\to}) \\ \mapsto \text{succ zero} & (\beta_{\to}) \end{array}$$

In general, *pred* is a constant time ($O(1)$) function over the size of its argument when following the call-by-name semantics, which computes the predecessor of any natural number in a fixed number of steps. In contrast, *pred* is a linear time ($O(n)$) function when following the call-by-value semantics, where *pred* (succ$^n$ zero) executes with a number of steps

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

proportional to the size $n$ of its argument because it requires at least $n$ applications of the $\beta_{\text{succ}}$ rule before an answer can be returned.

## 3 Recursion in an abstract machine

In order to explore the lower-level performance details of recursion, we can use an *abstract machine* for modeling an implementation of System T. Unlike the operational semantics given in Fig. 3, which requires a costly recursive search deep into an expression to find the next redex at every step, an abstract machine explicitly includes this search in the computation itself which can immediately resume from the same position as the previous reduction step. As such, every step of the machine can be applied by matching only on the top-level form of the machine state, which models the fact that a real implementations in a machine does not have to recursively search for the next reduction step to perform, but can identify and jump to the next step in a constant amount of time. Thus, in an abstract machine instead of working with terms one works with configurations of the form:

$$\langle M \| E \rangle$$

where $M$ is a term also called a producer and $E$ is a continuation or evaluation context, also called a consumer. A state, also called a command, puts together a producer and a consumer, so that the output of $M$ is given as the input to $E$. We first present distinct abstract machines for call-by-name and call-by-value, we then smooth out the differences in the uniform abstract machine.

### 3.1 Call-by-name abstract machine

The call-by-name abstract machine for System T is based on the Krivine machine (Krivine, 2007), which is defined in terms of these continuations (representing evaluation contexts, for example, **tp** represents the empty, top-level context) and transition rules:[1,2]

$$E ::= \mathbf{tp} \mid N \cdot E \mid \mathbf{rec}\{\text{zero} \to M \mid \text{succ}\, x \to z.N\}\ \mathbf{with}\ E$$

Refocusing rules:

$$\langle M\ N \| E \rangle \mapsto \langle M \| N \cdot E \rangle$$
$$\langle \mathbf{rec}\, M\ \mathbf{as}\{\ldots\} \| E \rangle \mapsto \langle M \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ E \rangle$$

Reduction rules:

$$\langle \lambda x.M \| N \cdot E \rangle \mapsto \langle M[N/x] \| E \rangle$$
$$\left\langle \text{zero} \,\middle\|\, \begin{array}{l} \mathbf{rec}\ \{\ \text{zero} \to N \\ \qquad \mid \text{succ}\, x \to y.N'\} \\ \mathbf{with}\ E \end{array} \right\rangle \mapsto \langle N \| E \rangle$$

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

[2] Note that the rule $\langle \mathbf{rec}\, M\ \mathbf{as}\{\ldots\} \| E \rangle \mapsto \langle M \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ E \rangle$ has an ellipsis "..." which matches the same syntactic form on both sides of the transition rule. In general, we will use this pattern of matching ellipsis to elide a repeated syntactic fragment on two sides of a transition rule to simplify the presentation of examples.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$\left\langle \text{succ } M \,\middle\|\, \begin{array}{l}\textbf{rec } \{\text{zero} \to N \\ \quad\;\; \mid \text{succ } x \to y.N'\} \\ \textbf{with } E\end{array} \right\rangle \mapsto \left\langle N' \left[ M/x \,,\; \begin{array}{l}\textbf{rec } M \textbf{ as } \{\text{zero} \to N \\ \qquad\qquad \mid \text{succ } x \to y.N'\}\end{array} \Big/ y \right] \middle\| E \right\rangle$$

The first two rules are *refocusing* rules that move the attention of the machine closer to the next reduction building a larger continuation: $N \cdot E$ corresponds to the context $E[\Box\ N]$, and the continuation $\textbf{rec}\{\text{zero} \to N \mid \text{succ } x \to y.N'\}\ \textbf{with}\ E$ corresponds to the context $E[\textbf{rec}\ \Box\ \textbf{as}\{\text{zero} \to N \mid \text{succ } x \to y.N'\}]$. The latter three rules are *reduction* rules which correspond to steps of the operational semantics in Fig. 3. While the distinction between refocusing and reduction rules is just a mere classification now, the difference will become an important tool for generalizing the abstract machine into a more uniform presentation later in Section 3.3.

### 3.2 Call-by-value abstract machine

A CEK-style (Felleisen & Friedman, 1986), call-by-value abstract machine for System T—which evaluates applications $M_1\ M_2\ \ldots\ M_n$ left-to-right to match the call-by-value semantics in Fig. 3—is given by these continuations $E$ and transition rules:

$$E ::= \textbf{tp} \mid n \cdot E \mid V \circ E \mid \text{succ} \circ E \mid \textbf{rec}\{\text{zero} \to M \mid \text{succ } x \to y.N\}\ \textbf{with}\ E$$

Refocusing rules:

$$\begin{aligned}
\langle M\ N \| E\rangle &\mapsto \langle M \| N \cdot E\rangle \\
\langle V \| R \cdot E\rangle &\mapsto \langle R \| V \circ E\rangle \\
\langle V \| V' \circ E\rangle &\mapsto \langle V' \| V \cdot E\rangle \\
\langle \textbf{rec } M \textbf{ as}\{\ldots\} \| E\rangle &\mapsto \langle M \| \textbf{rec}\{\ldots\}\ \textbf{with } E\rangle \\
\langle \text{succ } R \| E\rangle &\mapsto \langle R \| \text{succ} \circ E\rangle \\
\langle V \| \text{succ} \circ E\rangle &\mapsto \langle \text{succ } V \| E\rangle
\end{aligned}$$

Reduction rules:

$$\langle \lambda x.M \| V \cdot E\rangle \mapsto \langle M[V/x] \| E\rangle$$

$$\left\langle \text{zero} \,\middle\|\, \begin{array}{l}\textbf{rec } \{\text{zero} \to N \\ \quad\;\; \mid \text{succ } x \to y.N'\} \\ \textbf{with } E\end{array} \right\rangle \mapsto \langle N \| E\rangle$$

$$\left\langle \text{succ } V \,\middle\|\, \begin{array}{l}\textbf{rec } \{\text{zero} \to N \\ \quad\;\; \mid \text{succ } x \to y.N'\} \\ \textbf{with } E\end{array} \right\rangle \mapsto \left\langle V \,\middle\|\, \begin{array}{l}\textbf{rec } \{\text{zero} \to N \\ \quad\;\; \mid \text{succ } x \to y.N'\} \\ \textbf{with } ((\lambda y.N') \circ E)\end{array} \right\rangle$$

where $R$ stands for a *non-value* term. Since the call-by-value operational semantics has more forms of evaluation contexts, this machine has additional refocusing rules for accumulating more forms of continuations including applications of functions ($V \circ E$ corresponding to $E[V\ \Box]$) and the successor constructor ($\text{succ} \circ E$ corresponding to $E[\text{succ}\ \Box]$). Also note that the final reduction rule for the succ case of recursion is different, accounting for the fact that recursion in call-by-value follows a different order than in call-by-name. Indeed, the recursor must explicitly accumulate and build upon a continuation, "adding to"

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

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

$$\mathit{Command} \ni c ::= \langle v \| e \rangle \quad \mathit{Term} \ni v, w ::= \mu\alpha.c \mid V \quad \mathit{CoTerm} \ni e, f ::= \tilde{\mu}x.c \mid E$$

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

$$\begin{aligned} \mathit{Value} \ni V, W &::= \mu\alpha.c \mid x \mid \lambda x.v \mid \mathrm{zero} \mid \mathrm{succ}\, V \\ \mathit{CoValue} \ni E, F &::= \alpha \mid V \cdot E \mid \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\}\ \mathbf{with}\ E \end{aligned}$$

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

$$\begin{aligned} \mathit{Value} \ni V, W &::= x \mid \lambda x.v \mid \mathrm{zero} \mid \mathrm{succ}\, V \\ \mathit{CoValue} \ni E, F &::= \tilde{\mu}x.c \mid \alpha \mid V \cdot E \mid \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\}\ \mathbf{with}\ E \end{aligned}$$

*Operational reduction rules*:

$$(\mu) \qquad \langle \mu\alpha.c \| E \rangle \mapsto c[E/\alpha]$$

$$(\tilde{\mu}) \qquad \langle V \| \tilde{\mu}x.c \rangle \mapsto c[V/x]$$

$$(\beta_{\to}) \qquad \langle \lambda x.v \| V \cdot E \rangle \mapsto \langle v[V/x] \| E \rangle$$

$$(\beta_{\mathrm{zero}}) \qquad \left\langle \mathrm{zero} \,\middle\|\, \begin{array}{l} \mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \quad\ \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ E \end{array} \right\rangle \mapsto \langle v \| E \rangle$$

$$(\beta_{\mathrm{succ}}) \qquad \left\langle \mathrm{succ}\, V \,\middle\|\, \begin{array}{l} \mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \quad\ \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ E \end{array} \right\rangle \mapsto \left\langle \mu\alpha. \left\langle V \,\middle\|\, \begin{array}{l} \mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \quad\ \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ \alpha \end{array} \right\rangle \,\middle\|\, \tilde{\mu}y.\langle w[V/x] \| E \rangle \right\rangle$$

Fig. 4. Uniform, recursive abstract machine for System T.

the place it returns to with every recursive call. But otherwise, the reduction rules are the same.[3]

### 3.3 Uniform abstract machine

We now unite both evaluation strategies with a common abstract machine, shown in Fig. 4. As before, machine configurations are of the form

$$\langle v \| e \rangle$$

which put together a term $v$ and a continuation $e$ (often referred to as a coterm). However, both terms and continuations are more general than before. Our uniform abstract machine is based on the sequent calculus, a symmetric language reflecting many dualities of classical logic (Curien & Herbelin, 2000; Wadler, 2003). An advantage of this sequent-based system is that the explicit symmetry inherent in its syntax gives us a language to express and understand the implicit symmetries that are hiding in computations and types. This

[3] The astute reader might notice that, when given a value, the continuation $(\lambda x.M) \circ E$ always leads the same sequence of two transitions: $\langle V \| (\lambda x.M) \circ E \rangle \mapsto \langle \lambda x.M \| V \cdot E \rangle \mapsto \langle M[V/x] \| E \rangle$. Their next thought might be to merge these two steps together, to transition directly with the combined rule $\langle V \| (\lambda x.M) \circ E \rangle \mapsto \langle M[V/x] \| E \rangle$ that "saves" a step in the reduction sequence. However, our main motivation for showing the call-by-name and call-by-value abstract machines is to lead to the uniform abstract machine coming next in Section 3.3. Leaving these two steps separate more clearly illustrates the common ground of the two machines which makes it possible to combine them into a common definition.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

utility of a symmetric language is one of the key insights of our approach, which will let us syntactically *derive* a notion of corecursion which is dual to recursion.

Unlike the previous machines, continuations go beyond evaluation contexts and include $\tilde{\mu}x.\langle v \| e\rangle$, which is a continuation that binds its input value to $x$ and then steps to the machine state $\langle v \| e\rangle$. This new form allows us to express the additional call-by-value evaluation contexts: $V \circ E$ becomes $\tilde{\mu}x.\langle V \| x \cdot E\rangle$, and $\mathrm{succ} \circ E$ is $\tilde{\mu}x.\langle \mathrm{succ}\, x \| E\rangle$. Evaluation contexts are also more restrictive than before; only values can be pushed on the calling stack. We represent the application continuation $M \cdot E$ with a non-value argument $M$ by naming its partner—the generic value $V$—with $y$: $\tilde{\mu}y.\langle M \| \tilde{\mu}x.\langle y \| x \cdot E\rangle\rangle$.[4]

The refocusing rules can be subsumed all together by extending terms with a dual form of $\tilde{\mu}$-binding. The $\mu$-abstraction expression $\mu\alpha.\langle v \| e\rangle$ binds its continuation to $\alpha$ and then steps to the machine state $\langle v \| e\rangle$. With $\mu$-bindings, all the refocusing rules for call-by-name and call-by-value can be encoded in terms of $\mu$ and $\tilde{\mu}$.[5] It is also not necessary to do these steps at run-time, but can all be done before execution through a compilation step. The target language of this compilation step becomes the syntax of the uniform abstract machine, as shown in Fig. 4. This language does not include anymore applications $M\ N$ and the recursive term **rec** $M$ **as**$\{\ldots\}$. Also, unlike System T, the syntax of terms and coterms depends on the definition of values and covalues; succ $V$, **rec**$\{\ldots\}$ **with** $E$, and call stacks $V \cdot E$ are valid in both call-by-name and call-by-value, just with different definitions of $V$ and $E$. Yet, all System T terms can still be translated to the smaller language of the machine, as shown in Fig. 5. General terms also include the $\mu$- and $\tilde{\mu}$-binders described above: $\mu\alpha.c$ is not a value in call-by-value, and $\tilde{\mu}x.c$ is not a covalue in call-by-name. So for example, $\mathrm{succ}(\mu\alpha.c)$ is not a legal term in call-by-value, but can be translated instead to $\mu\beta.\langle \mu\alpha.c \| \tilde{\mu}x.\langle \mathrm{succ}\, x \| \beta\rangle\rangle$ following Fig. 5. Similarly $x \cdot \tilde{\mu}y.c$ is not a legal call stack in call-by-name, but it can be rewritten to $\tilde{\mu}z.\langle \mu\alpha.\langle z \| x \cdot \alpha\rangle \| \tilde{\mu}y.c\rangle$.

As with System T, the notions of values and covalues drive the reduction rules in Fig. 4. In particular, the $\mu$ and $\tilde{\mu}$ rules will only substitute a value for a variable or a covalue for a covariable, respectively. Likewise, the $\beta_{\to}$ rule implements function calls, but taking the next argument value off of a call stack and plugging it into the function. The only remaining rules are $\beta_{\mathrm{zero}}$ and $\beta_{\mathrm{succ}}$ for reducing a recursor when given a number constructed by zero or succ. While the $\beta_{\mathrm{zero}}$ is exactly the same as it was previously in both specialized machine, notice how $\beta_{\mathrm{succ}}$ is different. Rather than committing to one

[4] We choose to restrict certain continuations by expanding them into a $\tilde{\mu}$ in order to reduce the number of reduction rules that we have to consider in the abstract machine. Alternatively, we could accept $R \cdot E$ with a non-value $R$ as a valid continuation, but this requires another rule to perform the expansion which "lifts" out the argument $R$ to the top-level during execution like so: $\langle V \| R \cdot E\rangle \mapsto \langle V \| \tilde{\mu}y.\langle R \| \tilde{\mu}x.\langle y \| x \cdot E\rangle\rangle\rangle \mapsto \langle R \| \tilde{\mu}x.\langle V \| x \cdot E\rangle\rangle$. This family of "lifting" rules appeared as the $\varsigma$ rules of Wadler (2003). The choice to perform $\varsigma$ reduction at "run-time" while the machine executes as in Wadler (2003) versus rewriting the code using analogous $\varsigma$ expansions at "compile-time" as we do here leads to an equivalent system. See Downen & Ariola (2018*b*) for more details on this choice and how it relates to proof theory.

[5] The $\mu$-abstraction provides a form of control effects first-class continuations, analogous to the abstraction of the same name in the $\lambda\mu$-calculus (Parigot, 1992). The reason why $\mu$ is so useful in the abstract machine is that it lets us expand out all elimination forms, like function application and numeric recursion, into its continuation form, thereby eliminating the redundant syntax we would have to consider. For example, the refocusing rule for function application, $\langle M\ N \| \alpha\rangle \mapsto \langle M \| N \cdot \alpha\rangle$ rewrites $M\ N$ on the term side to $N \cdot \alpha$ on the continuation side at run-time, for any starting continuation $\alpha$. Instead, we can replace this step at compile-time by abstracting over the $\alpha$ with the program transformation $M\ N = \mu\alpha.\langle M \| N \cdot E\rangle$. The same can be done by transforming **rec** as an expression to **rec** as a continuation using $\mu$. In this way, $\mu$-abstractions reduce the number of reduction rules and syntactic forms needed by the uniform abstract machine.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$
\begin{aligned}
[\![x]\!] &:= x \\
[\![\lambda x.M]\!] &:= \lambda x.[\![M]\!] \\
[\![\mathrm{zero}]\!] &:= \mathrm{zero} \\
[\![\mathrm{succ}\ V]\!] &:= \mathrm{succ}\ [\![V]\!] \\
[\![\mathrm{succ}\ M]\!] &:= \mu\alpha.\langle [\![M]\!] \,\|\, \tilde{\mu}x.\langle \mathrm{succ}\ x \| \alpha\rangle\rangle \qquad (M \notin \mathit{Value}) \\
[\![M\ N]\!] &:= \mu\alpha.\langle [\![M]\!] \,\|\, \tilde{\mu}x.\langle [\![N]\!] \,\|\, \tilde{\mu}y.\langle x \| y \cdot \alpha\rangle\rangle\rangle \\
[\![\mathbf{rec}\ M\ \mathbf{as}\ \{\mathrm{zero} \to N \mid \mathrm{succ}\ x \to y.N'\}]\!] &:= \mu\alpha.\langle [\![M]\!] \,\|\, \mathbf{rec}\{\mathrm{zero} \to [\![N]\!] \mid \mathrm{succ}\ x \to y.[\![N']\!]\}\ \mathbf{with}\ \alpha\rangle
\end{aligned}
$$

Fig. 5. The translation from System T to the uniform abstract machine.

evaluation order or the other, $\beta_{\mathrm{succ}}$ is neutral: the recursive predecessor (expressed as the term $\mu\alpha.\langle V \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \alpha\rangle$ on the right-hand side) is neither given precedence (at the top of the command) nor delayed (by substituting it for $y$). Instead, this recursive predecessor is bound to $y$ with a $\tilde{\mu}$-abstraction. This way, the correct evaluation order can be decided in the next step by either an application of $\mu$ or $\tilde{\mu}$ reduction.

Notice one more advantage of translating System T at "compile-time" into a smaller language in the abstract machine. In the same way that the λ-calculus supports both a deterministic operational semantics along with a rewriting system that allows for optimizations that apply reductions to any sub-expression, so too does the syntax of this uniform abstract machine allow for optimizations that reduce sub-expressions in advance of the usual execution order. For example, consider the term $\mathbf{let}\, f = (\lambda x.(\lambda y.y)\ x)\ \mathbf{in}\ M$.[6] Its next step is to substitute the abstraction $(\lambda x.(\lambda y.y)\ x)$ for $f$ in $M$, and every time $M$ calls $(f\ V)$ we must repeat the steps $(\lambda x.(\lambda y.y)\ x)\ V \mapsto (\lambda y.y)\ V \mapsto V$ again. But applying a $\beta$ reduction under the λ bound to $f$, we can instead optimize this expression to $\mathbf{let}\, f = (\lambda x.x)\ \mathbf{in}\ M$. With this optimization, when $M$ calls $(f\ V)$ it reduces as $(\lambda x.x)\ V \mapsto V$ in one step. This same approach applies to the uniform abstract machine, too, because all elimination forms like $\square\ N$ and $\mathbf{rec}\ \square\ \mathbf{as}\{\dots\}$ have been compiled to continuations of the form $N \cdot \alpha$ and $\mathbf{rec}\{\dots\}\ \mathbf{with}\ \alpha$ that we can use to apply machine reductions. For example, the translation of $\mathbf{let}\, f = (\lambda x.(\lambda y.y)\ x)\ \mathbf{in}\ M$ is quite large, but we can optimize away several $\mu$ and $\tilde{\mu}$ reductions—which are responsible for the administrative duty of directing information and control flow—in advance inside sub-expressions to simplify the translation to:[7]

$$[\![\mathbf{let}\, f = (\lambda x.(\lambda y.y)\ x)\ \mathbf{in}\ M]\!] \twoheadrightarrow_{\mu\tilde{\mu}} \mu\alpha.\langle \lambda x.\mu\beta.\langle \lambda y.y \| x \cdot \beta\rangle \,\|\, \tilde{\mu}f.\langle [\![M]\!] \| \alpha\rangle\rangle$$

where $\twoheadrightarrow$ denotes zero or more reduction steps $\mapsto$ applied inside *any* context. From there, we can further optimize the program by simplifying the application of $\lambda y.y$ in advance, by applying the $\beta_{\to}$ rule inside the $\mu$s binding $\alpha$ and $\beta$ and the λ binding $x$ like so:

$$\mu\alpha.\langle \lambda x.\mu\beta.\langle \lambda y.y \| x \cdot \beta\rangle \,\|\, \tilde{\mu}f.\langle [\![M]\!] \| \alpha\rangle\rangle \to_{\beta_{\to}} \mu\alpha.\langle \lambda x.\mu\beta.\langle x \| \beta\rangle \,\|\, \tilde{\mu}f.\langle [\![M]\!] \| \alpha\rangle\rangle$$

This way, we are able to still optimize the program even *after* compilation to the abstract machine language, the same as if we were optimizing the original λ-calculus source code.

[6] Where we use the usual syntactic sugar $\mathbf{let}\, x = M\ \mathbf{in}\ N$ defined as $(\lambda x.N)\ M$.

[7] The $\mu$ and $\tilde{\mu}$ reductions that we simplify here are analogous to *administrative reductions* in continuation-passing style translations (Sabry & Felleisen, 1993). They explicitly direct the information flow and control flow of the program in terms of $\tilde{\mu}$- and $\mu$-bindings, but can sometimes make it harder to read example code. We don't formally distinguish between administrative and ordinary reductions in this paper, but we may selectively (and explicitly) reduce away these more administrative bindings in order to make examples easier to read.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

*Intermezzo* 3.1. We can now summarize how some basic concepts of recursion are directly modeled in our syntactic framework:

- *With inductive data types, values are constructed and the consumer is a recursive process that uses the data*. Natural numbers are terms or producers, and their use is a process which is triggered when the term becomes a value.
- *Construction of data is finite and its consumption is (potentially) infinite, in the sense that there must be no limit to the size of the data that a consumer can process*. We can only build values from a finite number of constructor applications. However, the consumer does not know how big of an input it will be given, so it has to be ready to handle data structures of any size. In the end, termination is preserved because only finite values are consumed.
- *Recursion uses the data, rather than producing it*. **rec** is a coterm, not a term.
- *Recursion starts big and potentially reduces down to a base case*. As shown in the reduction rules, the recursor breaks down the data structure and might end when the base case is reached.
- *The values of a data structures are all independent from each other but the results of the recursion potentially depend on each other*. In the reduction rule for the successor case, the result at a number $n$ might depend on the result at $n-1$.

### *3.4 Examples of recursion*

Restricting the $\mu$ and $\tilde{\mu}$ rules to only binding (co)values effectively implements the chosen evaluation strategy. For example, consider the application $(\lambda z.\,\mathrm{succ}\,\mathrm{zero})\ ((\lambda x.x)\ \mathrm{zero})$. Call-by-name evaluation will reduce the outer application first and return succ zero right away, whereas call-by-value evaluation will first reduce the inner application $((\lambda x.x)\ \mathrm{zero})$. How is this different order of evaluation made explicit in the abstract machine, which uses the same set of rules in both cases? First, consider the translation of $[\![(\lambda z.x)\ ((\lambda x.x)\ y)]\!]$:

$$[\![(\lambda z.x)\ ((\lambda x.x)\ y)]\!] := \mu\alpha.\langle \lambda z.x \| \tilde{\mu} f.\langle \mu\beta.\langle \lambda x.x \| \tilde{\mu} g.\langle y \| \tilde{\mu} y.\langle g \| y \cdot \beta\rangle\rangle\rangle \| \tilde{\mu} z'.\langle f \| z' \cdot \alpha\rangle\rangle$$

This is quite a large term for such a simple source expression. To make the example clearer, let us first simplify the administrative-style $\tilde{\mu}$-bindings out of the way (using applications of $\tilde{\mu}$ rules for to substitute $\lambda$-abstractions and variables in a way that is valid for both call-by-name and call-by-value) to get the shorter term:

$$[\![(\lambda z.x)\ ((\lambda x.x)\ y)]\!] \twoheadrightarrow_{\tilde{\mu}} \mu\alpha.\langle \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \| \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle$$

To execute it, we need to put it in interaction with an actual context. In our case, we can simply use a covariable $\alpha$. Call-by-name execution then proceeds from the simplified as:

$$\begin{aligned} \langle \mu\alpha.\langle \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \| \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle \| \alpha\rangle &\mapsto \langle \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \| \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle && (\mu) \\ &\mapsto \langle \lambda z.x \| \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \cdot \alpha\rangle && (\tilde{\mu}*) \\ &\mapsto \langle x \| \alpha\rangle && (\beta_{\to}) \end{aligned}$$

And call-by-value execution of the simplified proceeds as:

$$\begin{aligned} \langle \mu\alpha.\langle \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \| \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle \| \alpha\rangle &\mapsto \langle \mu\beta.\langle \lambda x.x \| y \cdot \beta\rangle \| \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle && (\mu) \\ &\mapsto \langle \lambda x.x \| y \cdot \tilde{\mu} z.\langle \lambda z.x \| z \cdot \alpha\rangle\rangle && (\mu*) \end{aligned}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$\begin{array}{ll} \mapsto \langle y \| \tilde{\mu} z. \langle \lambda z.x \| z \cdot \alpha \rangle \rangle & (\beta_{\to}) \\ \mapsto \langle \lambda z.x \| y \cdot \alpha \rangle & (\tilde{\mu}) \\ \mapsto \langle x \| \alpha \rangle & (\beta_{\to}) \end{array}$$

The first two steps are the same for either evaluation strategy. Where the two begin to diverge is in the third step (marked by a $*$), which is an interaction between a $\mu$- and a $\tilde{\mu}$-binder. In call-by-name, the $\tilde{\mu}$ rule takes precedence (because a $\tilde{\mu}$-coterm is not a covalue), leading to the next step which throws away the first argument, unevaluated. In call-by-value, the $\mu$ rule takes precedence (because a $\mu$-term is not a value), leading to the next step which evaluates the first argument, producing $y$ to bind to $z$ that eventually gets thrown away.

Consider the System T definition of *plus* from Example 2.1, which is expressed by the machine term

$$plus = \lambda x.\lambda y.\mu\beta.\langle x \| \mathbf{rec}\{\text{zero} \to y \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \beta \rangle$$

The application *plus* 2 3 is then expressed as $\mu\alpha.\langle plus \| 2 \cdot 3 \cdot \alpha \rangle$, which is obtained by reducing some $\mu$- and $\tilde{\mu}$-bindings in advance. Putting this term in the context $\alpha$, in call-by-value it executes (eliding the branches of the **rec** continuation, which are the same in every following step) like so:

$$\begin{array}{ll} \langle \mu\alpha.\langle plus \| 2 \cdot 3 \cdot \alpha \rangle \| \alpha \rangle & \\ \mapsto \langle plus \| 2 \cdot 3 \cdot \alpha \rangle & \\ \mapsto \langle \lambda y.\mu\beta.\langle 2 \| \mathbf{rec}\{\text{zero} \to y \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \beta \rangle \| 3 \cdot \alpha \rangle & (\beta_{\to}) \\ \mapsto \langle \mu\beta.\langle 2 \| \mathbf{rec}\{\text{zero} \to 3 \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \beta \rangle \| \alpha \rangle & (\beta_{\to}) \\ \mapsto \langle \text{succ}(\text{succ zero}) \| \mathbf{rec}\{\text{zero} \to 3 \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \alpha \rangle & (\mu) \\ \mapsto \langle \mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ \beta \rangle \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle & (\beta_{\text{succ}}) \\ \mapsto \langle \text{succ zero} \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle & (\mu) \\ \mapsto \langle \mu\beta.\langle \text{zero} \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ \beta \rangle \| \tilde{\mu} z'.\langle \text{succ}\ z' \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle \rangle & (\beta_{\text{succ}}) \\ \mapsto \langle \text{zero} \| \mathbf{rec}\{\ldots\}\ \mathbf{with}\ \tilde{\mu} z'.\langle \text{succ}\ z' \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle \rangle & (\mu) \\ \mapsto \langle 3 \| \tilde{\mu} z'.\langle \text{succ}\ z' \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle \rangle & (\beta_{\text{zero}}) \\ \mapsto \langle \text{succ}\ 3 \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle & (\tilde{\mu}) \\ \mapsto \langle \text{succ}(\text{succ}\ 3) \| \alpha \rangle & (\tilde{\mu}) \end{array}$$

Notice how this execution shows how, during the recursive traversal of the data structure, the return continuation of the recursor is updated (in blue) to keep track of the growing context of pending operations, which must be fully processed before the final value of 5 (succ(succ 3)) can be returned to the original caller ($\alpha$). This update is implicit in the $\lambda$-calculus-based System T, but becomes explicit in the abstract machine. In contrast, call-by-name only computes numbers as far as they are needed, otherwise stopping at the outermost constructor. The call-by-name execution of the above command proceeds as follows, after fast-forwarding to the first application of $\beta_{\text{succ}}$:

$$\begin{array}{ll} \langle \mu\alpha.\langle plus \| 2 \cdot 3 \cdot \alpha \rangle \| \alpha \rangle & \\ \mapsto\!\!\mapsto \langle \text{succ}(\text{succ zero}) \| \mathbf{rec}\{\text{zero} \to 3 \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \alpha \rangle & (\mu) \\ \mapsto \langle \mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\text{zero} \to 3 \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \beta \rangle \| \tilde{\mu} z.\langle \text{succ}\ z \| \alpha \rangle \rangle & (\beta_{\text{succ}}) \\ \mapsto \langle \text{succ}(\mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\text{zero} \to 3 \mid \text{succ}\ \_ \to z.\ \text{succ}\ z\}\ \mathbf{with}\ \beta \rangle) \| \alpha \rangle & (\tilde{\mu}) \end{array}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Unless $\alpha$ demands to know something about the predecessor of this number, the term $\mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \beta\rangle$ will not be computed.

Now consider *pred* (succ(succ zero)), which can be expressed in the machine as:

$$\mu\alpha.\langle \mathit{pred} \| \text{succ(succ zero)} \cdot \alpha\rangle$$
$$\mathit{pred} = \lambda x.\mu\beta.\langle x \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}\ \mathbf{with}\ \beta\rangle$$

In call-by-name, it executes with respect to $\alpha$ like so:

$$\begin{array}{ll}
\langle \mu\alpha.\langle \mathit{pred} \| \text{succ(succ zero)} \cdot \alpha\rangle \| \alpha\rangle & \\
\mapsto \langle \mathit{pred} \| \text{succ(succ zero)} \cdot \alpha\rangle & (\mu) \\
\mapsto \langle \mu\beta.\langle \text{succ(succ zero)} \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}\ \mathbf{with}\ \beta\rangle \| \alpha\rangle & (\beta_{\to}) \\
\mapsto \langle \text{succ(succ zero)} \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}\ \mathbf{with}\ \alpha\rangle & (\mu) \\
\mapsto \langle \mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}\ \mathbf{with}\ \beta\rangle \| \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle & (\beta_{\text{succ}}) \\
\mapsto \langle \text{succ zero} \| \alpha\rangle & (\tilde{\mu})
\end{array}$$

Notice how, after the first application of $\beta_{\text{succ}}$, the computation finishes in just one $\tilde{\mu}$ step, even though we began recursing on the number 2. In call-by-value instead, we have to continue with the recursion even though its result is not needed. Fast-forwarding to the first application of the $\beta_{\text{succ}}$ rule, we have:

$$\begin{array}{ll}
\langle \text{succ(succ zero)} \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\, x \to z.x\}\ \mathbf{with}\ \alpha\rangle & \\
\mapsto \langle \mu\beta.\langle \text{succ zero} \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \beta\rangle \| \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle & (\beta_{\text{succ}}) \\
\mapsto \langle \text{succ zero} \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle & (\mu) \\
\mapsto \langle \mu\beta.\langle \text{zero} \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \beta\rangle \| \tilde{\mu}z.\langle \text{zero} \| \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle\rangle & (\beta_{\text{succ}}) \\
\mapsto \langle \text{zero} \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \tilde{\mu}z.\langle \text{succ zero} \| \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle\rangle & (\mu) \\
\mapsto \langle \text{zero} \| \tilde{\mu}z.\langle \text{succ zero} \| \tilde{\mu}z.\langle \text{succ zero} \| \alpha\rangle\rangle\rangle & (\beta_{\text{zero}}) \\
\mapsto\!\!\mapsto \langle \text{succ zero} \| \alpha\rangle & (\tilde{\mu})
\end{array}$$

### *3.5 Recursion versus iteration: Expressiveness and efficiency*

Recall how the recursor performs two jobs at the same time: finding the predecessor of a natural number as well as calculating the recursive result given for the predecessor. These two functionalities can be captured separately by continuations that perform shallow *case analysis* and *iteration*, respectively. Rather than including them as primitives, both can be expressed as syntactic sugar in the form of macro-expansions in the language of the abstract machine like so:

$$\begin{array}{llll}
\mathbf{case}\ \{\text{zero} \to v & \mathbf{rec}\ \{\text{zero} \to v & \mathbf{iter}\ \{\text{zero} \to v & \mathbf{rec}\ \{\text{zero} \to v \\
\quad \mid \text{succ}\, x \to w\} := & \quad \mid \text{succ}\, x \to _.w\} & \quad \mid \text{succ} \to x.w\} := & \quad \mid \text{succ}\, _ \to x.w\} \\
\mathbf{with}\ E & \mathbf{with}\ E & \mathbf{with}\ E & \mathbf{with}\ E
\end{array}$$

The only cost of this encoding of **case** and **iter** is an unused variable binding, which is easily optimized away. In practice, this encoding of iteration will perform exactly the same as if we had taken **iter**ation as a primitive.

While it is less obvious, going the other way is still possible. It is well known that primitive recursion can be encoded as a macro-expansion of iteration using pairs. The

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

usual macro-expansion in System T is:

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

The trick to this encoding is to use **iter** to compute *both* a reconstruction of the number being iterated upon (the first component of the iterative result) alongside the desired result (the second component). Doing both at once gives access to the predecessor in the succ case, which can be extracted from the first component of the previous result (given by the variable $x$ in the pattern match $(x, y)$).

To express this encoding in the abstract machine, we need to extend it with pairs, which look like (Wadler, 2003):

$$\langle (v, w) \| \mathrm{fst}\, E \rangle \mapsto \langle v \| E \rangle \qquad \langle (v, w) \| \mathrm{snd}\, E \rangle \mapsto \langle w \| E \rangle \qquad (\beta_\times)$$

In the syntax of the abstract machine, the analogous encoding of a **rec** continuation as a macro-expansion looks like this:

$$\begin{array}{lll} \textbf{rec}\ \{\ \mathrm{zero} \to v & & \textbf{iter}\ \{\ \mathrm{zero} \to (\mathrm{zero}, v) \\ \quad \mid \mathrm{succ}\, x \to y.w\} & := & \quad \mid \mathrm{succ} \to (x, y).(\mathrm{succ}\, x, w)\} \\ \textbf{with}\ E & & \textbf{with}\ \mathrm{snd}\ E \end{array}$$

Since the inductive case $w$ might refer to both the predecessor $x$ and the recursive result for the predecessor (named $y$), the two parts must be extracted from the pair returned from iteration. Here we express this extraction in the form of pattern matching, which is shorthand for:

$$(x, y).(v_1, v_2) := z.\mu\alpha.\langle z \| \mathrm{fst}(\tilde{\mu}x.\langle z \| \mathrm{snd}(\tilde{\mu}y.\langle (v_1, v_2) \| \alpha \rangle)) \rangle)$$

Note that the recursor continuation is tasked with passing its final result to $E$ once it has finished. In order to give this same result to $E$, the encoding has to extract the second component of the final pair before passing it to $E$, which is exactly what snd $E$ expresses.

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. As such, redefining *pred* using **iter** in this way changes it from a constant time ($O(1)$) to a linear time ($O(n)$) function. 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**.

### *3.6 Types and correctness*

We can also give a type system directly for the abstract machine, as shown in Fig. 6. This system has judgments for assigning types to terms as usual: $\Gamma \vdash v : A$ says $v$ produces

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

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

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

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

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

$$\frac{}{\Gamma \vdash \mathrm{zero} : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{zero}} \qquad \frac{\Gamma \vdash V : \mathrm{Nat}}{\Gamma \vdash \mathrm{succ}\, V : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{succ}}$$

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

Fig. 6. Type system for the uniform, recursive abstract machine.

an output of type $A$. In addition, there are also judgments for assigning types to coterms ($\Gamma \vdash e \div A$ says $e$ consumes an input of type $A$) and commands ($\Gamma \vdash c$ says $c$ is safe to compute, and does not produce or consume anything).

This type system ensures that the machine itself is type safe: well-typed, executable commands don't get stuck while in the process of computing a final state. For our purposes, we will consider "programs" to be commands $c$ with just one free covariable (say $\alpha$), representing the initial, top-level continuation expecting a natural number as the final answer. Thus, well-typed executable commands will satisfy $\alpha \div \mathrm{Nat} \vdash c$. The only final states of these programs have the form $\langle \mathrm{zero} \| \alpha \rangle$, which sends 0 to the final continuation $\alpha$, or $\langle \mathrm{succ}\, V \| \alpha \rangle$, which sends the successor of some $V$ to $\alpha$. But the type system ensures more than just type safety: all well-typed programs will eventually terminate. That's because **rec**-expressions, which are the only form of recursion in the language, always decrement their input by 1 on each recursive step. So together, every well-typed executable command will eventually (termination) reach a valid final state (type safety).

**Theorem 3.2** (Type safety & Termination of Programs)**.** *For any command $c$ of the recursive abstract machine, if $\alpha \div \mathrm{Nat} \vdash c$ then $c \mapsto\!\!\!\!\to \langle \mathrm{zero} \| \alpha \rangle$ or $c \mapsto\!\!\!\!\to \langle \mathrm{succ}\, V \| \alpha \rangle$ for some $V$.*

The truth of this theorem follows directly from the latter development in Section 6, since it is a special case of Theorem 6.14.

*Intermezzo* 3.3. Since our abstract machine is based on the logic of Gentzen's sequent calculus (Gentzen, 1935), the type system in Fig. 6 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 becomes $\alpha : A$ on the right. Doing so gives a conventional two-sided sequent calculus as in (Ariola *et al.*, 2009; Downen & Ariola, 2018*b*), where the rules labeled $L$ with conclusions

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$\mathit{Type} \ni A, B ::= A \to B \mid \mathrm{Nat} \mid \mathrm{Stream}\, A$$

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

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

Fig. 7. Typing rules for streams in the uniform, (co)recursive abstract machine.

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. In general, the three forms of two-sided sequents correspond to these three different typing judgments used here (with $\Gamma$ rearranged for uniformity):

- $x_i : B_i, .\overset{i}{.}., \alpha_j \div C_j, .\overset{j}{.}. \vdash v : A$ corresponds to $x_i : B_i, .\overset{i}{.}. \vdash v : A \mid \alpha_j : C_j, .\overset{j}{.}..$
- $x_i : B_i, .\overset{i}{.}., \alpha_j \div C_j, .\overset{j}{.}. \vdash e \div A$ corresponds to $x_i : B_i, .\overset{i}{.}. \mid e : A \vdash \alpha_j : C_j, .\overset{j}{.}..$
- $x_i : B_i, .\overset{i}{.}., \alpha_j \div C_j, .\overset{j}{.}. \vdash c$ corresponds to $c : (x_i : B_i, .\overset{i}{.}. \vdash \alpha_j : C_j, .\overset{j}{.}.)$.

## 4 Corecursion in an abstract machine

Instead of coming up with an extension of System T with corecursion and then define an abstract machine, we start directly with the abstract machine which we obtain by applying duality. As a prototypical example of a coinductive type, we consider infinite streams of values, chosen for their familiarity (other coinductive types work just as well), which we represent by the type Stream $A$, as given in Fig. 7.

The intention is that Stream $A$ 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, Stream $A$ has two *destructors* for building covalues. First, the covalue head $E$ (the base case dual to zero) projects out the first element of its given stream and passes its value to $E$. Second, the covalue tail $E$ (the coinductive case dual to succ $V$) discards the first element of the stream and passes the remainder of the stream to $E$. The *corecursor* is defined by dualizing the recursor, whose general form is:

$$\mathbf{rec}\ \{\mathit{base\ case} \to v \mid \quad \mathit{inductive\ case} \to y.w\}\ \mathbf{with}\ E$$
$$\mathbf{corec}\ \{\mathit{base\ case} \to e \mid \mathit{coinductive\ case} \to \gamma.f\}\ \mathbf{with}\ V$$

Notice how the internal seed $V$ corresponds to the return continuation $E$. In the base case of the recursor, term $v$ is sent to the current value of the continuation $E$. Dually, in the base case of the corecursor, the coterm $e$ receives the current value of the internal seed $V$. In the recursor's inductive case, $y$ receives the result of the next recursive step (*i.e.,* the predecessor of the current one), whereas in the corecursor's coinductive case, $\gamma$ sends the updated seed to the next corecursive step (*i.e.,* the tail of the current one). The two cases of the recursor match the patterns zero (the base case) and succ $x$ (the inductive case). Analogously, the corecursor matches against the two possible copatterns: the base case is head $\alpha$, and the coinductive case is tail $\beta$. So the full form of the stream corecursor is:

$$\mathbf{corec}\ \{\mathrm{head}\, \alpha \to e \mid \mathrm{tail}\, \beta \to \gamma.f\}\ \mathbf{with}\ V$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

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

$$\mathit{Command} \ni c, d ::= \langle v \| e \rangle \quad \mathit{Term} \ni v, w ::= \mu\alpha.c \mid V \quad \mathit{CoTerm} \ni e, f ::= \tilde{\mu}x.c \mid E$$

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

$$\mathit{Value} \ni V, W ::= \mu\alpha.c \mid x \mid \lambda x.v \mid \mathrm{zero} \mid \mathrm{succ}\, V \mid \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ v$$
$$\mathit{CoValue} \ni E, F ::= \alpha \mid V \cdot E \mid \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\}\ \mathbf{with}\ E \mid \mathrm{head}\, E \mid \mathrm{tail}\, E$$

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

$$\mathit{Value} \ni V, W ::= x \mid \lambda x.v \mid \mathrm{zero} \mid \mathrm{succ}\, V \mid \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V$$
$$\mathit{CoValue} \ni E, F ::= \tilde{\mu}x.c \mid \alpha \mid V \cdot E \mid \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\}\ \mathbf{with}\ E \mid \mathrm{head}\, e \mid \mathrm{tail}\, e$$

*Operational rules:*

$$(\mu) \qquad \langle \mu\alpha.c \| E \rangle \mapsto c[E/\alpha]$$
$$(\tilde{\mu}) \qquad \langle V \| \tilde{\mu}x.c \rangle \mapsto c[V/x]$$
$$(\beta_{\to}) \qquad \langle \lambda x.v \| V \cdot E \rangle \mapsto \langle v[V/x] \| E \rangle$$
$$(\beta_{\mathrm{zero}}) \qquad \left\langle \mathrm{zero} \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ E\end{array} \right\rangle \mapsto \langle v \| E \rangle$$
$$(\beta_{\mathrm{succ}}) \qquad \left\langle \mathrm{succ}\, V \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ E\end{array} \right\rangle \mapsto \left\langle \mu\alpha. \left\langle V \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\, x \to y.w\} \\ \mathbf{with}\ \alpha\end{array} \right\rangle \,\middle\|\, \tilde{\mu}y.\langle w[V/x] \| E \rangle \right\rangle$$
$$(\beta_{\mathrm{head}}) \qquad \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\,\alpha \to e \\ \qquad \mid \mathrm{tail}\,\beta \to \gamma.f\} \\ \mathbf{with}\ V\end{array} \,\middle\|\, \mathrm{head}\, E \right\rangle \mapsto \langle V \| e[E/\alpha] \rangle$$
$$(\beta_{\mathrm{tail}}) \qquad \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\,\alpha \to e \\ \qquad \mid \mathrm{tail}\,\beta \to \gamma.f\} \\ \mathbf{with}\ V\end{array} \,\middle\|\, \mathrm{tail}\, E \right\rangle \mapsto \left\langle \mu\gamma.\langle V \| f[E/\beta] \rangle \,\middle\|\, \tilde{\mu}x. \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\,\alpha \to e \\ \qquad \mid \mathrm{tail}\,\beta \to \gamma.f\} \\ \mathbf{with}\ x\end{array} \,\middle\|\, E \right\rangle \right\rangle$$

Fig. 8. Uniform, (co)recursive abstract machine.

The uniform abstract machine is given in Fig. 8, where we highlight the extensions. Note how 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 head $\alpha \to e$ matching the head projection just passes the accumulator to $e$, which (may) compute the current element and send it to $\alpha$. The corecursive case tail $\beta \to \gamma.f$ also passes the accumulator to $f$, which may return an updated accumulator (through $\gamma$) *or* circumvent further corecursion by returning another stream directly to the remaining projection (via $\beta$). As with the syntax, the operational semantics is roughly symmetric to the rules for natural numbers, where roles of terms and coterms have been flipped.

*Intermezzo* 4.1. We review how the basic concepts of corecursion are reflected in our syntactic framework. Note how these observations are dual to the basic concepts of recursion.

– *With coinductive data types, covalues are constructed and the producer is a process that generates the data*. Observers of streams are constructed via the head and tail projections, and their construction is a process.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

- *Use of codata is finite and its creation is (potentially) infinite in the sense that there must be no limit to the size of the codata that a producer can process*. We can only build covalues from a finite number of destructor applications. However, the producer does not know how big of a request it will be given, so it has to be ready to handle codata structures of any size. In the end, termination is preserved because only finite covalues are consumed.
- *Corecursion produces the data, rather than using it*. **corec** is a term, not a coterm.
- *Corecursion starts from a seed and potentially produces bigger and bigger internal states*; the corecursor brakes down the codata structure and might end when the base case is reached.
- *The values of a codata structure potentially depend on each other*. The n-th value of a stream might depend on the value at the index $n$-1.

### 4.1 Examples of corecursion

The infinite streams $x, x, x, x, \ldots$ and $x, f\ x, f\ (f\ x), \ldots$, which are written in Haskell as

$$always\ x = x : always\ x \qquad\qquad repeat\ f\ x = x : repeat\ f\ (f\ x)$$

can be calculated by the abstract machine terms

$$always\ x = \mathbf{corec}\ \{\, \mathrm{head}\ \alpha \to \alpha \mid \mathrm{tail}\ \_ \to \gamma.\gamma \,\}\ \mathbf{with}\ x$$

$$repeat\ f\ x = \mathbf{corec}\ \{\, \mathrm{head}\ \alpha \to \alpha \mid \mathrm{tail}\ \_ \to \gamma.\tilde{\mu}x.\langle f \| x \cdot \gamma \rangle \,\}\ \mathbf{with}\ x$$

So when an observer asks *always* $x$ (*repeat* $f$ $x$) for its head element (matching the copattern $\mathrm{head}\,\alpha$), *always* $x$ (*repeat* $f$ $x$) returns (to $\alpha$) the current value of the seed $x$. Otherwise, when an observer asks for its tail (matching the copattern $\mathrm{tail}\,\beta$), *always* $x$ (*repeat* $f$ $x$) continues corecursing with the same seed $x$ ($\mu\gamma.\langle f \| x \cdot \gamma \rangle$).

The infinite streams containing all zeroes, and the infinite stream of all natural numbers counting up from 0, are then represented as:

$$zeroes = \mu\alpha.\langle always \| \mathrm{zero} \cdot \alpha \rangle$$
$$nats = \mu\alpha.\langle repeat \| succ \cdot \mathrm{zero} \cdot \alpha \rangle$$

where *succ* is defined as $\lambda x.\mu\alpha.\langle \mathrm{succ}\ x \| \alpha \rangle$.

To better understand execution, let's trace how computation works in both call-by-name and call-by-value. Asking for the third element of *zeroes* proceeds with the following calculation in call-by-value (we let $always_x$ stand for the stream with $x$ being the seed):

$$\begin{aligned}
&\langle zeroes \| \mathrm{tail}(\mathrm{tail}(\mathrm{head}\,\alpha)) \rangle \\
&\mapsto \langle always \| \mathrm{zero} \cdot \mathrm{tail}(\mathrm{tail}(\mathrm{head}\,\alpha)) \rangle && (\mu) \\
&\mapsto \langle always_{\mathrm{zero}} \| \mathrm{tail}(\mathrm{tail}(\mathrm{head}\,\alpha)) \rangle && (\beta_{\to}) \\
&\mapsto \langle \mu\gamma.\langle \mathrm{zero} \| \gamma \rangle \| \tilde{\mu}x.\langle always_x \| \mathrm{tail}(\mathrm{head}\,\alpha) \rangle \rangle && (\beta_{\mathrm{tail}}) \\
&\mapsto \langle \mathrm{zero} \| \tilde{\mu}x.\langle always_x \| \mathrm{tail}(\mathrm{head}\,\alpha) \rangle \rangle && (\mu) \\
&\mapsto \langle always_{\mathrm{zero}} \| \mathrm{tail}(\mathrm{head}\,\alpha) \rangle && (\tilde{\mu}) \\
&\mapsto\!\!\mapsto \langle always_{\mathrm{zero}} \| \mathrm{head}\,\alpha \rangle && (\beta_{\mathrm{tail}}\mu\tilde{\mu}) \\
&\mapsto \langle \mathrm{zero} \| \alpha \rangle && (\beta_{\mathrm{head}})
\end{aligned}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

In contrast, notice how the same calculation in call-by-name builds up a delayed computation in the seed:

$$
\begin{aligned}
&\langle \mathit{zeroes} \| \mathrm{tail}(\mathrm{tail}(\mathrm{head}\,\alpha)) \rangle \\
&\mapsto\!\!\!\!\mapsto \langle \mathit{always}_{\mathrm{zero}} \| \mathrm{tail}(\mathrm{tail}(\mathrm{head}\,\alpha)) \rangle && (\mu\beta_{\to}) \\
&\mapsto \langle \mu\gamma.\langle \mathrm{zero} \| \gamma \rangle \| \tilde{\mu}x.\langle \mathit{always}_x \| \mathrm{tail}(\mathrm{head}\,\alpha) \rangle \rangle && (\beta_{\mathrm{tail}}) \\
&\mapsto \langle \mathit{always}_{\mu\gamma.\langle \mathrm{zero} \| \gamma \rangle} \| \mathrm{tail}(\mathrm{head}\,\alpha) \rangle && (\tilde{\mu}) \\
&\mapsto\!\!\!\!\mapsto \langle \mathit{always}_{\mu\gamma'.\langle \mu\gamma.\langle \mathrm{zero} \| \gamma \rangle \| \gamma' \rangle} \| \mathrm{head}\,\alpha \rangle && (\beta_{\mathrm{tail}}\tilde{\mu}) \\
&\mapsto \langle \mu\gamma'.\langle \mu\gamma.\langle \mathrm{zero} \| \gamma \rangle \| \gamma' \rangle \| \alpha \rangle && (\beta_{\mathrm{head}}) \\
&\mapsto \langle \mu\gamma.\langle \mathrm{zero} \| \gamma \rangle \| \alpha \rangle && (\mu) \\
&\mapsto \langle \mathrm{zero} \| \alpha \rangle && (\mu)
\end{aligned}
$$

Consider the function that produces a stream that counts down from some initial number $n$ to 0, and then staying at 0. Informally, this function can be understood as:

$$
\begin{aligned}
&\mathit{countDown} : \mathrm{Nat} \to \mathrm{Stream\ Nat} \\
&\mathit{countDown}\ n = n, n-1, n-2, \ldots, 3, 2, 1, 0, 0, 0, \ldots
\end{aligned}
$$

which corresponds to the Haskell function

$$
\begin{aligned}
&\mathit{countDown}\ 0 = \mathit{zeroes} \\
&\mathit{countDown}\ n = n : \mathit{countDown}\ (n-1)
\end{aligned}
$$

It is formally defined in the abstract machine like so:

$$
\begin{aligned}
\mathit{countDown}\ n = \mathbf{corec}\ \{\ & \mathrm{head}\,\alpha \to \alpha \\
& \mid \mathrm{tail}\ \_ \to \gamma.\ \mathbf{rec}\ \{\ \mathrm{zero} \to \mathrm{zero} \\
& \qquad \mid \mathrm{succ}\ n \to n\}\ \mathbf{with}\ \gamma\} \\
\mathbf{with}\ n &
\end{aligned}
$$

This definition of *countDown* can be understood as follows:

- If the head of the stream is requested (matching the copattern $\mathrm{head}\,\alpha$), then the current value $x$ of the seed is returned (to $\alpha$) as-is.
- Otherwise, if the tail is requested (matching the copattern $\mathrm{tail}\ \_$), then the current value of the seed is inspected: if it is 0, then 0 is used again as the seed; otherwise, the seed is the successor of some $y$, in which case $y$ is given as the updated seed.

The previous definition of *countDown* is not very efficient; once $n$ reaches zero, one can safely returns the *zeroes* stream thus avoiding the test for each question. This can be avoided with the power of the corecursor as so:

$$
\begin{aligned}
\mathit{countDown}'\ n = \mathbf{corec}\ \{\ & \mathrm{head}\,\alpha \to \alpha \\
& \mid \mathrm{tail}\ \beta \to \gamma.\ \mathbf{rec}\ \{\ \mathrm{zero} \to \mu\_.\langle \mathit{zeroes} \| \beta \rangle \\
& \qquad \mid \mathrm{succ}\ n \to \gamma.n\}\ \mathbf{with}\ \gamma\} \\
\mathbf{with}\ n &
\end{aligned}
$$

Note that in the coinductive step, the decision on which continuation is taken (the observer of the tail $\beta$ or the continuation of corecursion $\gamma$) depends on the current value of the seed. If the seed reaches 0 the corecursion is stopped and the stream of zeros is returned instead.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Another example of this "switching" behavior, where one corecursive loop (like *countDown′ 0*) ends and switches to another corecursive loop (like *zeroes*) is the *switch0* function informally written as

$$\begin{aligned} \mathit{switch0}\ (x_0, x_1, \ldots, 0, x_{i+1}, \ldots)\ (y_0, y_1, \ldots) &= x_0, x_1, \ldots, 0, y_0, y_1, \ldots \quad (\text{if } \forall j < i . x_j \neq 0) \\ \mathit{switch0}\ (x_0, x_1, \ldots)\ (y_0, y_1, \ldots) &= x_0, x_1, \ldots \quad (\text{if } \forall j . x_j \neq 0) \end{aligned}$$

which corresponds to the Haskell code:

$$\begin{aligned} \mathit{switch0}\ (0 : \mathit{xs})\ \mathit{ys} &= 0 : \mathit{ys} \\ \mathit{switch0}\ (x : \mathit{xs})\ \mathit{ys} &= x : \mathit{switch0}\ \mathit{xs}\ \mathit{ys} \end{aligned}$$

Intuitively, *switch0 xs ys* will generate the stream produced by *xs* until a 0 is reached; at that point, the *switch0* loop ends by returning 0 followed by the whole stream *ys* as-is. This example is more interesting than *countDown* because there is no inductive argument (like a number or a list) that the function can inspect in advance to predict when the switch will occur. Instead, *switch0* must generate its elements on demand only when they are requested, and the switch to *ys* will only happen if an observer decides to look deep enough into the stream to hit an 0 inside *xs*, which might never even happen. *switch0* can be efficiently implemented in the abstract machine using both continuations provided by the corecursor like so:

$$\begin{aligned} \mathit{switch0}\ \mathit{xs}\ \mathit{ys} = \mathbf{corec}\{&\mathrm{head}\ \alpha \to \alpha \\ &\mid \mathrm{tail}\ \beta \to \gamma . \tilde{\mu} \mathit{xs} . \langle \mathit{xs} \| \mathrm{head}\ \mathbf{rec}\{\mathrm{zero} \to \mathit{ys} \\ &\qquad\qquad\qquad\qquad \mid \mathrm{succ}\ n \to \mu\_ . \langle \mathit{xs} \| \mathrm{tail}\ \gamma \rangle\} \\ &\qquad\qquad\qquad\qquad \mathbf{with}\ \beta \rangle\} \\ &\mathbf{with}\ \mathit{xs} \end{aligned}$$

The internal state to this corecursive loop contains the remainder of *xs* that hasn't been seen yet. While the *switch0* **corec** loop is active, it will always generate the head of the current remainder of *xs* as its own head element. When the tail is requested, this loop checks the head of the *xs* remainder to decide what to do:

- if it is zero, then the whole stream *ys* gets returned to $\beta$, which is the original caller who requested the tail of the stream, otherwise
- if it is succ *n*, the **corec** loop inside *switch0* will continue by returning the tail of the remaining *xs* to the continuation $\gamma$ which will update the internal state of the loop with one fewer element that has not yet been seen.

### *4.2 Properly de Morgan 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 tail *E* looks similar enough to succ *V*, the base cases of 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 Nat to Numbered *A* which represents a value of type *A* labeled with a natural number (also known as Burroni naturals). Numbered *A* has two constructors: the base case zero : *A* → Numbered *A* labels an *A* value

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

with the number 0, and $\mathrm{succ} : \mathrm{Numbered}\, A \to \mathrm{Numbered}\, A$ increments the numeric label while leaving the $A$ value alone, as shown by the following rules:

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

In order to match the generalized zero constructor, the base case of the recursor needs to be likewise generalized with an extra parameter. In System T, this looks like $\mathbf{rec}\, M\, \mathbf{as}\, \{\mathrm{zero}\, x \to N \mid \mathrm{succ}\, y \to z.N'\}$, while in the abstract machine we get the generalized continuation $\mathbf{rec}\, \{\mathrm{zero}\, x \to v \mid \mathrm{succ}\, y \to z.w\}\, \mathbf{with}\, E$. The typing rule for this continuation is:

$$\frac{\Gamma, x : A \vdash v : B \quad \Gamma, y : \mathrm{Numbered}\, A, z : B \vdash w : B \quad \Gamma \vdash E \div B}{\Gamma \vdash \mathbf{rec}\, \{\mathrm{zero}\, x \to v \mid \mathrm{succ}\, y \to z.w\}\, \mathbf{with}\, E \div \mathrm{Numbered}\, A}$$

It turns out that $\mathrm{Numbered}\, A$ *is* the proper de Morgan dual to $\mathrm{Stream}\, A$. Notice how the two constructors $\mathrm{zero}\, V$ and $\mathrm{succ}\, W$ exactly mirror the two destructors $\mathrm{head}\, E$ and $\mathrm{tail}\, F$ when we swap the roles of values and covalues. The recursor continuation of the form $\mathbf{rec}\, \{\mathrm{zero}\, x \to v \mid \mathrm{succ}\, y \to z.w\}\, \mathbf{with}\, E$ is the perfect mirror image of the stream corecursor $\mathbf{corec}\, \{\mathrm{head}\, \alpha \to e \mid \mathrm{tail}\, \beta \to \gamma.f\}\, \mathbf{with}\, V$ when we likewise swap variables with covariables in the (co)patterns. In more detail, we can write the duality relation between values and covalues as $\between$. Assuming that $V \between E$, we have the following duality between the constructors and destructors of these two types:

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

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

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

We could also express the proper de Morgan 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 no (closed) values, and a single covalue that represents an empty continuation. With this in mind, the type Nat is properly dual to $\mathrm{Stream}\, \bot$, *i.e.,* an infinite stream of computations which cannot return any value to their observer.[8]

In order to fully realize this de Morgan duality in an interesting way that can talk about functions, we need to refer to the duals of those functions. Logically, the dual to the function type $A \to B$ (classical equivalent to $(\neg A) \vee B$ and $\neg(A \wedge (\neg B))$, whose continuations $V \cdot E$ contain an argument $V$ of type $A$ paired with a consumer $E$ of $B$s) is a *subtraction* type $B - A$ (classically equivalent to $B \wedge (\neg A)$ for typing a pair $E \cdot V$ of a value $V$ of type $B$ and a continuation $E$ expecting $A$s). Following (Curien & Herbelin, 2000), this subtraction type, along with the units $\top$ and $\bot$, can be added to our uniform abstract machine with the following extended syntax of (co)values, reduction rule $\beta_-$ for when a subtraction pair $E \cdot V$ interacts with the continuation abstraction $\tilde{\lambda}\alpha.e$, and typing rules:

[8] Considering polarity in programming languages (Zeilberger, 2009; Munch-Maccagnoni, 2013), the $\top$ type for truth we use in "Numbered $\top$" should be interpreted as a positive type (written as 1 in linear logic (Girard, 1987)). Dually, the $\bot$ type for falsehood in "Stream $\bot$" is a negative type (also called $\bot$ in linear logic).

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Duality inversion of types $\boxed{A^\perp}$:

$$(\text{Numbered}\, A)^\perp := \text{Stream}(A^\perp) \qquad (\text{Stream}\, A)^\perp := \text{Numbered}(A^\perp)$$
$$(A \to B)^\perp := B^\perp - A^\perp \qquad (A - B)^\perp := B^\perp \to A^\perp$$
$$\top^\perp := \bot \qquad \bot^\perp := \top$$

Duality inversion of typing contexts $\boxed{\Gamma^\perp}$

$$\bullet^\perp := \bullet \qquad (\Gamma, x : A)^\perp := \Gamma^\perp, x^\perp \div A^\perp \qquad (\Gamma, \alpha \div A)^\perp := \Gamma^\perp, \alpha^\perp : A^\perp$$

Duality inversion of terms $\boxed{v^\perp}$, coterms $\boxed{e^\perp}$ and commands $\boxed{c^\perp}$

$$(\mu\alpha.c)^\perp := \tilde{\mu}\alpha^\perp.c^\perp \qquad (\tilde{\mu}x.c)^\perp := \mu x^\perp.c^\perp$$
$$(\text{zero}\, V)^\perp := \text{head}(V^\perp) \qquad (\text{head}\, E)^\perp := \text{zero}(E^\perp)$$
$$(\text{succ}\, V)^\perp := \text{tail}(V^\perp) \qquad (\text{tail}\, E)^\perp := \text{succ}(E^\perp)$$
$$(\lambda x.v)^\perp := \tilde{\lambda}x^\perp.v^\perp \qquad (\tilde{\lambda}\alpha.e)^\perp := \lambda\alpha^\perp.e^\perp$$
$$(e \cdot v)^\perp := e^\perp \cdot v^\perp \qquad (v \cdot e)^\perp := v^\perp \cdot e^\perp$$
$$()^\perp := [] \qquad []^\perp := ()$$
$$(\mathbf{corec}\{\text{head}\, \alpha \to e \mid \text{tail}\, \beta \to \gamma.f\}\ \mathbf{with}\ V)^\perp := \mathbf{rec}\{\text{zero}\, \alpha^\perp \to e^\perp \mid \text{succ}\, \beta^\perp \to \gamma^\perp.f^\perp\}\ \mathbf{with}\ V^\perp$$
$$(\mathbf{rec}\{\text{zero}\, x \to v \mid \text{succ}\, y \to z.w\}\ \mathbf{with}\ E)^\perp := \mathbf{corec}\{\text{head}\, x^\perp \to v^\perp \mid \text{tail}\, y^\perp \to z^\perp.w^\perp\}\ \mathbf{with}\ E^\perp$$
$$\langle v \| e \rangle^\perp := \langle e^\perp \| v^\perp \rangle$$

Fig. 9. Duality of types, (co)terms, and commands in the uniform abstract machine.

$$V ::= \cdots \mid () \mid V \cdot E \qquad e ::= \cdots \mid [] \mid \tilde{\lambda}\alpha.e \qquad \langle E \cdot V \| \tilde{\lambda}\alpha.e \rangle \mapsto \langle V \| e[E/\alpha] \rangle \qquad (\beta_-)$$

$$\frac{}{\Gamma \vdash () : \top}\ \top R \qquad \frac{}{\Gamma \vdash [] \div \bot}\ \bot L$$

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash E \div B}{\Gamma \vdash E \cdot V : A - B}\ -R \qquad \frac{\Gamma, \alpha \div B \vdash e \div A}{\Gamma \vdash \tilde{\lambda}\alpha.e \div A - B}\ -L$$

Extending the uniform abstract machine with these new rules, lets us formally define a de Morgan duality transformation, shown in Fig. 9, that converts any term $v$ producing type $A$ into the coterm $v^\perp$ consuming type $A^\perp$, converts any coterm $e$ expecting type $A$ to a term $e^\perp$ giving type $A^\perp$, and converts a command $c$ to its dual command $c^\perp$. The duality transformation assumes a bijection between variables and covariables (so that $x^\perp$ denotes a unique covariable $\alpha$ such that $\alpha^\perp = x$, and vice versa), which makes the whole transformation involutive (*e.g.,* $c^{\perp\perp} = c$ and $A^{\perp\perp} = A$) by definition. This generalizes the previous duality theorems for the classical sequent calculus (Curien & Herbelin, 2000; Wadler, 2003)—namely that involutive duality inverts typing and operational semantics in a meaningful way—to also incorporate (co)recursion on numbers and streams.

**Theorem 4.2** (Duality)**.** *In the fully dual abstract machine (the uniform abstract machine extended with subtraction $A - B$ and units $\top$ and $\bot$), the following symmetries hold:*

1. $\Gamma \vdash v : A$ *is derivable if and only if* $\Gamma^\perp \vdash v^\perp \div A^\perp$ *is.*
2. $\Gamma \vdash c$ *is derivable if and only if* $\Gamma^\perp \vdash c^\perp$ *is.*

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

3. *$v$ is a call-by-value value if and only if $e^\perp$ is a call-by-name covalue.*
4. *$c \mapsto c'$ in call-by-name if and only if $c^\perp \mapsto c'^\perp$ in call-by-value.*

The proof of Theorem 4.2 just follows by induction on the given typing derivation or reduction rule and is an application of the duality theorem of general data and codata types (Downen, 2017), extended with the new form of primitive (co)recursor.

*Intermezzo* 4.3. The reader already familiar with other work on streams and duality, who has heard that finite lists are "dual" to (possibly) infinite streams, might be surprised to see the statement in Fig. 9 and Theorem 4.2 that streams are dual to (a generalization of) natural numbers. How can this be? The conflict is in two different meanings of the word "dual."

The meaning of "dual" we use here corresponds exactly to the usual notion of de Morgan duality from classical logic. De Morgan duality flips between "true" ($\top$) and "false" ($\bot$), and between "and" ($A \times B$) and "or" ($A + B$), so that the dual of a proposition $A$ is $A^\perp$ (logically equivalent to $\neg A$) defined like so:

$$\top^\perp := \bot \qquad \bot^\perp := \top \qquad (A + B)^\perp := (A^\perp) \times (B^\perp) \qquad (A \times B)^\perp := (A^\perp) + (B^\perp)$$

Importantly, notice how the duality transformation is *deep*: $A^\perp$ does not just flip the top connective, it flips *all* the connectives down to the leaves. For example, the proposition dual to $(\top + \bot) \times \top$ (which is tautologically *true*) is $((\top + \bot) \times \top)^\perp = (\bot \times \top) + \bot$ (which denotes a *falsehood*, the dual to *truth*) and *not* the erroneous illogical inequality $((\top + \bot) \times \top)^\perp \neq (\top + \bot) + \top$ (which still tautology true, and *not* the dual).

Previous work (Curien & Herbelin, 2000; Wadler, 2003) has showed how this duality of propositions in a logic can naturally extend to a duality of types in a programming language based on the classical sequent calculus. Because we are interested in (co)inductive types here, we also need to consider how this relates to recursion in types. As usual, inductive types like Nat are modeled as a *least fixed point*, that we write as $\mathrm{LFP}\, X.A$, whereas coinductive types like $\mathrm{Stream}\, A$ are modeled as a *greatest fixed point*, that we write as $\mathrm{GFP}\, X.A$.[9] A key aspect of this paper is to effectively generalize the classical de Morgan duality of Curien & Herbelin (2000); Wadler (2003) to the known dualities of least and greatest fixed points:

$$(\mathrm{LFP}\, X.A)^\perp := \mathrm{GFP}\, X.(A^\perp) \qquad\qquad (\mathrm{GFP}\, X.A)^\perp := \mathrm{LFP}\, X.(A^\perp)$$

The $\mathrm{Stream}\, A$, Nat, and $\mathrm{Numbered}\, A$ types we have seen thus far are isomorphic (denoted as $\approx$) to these usual encodings in terms of greatest and least fixed points built on top of basic type constructors like $+$ and $\times$:

$$\mathrm{Stream}\, A \approx \mathrm{GFP}\, X.\, A \times X \qquad \mathrm{Nat} \approx \mathrm{LFP}\, X.\, \top + X \qquad \mathrm{Numbered}\, A \approx \mathrm{LFP}\, X.\, A + X$$

By applying the usual de Morgan duality of classical logic along with the duality of greatest and least fixed points above, we can calculate these duals of streams and natural numbers:

$$(\mathrm{Stream}\, A)^\perp \approx \mathrm{LFP}\, X.\, A^\perp + X \approx \mathrm{Numbered}(A^\perp) \qquad \mathrm{Nat}^\perp \approx \mathrm{GFP}\, X.\, \bot \times X \approx \mathrm{Stream}\, \bot$$

[9] Traditionally, the least fixed point of a type function $F$ is denoted by a $\mu$ as in $\mu X.F(X)$, and the greatest fixed point of a type function $F$ is commonly denoted by a $\nu$ as in $\nu X.F(X)$. Because we have already reserved $\mu$ to denote the binder for continuations in a term, à la (Curien & Herbelin, 2000), we will use the more cumbersome notation LFP and GFP for the purposes of this short intermezzo to avoid confusion.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Contrastingly, other work on (co)inductive types instead calls two types "duals" by just flipping between the greatest or fixed point at the top of the top, but *leaving everything else alone*. In other words, this other literature says that the dual of $\mathrm{LFP}\, X.A$ is just $\mathrm{GFP}\, X.A$, *instead of* $\mathrm{GFP}\, X.(A^{\perp})$. In comparison, this is a *shallow* notion of duality that does not look deeper into the type. The typical working example of this notion of duality is that the inductive type of lists ($\mathrm{List}\, A$) is dual to the coinductive type of *possibly infinite or finite* lists ($\mathrm{InfList}\, A$), modeled like so:

$$\mathrm{List}\, A \approx \mathrm{LFP}\, X.\ \top + (A \times X) \qquad\qquad \mathrm{InfList}\, A \approx \mathrm{GFP}\, X.\ \top + (A \times X)$$

This other notion of duality, which only swaps a greatest for a list fixed point, is closest to the work on corecursion in lazy functional languages like Haskell, where $\mathrm{InfList}\, A$ represents Haskell's usual list type `[a]`. The work has undoubtedly been very productive in showing how to solve practical programming problems involving infinite data in the context of lazy functional languages, but covers different ground than our main objective. For example, shallow duality that just swaps the top-most fixed point—and ignores the rest of the underlying type—fails at providing an involutive duality of types and computation like the one shown in Theorem 4.2 and Fig. 9. Intuitively, $(\mathrm{List}\, A)^{\perp} \neq \mathrm{InfList}\, A$ because a finite list of $A$s looks nothing like the continuation expecting a (possibly infinite) list of $A$, and vice versa!

## 5 Corecursion versus coiteration: Expressiveness and efficiency

Similar to recursion, we can define two special cases of corecursion which only use part of its functionality by just ignoring a parameter in the corecursive branch. We derive the encodings for the creation of streams by applying the syntactic duality to the encodings presented in Section 3.5:

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

Specifically, **cocase** simply matches on the shape of its projection, which has the form $\mathrm{head}\, \alpha$ or $\mathrm{tail}\, \beta$, without corecursing at all. In contrast, **coiter** *always* corecurses by providing an updated accumulator in the $\mathrm{tail}\, \beta$ case without ever referring to $\beta$. We have seen already several examples of coiteration. Indeed, all examples given in the previous section, except *countDown*′, are examples of coiteration, as indicated by the absence of the rest of the continuation (named _). From the perspective of logic, coiteration can be seen as *intuitionistic* because it only ever involves one continuation (corresponding to one conclusion) at a time. In contrast, the most general form of the corecursor is inherently *classical* because the corecursive step $\mathrm{tail}\, \beta \to \gamma.f$ introduces two continuations $\beta$ and $\gamma$ (corresponding to two different conclusion) at the same time.

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. Indeed, in call-by-value the simple cocase continuation avoids the

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

corecursive step entirely in two steps:

$$\begin{aligned}
&\langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to f\}\ \mathbf{with}\ V \| \mathrm{tail}\,E\rangle \\
&:= \langle \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to _.f\}\ \mathbf{with}\ V \| \mathrm{tail}\,E\rangle \\
&\mapsto \langle \mu_.\langle V \| f[E/\beta]\rangle \| \tilde{\mu}x.\langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to f\}\ \mathbf{with}\ x \| E\rangle\rangle && (\beta_{\mathrm{tail}}) \\
&\mapsto \langle V \| f[E/\beta]\rangle && (\mu)
\end{aligned}$$

whereas cocase continues corecursing in call-by-name, despite the fact that this work will ultimately be thrown away once any element is requested via head:

$$\begin{aligned}
&\langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to f\}\ \mathbf{with}\ V \| \mathrm{tail}\,E\rangle \\
&:= \langle \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to _.f\}\ \mathbf{with}\ V \| \mathrm{tail}\,E\rangle \\
&\mapsto \langle \mu_.\langle V \| f[E/\beta]\rangle \| \tilde{\mu}x.\langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to f\}\ \mathbf{with}\ x \| E\rangle\rangle && (\beta_{\mathrm{tail}}) \\
&\mapsto \langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to f\}\ \mathbf{with}\ \mu_.\langle V \| f[E/\beta]\rangle \| E\rangle && (\tilde{\mu})
\end{aligned}$$

As an example of cocase, consider the following stream:

$$scons\ x\ (y_1, y_2, \dots) = x, y_1, y_2, \dots$$

*scons x s* appends a new element $x$ on top of the stream $s$. This informal definition can be formalized in terms of **cocase** like so:

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

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, where we write $\mathrm{tail}^n\,E$ to mean the $n$-fold application of tail over $E$ (*i.e.,* $\mathrm{tail}^0\,E = E$ and $\mathrm{tail}^{n+1}\,E = \mathrm{tail}(\mathrm{tail}^n\,E)$):

$$\begin{aligned}
&\langle scons \| x \cdot s \cdot \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \\
&\mapsto\!\!\mapsto \langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail}\,\beta \to \tilde{\mu}_.\langle s \| \beta\rangle\}\ \mathbf{with}\ x\} \| \mathrm{tail}(\mathrm{tail}^n(\mathrm{head}\,\alpha))\rangle && (\beta_{\to}) \\
&\mapsto\!\!\mapsto \langle x \| \tilde{\mu}_.\langle s \| \mathrm{tail}^n(\mathrm{head}\,\alpha)\rangle\rangle && (\beta_{\mathrm{tail}}\mu) \\
&\mapsto \langle s \| \mathrm{tail}^n(\mathrm{head}\,\alpha)\rangle && (\tilde{\mu})
\end{aligned}$$

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:

$$\begin{aligned}
&\langle scons \| x \cdot s \cdot \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \\
&\mapsto\!\!\mapsto \langle \mathbf{cocase}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail}\,\beta \to \tilde{\mu}_.\langle s \| \beta\rangle\}\ \mathbf{with}\ x\} \| \mathrm{tail}(\mathrm{tail}^n(\mathrm{head}\,\alpha))\rangle && (\beta_{\to}) \\
&\mapsto\!\!\mapsto \langle \mathbf{cocase}\{\dots\}\ \mathbf{with}\ \mu_.\langle x \| \tilde{\mu}_.\langle s \| \mathrm{tail}^n(\mathrm{head}\,\alpha)\rangle\rangle \| \mathrm{tail}^n(\mathrm{head}\,\alpha)\rangle && (\beta_{\mathrm{tail}}\tilde{\mu})
\end{aligned}$$

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 backtrack to directly ask $s$ for $\mathrm{tail}^n\,E$.

However, one question remains: how do we accurately measure the cost of a stream? The answer is more subtle than the cost of numeric loops, because streams more closely resemble functions. With functions, it is not good enough to just count the steps it takes to calculate the closure. We also need to count the steps taken in the body of the function when it is called, *i.e.,* what happens when the function is used. The same issue occurs

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

with streams, where we also need to count what happens when the stream is used, *i.e.,* the number of steps taken inside the stream in response to a projection. Of course, the internal number of steps in both cases can depend on the "size" of the input. For functions, this is the size of its argument; in the simple case of $\mathrm{Nat} \to \mathrm{Nat}$, this size is just the value $n$ of the numeric argument $\mathrm{succ}^n\,\mathrm{zero}$ itself. For streams, the input is the stream projection of the form $\mathrm{tail}^n(\mathrm{head}\,\alpha)$, whose size is proportional to the number $n$ of tail projections. Therefore, the computational complexity of a stream value $s$—perhaps defined by a **corec** term—is expressed as some $O(f(n))$, where $n$ denotes the depth of the projection given by the number of tails.

Now, let us consider the asymptotic cost of *scons* under both call-by-value and call-by-name evaluation. In general, the difference in performance between $\langle \mathit{scons} \| x \cdot s \cdot E \rangle$ and $\langle s \| E \rangle$ in call-by-value is just a constant time ($O(1)$) overhead. So, given that the cost of $s$ is $O(f(n))$, then the cost of $\langle \mathit{scons} \| x \cdot s \cdot E \rangle$ will also be $O(f(n))$. In contrast, the call-by-name evaluation of *scons* incurs an additional linear time ($O(n)$) overhead based on depth of the projection: $\langle \mathit{scons} \| x \cdot s \cdot \mathrm{tail}^{n+1}(\mathrm{head}\,E) \rangle$ takes an additional number of steps proportional to $n$ compared to the cost of executing $\langle s \| \mathrm{tail}^n(\mathrm{head}\,E) \rangle$. As a consequence, the call-by-name cost of $\langle \mathit{scons} \| x \cdot s \cdot E \rangle$ is $O(n + f(n))$, given that the cost of $s$ is $O(f(n))$. So the efficiency of corecursion is better in call-by-value than in call-by-name.

To make the analysis more concrete, let's look at an application of *scons* to a specific underlying stream. Recall the *countDown* function from Section 4.1 which produces a stream counting down from a given $n$: $n, n-1, \ldots, 2, 1, 0, 0, \ldots$. This function (and the similar *countDown′*) is defined to immediately return a stream value that captures the starting number $n$, and then incrementally counts down one at a time with each tail projection. An alternative method of generating this same stream is to do all the counting up-front: recurse right away on the starting number and use *scons* to piece together the stream from each step towards zero. This algorithm can be expressed in terms of System T-style recursion like so:

$$\begin{aligned} \mathit{countNow}\ n = \mathbf{rec}\ n\ \mathbf{as}\ \{\ & \mathrm{zero} \to \mathit{zeroes} \\ & \mid \mathrm{succ}\ x \to \mathit{xs}.\ \mathit{scons}\ (\mathrm{succ}\ x)\ \mathit{xs}\} \end{aligned}$$

So that the translation of *countNow* to the abstract machine language is:

$$\mathit{countNow} = \lambda n.\mu\alpha.\left\langle n \,\middle\|\, \begin{array}{l} \mathbf{rec}\ \{\ \mathrm{zero} \to \mathit{zeroes} \\ \qquad \mid \mathrm{succ}\ x \to \mathit{xs}.\mu\beta.\langle \mathit{scons} \| \mathrm{succ}\ x \cdot \mathit{xs} \cdot \beta \rangle\} \\ \mathbf{with}\ \alpha \end{array} \right\rangle$$

What is the cost of *countNow*? First, the up-front cost of calling the function with the argument $\mathrm{succ}^n\,\mathrm{zero}$ is unsurprisingly $O(n)$, due to the use of the recursor. But this doesn't capture the full cost; what is the efficiency of using the stream value returned by *countNow*? To understand the efficiency of the stream, we have to consider *both* the initial numeric argument as well as the depth of the projection: $\langle \mathit{countNow} \| \mathrm{succ}^n\,\mathrm{zero} \cdot \mathrm{tail}^m(\mathrm{head}\,\alpha) \rangle$ in the abstract machine language, which corresponds to $\mathrm{head}(\mathrm{tail}^m(\mathit{countNow}\ (\mathrm{succ}^n\,\mathrm{zero})))$ in a functional style. In the base case, we have the stream *zeroes*, whose cost is $O(m)$ because it must traverse past all $m$ tail projections before it can return a 0 to the head projection. On top of this, we apply $n$ applications of *scons*. Recall that in call-by-value, we said that the cost of *scons* is the same as the underlying stream. Thus, the call-by-value

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

efficiency of the stream returned by $\langle countNow \| \text{succ}^n \text{ zero} \cdot \text{tail}^m(\text{head}\,\alpha)\rangle$ is just $O(m)$. In call-by-name in contrast, *scons* adds an additional linear time overhead to the underlying stream. Since there are $n$ applications of *scons* and the projection is $m$ elements deep, the call-by-name efficiency of the stream returned by $\langle countNow \| \text{succ}^n \text{ zero} \cdot \text{tail}^m(\text{head}\,\alpha)\rangle$ is $O(m(n+1))$. If the count $n$ and depth $m$ are roughly proportional to each other (so that $n \approx m$), the difference between call-by-value and call-by-name evaluation of *countNow*'s stream is a jump between a linear time and quadratic time computation.

### 5.1 Corecursion in terms of coiteration

Recall in Section 3.5 that we were able to encode **rec** in terms of **iter**, though at an increased computational cost. Applying syntactic duality, we can write a similar encoding of **corec** as a macro-expansion in terms of **coiter**. Doing so requires the dual of pairs: sum types. Sum types in the abstract machine look like (Wadler, 2003):

$$\langle \text{left}\, V \| [e_1, e_2]\rangle \mapsto \langle V \| e_1\rangle \qquad \langle \text{right}\, V \| [e_1, e_2]\rangle \mapsto \langle V \| e_2\rangle \qquad (\beta_+)$$

The macro-expansion for encoding **corec** as **coiter** derived from duality is:

$$\begin{array}{lll} \textbf{corec} & \{\,\text{head}\,\alpha \to e & \textbf{coiter} \;\{\,\text{head}\,\alpha \to [\text{head}\,\alpha, e] \\ & \mid \text{tail}\,\beta \;\to \gamma.f\} := & \quad\;\; \mid \text{tail} \quad\;\; \to [\beta, \gamma].[\text{tail}\,\beta, f]\} \\ \textbf{with}\; V & & \textbf{with}\; \text{right}\, V \end{array}$$

Note how the coinductive step of **coiter** has access to both an option to update the internal seed, or to return some other, fully formed stream as its tail. So dually to the encoding of the recursor, this encoding keeps both options open by reconstructing the original continuation alongside the continuation which uses its internal seed. In the base case matching $\text{head}\,\alpha$, it rebuilds the observation $\text{head}\,\alpha$ and pairs it up with the original response $e$ to the base case. In the coinductive case matching $\text{tail}\,\beta$, the projection $\text{tail}\,\beta$ is combined with the continuation $f$ which can update the internal seed via $\gamma$. Since $f$ might refer to one or both of $\beta$ and $\gamma$, we need to "extract" the two parts from the corecursive tail observation. Above, we express this extraction as copattern matching (Abel *et al.*, 2013), which is shorthand for

$$[\beta, \gamma].[e, f] := \alpha.\tilde{\mu}x.\langle \text{left}\,\mu\beta.\langle \text{right}\,\mu\gamma.\langle x \| [e, f]\rangle \| \alpha\rangle \| \alpha\rangle$$

Because this encoding is building up a pair of two continuations, the internal seed which is passed to them needs to be a value of the appropriately matching sum type. Thus, the encoding has two modes:

- If the seed has the form $\text{right}\,x$, then the **coiter** is simulating the original **corec** process with the seed $x$. This is because the right continuations for the base and coinductive steps are exactly those of the encoded recursor ($e$ and $f$, respectively) which get applied to $x$.
- If the seed has the form $\text{left}\,s$, containing some stream $s$, then the **coiter** is mimicking $s$ as-is. This is because the left continuations for the base and coinductive steps exactly mimic the original observations ($\text{head}\,\alpha$ and $\text{tail}\,\beta$, respectively) which get applied to the stream $s$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

To start the corecursive process off, we begin with right $V$, which corresponds to the original seed $V$ given to the corecursor. Then in the coinductive step, if $f$ updates the seed to $V'$ via $\gamma$, the seed is really updated to right $V'$, continuing the simulation of corecursion. Otherwise, if $f$ returns some other stream $s$ to $\beta$, the seed gets updated to left $s$, and the coiteration continues but instead mimics $s$ at each step.

As with recursion, encoding **corec** in terms of **coiter** forces a performance penalty for functions like *scons* which can return a stream directly in the coinductive case instead of updating the accumulator. Recall how call-by-value execution was more efficient than call-by-name. Yet, the above encoding results in this same inefficient execution even in call-by-value, as in this command which accesses the $(n+2)^{th}$ element:

$$
\begin{aligned}
&\langle \mathit{scons} \| x \cdot s \cdot \mathrm{tail}^{n+2}(\mathrm{head}\,\alpha)\rangle \\
&\mapsto\!\!\!\!\mapsto \left\langle \begin{array}{l} \textbf{corec}\,\{\,\mathrm{head}\,\alpha \to \alpha \\ \qquad\quad \mid \mathrm{tail}\,\beta \to \_.\tilde{\mu}\_.\langle s \| \beta\rangle\} \\ \quad \textbf{with}\, x \end{array} \middle\| \mathrm{tail}^{n+2}(\mathrm{head}\,\alpha) \right\rangle && (\beta_{\to}) \\
&:= \left\langle \begin{array}{l} \textbf{coiter}\,\{\,\mathrm{head}\,\alpha \to [\mathrm{head}\,\alpha, \alpha] \\ \qquad\quad \mid \mathrm{tail} \to [\beta, \_].[\mathrm{tail}\,\beta, \tilde{\mu}\_.\langle s \| \beta\rangle]\} \\ \quad \textbf{with}\, \mathrm{right}\, x \end{array} \middle\| \mathrm{tail}(\mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)) \right\rangle \\
&\mapsto\!\!\!\!\mapsto \langle \mathrm{right}\, x \| [\mathrm{tail}\,\tilde{\mu}y.\langle \mathrm{left}\, y \| \gamma\rangle, \tilde{\mu}\_.\langle s \| \tilde{\mu}y.\langle \mathrm{left}\, y \| \gamma\rangle\rangle]\rangle && (\beta_{\mathrm{tail}}, \ldots) \\
&\quad\ \textbf{where}\ \gamma = \tilde{\mu}z.\langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ z \| \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle \\
&\mapsto \langle x \| \tilde{\mu}\_.\langle s \| \tilde{\mu}y.\langle \mathrm{left}\, y \| \tilde{\mu}z.\langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ z \| \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle\rangle\rangle\rangle && (\beta_{+}) \\
&\mapsto\!\!\!\!\mapsto \langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ \mathrm{left}\, s \| \mathrm{tail}^{n+1}(\mathrm{head}\,\alpha)\rangle && (\tilde{\mu}) \\
&\mapsto\!\!\!\!\mapsto \langle \mathrm{left}\, s \| [\mathrm{tail}\,\tilde{\mu}y.\langle \mathrm{left}\, y \| \gamma\rangle, \tilde{\mu}\_.\langle s \| \tilde{\mu}y.\langle \mathrm{left}\, y \| \gamma\rangle\rangle]\rangle && (\beta_{\mathrm{tail}}, \ldots) \\
&\quad\ \textbf{where}\ \gamma = \tilde{\mu}z.\langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ z \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle \\
&\mapsto \langle s \| \mathrm{tail}\,\tilde{\mu}y.\langle \mathrm{left}\, y \| \tilde{\mu}z.\langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ z \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle\rangle\rangle && (\beta_{+})
\end{aligned}
$$

Notice how the internal seed (in blue) changes through this computation. To begin, the seed is right $x$. The first tail projection (triggering the $\beta_{\mathrm{tail}}$ rule) leads to the decision point (by $\beta_{+}$) which chooses to update the seed with left $s$. From that point on, each tail projection to follow will trigger the next step of this coiteration (and another $\beta_{\mathrm{tail}}$ rule). Each time, this will end up asking $s$ for its tail, $s_1$, which will be then used to build the next seed, left $s_1$.

In order to continue, we need to know something about $s$, specifically, how it responds to a tail projection. For simplicity, assume that the tail of $s$ is $s_1$, *i.e.,* $\langle s \| \mathrm{tail}\, E\rangle \mapsto\!\!\!\!\mapsto \langle s_1 \| E\rangle$. And then for each following $s_i$, assume its tail is $s_{i+1}$. Under this assumption, execution will proceed to the base case head projection like so:

$$
\begin{aligned}
&\langle s \| \mathrm{tail}\,\tilde{\mu}y.\langle \mathrm{left}\, y \| \tilde{\mu}z.\langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ z \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle\rangle\rangle \\
&\mapsto\!\!\!\!\mapsto \langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ \mathrm{left}\, s_1 \| \mathrm{tail}^{n}(\mathrm{head}\,\alpha)\rangle && (\mathrm{tail}\, s \mapsto\!\!\!\!\mapsto s_1) \\
&\mapsto\!\!\!\!\mapsto \langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ \mathrm{left}\, s_2 \| \mathrm{tail}^{n-1}(\mathrm{head}\,\alpha)\rangle && (\mathrm{tail}\, s_1 \mapsto\!\!\!\!\mapsto s_2) \\
&\mapsto\!\!\!\!\mapsto \ldots && (\mathrm{tail}\, s_i \mapsto\!\!\!\!\mapsto s_{i+1}) \\
&\mapsto\!\!\!\!\mapsto \langle \textbf{coiter}\{\ldots\}\ \textbf{with}\ \mathrm{left}\, s_{n+1} \| \mathrm{head}\,\alpha\rangle && (\mathrm{tail}\, s_n \mapsto\!\!\!\!\mapsto s_{n+1}) \\
&\mapsto \langle \mathrm{left}\, s_{n+1} \| [\mathrm{head}\,\alpha, \alpha]\rangle && (\beta_{\mathrm{head}}) \\
&\mapsto \langle s \| \mathrm{head}\,\alpha\rangle && (\beta_{+})
\end{aligned}
$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

In total, this computation incurs additional $\beta_{\text{tail}}$ steps linearly proportional to $n + 2$, on top of any additional work needed to compute the same number of tail projections for each $\langle s_i \| \text{tail } E \rangle \mapsto\!\!\!\!\rightarrow \langle s_{i+1} \| E \rangle$ along the way.

## 6 Safety and termination

Just like System T Theorem 3.2, the (co)recursive abstract machine is both *type safe* (meaning that well-typed commands never get stuck) and *terminating* (meaning that well-typed commands always cannot execute forever). In order to prove this fact, we can build a model of type safety and termination rooted in the idea of Girard's reducibility candidates (Girard *et al.*, 1989), but which matches closer to the structure of the abstract machine. In particular, we will use a model from Downen *et al.* (2020, 2019) which first identifies a set of commands that are *safe* to execute (in our case, commands which terminate on a valid final configuration). From there, types are modeled as a combination of terms and coterms that embed this safety property of executable commands.

### *6.1 Safety and candidates*

To begin, we derive our notion of safety from the conclusion of Theorem 3.2. Ultimately we will only run commands that are closed, save for one free covariable $\alpha$ taking a Nat, so we expect all such *safe* commands to eventually finish execution in a valid final state that provides that $\alpha$ with a Nat construction: either a zero or successor.

**Definition 6.1** (Safety)**.** The set of *safe* commands, $\perp\!\!\!\perp$, is:

$$\perp\!\!\!\perp := \{c \mid \exists c' \in \mathit{Final}.\ c \mapsto\!\!\!\!\rightarrow c'\}$$
$$\mathit{Final} := \{\langle \text{zero} \| \alpha \rangle \mid \alpha \in \mathit{CoVar}\} \cup \{\langle \text{succ } V \| \alpha \rangle \mid V \in \mathit{Value}, \alpha \in \mathit{CoVar}\}$$

From here, we will model types as collections of terms and coterms that work well together. A sensible place to start is to demand *soundness*: all of these terms and coterms can only form safe commands (i.e., ones found in $\perp\!\!\!\perp$). However, we will quickly find that we also need *completeness*: any terms and coterms that do not break safety are included.

**Definition 6.2** (Candidates)**.** A *pre-candidate* is any pair $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ where $\mathbb{A}^+$ is a set of terms, and $\mathbb{A}^-$ is a set of coterms, *i.e.,* $\mathbb{A} \in \wp(\mathit{Term}) \times \wp(\mathit{CoTerm})$.
A *sound* (pre-)candidate satisfies this additional requirement:

- *Soundness:* for all $v \in \mathbb{A}^+$ and $e \in \mathbb{A}^-$, the command $\langle v \| e \rangle$ is safe (*i.e.,* $\langle v \| e \rangle \in \perp\!\!\!\perp$).

A *complete* (pre-)candidate satisfies these two *completeness* requirements:

- *Positive completeness:* if $\langle v \| E \rangle$ is safe (*i.e.,* $\langle v \| E \rangle \in \perp\!\!\!\perp$) for all $E \in \mathbb{A}^-$ then $v \in \mathbb{A}^+$.
- *Negative completeness:* if $\langle V \| e \rangle$ is safe (*i.e.,* $\langle V \| e \rangle \in \perp\!\!\!\perp$) for all $V \in \mathbb{A}^+$ then $e \in \mathbb{A}^-$.

A *reducibility candidate* is any sound and complete (pre-)candidate. $\mathscr{PC}$ denotes the set of all pre-candidates, $\mathscr{SC}$ denotes the set of sound ones, $\mathscr{CC}$ the set of complete ones, and $\mathscr{RC}$ denotes the set of all reducibility candidates.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

As notation, given any pre-candidate $\mathbb{A}$, we will always write $\mathbb{A}^+$ to denote the first component of $\mathbb{A}$ (the term set $\pi_1(\mathbb{A})$) and $\mathbb{A}^-$ to denote the second one (the coterm set $\pi_2(\mathbb{A})$). As shorthand, given a reducibility candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$, we write $v \in \mathbb{A}$ to mean $v \in \mathbb{A}^+$ and likewise $e \in \mathbb{A}$ to mean $e \in \mathbb{A}^-$. Given a set of terms $\mathbb{A}^+$, we will occasionally write the pre-candidate $(\mathbb{A}^+, \{\})$ as just $\mathbb{A}^+$ when the difference is clear from context. Likewise, we will occasionally write the pre-candidate $(\{\}, \mathbb{A}^-)$ as just the coterm set $\mathbb{A}^-$ when unambiguous.

The motivation behind soundness may seem straightforward. It ensures that the *Cut* rule is safe. But soundness is not enough, because the type system does much more than *Cut*: it makes many promises that several terms and coterms inhabit the different types. For example, the function type contains $\lambda$-abstractions and call stacks, and *every* type contains $\mu$- and $\tilde{\mu}$-abstractions over free (co)variables of the type. Yet, there is nothing in soundness that keeps these promises. For example, the trivial pre-candidate $(\{\}, \{\})$ is sound but it contains nothing, even though *ActR* and *ActL* promise many $\mu$- and $\tilde{\mu}$-abstractions that are left out! So to fully reflect the rules of the type system, we require a more informative model.

Completeness ensures that every reducibility candidate has "enough" (co)terms that are promised by the type system. For example, the completeness requirements given in Definition 6.2 are enough to guarantee every complete candidate contains all the appropriate $\mu$- and $\tilde{\mu}$-abstractions that always step to safe commands for any allowed binding.

**Lemma 6.3** (Activation)**.** *For any complete candidate* $\mathbb{A}$*:*

1. *If* $c[E/\alpha] \in \perp\!\!\!\perp$ *for all* $E \in \mathbb{A}$*, then* $\mu\alpha.c \in \mathbb{A}$.
2. *If* $c[V/x] \in \perp\!\!\!\perp$ *for all* $V \in \mathbb{A}$*, then* $\tilde{\mu}x.c \in \mathbb{A}$.

**Proof** Consider the first fact (the second is perfectly dual to it and follows analogously) and assume that $c[E/\alpha] \in \perp\!\!\!\perp$ for all $E \in \mathbb{A}$. In other words, the definition of $c[E/\alpha] \in \perp\!\!\!\perp$ says $c[E/\alpha] \mapsto\!\!\!\!\mapsto c'$ for some valid command $c' \in \mathit{Final}$. For any specific $E \in \mathbb{A}$, we have:

$$\langle \mu\alpha.c \| E \rangle \mapsto c[E/\alpha] \mapsto\!\!\!\!\mapsto c' \in \mathit{Final}$$

By definition of $\perp\!\!\!\perp$ (Definition 6.1) and transitivity of reduction, $\langle \mu\alpha.c \| E \rangle \in \perp\!\!\!\perp$ as well for any $E \in \mathbb{A}$. Thus, $\mathbb{A}$ *must* contain $\mu\alpha.c$, as required by positive completeness (Definition 6.2). ■

Notice that the restriction to values and covalues in the definition of completeness (Definition 6.2) is crucial in proving Lemma 6.3. We can easily show that the $\mu$-abstraction steps to a safe command for every given (co)value, but if we needed to say the same for every (co)term we would be stuck in call-by-name evaluation where the $\mu$-rule might not fire. Dually, it is always easy to show that the $\tilde{\mu}$ safely steps for every value, but we cannot say the same for every term in call-by-value. The pattern in Lemma 6.3 of reasoning about commands based on the ways they reduce is a crucial key to proving properties of particular (co)terms of interest. The very definition of safe commands $\perp\!\!\!\perp$ is *closed under expansion* of the machine's operational semantics.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Property 6.4** (Expansion)**.** *If* $c \mapsto c' \in \perp\!\!\!\perp$ *then* $c \in \perp\!\!\!\perp$.

**Proof** Follows from the definition of $\perp\!\!\!\perp$ (Definition 6.1) and transitivity of reduction. ■

This property is the key step used to conclude the proof Lemma 6.3 and can be used to argue that other (co)terms are in specific reducibility candidates due to the way they reduce.

### *6.2 Subtyping and completion*

Before we delve into the interpretations of types as reducibility candidates, we first need to introduce another important concept that the model revolves around: *subtyping*. The type system (Figs. 6 and 7) for the abstract machine has no rules for subtyping, but nevertheless, a semantic notion of subtyping is useful for organizing and building reducibility candidates. More specifically, notice that there are exactly two basic ways to order pre-candidates based on the inclusion of their underlying sets:

**Definition 6.5** (Refinement and Subtype Order)**.** The *refinement order* ($\mathbb{A} \sqsubseteq \mathbb{B}$) and *subtype order* ($\mathbb{A} \leq \mathbb{B}$) between pre-candidates is:

$$(\mathbb{A}^+, \mathbb{A}^-) \sqsubseteq (\mathbb{B}^+, \mathbb{B}^-) := (\mathbb{A}^+ \subseteq \mathbb{B}^+) \text{ and } (\mathbb{A}^- \subseteq \mathbb{B}^-)$$
$$(\mathbb{A}^+, \mathbb{A}^-) \leq (\mathbb{B}^+, \mathbb{B}^-) := (\mathbb{A}^+ \subseteq \mathbb{B}^+) \text{ and } (\mathbb{A}^- \supseteq \mathbb{B}^-)$$

The reverse *extension order* $\mathbb{A} \sqsupseteq \mathbb{B}$ is defined as $\mathbb{B} \sqsubseteq \mathbb{A}$, and *supertype order* $\mathbb{A} \geq \mathbb{B}$ is $\mathbb{B} \leq \mathbb{A}$.

Refinement just expresses basic inclusion: $\mathbb{A}$ refines $\mathbb{B}$ when everything in $\mathbb{A}$ (both term and coterm) is contained within $\mathbb{B}$. With subtyping, the underlying set orderings go in *opposite* directions! If $\mathbb{A}$ is a subtype of $\mathbb{B}$, then $\mathbb{A}$ can have *fewer* terms and *more* coterms than $\mathbb{B}$. While this ordering may seem counter-intuitive, it closely captures the understanding of sound candidates where coterms are *tests* on terms. If $\perp\!\!\!\perp$ expresses which terms pass which tests (*i.e.,* coterms), the soundness requires that all of its terms passes each of its tests. If a sound candidate has fewer terms, then it might be able to safely include more tests which were failed by the removed terms. But if a sound candidate has more terms, it might be required to remove some tests that the new terms don't pass.

This semantics for subtyping formalizes Liskov's *substitution principle* (Liskov, 1987): if $\mathbb{A}$ is a subtype of $\mathbb{B}$, then terms of $\mathbb{A}$ are also terms of $\mathbb{B}$ because they can be safely used in any context expecting inputs from $\mathbb{B}$ (*i.e.,* with any coterm of $\mathbb{B}$). Interestingly, our symmetric model lets us express the logical dual of this substitution principle: if $\mathbb{A}$ is a subtype of $\mathbb{B}$, then the coterms of $\mathbb{B}$ (*i.e.,* contexts expecting inputs of $\mathbb{B}$) are coterms of $\mathbb{A}$ because they can be safely given any input from $\mathbb{A}$. These two principles lead to a natural subtype ordering of reducibility candidates, based on the sets of (co)terms they include.

The usefulness of this semantic, dual notion of subtyping comes from the way it gives us a complete lattice, which makes it possible to combine and build new candidates from other simpler ones.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Definition 6.6** (Subtype Lattice)**.** There is a complete lattice of pre-candidates in $\mathscr{PC}$ with respect to subtyping order, where the binary intersection ($\mathbb{A} \wedge \mathbb{B}$, *a.k.a meet* or *greatest lower bound*) and union ($\mathbb{A} \vee \mathbb{B}$, *a.k.a join* or *least upper bound*) are defined as:

$$\mathbb{A} \wedge \mathbb{B} := (\mathbb{A}^+ \cap \mathbb{B}^+, \mathbb{A}^- \cup \mathbb{B}^-) \qquad \mathbb{A} \vee \mathbb{B} := (\mathbb{A}^+ \cup \mathbb{B}^+, \mathbb{A}^- \cap \mathbb{B}^-)$$

Moreover, these generalize to the intersection ($\bigwedge \mathscr{A}$, *a.k.a* infimum) and union ($\bigvee \mathscr{A}$, *a.k.a* supremum) of any set $\mathscr{A} \subseteq \mathscr{PC}$ of pre-candidates

$$\bigwedge \mathscr{A} := \left(\bigcap \{\mathbb{A}^+ \mid \mathbb{A} \in \mathscr{A}\}, \bigcup \{\mathbb{A}^- \mid \mathbb{A} \in \mathscr{A}\}\right)$$
$$\bigvee \mathscr{A} := \left(\bigcup \{\mathbb{A}^+ \mid \mathbb{A} \in \mathscr{A}\}, \bigcap \{\mathbb{A}^- \mid \mathbb{A} \in \mathscr{A}\}\right)$$

The binary least upper bounds and greatest lower bounds have these standard properties:

$$\mathbb{A} \wedge \mathbb{B} \leq \mathbb{A}, \mathbb{B} \leq \mathbb{A} \vee \mathbb{B} \qquad \mathbb{C} \leq \mathbb{A}, \mathbb{B} \implies \mathbb{C} \leq \mathbb{A} \wedge \mathbb{B} \qquad \mathbb{A}, \mathbb{B} \leq \mathbb{C} \implies \mathbb{A} \vee \mathbb{B} \leq \mathbb{C}$$

In the general case for the bounds on an entire set of pre-candidates, we know:

$$\forall \mathbb{A} \in \mathscr{A}.\ \left(\bigwedge \mathscr{A} \leq \mathbb{A}\right) \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{C} \leq \mathbb{A}) \implies \mathbb{C} \leq \bigwedge \mathscr{A}$$
$$\forall \mathbb{A} \in \mathscr{A}.\ \left(\mathbb{A} \leq \bigvee \mathscr{A}\right) \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A} \leq \mathbb{C}) \implies \bigvee \mathscr{A} \leq \mathbb{C}$$

These intersections and unions both preserve soundness, and so they form a lattice of sound candidates in $\mathscr{SC}$ as well. However, they do not preserve completeness in general, so they do not form a lattice of reducibility candidates, *i.e.,* sound and complete pre-candidates. Completeness is not preserved because $\mathbb{A} \vee \mathbb{B}$ might be missing some terms (such as $\mu$s) which could be soundly included, and dually $\mathbb{A} \wedge \mathbb{B}$ might be missing some co-terms (such as $\tilde{\mu}$s). So because all reducibility candidates are sound pre-candidates, $\mathbb{A} \vee \mathbb{B}$ and $\mathbb{A} \wedge \mathbb{B}$ are well-defined, but their results will only be sound candidates (not another reducibility candidate).[10]

What we need is a way to extend arbitrary sound candidates, adding "just enough" to make them full-fledged reducibility candidates. Since there are two possible ways to do this (add the missing terms or add the missing coterms), there are two completions which go in different directions. Also, since the completeness of which (co)terms are guaranteed to be in reducibility candidates is specified up to (co)values, we cannot be sure that absolutely everything ends up in the completed candidate. Instead, we can only ensure that the (co)-values that started in the pre-candidate are contained in its completion. This restriction to just the (co)values of a pre-candidate, written $\mathbb{A}^v$ and defined as

$$(\mathbb{A}^+, \mathbb{A}^-)^v := (\{V \mid V \in \mathbb{A}^+\}, \{E \mid E \in \mathbb{A}^-\})$$

becomes a pivotal part of the semantics of types. With this in mind, it follows there are exactly two "ideal" completions, the positive and negative ones, which give a reducibility candidate that is the closest possible to the starting point.

[10] The refinement lattice, on the other hand, interacts very differently with soundness and completeness. The refinement union $\mathbb{A} \sqcup \mathbb{B} := (\mathbb{A}^+ \cup \mathbb{B}^+, \mathbb{A}^- \cup \mathbb{B}^-)$ preserves completeness, but might break soundness by putting together a term of $\mathbb{A}$ which is incompatible with a (co)term of $\mathbb{B}$, or vice versa. Dually, the refinement intersection of two pre-candidates, $\mathbb{A} \sqcap \mathbb{B} := (\mathbb{A}^+ \cap \mathbb{B}^+, \mathbb{A}^- \cap \mathbb{B}^-)$, preserves soundness but can break completeness if a safe term or coterm is left out of the underlying intersections. So while refinement may admit a complete lattice for pre-candidates in $\mathscr{PC}$, we only get two dual refinement semi-lattices for sound candidates in $\mathscr{SC}$ and complete candidates in $\mathscr{CC}$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Lemma 6.7** (Positive & Negative Completion)**.** *There are two completions,* Pos *and* Neg*, of any sound candidate* $\mathbb{A}$ *with these three properties:*

1. They are reducibility candidates: $\mathrm{Pos}(\mathbb{A})$ *and* $\mathrm{Neg}(\mathbb{A})$ *are both sound and complete.*
2. They are (co)value extensions: *Every (co)value of* $\mathbb{A}$ *is included in* $\mathrm{Pos}(\mathbb{A})$ *and* $\mathrm{Neg}(\mathbb{A})$.

$$\mathbb{A}^v \sqsubseteq \mathrm{Pos}(\mathbb{A}) \qquad\qquad \mathbb{A}^v \sqsubseteq \mathrm{Neg}(\mathbb{A})$$

3. They are the least/greatest such candidates: *Any reducibility candidate that extends the (co)values of* $\mathbb{A}$ *lies between* $\mathrm{Pos}(\mathbb{A})$ *and* $\mathrm{Neg}(\mathbb{A})$*, with* $\mathrm{Pos}(\mathbb{A})$ *being smaller and* $\mathrm{Neg}(\mathbb{A})$ *being greater. In other words, given any reducibility candidate* $\mathbb{C}$ *such that* $\mathbb{A}^v \sqsubseteq \mathbb{C}$*:*

$$\mathrm{Pos}(\mathbb{A}) \leq \mathbb{C} \leq \mathrm{Neg}(\mathbb{A})$$

The full proof of these completions (and proofs of the other remaining propositions not given in this section) is given in Appendix 1. We refer to $\mathrm{Pos}(\mathbb{A})$ as the *positive completion of* $\mathbb{A}$ because it is based entirely on the values of $\mathbb{A}$ (*i.e.,* its positive components): $\mathrm{Pos}(\mathbb{A})$ collects the complete set of coterms that are safe with $\mathbb{A}$'s values and then collects the terms that are safe with the covalues from the previous step, and so on until a fixed point is reached (taking three rounds total). As such, the covalues included in $\mathbb{A}$ can't influence the result of $\mathrm{Pos}(\mathbb{A})$. Dually, $\mathrm{Neg}(\mathbb{A})$ is the *negative completion of* $\mathbb{A}$ because it is based entirely on $\mathbb{A}$'s covalues in the same manner.

**Lemma 6.8** (Positive & Negative Invariance)**.** *For any sound candidates* $\mathbb{A}$ *and* $\mathbb{B}$*:*

- *If the values of* $\mathbb{A}$ *and* $\mathbb{B}$ *are the same, then* $\mathrm{Pos}(\mathbb{A}) = \mathrm{Pos}(\mathbb{B})$.
- *If the covalues of* $\mathbb{A}$ *and* $\mathbb{B}$ *are the same, then* $\mathrm{Neg}(\mathbb{A}) = \mathrm{Neg}(\mathbb{B})$.

This extra fact gives us another powerful completeness property. We can reason about covalues in the positive candidate $\mathrm{Pos}(\mathbb{A})$ purely in terms of how they interact with $\mathbb{A}$'s values, ignoring the rest of $\mathrm{Pos}(\mathbb{A})$. Dually, we can reason about values in the negative candidate $\mathrm{Neg}(\mathbb{A})$ purely based on of $\mathbb{A}$'s covalues.

**Corollary 6.9** (Strong Positive & Negative Completeness)**.** *For any sound candidate* $\mathbb{A}$*:*

- $E \in \mathrm{Pos}(\mathbb{A})$ *if and only if* $\langle V \| E \rangle \in \perp\!\!\!\perp$ *for all* $V \in \mathbb{A}$.
- $V \in \mathrm{Neg}(\mathbb{A})$ *if and only if* $\langle V \| E \rangle \in \perp\!\!\!\perp$ *for all* $E \in \mathbb{A}$.

**Proof** Follows directly from Lemmas 6.7 and 6.8. ■

Now that we know how to turn sound pre-candidates $\mathbb{A}$ into reducibility candidates $\mathrm{Pos}(\mathbb{A})$ or $\mathrm{Neg}(\mathbb{A})$—Pos and Neg take anything sound and deliver something sound *and* complete—we can spell out precisely the subtype-based lattice of reducibility candidates.

**Theorem 6.10** (Reducibility Subtype Lattice)**.** *There is a complete lattice of reducibility candidates in* $\mathscr{RC}$ *with respect to subtyping order, with this binary intersection* $\mathbb{A} \curlywedge \mathbb{B}$ *and union* $\mathbb{A} \curlyvee \mathbb{B}$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

$$
\begin{aligned}
[\![A \to B]\!] &:= \mathrm{Neg}\{V \cdot E \mid V \in [\![A]\!], E \in [\![B]\!]\} \\
&= \bigvee \{\mathbb{C} \in \mathscr{RC} \mid \forall\, V \in [\![A]\!], E \in [\![B]\!].\ V \cdot E \in \mathbb{C}\} \\
[\![\mathrm{Nat}]\!] &:= \bigwedge\!\!\!\bigwedge \{\mathbb{C} \in \mathscr{RC} \mid (\mathrm{zero} \in \mathbb{C}) \text{ and } (\forall\, V \in \mathbb{C}.\ \mathrm{succ}\, V \in \mathbb{C})\} \\
[\![\mathrm{Stream}\, A]\!] &:= \bigvee \{\mathbb{C} \in \mathscr{RC} \mid (\forall E \in [\![A]\!].\ \mathrm{head}\, E \in \mathbb{C}) \text{ and } (\forall E \in \mathbb{C}.\ \mathrm{tail}\, E \in \mathbb{C})\} \\
[\![\Gamma]\!] &:= \{\rho \in \mathit{Subst} \mid (\forall (x{:}A) \in \Gamma.\ x[\rho] \in [\![A]\!]) \text{ and } (\forall (\alpha \div A) \in \Gamma.\ \alpha[\rho] \in [\![A]\!])\} \\
\mathit{Subst} \ni \rho &::= V/x, \ldots, E/\alpha, \ldots
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash c]\!] &:= \forall \rho \in [\![\Gamma]\!].\ c[\rho] \in \perp\!\!\!\perp \\
[\![\Gamma \vdash v : A]\!] &:= \forall \rho \in [\![\Gamma]\!].\ v[\rho] \in [\![A]\!] \\
[\![\Gamma \vdash e \div A]\!] &:= \forall \rho \in [\![\Gamma]\!].\ e[\rho] \in [\![A]\!]
\end{aligned}
$$

Fig. 10. Model of termination and safety of the abstract machine.

$$\mathbb{A} \curlywedge \mathbb{B} := \mathrm{Neg}(\mathbb{A} \wedge \mathbb{B}) \qquad\qquad \mathbb{A} \curlyvee \mathbb{B} := \mathrm{Pos}(\mathbb{A} \vee \mathbb{B})$$

*and this intersection* ($\bigwedge\!\!\!\bigwedge \mathscr{A}$) *and union* ($\bigvee \mathscr{A}$) *of any set* $\mathscr{A} \subseteq \mathscr{RC}$ *of reducibility candidates*

$$\bigwedge\!\!\!\bigwedge \mathscr{A} := \mathrm{Neg}\left(\bigwedge \mathscr{A}\right) \qquad\qquad \bigvee \mathscr{A} := \mathrm{Pos}\left(\bigvee \mathscr{A}\right)$$

### *6.3 Interpretation and adequacy*

Using the subtyping lattice, we have enough infrastructure to define an interpretation of types and type-checking judgments as given in Fig. 10. Each type is interpreted as a reducibility candidate. Even though we are dealing with recursive types (Nat and Stream), the candidates for them are defined in a non-recursive way based on Knaster-Tarski's fixed point construction (Knaster, 1928; Tarski, 1955) and can be read with these intuitions:

- $[\![A \to B]\!]$ is the negatively complete candidate containing all the call stacks built from $[\![A]\!]$ arguments and $[\![B]\!]$ return continuations. Note that this is the same thing (via Lemmas 6.7 and 6.8) as *largest* candidate containing those call stacks.
- $[\![\mathrm{Nat}]\!]$ is the *smallest* candidate containing zero and closed successor constructors.
- $[\![\mathrm{Stream}\, A]\!]$ is the *largest* candidate containing head projections expecting an $[\![A]\!]$ element and closed under tail projections.

Typing environments ($\Gamma$) are interpreted as the set of valid substitutions $\rho$ which map variables $x$ to values and covariables $\alpha$ to covalues. The interpretation of the typing environment $\Gamma, x : A$ places an additional requirement on these substitutions: a valid substitution $\rho \in [\![\Gamma, x : A]\!]$ must substitute a value of $[\![A]\!]$ for the variable $x$, *i.e.,* $x[\rho] \in [\![A]\!]$. Dually, a valid substitution $\rho \in [\![\Gamma, \alpha \div A]\!]$ must substitute a covalue of $[\![A]\!]$ for $\alpha$, *i.e.,* $\alpha[\rho] \in [\![A]\!]$. Finally, typing judgments (*e.g.,* $\Gamma \vdash c$) are interpreted as statements which assert that the command or (co)term belongs to the safe set $\perp\!\!\!\perp$ or the assigned reducibility candidate for any substitution allowed by the environment. The key lemma is that typing derivations of a judgment ensure that the statement they correspond to holds true.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Lemma 6.11** (Adequacy)**.**

1. *If* $\Gamma \vdash c$ *is derivable then* $[\![\Gamma \vdash c]\!]$ *is true.*
2. *If* $\Gamma \vdash v : A$ *is derivable then* $[\![\Gamma \vdash v : A]\!]$ *is true.*
3. *If* $\Gamma \vdash e \div A$ *is derivable then* $[\![\Gamma \vdash e \div A]\!]$ *is true.*

In order to prove adequacy (Lemma 6.11), we need to know something more about which (co)terms are in the interpretation of types. For example, how do we know that the well-typed call stacks and λ-abstractions given by the rules in Fig. 6 end up in $[\![A \to B]\!]$? Intuitively, function types themselves are non-recursive. In Fig. 10, the union over the possible candidates $\mathbb{C}$ defining $[\![A \to B]\!]$ requires that certain call stacks must be in each $\mathbb{C}$, but it does *not* quantify over the (co)values already in $\mathbb{C}$ to build upon them. Because of this, $[\![A \to B]\!]$ is equivalent to the negative candidate $\mathrm{Neg}\{V \cdot E \mid V \in [\![A]\!], E \in [\![B]\!]\}$, as noted in Fig. 10. It follows from Corollary 6.9 that $[\![A \to B]\!]$ must contain any value which is compatible with just these call stacks, regardless of whatever else might be in $[\![A \to B]\!]$. This means we can use expansion (Property 6.4) to prove that $[\![A \to B]\!]$ contains all λ-abstractions that, when given one of these call stacks, step via $\beta_\to$ reduction to a safe command.

**Lemma 6.12** (Function Abstraction)**.** *If* $v[V/x] \in [\![B]\!]$ *for all* $V \in [\![A]\!]$*, then* $\lambda x.v \in [\![A \to B]\!]$.

**Proof** Observe that, for any $V \in [\![A]\!]$ and $E \in [\![B]\!]$:

$$\langle \lambda x.v \| V \cdot E \rangle \mapsto \langle v[V/x] \| E \rangle \in \perp\!\!\!\perp$$

where $\langle v[V/x] \| E \rangle \in \perp\!\!\!\perp$ is guaranteed due to soundness for the reducibility candidate $[\![B]\!]$. By expansion (Property 6.4), we know that $\langle \lambda x.v \| V \cdot E \rangle \in \perp\!\!\!\perp$ as well. So from Corollary 6.9:

$$\lambda x.v \in \mathrm{Neg}\{V \cdot E \mid V \in [\![A]\!], E \in [\![B]\!]\} = [\![A \to B]\!]$$

■

But what about $[\![\mathrm{Nat}]\!]$ and $[\![\mathrm{Stream}\, A]\!]$? The interpretations of (co)inductive types are not exactly instances of Pos or Neg as written, because unlike $[\![A \to B]\!]$, they quantify over elements in the possible $\mathbb{C}$s they are made from. This lets us say these $\mathbb{C}$s are closed under succ or tail, but it means that we cannot identify *a priori* a set of (co)values that generate the candidate independent of each $\mathbb{C}$.

A solution to this conundrum is to instead describe these (co)inductive types incrementally, building them step-by-step instead of all-at once à la Kleene's fixed point construction (Kleene, 1952). For example, the set of the natural numbers can be defined incrementally from a series of finite approximations by beginning with the empty set, and then at each step adding the number 0 and the successor of the previous set:

$$\mathbb{N}_0 := \{\} \qquad \mathbb{N}_{i+1} := \{0\} \cup \{n + 1 \mid n \in \mathbb{N}_i\} \qquad \mathbb{N} := \bigcup_{i=0}^{\infty} \{\mathbb{N}_i\}$$

So that each $\mathbb{N}_i$ contains only the numbers less than $i$. The final set of *all* natural numbers, $\mathbb{N}$, is then the union of each approximation $\mathbb{N}_i$ along the way. Likewise, we can do a similar

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

incremental definition of finite approximations of $[\![\text{Nat}]\!]$ like so:

$$[\![\text{Nat}]\!]_0 := \text{Pos}\{\} = \bigwedge \mathscr{RC} = \bigvee \{\}$$
$$[\![\text{Nat}]\!]_{i+1} := \text{Pos}(\{\text{zero}\} \vee \{\text{succ } V \mid V \in [\![\text{Nat}]\!]_i\})$$

The starting approximation $[\![\text{Nat}]\!]_0$ is the *smallest* reducibility candidate, which is given by Pos{}. From there, the next approximations are given by the positive candidate containing at least zero and the successor of every value of their predecessor. This construction mimics the approximations $\mathbb{N}_i$, where we start with the smallest possible base and incrementally build larger and larger reducibility candidates that more accurately approximate the limit.

The typical incremental coinductive definition is usually presented in the reverse direction: start out with the "biggest" set (whatever that is), and trim it down step-by-step. Instead, the coinductive construction of reducibility candidates is much more concrete, since they form a complete lattice (with respect to subtyping). There is a specific biggest candidate $\bigvee \mathscr{RC}$ (equal to $\bigwedge\{\}$) containing every term possible and the fewest coterms allowed. From there, we can add explicitly more coterms to each successive approximation, which shrinks the candidate by ruling out some terms that do not run safely with them. Thus, the incremental approximations of $[\![\text{Stream } A]\!]$ are defined negatively as:

$$[\![\text{Stream } A]\!]_0 := \text{Neg}\{\} = \bigvee \mathscr{RC} = \bigwedge \{\}$$
$$[\![\text{Stream } A]\!]_{i+1} := \text{Neg}(\{\text{head } E \mid E \in [\![A]\!]\} \wedge \{\text{tail } E \mid E \in [\![\text{Stream } A]\!]_i\})$$

We start with the biggest possible candidate given by Neg{}. From there, the next approximations are given by the negative candidate containing at least head $E$ (for any $E$ expecting an element of type $[\![A]\!]$) and the tail of every covalue in the previous approximation. The net effect is that $[\![\text{Stream } A]\!]_i$ definitely contains all continuations built from (at most) $i$ head and tail destructors. As with $[\![\text{Nat}]\!]_i$, the goal is to show that $[\![\text{Stream } A]\!]$ is the limit of $[\![\text{Stream } A]\!]_i$, *i.e.,* that it is the intersection of all finite approximations.

**Lemma 6.13** ((Co)Induction Inversion)**.**

$$[\![\text{Nat}]\!] = \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \qquad\qquad [\![\text{Stream } A]\!] = \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$$

This fact makes it possible to use expansion (Property 6.4) and strong completeness (Corollary 6.9) to prove that the recursor belongs to each of the approximations $[\![\text{Nat}]\!]_i$; and thus also to $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i = [\![\text{Nat}]\!]$ by induction on $i$. Dually, the corecursor belongs to each approximation $[\![\text{Stream } A]\!]_i$, so it is included in $\bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i = [\![\text{Stream } A]\!]$. This proof of safety for (co)recursion is the final step in proving overall adequacy (Lemma 6.11). In turn, the ultimate type safety and termination property we are after is a special case of a more general notion of *observable* safety and termination, which follows directly from adequacy for certain typing environments.

**Theorem 6.14** (Type safety & Termination of Programs)**.** *If* $\alpha \div \text{Nat} \vdash c$ *in the (co)-recursive abstract machine then* $c \mapsto\!\!\!\!\rightarrow \langle \text{zero} \| \alpha \rangle$ *or* $c \mapsto\!\!\!\!\rightarrow \langle \text{succ } V \| \alpha \rangle$ *for some* $V$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Proof** From Lemma 6.11, we have $[\![\alpha \div \mathrm{Nat} \vdash c]\!]$. That is, for all $E \in [\![\mathrm{Nat}]\!]$, $c[E/\alpha] \in \perp\!\!\!\perp$. Note that both $\langle \mathrm{zero} \| \alpha \rangle \in \perp\!\!\!\perp$ and $\langle \mathrm{succ}\, V \| \alpha \rangle \in \perp\!\!\!\perp$ (for any $V$ whatsoever) by definition of $\perp\!\!\!\perp$ (Definition 6.1). So by Corollary 6.9, $\alpha$ itself is a member of $[\![\mathrm{Nat}]\!]$, *i.e.,* $\alpha \in [\![\mathrm{Nat}]\!]$. Thus, $c[\alpha/\alpha] = c \in \perp\!\!\!\perp$, and the necessary reduction follows from the definition of $\perp\!\!\!\perp$. ■

*Intermezzo* 6.15. The wordings of Theorems 3.2 and 6.14 are meant to reflect the typical statement of type safety for functional and λ-calculus-based languages. In a λ-based language, type safety (and notably the "progress" half of a *progress and preservation* proof) is limited to *closed* programs which return some basic, observable data type like a boolean value, string, number, or list. For example, type safety might ensure that a term $M$ never gets stuck when $\bullet \vdash M : \mathrm{Nat}$. In our typed abstract machine, commands are the unit of execution rather than terms. So instead of a closed term $\bullet \vdash v : \mathrm{Nat}$, we state type safety and termination for a starting command like $\alpha \div \mathrm{Nat} \vdash \langle v \| \alpha \rangle$, where $v$ is the closed "source program" and $\alpha$ represents an initial continuation or empty context waiting for the final result, which is expected to be a natural number.

The fact that the safe command described by Theorems 3.2 and 6.14 is *not* closed,[11] but has a free covariable, raises the question: can safety and termination be generalized to cover *other* open commands with additional variables or covariables? And what might we require of those open commands in general to still ensure termination?

It turns out, very little of the proof methodology depends the specifics of our particular notion of safety captured by the sets of commands $\perp\!\!\!\perp$ and *Final* from Definition 6.1. Rather, the only property of these sets that is pervasively used is the Property 6.4 that $\perp\!\!\!\perp$ is closed under expansion ($c \mapsto c' \in \perp\!\!\!\perp$ implies $c \in \perp\!\!\!\perp$). In fact, there is only *one* place in the proof that requires another fact about $\perp\!\!\!\perp$, which is showing that $[\![\mathrm{Nat}]\!] = \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ for Lemma 6.13; this step uses the fact that there is at least one command, namely $\langle \mathrm{zero} \| \alpha \rangle$, in $\perp\!\!\!\perp$. However, this choice of example inhabitant of $\perp\!\!\!\perp$ is arbitrary, and could be replaced with any other choice of $\langle V_0 \| E_0 \rangle \in \perp\!\!\!\perp$ without otherwise changing the proof.

This independence of the general proof methodology from the notion of safety (captured by $\perp\!\!\!\perp$) makes it a rather straightforward exercise to generalize to other results by just redefining the set $\perp\!\!\!\perp$ of safe commands. For example, we could consider additional examples of *Final* commands which also includes stopping with any variable $x$ observed by a destructor $\mathrm{head}\, E$, $\mathrm{tail}\, E$, or $V \cdot E$, letting us observe a larger collection of safe commands $\perp\!\!\!\perp$.

*Definition* 6.16 (Observable Safety). The set of *observably safe* commands, $\perp\!\!\!\perp$, is:

$$\begin{aligned} \perp\!\!\!\perp &:= \{c \mid \exists c' \in \mathit{Final}. c \mapsto\!\!\!\!\rightarrow c'\} \\ \mathit{Final} &:= \{\langle \mathrm{zero} \| \alpha \rangle \mid \alpha \in \mathit{CoVar}\} \cup \{\langle \mathrm{succ}\, V \| \alpha \rangle \mid V \in \mathit{Value}, \alpha \in \mathit{CoVar}\} \\ &\quad \cup \{\langle x \| \mathrm{head}\, E \rangle \mid x \in \mathit{Var}, E \in \mathit{CoValue}\} \cup \{\langle x \| \mathrm{tail}\, E \rangle \mid x \in \mathit{Var}, E \in \mathit{CoValue}\} \\ &\quad \cup \{\langle x \| V \cdot E \rangle \mid x \in \mathit{Var}, V \in \mathit{Value}, E \in \mathit{CoValue}\} \end{aligned}$$

The same theorems still hold, up to and including adequacy (Lemma 6.11) and the specific lemmas about the nature of function and (co)inductive types (Lemmas 6.12

[11] In fact, there is no well-typed, closed command of the uniform abstract machine language (Downen, 2017)! This corresponds to the notion of logical consistency, which says that a consistent logic cannot derive a contradiction. In the connection between the logic of the sequent calculus and the abstract machine, a closed command represents a logical contradiction.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

and 6.13), after replacing Definition 6.1 with Definition 6.16. As a consequence, we can use this bigger definition of $\perp\!\!\!\perp$ to meaningfully observe a larger collection of safe commands than before.

*Definition* 6.17 (Observable Typing Environment). A typing environment $\Gamma$ is *observable* when every covariable in $\Gamma$ has the type Nat, and every variable in $\Gamma$ has the type $A \to B$ or Stream $A$ for some type(s) $A$ and $B$.

*Theorem* 6.18 (Observable Safety & Termination). *Given any observable typing environment* $\Gamma$*, if* $\Gamma \vdash c$ *in the (co)recursive abstract machine then* $c \mapsto\!\!\!\!\rightarrow c' \in \mathit{Final}$*, as per Definition 6.16.*

**Proof** First, note that the interpretations of specific types are guaranteed to contain variables or covariables as follows:

1. $\alpha \in [\![\mathrm{Nat}]\!]$. This is because $\langle \mathrm{zero} \| \alpha \rangle \in \mathit{Final} \supseteq \perp\!\!\!\perp$ and $\langle \mathrm{succ}\, V \| \alpha \rangle \in \mathit{Final} \supseteq \perp\!\!\!\perp$ for *any* $V$ whatsoever by definition of $\perp\!\!\!\perp$ (Definition 6.16). So by the updated Corollary 6.9, $\alpha$ itself is a member of $[\![\mathrm{Nat}]\!]$.
2. $x \in [\![\mathrm{Stream}\, A]\!]$. This is because $\langle x \| \mathrm{head}\, E \rangle \in \mathit{Final} \supseteq \perp\!\!\!\perp$ and $\langle x \| \mathrm{tail}\, E \rangle \in \mathit{Final} \supseteq \perp\!\!\!\perp$ for *any* $E$ by definition of $\perp\!\!\!\perp$ (Definition 6.16), so $x$ is a member of $[\![\mathrm{Stream}\, A]\!]$ for any $A$ by the updated Corollary 6.9.
3. $x \in [\![A \to B]\!]$. This is because $\langle x \| V \cdot E \rangle \in \mathit{Final} \supseteq \perp\!\!\!\perp$ for *any* $V$ and $E$ by definition of $\perp\!\!\!\perp$ (Definition 6.16), so $x$ is a member of $[\![A \to B]\!]$ for any $A$ and $B$ by the updated Corollary 6.9.

It follows that for any observable typing environment $\Gamma$, there is $id_\Gamma \in [\![\Gamma]\!]$ where the identity substitution $id_\Gamma$ is defined as:

$$id_\Gamma(x) = x \qquad (\text{if } (x : A) \in \Gamma) \qquad\qquad id_\Gamma(\alpha) = \alpha \qquad (\text{if } (\alpha \div A) \in \Gamma)$$

Now, updated adequacy (Lemma 6.11) ensures $[\![\Gamma \vdash c]\!]$. That is, for all $\rho \in [\![\Gamma]\!]$, $c[\rho] \in \perp\!\!\!\perp$. Since $\Gamma$ is observable, $id_\Gamma \in [\![\Gamma]\!]$, and thus $c[id_\Gamma] = c \in \perp\!\!\!\perp$. So by definition of $\perp\!\!\!\perp$ (Definition 6.16), $c \mapsto\!\!\!\!\rightarrow c' \in \mathit{Final}$. ■

]

## 7 Related work

The corecursor presented here is a computational interpretation of the categorical model of corecursion in a coalgebra. A coalgebra for a functor $F$ is defined by a morphism $\alpha : A \to F(A)$, and it is *(strongly) terminal* if there always exists a *unique* morphism from any other coalgebra of $F$ into it, satisfying the usual commutation properties (i.e., the coalgebra given by $\alpha : A \to F(A)$ is the terminal object in the category of $F$-coalgebras). As a way to characterize the difference between coiteration and corecursion, Geuvers (1992) relaxes this usual requirement to *weakly terminal coalgebras*, for which there might be several *non-unique* morphisms with the correct properties into the weakly terminal coalgebra.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

A *corecursive coalgebra* extends this idea by ensuring commutation with other morphisms of the form $X \to F(A + X)$, rather than just another $F$-coalgebra $X \to F(X)$.

The dual notion to Geuvers (1992)'s corecursive $F$-coalgebra is (therein) called a *recursive $F$-algebra*: an algebra $\alpha : F(A) \to A$ that is only *weakly initial*—there is always a morphism from $\alpha$ to any other $F$-algebra, but it need not be unique—such that it commutes with every morphism of the form $F(A \times X) \to X$, not just $F$-algebras $F(X) \to X$. Computationally speaking, the generalization to $F(A \times X) \to X$ corresponds to the difference between the System T recursor and iterator: the $A \times X$ corresponds to the two inputs in the successor case of the recursor (with $A$ being the immediate predecessor and $X$ being the solution on the predecessor), in comparison to the single input for the successor case of the iterator (who is only given the solution on the predecessor, but not the predecessor itself). Analogously, the $A + X$ in the commutation with $X \to F(A + X)$ for a corecursive coalgebra is interpreted as an (intuitionistic) sum type in the pure (*i.e.,* side-effect free) λ-calculus model by Geuvers (1992): $X + A$ stands for the ordinary data type with two constructors for injecting either of $X$ or $A$ into the sum. Here, we use an interpretation of $A + X$ as expressing the classical disjunction of multiple conclusions, represented in the calculus by multiple continuations passed simultaneously to one term. The multi-continuation interpretation of a classical disjunction gives improved generality, and can express some corecursive algorithms that the intuitionistic interpretation cannot (Downen & Ariola, 2021).

The coiterator, which we define as the restriction of the corecursor to never short-cut corecursion, corresponds exactly to the Harper's `strgen` (Harper, 2016). In this sense, the corecursor is a conservative extension of the purely functional coiterator. Coiteration with control operators is considered in Barthe & Uustalu (2002), which gives a call-by-name CPS translation for a stream coiterator and constructor, corresponding to **coiter** and **cocase**, but not for **corec**. 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 call-by-name. An alternative approach to (co)recursive combinators is *sized types* (Hughes *et al.*, 1996; Abel, 2006), which give the programmer control over recursion while still ensuring termination, and have been used for both purely functional (Abel & Pientka, 2013) and classical (Downen *et al.*, 2015) coinductive types. Both of these approaches express (co)recursive algorithms directly in terms of (co)pattern matching.

In the context of more practical programming, Charity (Fukushima & Cockett, 1992) is a total functional programming language whose design is based closely on categorical semantics, which can express the duality between inductive and coinductive types. As such, it included primitives for iteration and coiteration, as well as case and cocase, similar to **coiter** and **cocase** primitives in the calculus shown here. More recently, the OCaml language has been extended with copatterns (Regis-Gianas & Laforgue, 2017) as well as an explicit `corec` form Jeannin *et al.* (2017) to allow for corecursive functional programming. In other programming paradigms, applications of corecursion have been studied in the context of object-oriented programming (Ancona & Zucca, 2013, 2012a,b)—with a focus on modeling finite structures with cycles via *regular corecursion*—and logic programming (Ancona, 2013; Dagnino *et al.*, 2020; Dagnino, 2020) with non-well-founded structures in terms of *coaxioms* that are applied at "infinite" depth in an proof tree.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Our investigation on evaluation strategy showed the (dual) impact of call-by-value versus call-by-name evaluation (Curien & Herbelin, 2000; Wadler, 2003) on the efficiency of (co)recursion. In contrast to having a monolithic evaluation strategy, another approach is to use a hybrid evaluation strategy as done by call-by-push-value (Levy, 2001) or polarized languages (Zeilberger, 2009; Munch-Maccagnoni, 2013). With a hybrid approach, we could define one language which has the efficient version of both the recursor and corecursor. Polarity also allows for incorporating other evaluation strategies, such as call-by-need which shares the work of computations (Downen & Ariola, 2018*a*; McDermott & Mycroft, 2019). We leave the investigation of a polarized version of corecursion to future work.

## 8 Conclusion

This paper provides a foundational calculus for (co)recursion in programs phrased in terms of an abstract machine language. The impact of evaluation strategy is also illustrated, where call-by-value and call-by-name have (opposite) advantages for the efficiency of corecursion and recursion, respectively. These (co)recursion schemes are captured by (co)data types whose duality is made apparent by the language of the abstract machine. In particular, inductive data types, like numbers, revolve around constructing concrete, finite values, so that observations on numbers may be abstract and unbounded. Dually, coinductive codata types, like streams, revolve around concrete, finite observations, so that values may be abstract and unbounded objects. The computational interpretation of this duality lets us bring out hidden connections underlying the implementation of recursion and corecursion. For example, the explicit "seed" or accumulator usually used to generate infinite streams is, in fact, dual to the implicitly growing evaluation context of recursive calls. To show that the combination of primitive recursion and corecursion is well-behaved—that is, every program safely terminates with an answer—we interpreted the type system as a form of classical (bi)orthogonality model capable of handling first-class control effects, and extended with (co)inductive reducibility candidates. Our model reveals how the incremental Kleene-style and wholesale Knaster-Tarski-style constructions of greatest and least fixed points have different advantages for reasoning about program behavior. By showing the two fixed point constructions are the same—a non-trivial task for types of effectful computation—we get a complete picture of the mechanics of classical (co)recursion.

## Acknowledgments

We would like to thank the anonymous reviewers for their helpful suggestions and feedback for improving the presentation of this article. This work is supported by the National Science Foundation under Grant No. 1719158.

## Conflicts of interest

None.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

## References


Abel, A. (2006) *A Polymorphic Lambda Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München.

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

Abel, A. M. & Pientka, B. (2013) Wellfounded recursion with copatterns: A unified approach to termination and productivity. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. *ICFP '13*. ACM, pp. 185–196.

Ancona, D. (2013) Regular corecursion in Prolog. *Comput. Lang. Syst. Struct.* **39**, 142–162.

Ancona, D. & Zucca, E. (2012a) Corecursive featherweight Java. *Formal Techniques for Java-like Programs (FTfJP12)*.

Ancona, D. & Zucca, E. (2012b) Translating corecursive Featherweight Java in coinductive logic programming. In *Co-LP 2012 - A Workshop on Coinductive Logic Programming*.

Ancona, D. & Zucca, E. (2013) Safe Corecursion in coFJ. In *Formal Techniques for Java-Like Programs (FTfJP13),* pp. 2:1–2:7.

Ariola, Z. M., Bohannon, A. & Sabry, A. (2009) Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.* **31**, 13:1–13:48.

Barthe, G. & Uustalu, T. (2002) CPS translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, *PEPM'02*. Association for Computing Machinery, pp. 131–142.

Böhm, C. & Berarducci, A. (1985) Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.* **39**, 135–154.

Cockett, J. R. B. & Spencer, D. (1995) Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comput. Sci.* **139**, 69–113.

Crole, R. L. (2003) Coinduction and bisimilarity. *Oregon Programming Languages Summer School*. OPLSS.

Curien, P.-L. & Herbelin, H. (2000) The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, *ICFP '00*. ACM, pp. 233–243.

Dagnino, F. (2020) Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.* **15**.

Dagnino, F., Ancona, D. & Zucca, E. (2020) Flexible coinductive logic programming. *Theory Pract. Logic Program.* **20**, 818–833.

Downen, P. (2017) *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon.

Downen, P. & Ariola, Z. M. (2018a) 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 **119**, pp. 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Downen, P. & Ariola, Z. M. (2018b) A tutorial on computational classical logic and the sequent calculus. *J. Funct. Program.* **28**, e3.

Downen, P. & Ariola, Z. M. (2021) *Classical (Co)Recursion: Programming*. https://arxiv.org/abs/2103.06913.

Downen, P., Johnson-Freyd, P. & Ariola, Z. M. (2015) Structures for structural recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, *ICFP '15*. ACM, pp. 127–139.

Downen, P., Ariola, Z. M. & Ghilezan, S. (2019) The duality of classical intersection and union types. *Fundam. Inform.* **170**, 39–92.

Downen, P., Johnson-Freyd, P. & Ariola, Z. M. (2020) Abstracting models of strong normalization for classical calculi. *J. Log. Algebr. Methods Program.* **111**, 100512.

Felleisen, M. & Friedman, D. P. (1986) Control operators, the SECD machine, and the λ-calculus. In *Proceedings of the IFIP TC 2/WG2.2 Working Conference on Formal Descriptions of Programming Concepts Part III,* pp. 193–219.


https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Fukushima, T. & Cockett, R. (1992) *About Charity*.
Gentzen, G. (1935) Untersuchungen über das logische schließen. I. *Math. Z.* **39**, 176–210.
Geuvers, H. (1992) Inductive and coinductive types with iteration and recursion. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Bastad* pp. 193–217.
Gibbons, J. (2003) *The Fun of Programming*. In Chap. Origami programming, pp. 41–60.
Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50**, 1–101.
Girard, J.-Y., Lafont, Y. & Taylor, P. (1989) *Proofs and Types*. Cambridge University Press.
Gödel, K. (1980) On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic* **9**, 133–142. English translation of *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes*, first published 1958.
Hagino, T. (1987) A typed lambda calculus with categorical type constructors. *Category Theory and Computer Science*. Springer Berlin Heidelberg, pp. 140–157.
Harper, R. (2016) *Practical Foundations for Programming Languages*. 2nd edn. Cambridge University Press.
Hinze, R., Wu, N. & Gibbons, J. (2013) Unifying structured recursion schemes. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, *ICFP '13*. Association for Computing Machinery, pp. 209–220.
Hughes, J., Pareto, L. & Sabry, A. (1996) Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL'96*. Association for Computing Machinery, pp. 410–423.
Jeannin, J.-B., Kozen, D. & Silva, A. (2017) Cocaml: Functional programming with regular coinductive types. *Fundam. Inform.* **150**, 347–377.
Kleene, S. C. (1952) *Introduction to Metamathematics*. D. van Nostrand Company.
Knaster, B. (1928) Un theoreme sur les functions d'ensembles. *Ann. Soc. Pol. Math.* **6**, 133–134.
Krivine, J.-L. (2007) A call-by-name lambda-calculus machine. *Higher-Order Symb. Comput.* **20**, 199–207.
Levy, P. B. (2001) *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London.
Liskov, B. (1987) Keynote address-data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum), OOPSLA '87*. ACM, pp. 17–34.
Malcolm, G. (1990) Data structures and program transformation. *Sci. Comput. Program.* **14**, 255–279.
McDermott, D. & Mycroft, A. (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, pp. 235–262.
Meertens, L. (1987) *First Steps towards the Theory of Rose Trees*. Draft report. CWI, Amsterdam.
Meertens, L. (1992) Paramorphisms. *Formal Aspects Comput.* **4**, 413–424.
Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, pp. 124–144.
Mendler, N. P. (1987) Recursive types and type constraints in second-order lambda calculus. *Log. Comput. Sci*.
Mendler, N. P. (1988) *Inductive Definition in Type Theory*. PhD thesis, Cornell University.
Munch-Maccagnoni, G. (2013) *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot.
Parigot, M. (1992) $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning: International Conference, LPAR '92*. Springer Berlin Heidelberg, pp. 190–201.
Regis-Gianas, Y. & Laforgue, P. (2017) Copattern-matchings and first-class observations in OCaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, *PPDP '17*.

Rutten, J. (2019) *The Method of Coalgebra: Exercises in Coinduction*. CWI, Amsterdam, The Netherlands.
Sabry, A. & Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* **6**, 289–360.
Sangiorgi, D. (2011) *Introduction to Bisimulation and Coinduction*. Cambridge University Press.
Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Appl. Math.* **5**, 285–309.
Vene, V. & Uustalu, T. (1998) Functional programming with apomorphisms (corecursion). *Proc. Estonian Acad. Sci. Phys. Math.* **47**, 147–161.
Vos, T. E. J. (1995) *Program Construction and Generation Based on Recursive Data Types*. Masters thesis, Philips Research, Eindhoven.
Wadler, P. (2003) Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 189–201.
Zeilberger, N. (2009) *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University.

## 1 Proof of type safety and termination

Here we give the full details to the proof of the main result, Theorem 6.14, ensuring both safety and termination for all well-typed, executable commands. We use a proof technique suitable for abstract machines based on (Downen *et al.*, 2020, 2019), which we extend with the inductive type Nat and the coinductive type Stream $A$. To begin in Appendices 1.1 to 1.3, we give a self-contained introduction and summary of the fundamental concepts and results from Downen *et al.* (2020, 2019). Appendix 1.2 in particular gives a new account of *positive* and *negative completion* which simplifies the sections that follow. From there, Appendix 1.4 establishes the definition and properties of the (co)inductive types Nat and Stream $A$ in this model, which lets us prove the fundamental *adequacy* lemma in Appendix 1.5.

### *1.1 Orthogonal fixed point candidates*

Our proof technique revolves around *pre-candidates* (Definition 6.2) and their more informative siblings *reducibility candidates*. The first, and most important, operation on pre-candidates is *orthogonality*. Intuitively, on the one side orthogonality identifies *all* the terms that are safe with *everything* in a given set of coterms, and on the other side, it identifies the coterms that are safe with a set of terms. These two dual operations converting back and forth between terms and coterms naturally extend to a single operation on pre-candidates.

**Definition 1.1** (Orthogonality)**.** The *orthogonal* of any set of terms, $\mathbb{A}^+$, written $\mathbb{A}^{+\perp\!\!\!\perp}$, is the set of coterms that form safe commands (*i.e.,* in $\perp\!\!\!\perp$) with all of $\mathbb{A}^+$:

$$\mathbb{A}^{+\perp\!\!\!\perp} := \{e \mid \forall v \in \mathbb{A}^+.\ \langle v \| e \rangle \in \perp\!\!\!\perp\}$$

Dually, the *orthogonal* of any set of coterms $\mathbb{A}^-$, also written $\mathbb{A}^{-\perp\!\!\!\perp}$ and disambiguated by context, is the set of terms that form safe commands with all of $\mathbb{A}^-$:

$$\mathbb{A}^{-\perp\!\!\!\perp} := \{v \mid \forall e \in \mathbb{A}^-.\ \langle v \| e \rangle \in \perp\!\!\!\perp\}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Finally, the *orthogonal* of any pre-candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ is:

$$(\mathbb{A}^+, \mathbb{A}^-)^{\perp\!\!\!\perp} := (\mathbb{A}^{-\perp\!\!\!\perp}, \mathbb{A}^{+\perp\!\!\!\perp})$$

As a shorthand for mapping over sets, given any set of pre-candidates $\mathscr{A} \subseteq \mathscr{PC}$, we write $\mathscr{A}^{\perp\!\!\!\perp *}$ for the set of orthogonals to each pre-candidate in $\mathscr{A}$:

$$\mathscr{A}^{\perp\!\!\!\perp *} := \{\mathbb{A}^{\perp\!\!\!\perp} \mid \mathbb{A}^{\perp\!\!\!\perp} \in \mathscr{A}\}$$

We use the same notation for the orthogonals of any set of term-sets ($\mathscr{A} \subseteq \wp(\mathit{Term})$) or coterm-sets ($\mathscr{A} \subseteq \wp(\mathit{CoTerm})$), individually.

Orthogonality is interesting primarily because of the logical structure it creates among pre-candidates. In particular, orthogonality behaves very much like *intuitionistic negation* ($\neg$). Intuitionistic logic rejects *double negation elimination* ($\neg\neg A \iff A$) in favor of the weaker principle of *double negation introduction* ($A \implies \neg\neg A$). This fundamental property of intuitionistic negation is mimicked by pre-candidate orthogonality.

**Property 1.2** (Orthogonal Negation)**.** *The following holds for any pre-candidates* $\mathbb{A}$ *and* $\mathbb{B}$*:*

1. Contrapositive (*i.e.,* antitonicity): $\mathbb{A} \sqsubseteq \mathbb{B}$ *implies* $\mathbb{B}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$.
2. Double orthogonal introduction (DOI): $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp}$.
3. Triple orthogonal elimination (TOE): $\mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp} = \mathbb{A}^{\perp\!\!\!\perp}$.

**Proof**

1. *Contrapositive*: Let $v \in \mathbb{B}^{\perp\!\!\!\perp}$ and $e \in \mathbb{A}$. We know $e \in \mathbb{B}$ (because $\mathbb{A} \sqsubseteq \mathbb{B}$ implies $\mathbb{A}^- \subseteq \mathbb{B}^-$) and thus $\langle v \| e \rangle \in \perp\!\!\!\perp$ (because $v \in \mathbb{B}^{-\perp\!\!\!\perp}$). Therefore, $e \in \mathbb{A}^{\perp\!\!\!\perp}$ by definition of orthogonality (Definition 1.1). Dually, given any $e \in \mathbb{B}^{\perp\!\!\!\perp}$ and $v \in \mathbb{A}$, we know $v \in \mathbb{B}$ and thus $\langle v \| e \rangle \in \perp\!\!\!\perp$, so $e \in \mathbb{B}^{\perp\!\!\!\perp}$ as well.
2. *DOI*: Suppose $v \in \mathbb{A}$. For any $e \in \mathbb{A}^{\perp\!\!\!\perp}$, we know $\langle v \| e \rangle \in \perp\!\!\!\perp$ by definition of orthogonality (Definition 1.1). Therefore, $v \in \mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp}$ also by definition of orthogonality. Dually, every $e \in \mathbb{A}$ yields $\langle v \| e \rangle \in \perp\!\!\!\perp$ for all $v \in \mathbb{A}^{\perp\!\!\!\perp}$, so $e \in \mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp}$ as well.
3. *TOE*: Note that $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp}$ is an instance of double orthogonal introduction above for $\mathbb{A}$, so by contrapositive, $\mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$. Another instance of double orthogonal introduction for $\mathbb{A}^{\perp\!\!\!\perp}$ is $\mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp}$. Thus, the two pre-candidates are equal.

■

The second operation on pre-candidates is the *(co)value restriction*. This just limits a given pre-candidate to only the values and covalues contained within it and gives us a way to handle the chosen evaluation strategy (here, call-by-name or call-by-value) in the model. In particular, the (co)value restriction is useful for capturing the *completeness* requirement of reducibility candidates (Definition 6.2), which only tests (co)terms with respect to the (co)values already in the candidate.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Definition 1.3** ((Co)value Restriction)**.** The *(co)value restriction* of a set of terms $\mathbb{A}^+$, set of coterms $\mathbb{A}^-$, and pre-candidates $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ is:

$$\mathbb{A}^{+v} := \{V \mid V \in \mathbb{A}^+\} \qquad \mathbb{A}^{-v} := \{E \mid E \in \mathbb{A}^-\} \qquad (\mathbb{A}^+, \mathbb{A}^-)^v := (\mathbb{A}^{+v}, \mathbb{A}^{-v})$$

As another shorthand, given any set of pre-candidates $\mathscr{A}$, we will occasionally write $\mathscr{A}^{v*}$ to be the set of (co)value restrictions of each pre-candidate in $\mathscr{A}$:

$$\mathscr{A}^{v*} := \{\mathbb{A}^v \mid \mathbb{A} \in \mathscr{A}\}$$

We use the same notation for the (co)value restriction of any set of term-sets or coterm-sets.

**Property 1.4** (Monotonicity)**.** *Given any pre-candidates* $\mathbb{A}$ *and* $\mathbb{B}$*,*

1. $\mathbb{A} \leq \mathbb{B}$ *implies* $\mathbb{A}^{\perp\!\!\!\perp} \leq \mathbb{B}^{\perp\!\!\!\perp}$*, and*
2. $\mathbb{A} \leq \mathbb{B}$ *implies* $\mathbb{A}^v \leq \mathbb{B}^v$.
3. $\mathbb{A} \sqsubseteq \mathbb{B}$ *implies* $\mathbb{A}^v \sqsubseteq \mathbb{B}^v$.

**Proof** Subtype monotonicity of orthogonality follows from contrapositive (Property 1.2) and the opposed definitions of refinement versus subtyping. Specifically, $\mathbb{A} \leq \mathbb{B}$ means the same thing as $(\mathbb{A}^+, \mathbb{B}^-) \sqsubseteq (\mathbb{B}^+, \mathbb{A}^-)$, which contrapositive (Property 1.2) turns into $(\mathbb{B}^+, \mathbb{A}^-)^{\perp\!\!\!\perp} = (\mathbb{A}^{-\perp\!\!\!\perp}, \mathbb{B}^{+\perp\!\!\!\perp}) \sqsubseteq (\mathbb{B}^{-\perp\!\!\!\perp}, \mathbb{A}^{+\perp\!\!\!\perp}) = (\mathbb{A}^+, \mathbb{B}^-)^{\perp\!\!\!\perp}$ which is equivalent to $\mathbb{A}^{\perp\!\!\!\perp} \leq \mathbb{B}^{\perp\!\!\!\perp}$. Monotonicity of the (co)value restriction with respect to both subtyping and refinement follows directly from its definition. ■

Putting the two operations together, *(co)value restricted orthogonality* ($\mathbb{A}^{v\perp\!\!\!\perp}$) becomes our primary way of handling reducibility candidates. This combined operation shares essentially the same negation-inspired properties of plain orthogonality (Property 1.2), but is restricted to just (co)values rather than general (co)terms.

**Property 1.5** (Restricted Orthogonal Negation)**.** *Given any pre-candidate* $\mathbb{A}$*:*

1. Restriction idempotency: $\mathbb{A}^{vv} = \mathbb{A}^v \sqsubseteq \mathbb{A}$
2. Restricted orthogonal: $\mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp}$
3. Restricted double orthogonal introduction (DOI): $\mathbb{A}^v \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v}$.
4. Restricted triple orthogonal elimination (TOE): $\mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \mathbb{A}^{v\perp\!\!\!\perp v}$.

**Proof**

1. Because $V \in \mathbb{A}$ if and only if $V \in \mathbb{A}^v$ (and symmetrically for covalues).
2. Follows from contrapositive (Property 1.2) of the above fact that $\mathbb{A}^v \sqsubseteq \mathbb{A}$.
3. Double orthogonal introduction (Property 1.2) on $\mathbb{A}^v$ gives $\mathbb{A}^v \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp\perp\!\!\!\perp}$. The restricted orthogonal (above) of $\mathbb{A}^{v\perp\!\!\!\perp}$ implies $\mathbb{A}^{v\perp\!\!\!\perp\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$. Thus from monotonicity (Property 1.4) and restriction idempotency, we have: $\mathbb{A}^v \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp\perp\!\!\!\perp v} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v}$.
4. Follows similarly to triple orthogonal elimination in (Property 1.2). $\mathbb{A}^v \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v}$ is an instance of restricted double orthogonal introduction above, and by contrapositive (Property 1.2) and monotonicity (Property 1.4), $\mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v}$. Another instance of restricted double orthogonal introduction on $\mathbb{A}^{v\perp\!\!\!\perp v}$ is $\mathbb{A}^{v\perp\!\!\!\perp v} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v}$. Thus, the two sets are equal.

■

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

With these restricted logical properties, we can recast the *soundness* and *completeness* properties of reducibility candidates (Definition 6.2) in terms of orthogonality to show reducibility candidates are exactly the same as fixed points of (co)value restricted orthogonality.

**Lemma 1.6** (Fixed Point Candidates)**.**

1. *A pre-candidate* $\mathbb{A}$ *is sound if and only if* $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$.
2. *A pre-candidate* $\mathbb{A}$ *is complete if and only if* $\mathbb{A}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A}$.
3. *A pre-candidate* $\mathbb{A}$ *is a reducibility candidate if and only if* $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp}$.
4. *Every reducibility candidate is a fixed point of orthogonality:* $\mathbb{A} \in \mathscr{RC}$ *implies* $\mathbb{A} = \mathbb{A}^{\perp\!\!\!\perp}$.

**Proof** Unfolding the definitions of orthogonality (Definition 1.1) and the (co)-value restriction (Definition 1.3) shows that the first two refinements are equivalent to soundness and completeness from Definition 6.2.

For the last fact, first recall $\mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp}$ (Property 1.5). So if a pre-candidate $\mathbb{A}$ is both sound and complete, $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp} = \mathbb{A}^{\perp\!\!\!\perp}$ because

$$\mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp}$$

Going the other way, suppose that $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp}$. Completeness is guaranteed by definition, but what of soundness? Suppose that $v$ and $e$ come from the fixed point pre-candidate $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp} = (\mathbb{A}^{-v\perp\!\!\!\perp}, \mathbb{A}^{+v\perp\!\!\!\perp})$. The reason why $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp}$ forces $\langle v \| e \rangle \in \perp\!\!\!\perp$ depends on the evaluation strategy.

- *Call-by-value*, where every coterm is a covalue. Thus, the positive requirement on terms of reducibility candidates is equivalent to: $v \in \mathbb{A}^+$ if and only if, for all $e \in \mathbb{A}^-$, $\langle v \| e \rangle \in \perp\!\!\!\perp$.
- *Call-by-name*, where every term is a value. Thus, the negative requirement on coterms of reducibility candidates is equivalent to: $e \in \mathbb{A}^-$ if and only if, for all $v \in \mathbb{A}^+$, $\langle v \| e \rangle \in \perp\!\!\!\perp$.

In either case, $\langle v \| e \rangle \in \perp\!\!\!\perp$ for one of the above reasons, since $v, e \in \mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp}$. ■

### *1.2 Positive and negative completion*

Now that we know reducibility candidates are the same thing as fixed points of (co)value restricted orthogonality ($\_^{v\perp\!\!\!\perp}$), we have a direct method to define the completion of a sound pre-candidate into a sound *and* complete one. To complete some $\mathbb{A}$, there are two opposite points of view: (Pos) start with the terms of $\mathbb{A}$ and build everything else around those, or (Neg) start with the coterms of $\mathbb{A}$ and build around them. Both of these definitions satisfy all the defining criteria promised by Lemmas 6.7 to 6.8 due to the logical properties of orthogonality (Property 1.5).

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Definition 1.7** (Positive & Negative Reducibility Candidates)**.** For any sound candidate $\mathbb{A} \in \mathscr{SC}$, the positive (Pos($\mathbb{A}$)) and the negative (Neg($\mathbb{A}$)) completion of $\mathbb{A}$ are:

$$\mathrm{Pos}(\mathbb{A}) = (\mathbb{A}^{+}, \mathbb{A}^{+v\perp\!\!\!\perp})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = (\mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp})$$
$$\mathrm{Neg}(\mathbb{A}) = (\mathbb{A}^{-v\perp\!\!\!\perp}, \mathbb{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = (\mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp})$$

**Lemma 6.7** (Positive & Negative Completion)**.** *There are two completions,* Pos *and* Neg*, of any sound candidate* $\mathbb{A}$ *with these three properties:*

1. They are reducibility candidates: Pos($\mathbb{A}$) *and* Neg($\mathbb{A}$) *are both sound and complete.*
2. They are (co)value extensions: *Every (co)value of* $\mathbb{A}$ *is included in* Pos($\mathbb{A}$) *and* Neg($\mathbb{A}$)*.*
$$\mathbb{A}^{v} \sqsubseteq \mathrm{Pos}(\mathbb{A}) \qquad\qquad \mathbb{A}^{v} \sqsubseteq \mathrm{Neg}(\mathbb{A})$$
3. They are the least/greatest such candidates: *Any reducibility candidate that extends the (co)values of* $\mathbb{A}$ *lies between* Pos($\mathbb{A}$) *and* Neg($\mathbb{A}$)*, with* Pos($\mathbb{A}$) *being smaller and* Neg($\mathbb{A}$) *being greater. In other words, given any reducibility candidate* $\mathbb{C}$ *such that* $\mathbb{A}^{v} \sqsubseteq \mathbb{C}$*:*
$$\mathrm{Pos}(\mathbb{A}) \leq \mathbb{C} \leq \mathrm{Neg}(\mathbb{A})$$

**Proof** The definitions given in Definition 1.7 satisfy all three requirements:

1. *They are reducibility candidates:* Observe that by restricted triple orthogonal elimination (Property 1.5), Pos($\mathbb{A}$) and Neg($\mathbb{A}$) are reducibility candidates because they are fixed points of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6):

| $(\mathrm{Pos}(\mathbb{A}))^{v\perp\!\!\!\perp}$ | $(\mathrm{Neg}(\mathbb{A}))^{v\perp\!\!\!\perp}$ | |
|---|---|---|
| $= (\mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp})^{v\perp\!\!\!\perp}$ | $= (\mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp})^{v\perp\!\!\!\perp}$ | (Definition 1.7) |
| $= (\mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp})$ | $= (\mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp})$ | (Definition 1.1) |
| $= (\mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp})$ | $= (\mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp})$ | (Property 1.5) |
| $= \mathrm{Pos}(\mathbb{A})$ | $= \mathrm{Neg}(\mathbb{A})$ | (Definition 1.7) |

2. *They are (co)value extensions:* First, note that
$$\mathbb{A}^{+v} \subseteq \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \mathrm{Pos}(\mathbb{A})^{+v} \subseteq \mathrm{Pos}(\mathbb{A})^{+}$$
$$\mathbb{A}^{-v} \subseteq \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \mathrm{Neg}(\mathbb{A})^{-v} \subseteq \mathrm{Neg}(\mathbb{A})^{-}$$
by restricted double orthogonal introduction (Property 1.5). Furthermore, soundness of $\mathbb{A}$ means $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$ (*i.e.,* $\mathbb{A}^{+} \subseteq \mathbb{A}^{-\perp\!\!\!\perp}$ and $\mathbb{A}^{-} \subseteq \mathbb{A}^{+\perp\!\!\!\perp}$), so again by Property 1.5:
$$\mathbb{A}^{-v} \subseteq \mathbb{A}^{+\perp\!\!\!\perp v} \subseteq \mathbb{A}^{+v\perp\!\!\!\perp v} \subseteq \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \mathrm{Pos}(\mathbb{A})^{-v} \subseteq \mathrm{Pos}(\mathbb{A})^{-}$$
$$\mathbb{A}^{+v} \subseteq \mathbb{A}^{-\perp\!\!\!\perp v} \subseteq \mathbb{A}^{-v\perp\!\!\!\perp v} \subseteq \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \mathrm{Neg}(\mathbb{A})^{+v} \subseteq \mathrm{Neg}(\mathbb{A})^{+}$$
3. *They are the least/greatest such candidates:* Suppose there is a reducibility candidate $\mathbb{C}$ such that $\mathbb{A}^{v} \sqsubseteq \mathbb{C}$. Because $\mathbb{C}$ is a fixed point of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6), iterating contrapositive (Property 1.2) on this refinement gives:
$$\mathbb{C} = \mathbb{C}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{vv\perp\!\!\!\perp} = \mathbb{A}^{v\perp\!\!\!\perp} \qquad \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \sqsubseteq \mathbb{C}^{v\perp\!\!\!\perp} = \mathbb{C} \qquad \mathbb{C} = \mathbb{C}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Expanding the definition of Pos, Neg, and refinement, this means:

$$\mathrm{Pos}(\mathbb{A})^+ = \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp} \subseteq \mathbb{C} \qquad \mathrm{Pos}(\mathbb{A})^- = \mathbb{A}^{+v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp} \supseteq \mathbb{C}$$
$$\mathrm{Neg}(\mathbb{A})^+ = \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp} \supseteq \mathbb{C} \qquad \mathrm{Neg}(\mathbb{A})^- = \mathbb{A}^{-v\perp\!\!\!\perp v\perp\!\!\!\perp} \subseteq \mathbb{C}$$

Or in other words, $\mathrm{Pos}(\mathbb{A}) \leq \mathbb{C} \leq \mathrm{Neg}(\mathbb{A})$.

■

**Lemma 6.8** (Positive & Negative Invariance)**.** *For any sound candidates* $\mathbb{A}$ *and* $\mathbb{B}$*:*

- *If the values of* $\mathbb{A}$ *and* $\mathbb{B}$ *are the same, then* $\mathrm{Pos}(\mathbb{A}) = \mathrm{Pos}(\mathbb{B})$.
- *If the covalues of* $\mathbb{A}$ *and* $\mathbb{B}$ *are the same, then* $\mathrm{Neg}(\mathbb{A}) = \mathrm{Neg}(\mathbb{B})$.

**Proof** Because the definition of $\mathrm{Pos}(\mathbb{A})$ depends only on $\mathbb{A}^+$ and not $\mathbb{A}^-$, and dually the definition of $\mathrm{Neg}(\mathbb{A})$ depends only on $\mathbb{A}^-$. ■

In addition to these defining properties of Pos and Neg, the two completions are also *idempotent* (*i.e.,* they are closure operations, because multiple applications are the same as just one) and *monotonic* (*i.e.,* they preserve the subtyping order, by converting any two sound subtype candidates to two sound and complete subtype reducibility candidates).

**Corollary 1.8** (Idempotency)**.** *For all reducibility candidates* $\mathbb{A}$*,* $\mathrm{Pos}(\mathbb{A}) = \mathbb{A} = \mathrm{Neg}(\mathbb{A})$*. It follows that, for all sound candidates* $\mathbb{A}$*:*

$$\mathrm{Pos}(\mathrm{Pos}(\mathbb{A})) = \mathrm{Pos}(\mathbb{A}) \qquad \mathrm{Neg}(\mathrm{Neg}(\mathbb{A})) = \mathrm{Neg}(\mathbb{A})$$

**Proof** $\mathrm{Pos}(\mathbb{A}) = \mathbb{A} = \mathrm{Neg}(\mathbb{A})$ follows from Definition 1.7 because the reducibility candidate $\mathbb{A}$ is a fixed point of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6). The idempotency of Pos and Neg is then immediate from the fact that they produce reducibility candidates from any sound candidate. ■

**Lemma 1.9** (Monotonicity)**.** *Given any sound candidates* $\mathbb{A} \leq \mathbb{B}$*: 1.* $\mathrm{Pos}(\mathbb{A}) \leq \mathrm{Pos}(\mathbb{B})$*, and 2.* $\mathrm{Neg}(\mathbb{A}) \leq \mathrm{Neg}(\mathbb{B})$*.*

**Proof** Given $\mathbb{A} \leq \mathbb{B}$, Lemmas 6.7 to 6.8 imply that

$$\mathrm{Pos}(\mathbb{A}) = \mathrm{Pos}(\mathbb{A}^+, \{\}) \leq \mathrm{Pos}(\mathbb{B}^+, \{\}) = \mathrm{Pos}(\mathbb{B})$$
$$\mathrm{Neg}(\mathbb{A}) = \mathrm{Neg}(\{\}, \mathbb{A}^-) \leq \mathrm{Neg}(\{\}, \mathbb{B}^-) = \mathrm{Neg}(\mathbb{B})$$

because $\mathbb{A}^v \leq \mathbb{B}^v$ (Property 1.4), which means $\mathbb{A}^{+v} \subseteq \mathbb{B}^{+v}$ and $\mathbb{A}^{-v} \supseteq \mathbb{B}^{-v}$ by definition of subtyping. Thus from Lemma 6.7, we know that $(\mathbb{A}^{+v}, \{\}) \sqsubseteq (\mathbb{B}^{+v}, \{\}) \sqsubseteq \mathrm{Pos}(\mathbb{B}^+, \{\})$ and $\mathrm{Pos}(\mathbb{A}^+, \{\})$ is the least one to do so, forcing $\mathrm{Pos}(\mathbb{A}^+, \{\}) \leq \mathrm{Pos}(\mathbb{B}^+, \{\})$. Likewise from Lemma 6.7, we know that $(\{\}, \mathbb{B}^{-v}) \sqsubseteq (\{\}, \mathbb{A}^{v-}) \sqsubseteq \mathrm{Neg}(\{\}, \mathbb{A}^{-v})$ and $\mathrm{Neg}(\mathbb{B}^-, \{\})$ is the greatest one to do so, forcing $\mathrm{Neg}(\{\}, \mathbb{A}^-) \leq \mathrm{Neg}(\{\}, \mathbb{B}^-)$. ■

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

### *1.3 Refinement and subtyping lattices*

Because pre-candidates have two different orderings (Definition 6.5), they also have two very different lattice structures. We are primarily interested in the *subtyping lattice* because it is compatible with both soundness and completeness in both directions. In particular, the naïve subtype lattice as-is always preserves soundness, and combined with the dual completions (Pos and Neg) the subtype lattice preserves completeness as well. This gives us a direct way to assemble complex reducibility candidates from simpler ones.

**Theorem 1.10** (Sound Subtype Lattice)**.** *The subtype intersection* $\bigwedge$ *and union* $\bigvee$ *forms a complete semi-lattice over sound candidates in* $\mathscr{SC}$.

**Proof** Let $\mathscr{A} \subseteq \mathscr{SC}$ be a set of sound candidates, and suppose $v, e \in \bigwedge \mathscr{A}$. By definition:

- for all $\mathbb{A} \in \mathscr{A}$, $v \in \mathbb{A}$, and
- there exists an $\mathbb{A} \in \mathscr{A}$ such that $e \in \mathbb{A}$.

Therefore, we know that $v \in \mathbb{A}$ for the particular sound candidate that $e$ inhabits and thus $\langle v \| e \rangle$ by soundness of $\mathbb{A}$. Soundness of $\bigvee \mathscr{A}$ follows dually, because $v, e \in \bigvee \mathscr{A}$ implies:

- there exists an $\mathbb{A} \in \mathscr{A}$ such that $v \in \mathbb{A}$, and
- for all $\mathbb{A} \in \mathscr{A}$, $e \in \mathbb{A}$.

■

**Theorem 6.10** (Reducibility Subtype Lattice)**.** *There is a complete lattice of reducibility candidates in* $\mathscr{RC}$ *with respect to subtyping order, with this binary intersection* $\mathbb{A} \curlywedge \mathbb{B}$ *and union* $\mathbb{A} \curlyvee \mathbb{B}$

$$\mathbb{A} \curlywedge \mathbb{B} := \mathrm{Neg}(\mathbb{A} \wedge \mathbb{B}) \qquad\qquad \mathbb{A} \curlyvee \mathbb{B} := \mathrm{Pos}(\mathbb{A} \vee \mathbb{B})$$

*and this intersection* ($\bigcurlywedge \mathscr{A}$) *and union* ($\bigcurlyvee \mathscr{A}$) *of any set* $\mathscr{A} \subseteq \mathscr{RC}$ *of reducibility candidates*

$$\bigcurlywedge \mathscr{A} := \mathrm{Neg}\left(\bigwedge \mathscr{A}\right) \qquad\qquad \bigcurlyvee \mathscr{A} := \mathrm{Pos}\left(\bigvee \mathscr{A}\right)$$

**Proof** Let $\mathscr{A} \subseteq \mathscr{RC}$ be any set of reducibility candidates. First, note that $\bigwedge \mathscr{A}$ and $\bigvee \mathscr{A}$ are sound (Theorem 1.10) because every reducibility candidate is sound. Thus, for all $\mathbb{A} \in \mathscr{A}$, monotonicity (Lemma 1.9) and idempotency (Corollary 1.8) of Pos and Neg implies:

$$\bigwedge \mathscr{A} \leq \mathbb{A} \qquad\qquad \bigvee \mathscr{A} \leq \mathbb{A}$$

$$\bigcurlywedge \mathscr{A} = \mathrm{Pos} \bigwedge \mathscr{A} \leq \mathrm{Pos}(\mathbb{A}) = \mathbb{A} \qquad\qquad \bigcurlyvee \mathscr{A} = \mathrm{Neg} \bigvee \mathscr{A} \leq \mathrm{Neg}(\mathbb{A}) = \mathbb{A}$$

Lemma 1.9 and Corollary 1.8 also imply that these are the tightest such bounds. Suppose there are reducibility candidates $\mathbb{B}$ and $\mathbb{C}$ such that

$$\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{B} \leq \mathbb{A} \qquad\qquad \forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A} \leq \mathbb{C}$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

From the lattice properties of $\bigwedge$ and $\bigvee$, monotonicity, and idempotency, we have:

$$\mathbb{B} \leq \bigwedge \mathscr{A} \qquad \bigvee \mathscr{A} \leq \mathbb{C}$$
$$\mathbb{B} = \mathrm{Pos}(\mathbb{B}) \leq \mathrm{Pos} \bigwedge \mathscr{A} = \bigvee \mathscr{A} \qquad \bigwedge \mathscr{A} = \mathrm{Neg} \bigvee \mathscr{A} \leq \mathrm{Neg}(\mathbb{C}) = \mathbb{C}$$

■

The other lattice is based on refinement, instead of subtyping. In contrast, the refinement lattice has a opposing relationship with soundness and completeness: one direction of the lattice preserves only soundness, and the other one preserves only completeness.

**Definition 1.11** (Refinement Lattice)**.** There is a complete lattice of pre-candidates in $\mathscr{PC}$ with respect to refinement order, where the binary intersection ($\mathbb{A} \sqcap \mathbb{B}$ *a.k.a meet*) and union ($\mathbb{A} \sqcap \mathbb{B}$, *a.k.a join*) are defined as:

$$\mathbb{A} \sqcap \mathbb{B} := (\mathbb{A}^+ \cap \mathbb{B}^+, \mathbb{A}^- \cap \mathbb{B}^-) \qquad \mathbb{A} \sqcup \mathbb{B} := (\mathbb{A}^+ \cup \mathbb{B}^+, \mathbb{A}^- \cup \mathbb{B}^-)$$

Moreover, these generalize to the intersection ($\bigsqcap \mathbb{A}$, *a.k.a* infimum) and union ($\bigsqcup \mathbb{A}$, *a.k.a* supremum) of any set $\mathscr{A} \in \mathscr{PC}$ of pre-candidates

$$\bigsqcap \mathscr{A} := \left(\bigcap \{\mathbb{A}^+ \mid \mathbb{A} \in \mathscr{A}\}, \bigcap \{\mathbb{A}^- \mid \mathbb{A} \in \mathscr{A}\}\right)$$
$$\bigsqcup \mathscr{A} := \left(\bigcup \{\mathbb{A}^+ \mid \mathbb{A} \in \mathscr{A}\}, \bigcup \{\mathbb{A}^- \mid \mathbb{A} \in \mathscr{A}\}\right)$$

**Theorem 1.12** (Sound and Complete Refinement Semi-Lattices)**.** *The refinement intersection* $\bigsqcap$ *forms a meet semi-lattice over sound candidates in* $\mathscr{SC}$*, and the refinement union* $\bigsqcup$ *forms a join semi-lattice over complete candidates in* $\mathscr{CC}$*.*

**Proof** Let $\mathscr{A} \in \mathscr{SC}$ be a set of sound candidates, *i.e.,* for all $\mathbb{A} \in \mathscr{A}$, we know $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\perp}$. In the refinement lattice on pre-candidates, de Morgan duality (Property 1.13) implies:

$$\forall \mathbb{A} \in \mathscr{A}.\ \bigsqcap \mathscr{A} \sqsubseteq \mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\perp} \sqsubseteq \bigsqcup (\mathscr{A}^{\perp\!\!\perp *}) \sqsubseteq \left(\bigsqcap \mathscr{A}\right)^{\perp\!\!\perp}$$

So that $\bigsqcap \mathscr{A}$ is also sound.

Let $\mathscr{A} \in \mathscr{CC}$ be a set of complete candidates, *i.e.,* for all $\mathbb{A} \in \mathscr{A}$, we know $\mathbb{A}^{v\perp\!\!\perp} \sqsubseteq \mathbb{A}$. In the refinement lattice on pre-candidates, de Morgan duality (Property 1.13) and the fact that the (co)value restriction $\_^v$ distributes over unions implies:

$$\forall \mathbb{A} \in \mathscr{A}.\ \bigsqcup \mathscr{A} \sqsupseteq \mathbb{A} \sqsupseteq \mathbb{A}^{v\perp\!\!\perp} \sqsupseteq \bigsqcap (\mathscr{A}^{v*\perp\!\!\perp *}) = \bigsqcup (\mathscr{A})^{v\perp\!\!\perp}$$

So that $\bigsqcup \mathscr{A}$ is also complete. ■

Because soundness and completeness are each broken by different directions of this refinement lattice, it doesn't give us a complete lattice for assembling new reducibility candidates. However, what it does give us is additional insight into the logical properties of orthogonality. That is, while orthogonality behaves like intuitionistic negation, the intersections ($\bigsqcap$) and unions ($\bigsqcup$) act like conjunction and disjunction, respectively. Together, these give us properties similar to the familiar de Morgan laws of duality intuitionistic logic.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Property 1.13** (Orthogonal de Morgan)**.** *Given any pre-candidates* $\mathbb{A}$ *and* $\mathbb{B}$*:*

1. $(\mathbb{A} \sqcup \mathbb{B})^{\perp\!\!\!\perp} = (\mathbb{A}^{\perp\!\!\!\perp}) \sqcap (\mathbb{B}^{\perp\!\!\!\perp})$.
2. $(\mathbb{A} \sqcap \mathbb{B})^{\perp\!\!\!\perp} \sqsupseteq (\mathbb{A}^{\perp\!\!\!\perp}) \sqcup (\mathbb{B}^{\perp\!\!\!\perp})$.
3. $(\mathbb{A}^{\perp\!\!\!\perp} \sqcap \mathbb{B}^{\perp\!\!\!\perp})^{\perp\!\!\!\perp\perp\!\!\!\perp} = (\mathbb{A}^{\perp\!\!\!\perp}) \sqcap (\mathbb{B}^{\perp\!\!\!\perp}) = (\mathbb{A} \sqcup \mathbb{B})^{\perp\!\!\!\perp} = (\mathbb{A}^{\perp\!\!\!\perp\perp\!\!\!\perp} \sqcup \mathbb{B}^{\perp\!\!\!\perp\perp\!\!\!\perp})^{\perp\!\!\!\perp}$.

*Furthermore, given any set of pre-candidates* $\mathscr{A} \subseteq \mathscr{P}\mathscr{C}$*:*

1. $(\bigsqcup \mathscr{A})^{\perp\!\!\!\perp} = \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$.
2. $(\bigsqcap \mathscr{A})^{\perp\!\!\!\perp} \sqsupseteq \bigsqcup(\mathscr{A}^{\perp\!\!\!\perp *})$.
3. $(\bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *}))^{\perp\!\!\!\perp\perp\!\!\!\perp} = \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *}) = \left(\bigsqcup \mathscr{A}\right)^{\perp\!\!\!\perp} = \left(\bigsqcup \mathscr{A}^{\perp\!\!\!\perp *\perp\!\!\!\perp *}\right)^{\perp\!\!\!\perp}$.

**Proof** We will show only the de Morgan properties for union and intersection over any sets of pre-candidate $\mathscr{A}$; the binary versions are special cases of these. Note that the union and intersection of the refinement lattice on pre-candidates have these lattice properties:

$$\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A} \sqsubseteq \bigsqcup \mathscr{A} \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A} \sqsubseteq \mathbb{C}) \Longrightarrow \bigsqcup \mathscr{A} \sqsubseteq \mathbb{C}$$
$$\forall \mathbb{A} \in \mathscr{A}.\ \bigsqcap \mathscr{A} \sqsubseteq \mathbb{A} \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{C} \sqsubseteq \mathbb{A}) \Longrightarrow \mathbb{C} \sqsubseteq \bigsqcap \mathscr{A}$$

Taking the contrapositive (Property 1.2) to the facts on the left, and instantiating the facts on the right to $\mathscr{A}^{\perp\!\!\!\perp *}$, gives:

$$\forall \mathbb{A} \in \mathscr{A}.\ \left(\bigsqcup \mathscr{A}\right)^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp} \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \mathbb{C}) \Longrightarrow \bigsqcup(\mathscr{A}^{\perp\!\!\!\perp *}) \sqsubseteq \mathbb{C}$$
$$\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq \left(\bigsqcap \mathscr{A}\right)^{\perp\!\!\!\perp} \qquad (\forall \mathbb{A} \in \mathscr{A}.\ \mathbb{C} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}) \Longrightarrow \mathbb{C} \sqsubseteq \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$$

1. We know $(\bigsqcup \mathscr{A})^{\perp\!\!\!\perp} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$ (for each $\mathbb{A}^{\perp\!\!\!\perp} \in \mathscr{A}$), so $(\bigsqcup \mathscr{A})^{\perp\!\!\!\perp} \sqsubseteq \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$. In the reverse direction, suppose $v \in \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$. For every $e \in \bigsqcup \mathscr{A}$, we know there is (at least) one $\mathbb{A} \in \mathscr{A}$ such that $e \in \mathbb{A}$. So since $v \in \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *}) \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$, we know $\langle v \| e \rangle \in \perp\!\!\!\perp$ by definition of orthogonality (Definition 1.1). Therefore, $v \in (\bigsqcup \mathscr{A})^{\perp\!\!\!\perp}$ as well. Dually, for every $e \in \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$ and $v \in \bigsqcup \mathscr{A}$, there it at least one $v \in \mathbb{A} \in \mathscr{A}$, forcing $\langle v \| e \rangle \in \perp\!\!\!\perp$ and thus $e \in (\bigsqcup \mathscr{A})^{\perp\!\!\!\perp}$. So in general $\bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *}) \sqsubseteq (\bigsqcup \mathscr{A})^{\perp\!\!\!\perp}$, making the two sets equal.
2. We know $\mathbb{A}^{\perp\!\!\!\perp} \sqsubseteq (\bigsqcap \mathscr{A})^{\perp\!\!\!\perp}$ (for each $\mathbb{A}^{\perp\!\!\!\perp} \in \mathscr{A}$), so $\bigsqcup(\mathscr{A}^{\perp\!\!\!\perp *}) \sqsubseteq (\bigsqcap \mathscr{A})^{\perp\!\!\!\perp}$. But the reverse direction may not be true: $(\bigsqcap \mathscr{A})^{\perp\!\!\!\perp} \not\sqsubseteq \bigsqcup(\mathscr{A}^{\perp\!\!\!\perp *})$. Suppose that $e \in (\bigsqcap \mathscr{A})^{\perp\!\!\!\perp}$. Consider the possibility that each $\mathbb{A} \in \mathscr{A}$ might contain a term $v_{\mathbb{A}}$ incompatible with $e$ (*i.e.,* $\langle v_{\mathbb{A}} \| e \rangle \notin \perp\!\!\!\perp$), and yet each such $v_{\mathbb{A}}$ might not end up in the intersection of $\mathscr{A}$ ($v_{\mathbb{A}} \notin \bigsqcap \mathscr{A}$). In this case, $e$ is still orthogonal to every term in $\bigsqcap \mathscr{A}$, but there is no individual $\mathbb{A} \in \mathscr{A}$ such that $e \in \mathbb{A}^{\perp\!\!\!\perp}$ because each one has an associated counter-example $v_{\mathbb{A}}$ ruling it out.
3. The last fact follows from the above and triple orthogonal elimination (Property 1.2).

$$\left(\bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})\right)^{\perp\!\!\!\perp\perp\!\!\!\perp} = \left(\bigsqcup \mathscr{A}\right)^{\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp} = \left(\bigsqcup \mathscr{A}\right)^{\perp\!\!\!\perp} = \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *})$$
$$\left(\bigsqcup(\mathscr{A}^{\perp\!\!\!\perp *\perp\!\!\!\perp *})\right)^{\perp\!\!\!\perp} = \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *\perp\!\!\!\perp *\perp\!\!\!\perp *}) = \bigsqcap(\mathscr{A}^{\perp\!\!\!\perp *}) = \left(\bigsqcup \mathscr{A}\right)^{\perp\!\!\!\perp}$$

■

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Take note that the missing direction in the asymmetric property (2) ($(\mathbb{A} \sqcap \mathbb{B})^{\perp\!\!\!\perp} \not\sqsupseteq (\mathbb{A}^{\perp\!\!\!\perp}) \sqcup (\mathbb{B}^{\perp\!\!\!\perp})$) exactly corresponds to the direction of the de Morgan laws which is rejected by intuitionistic logic (the negation of a conjunction does not imply the disjunction of the negations). Instead, we have a weakened version of (2) presented in (3), adding additional applications of orthogonality to restore the symmetric equality rather than an asymmetric refinement. As with other properties like triple orthogonal elimination, this also has a (co)-value restricted variant.

**Lemma 1.14** (Restricted de Morgan)**.** *For any set of pre-candidates $\mathscr{A} \subseteq \mathscr{PC}$:*

$$\left(\bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v*})\right)^{\perp\!\!\!\perp v\perp\!\!\!\perp v} = \bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v*}) = \left(\bigsqcup \mathscr{A}\right)^{v\perp\!\!\!\perp v} = \left(\bigsqcup(\mathscr{A}^{v*\perp\!\!\!\perp*v*\perp\!\!\!\perp*})\right)^{v\perp\!\!\!\perp v}$$

**Proof** Follows from the de Morgan laws (Property 1.13), restricted triple orthogonal elimination (Property 1.5), and the fact that the (co)value restriction $\_^{v}$ distributes over intersection and unions:

$$\left(\bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v*})\right)^{\perp\!\!\!\perp v\perp\!\!\!\perp v} = \left(\bigsqcup \mathscr{A}\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp v} = \left(\bigsqcup \mathscr{A}\right)^{v\perp\!\!\!\perp v} = \bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v*})$$

$$\left(\bigsqcup(\mathscr{A}^{v*\perp\!\!\!\perp*v*\perp\!\!\!\perp*})\right)^{v\perp\!\!\!\perp v} = \bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v*\perp\!\!\!\perp*v*\perp\!\!\!\perp*v*}) = \bigsqcap(\mathscr{A}^{v*\perp\!\!\!\perp*v}) = \left(\bigsqcup \mathscr{A}\right)^{v\perp\!\!\!\perp v}$$

■

With these de Morgan properties of intersection and union, we can be more specific about how the subtype lattice operations $\bigwedge$ and $\bigvee$ on pre-candidates are lifted into the complete versions $\curlywedge$ and $\curlyvee$ that form the subtype lattice among reducibility candidates.

**Lemma 1.15.** *Let $\mathscr{A} \subseteq \mathscr{RC}$ be any set of reducibility candidates.*

$$\curlywedge \mathscr{A} = \left(\bigwedge \mathscr{A}\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \left(\left(\bigcap\{\mathbb{A}^{+} \mid \mathbb{A} \in \mathscr{A}\}\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, \left(\bigcap\{\mathbb{A}^{+} \mid \mathbb{A} \in \mathscr{A}\}\right)^{v\perp\!\!\!\perp}\right)$$

$$\curlyvee \mathscr{A} = \left(\bigvee \mathscr{A}\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \left(\left(\bigcap\{\mathbb{A}^{-} \mid \mathbb{A} \in \mathscr{A}\}\right)^{v\perp\!\!\!\perp}, \left(\bigcap\{\mathbb{A}^{-} \mid \mathbb{A} \in \mathscr{A}\}\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp}\right)$$

**Proof** Follows from de Morgan duality (Property 1.13 and Lemma 1.14) and the fact that reducibility candidates are fixed points of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6). Let $\mathscr{A}^{+} = \{\mathbb{A}^{+} \mid \mathbb{A} \in \mathscr{A}\}$ and $\mathscr{A}^{-} = \{\mathbb{A}^{-} \mid \mathbb{A} \in \mathscr{A}\}$ in the following:

$$\begin{aligned}
\curlywedge \mathscr{A} &= \mathrm{Neg} \bigwedge \mathscr{A} \\
&= ((\bigcup \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcup \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{-v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{-v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{+})^{v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{-v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcup \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= (\bigwedge \mathscr{A})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}
\end{aligned}$$

$$\begin{aligned}
\curlyvee \mathscr{A} &= \mathrm{Pos} \bigvee \mathscr{A} \\
&= ((\bigcup \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcup \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{+v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{+v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{-})^{v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= ((\bigcap \mathscr{A}^{+v*\perp\!\!\!\perp*})^{v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= ((\bigcup \mathscr{A}^{+})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, (\bigcap \mathscr{A}^{-})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}) \\
&= (\bigvee \mathscr{A})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}
\end{aligned}$$

■

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

### *1.4 (Co)induction and (Co)recursion*

We now examine how the (co)inductive types Nat and Stream $A$ are properly defined as reducibility candidates in this orthogonality-based, symmetric model. As shorthand, we will use these two functions on reducibility candidates

$$N : \mathscr{RC} \to \mathscr{RC} \qquad S : \mathscr{RC} \to \mathscr{RC} \to \mathscr{RC}$$
$$N(\mathbb{C}) := \mathrm{Pos}(\{\mathrm{zero}\} \vee \mathrm{succ}(\mathbb{C})) \qquad S_{\mathbb{A}}(\mathbb{C}) := \mathrm{Neg}(\mathrm{head}(\mathbb{A}) \wedge \mathrm{tail}(\mathbb{C}))$$

defined in terms of these operations that lift constructors and destructors onto candidates:

$$\mathrm{succ}(\mathbb{C}) := \{\mathrm{succ}\, V \mid V \in \mathbb{C}\} \quad \mathrm{tail}(\mathbb{C}) := \{\mathrm{tail}\, E \mid E \in \mathbb{C}\} \quad \mathrm{head}(\mathbb{A}) := \{\mathrm{head}\, E \mid E \in \mathbb{A}\}$$

These capture the (co)inductive steps for the iterative definitions of $[\![\mathrm{Nat}]\!]_i$ and $[\![\mathrm{Stream}\, A]\!]_i$:

$$[\![\mathrm{Nat}]\!]_{i+1} = N([\![\mathrm{Nat}]\!]_i) \qquad [\![\mathrm{Stream}\, A]\!]_{i+1} = S_{[\![A]\!]}([\![\mathrm{Stream}\, A]\!]_i)$$

They also capture the all-at-once definitions of $[\![\mathrm{Nat}]\!]$ and $[\![\mathrm{Stream}\, A]\!]$ as

$$[\![\mathrm{Nat}]\!] = \bigwedge\!\!\!\bigwedge \{\mathbb{C} \in \mathscr{RC} \mid N(\mathbb{C}) \leq \mathbb{C}\} \qquad [\![\mathrm{Stream}\, A]\!] = \bigvee \{\mathbb{C} \in \mathscr{RC} \mid \mathbb{C} \leq S_{[\![A]\!]}(\mathbb{C})\}$$

due to the fact that their closure conditions (under zero, succ, and head, tail, respectively) are equivalent to these subtyping conditions.

**Lemma 1.16.** *For all reducibility candidates* $\mathbb{C}$

1. $N(\mathbb{C}) \leq \mathbb{C}$ *if and only if* $\mathrm{zero} \in \mathbb{C}$ *and* $\mathrm{succ}\, V \in \mathbb{C}$ *(for all* $V \in \mathbb{C}$*).*
2. $\mathbb{C} \leq S_{\mathbb{A}}(\mathbb{C})$ *if and only if* $\mathrm{head}\, E \in \mathbb{C}$ *(for all* $E \in \mathbb{A}$*) and* $\mathrm{tail}\, E \in \mathbb{C}$ *(for all* $E \in \mathbb{C}$*).*

**Proof** The "only if" directions follow directly from Lemma 6.7 by the definitions of $N$, $S_{\mathbb{A}}$, and subtyping. That is, we know that $\mathrm{zero} \in N(\mathbb{C})$ and $\mathrm{succ}\, V \in N(\mathbb{C})$ (for all $V \in \mathbb{C}$) by definition of $N$ in terms of Pos, and thus, they must be in $\mathbb{C} \geq N(\mathbb{C})$ by subtyping. Similarly, $\mathrm{head}\, E, \mathrm{tail}\, F \in S_{\mathbb{A}}(\mathbb{C}) \leq \mathbb{C}$ (for all $E \in \mathbb{A}$ and $F \in \mathbb{C}$) by subtyping and the definition of $S$ in terms of Neg.

The "if" direction follows from the universal properties of Pos and Neg (Lemma 6.7): for any reducibility candidate $\mathbb{C} \sqsupseteq \mathbb{B}^{\upsilon}$ $\mathrm{Pos}(\mathbb{B}) \leq \mathbb{C} \leq \mathrm{Neg}(\mathbb{B})$. Now note that

$$N(\mathbb{C}) = \mathrm{Pos}(\{\mathrm{zero}\} \cup \{\mathrm{succ}\, V \mid V \in \mathbb{C}\}, \{\})$$
$$S_{\mathbb{A}}(\mathbb{C}) = \mathrm{Neg}(\{\}, \{\mathrm{head}\, E \mid E \in \mathbb{A}\} \cup \{\mathrm{tail}\, F \mid F \in \mathbb{C}\})$$

Therefore, if $\mathrm{zero} \in \mathbb{C}$ and $\mathrm{succ}\, V \in \mathbb{C}$ (for all $V \in \mathbb{C}$) then

$$N(\mathbb{C}) \leq \mathbb{C} \sqsupseteq (\{\mathrm{zero}\} \cup \{\mathrm{succ}\, V \mid V \in \mathbb{C}\}, \{\})$$

Likewise if $\mathrm{head}\, E \in \mathbb{C}$ and $\mathrm{tail}\, F \in \mathbb{C}$ (for all $E \in \mathbb{A}$ and $F \in \mathbb{C}$) then

$$(\{\}, \{\mathrm{head}\, E \mid E \in \mathbb{A}\} \cup \{\mathrm{tail}\, F \mid F \in \mathbb{C}\}) \sqsubseteq \mathbb{C} \leq S_{\mathbb{A}}(\mathbb{C})$$

■

First, consider the model of the Nat type in terms of the infinite union of approximations: $\bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$. This reducibility candidate should contain safe instances of the recursor. The reason it does is because the presence of a recursor is preserved by the $N$ stepping operation

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

on reducibility candidates, and so it must remain in the final union $\bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ because it is in each approximation $[\![\mathrm{Nat}]\!]_i$.

**Lemma 1.17** (Nat Recursion Step)**.** *For any reducibility candidates* $\mathbb{B}$ *and* $\mathbb{C}$*,*

$$\mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E \in N(\mathbb{C})$$

*for all* $E \in \mathbb{B}$ *whenever the following conditions hold:*

- $v \in \mathbb{B}$*,*
- $w[V/x, W/y] \in \mathbb{B}$ *for all* $V \in \mathbb{C}$ *and* $W \in \mathbb{B}$*, and*
- $\mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E \in \mathbb{C}$ *for all* $E \in \mathbb{B}$*.*

**Proof** Note that the accumulator continuation $E$ changes during the successor step, so we will need to generalize over it. As shorthand, let $E'^{\mathbf{rec}}_{E'} := \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E'$ where $E'$ stands for the given continuation accumulator. It suffices to show (via Corollary 6.9) that for all $E' \in \mathbb{B}$, $\langle \mathrm{zero} \| E'^{\mathbf{rec}}_{E'} \rangle$ and $\langle \mathrm{succ}\, V \| E'^{\mathbf{rec}}_{E'} \rangle$ (for all $V \in \mathbb{C}$). Observe that, given any $E' \in \mathbb{B}$ and $V \in \mathbb{C}$, we have these two possible reductions:

$$\langle \mathrm{zero} \| E'^{\mathbf{rec}}_{E'} \rangle \mapsto \langle v \| E' \rangle \qquad \langle \mathrm{succ}\, V \| E'^{\mathbf{rec}}_{E'} \rangle \mapsto \langle \mu\alpha.\langle V \| E'^{\mathbf{rec}}_{\alpha} \rangle \| \tilde{\mu}y.\langle w[V/x] \| E' \rangle \rangle$$

Now, we note the following series facts:

1. $\langle v \| E' \rangle \in \perp\!\!\!\perp$ because $v, E' \in \mathbb{B}$ by assumption.
2. $w[V/x, W/y] \in \mathbb{B}$ for all $W \in \mathbb{B}$ because $V \in \mathbb{C}$.
3. $\langle w[V/x, W/y] \| E' \rangle \in \perp\!\!\!\perp$ for all $W \in \mathbb{B}$.
4. $\tilde{\mu}y.\langle w[V/x] \| E' \rangle \in \mathbb{B}$ by activation (Lemma 6.3).
5. $\langle V \| E'^{\mathbf{rec}}_{E'} \rangle \in \perp\!\!\!\perp$ for all $E' \in \mathbb{B}$ by the assumption $V, E'^{\mathbf{rec}}_{E'} \in \mathbb{C}$.
6. $\mu\alpha.\langle V \| E'^{\mathbf{rec}}_{\alpha} \rangle \in \mathbb{B}$ by activation (Lemma 6.3).
7. $\langle \mu\alpha.\langle V \| E'^{\mathbf{rec}}_{\alpha} \rangle \| \tilde{\mu}y.\langle w[V/x] \| E' \rangle \rangle \in \perp\!\!\!\perp$.

Therefore, $\langle V \| E'^{\mathbf{rec}}_{E'} \rangle$ reduces to a command in $\perp\!\!\!\perp$ for any $V \in N(\mathbb{C})$. It follows from Property 6.4 and Corollary 6.9 that $E'^{\mathbf{rec}}_{E'} \in N(\mathbb{C})$. ■

**Lemma 1.18** (Nat Recursion)**.** *For any reducibility candidate* $\mathbb{B}$*, if*

- $v \in \mathbb{B}$*,*
- $w[V/x, W/y] \in \mathbb{B}$ *for all* $V \in \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ *and* $W \in \mathbb{B}$*, and*
- $E \in \mathbb{B}$*,*

*then* $\mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E \in \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$.

**Proof** By induction on $i$, $\mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E' \in [\![\mathrm{Nat}]\!]_i$ for all $E' \in \mathbb{B}$:

- (0) $[\![\mathrm{Nat}]\!]_0 = \mathrm{Pos}\{\}$ is the least reducibility candidate w.r.t subtyping, *i.e.,* with the fewest terms and the most coterms, so $\mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\} \mathbf{\ with\ } E' \in [\![\mathrm{Nat}]\!]_0$ trivially.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

- $(i+1)$ Assume the inductive hypothesis: $\mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E' \in [\![\text{Nat}]\!]_i$ for all $E' \in \mathbb{B}$. Applying Lemma 1.17 to $[\![\text{Nat}]\!]_i$, we have (for all $E' \in \mathbb{B}$):

$$\mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E' \in N([\![\text{Nat}]\!]_{i+1}) = [\![\text{Nat}]\!]_{i+1}$$

Thus, we know that the least upper bound of all $[\![\text{Nat}]\!]_i$ in the subtype lattice (Theorem 6.10) contains the instance of this recursor with $E' = E \in \mathbb{B}$ because it is a covalue (Lemma 6.7):

$$\mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$$

■

Showing that the union $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is closed under the succ constructor is more difficult. The simple union $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is clearly closed under succ: every value in $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ must come from some individual approximation $[\![\text{Nat}]\!]_n$, so its successor is in the next one $[\![\text{Nat}]\!]_{n+1}$. However, $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is not just a simple union; it has been completed by Pos, so it might—hypothetically—contain *more* terms which do not come from any individual $[\![\text{Nat}]\!]_n$. Thankfully, this does not happen. It turns out the two unions are one in the same—the Pos completion cannot add anything more because of infinite recursors which can inspect numbers of any size—which lets us show that $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is indeed closed under succ.

**Lemma 1.19** (Nat Choice). $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i = \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$.

**Proof** We already know that $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \leq \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ by definition (since all reducibility candidates are pre-candidates), so it suffices to show the reverse: $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \leq \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$.

Note from Lemma 1.15 that

$$\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i = \left(\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \left(\left(\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i\right)^{-v\perp\!\!\!\perp}, \left(\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i\right)^{-v\perp\!\!\!\perp v\perp\!\!\!\perp}\right)$$

We will proceed by showing there is an $E \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ such that $\langle V \| E\rangle \in \perp\!\!\!\perp$ forces $V \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$. Since we know that every $V \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ and $E \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ forms a safe command $\langle V \| E\rangle \in \perp\!\!\!\perp$, this proves the result.

First, observe that $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ contains the following instance of the recursor (Lemma 1.18):

$$\mathbf{rec}_\infty := \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\ \_ \to x.x\}\ \mathbf{with}\ \alpha \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$$

which has these reductions with zero and succ:

$$\langle \text{zero} \| \mathbf{rec}_\infty\rangle \mapsto \langle \text{zero} \| \alpha\rangle \qquad \langle \text{succ}\, V \| \mathbf{rec}_\infty\rangle \mapsto \langle \mu\alpha.\langle V \| \mathbf{rec}_\infty\rangle \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle$$

The reason why this covalue forces values of $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ into values of $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ depends on the evaluation strategy.

*In call-by-value*, the first successor reduction proceeds as:

$$\langle \text{succ}\, V \| \mathbf{rec}_\infty\rangle \mapsto\!\!\!\mapsto \langle V \| \mathbf{rec}_\infty[\tilde{\mu}x.\langle x \| \alpha\rangle/\alpha]\rangle$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

where the continuation accumulator has been $\tilde{\mu}$-expanded. In general, an arbitrary step in this reduction sequence looks like:

$$\langle \text{succ } V \| \mathbf{rec}_\infty[E/\alpha] \rangle \mapsto\!\!\!\!\rightarrow \langle V \| \mathbf{rec}_\infty[\tilde{\mu}x.\langle x \| E \rangle / \alpha] \rangle$$

The only values (in call-by-value) which do not get stuck with $\mathbf{rec}_\infty$ (*i.e.,* values $V$ such that $\langle V \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$) have the form $\text{succ}^n$ zero which is in $[\![\text{Nat}]\!]_{n+1} \leq \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$.

*In call-by-name*, the successor reduction proceeds as:

$$\langle \text{succ } V \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow \langle V \| \mathbf{rec}_\infty \rangle$$

Call-by-name includes another form of value, $\mu\beta.c$, which is not immediately stuck with $\mathbf{rec}_\infty$. We now need to show that $\langle V \| \mathbf{rec}_\infty \rangle \in \perp\!\!\!\perp$, *i.e.,* $\langle V \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$, forces $V \in [\![\text{Nat}]\!]_n$ for some $n$. Let's examine this reduction more closely and check the intermediate results by abstracting out $\mathbf{rec}_\infty$ with a fresh covariable $\beta$: $\langle V \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$ because $\langle V \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c'[\mathbf{rec}_\infty/\beta]$ for some $\langle V \| \beta \rangle \mapsto\!\!\!\!\rightarrow c' \not\mapsto$ and then $c'[\mathbf{rec}_\infty/\beta] \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$. We now proceed by (strong) induction on the length of the remaining reduction sequence (*i.e.,* the number of steps in $c'[\mathbf{rec}_\infty/\beta] \mapsto\!\!\!\!\rightarrow c$) and by cases on the shape of the intermediate $c'$:

- $c' \neq \langle W \| \beta \rangle$. Then, $c'[\mathbf{rec}_\infty/\beta] \in \mathit{Final}$ already. In this case, $\langle W \| E \rangle \in \perp\!\!\!\perp$ for any $E$ whatsoever by expansion (Property 6.4), and so $W \in [\![\text{Nat}]\!]_0 = \text{Pos}\{\}$.
- $c' = \langle W \| \beta \rangle$. Then we know that $c'[\mathbf{rec}_\infty/\beta] = \langle W[\mathbf{rec}_\infty/\beta] \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$. Since $\langle W \| \beta \rangle \not\mapsto$ we know $W[\mathbf{rec}_\infty/\beta]$ is not a $\mu$-abstraction. The only other possibilities for $W[\mathbf{rec}_\infty/\beta]$, given the known reduction to $c$, are zero or succ $V'$ for some $V'$. zero $\in [\![\text{Nat}]\!]_1$ by definition. In the case of succ $V'$, we have the (non-reflexive) reduction sequence $c'[\mathbf{rec}_\infty/\beta] = \langle \text{succ } V' \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow \langle V' \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c$. The inductive hypothesis on the smaller reduction $\langle V' \| \mathbf{rec}_\infty \rangle \mapsto\!\!\!\!\rightarrow c$ ensures $V' \in [\![\text{Nat}]\!]_n$ for some $n$, so that succ $V' \in [\![\text{Nat}]\!]_{n+1}$ by definition, and thus $V \in [\![\text{Nat}]\!]_{n+1}$ as well by expansion.

So in both call-by-value and call-by-name, we have $\left(\bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^-\right)^{v\perp\!\!\!\perp} \subseteq \bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+$. De Morgan duality (Lemma 1.6) ensures the reverse, so $\left(\bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^-\right)^{v\perp\!\!\!\perp} = \bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+$. Finally, because all reducibility candidates are fixed points of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6), de Morgan duality further implies:

$$\begin{aligned}\curlyvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i &= \left(\left(\bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^-\right)^{v\perp\!\!\!\perp}, \left(\bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^-\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp}\right) = \left(\bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+, \left(\bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+\right)^{v\perp\!\!\!\perp}\right) \\ &= \left(\bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+, \bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^{+v\perp\!\!\!\perp}\right) = \left(\bigcup_{i=0}^{\infty} [\![\text{Nat}]\!]_i^+, \bigcap_{i=0}^{\infty} [\![\text{Nat}]\!]_i^-\right) = \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i\end{aligned}$$

■

Due to the symmetry of the model, the story for Stream $A$ is very much the same as Nat. We can show that the intersection of approximations—$\curlywedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$—contains safe instances of the corecursor because its presence is preserved by the stepping function $S$. The task of showing that this intersection is closed under the tail destructor is more challenging in the same way as succ closure. We solve it with the dual method: the presence of corecursors which "inspect" stream continuations of any size ensures that there are no new surprises that are not already found in one of the approximations $[\![\text{Stream } A]\!]_n$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Lemma 1.20** (Stream Corecursion Step)**.** *For any reducibility candidates* $\mathbb{A}$*,* $\mathbb{B}$ *and* $\mathbb{C}$*,*

$$\mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V \in S_{\mathbb{A}}(\mathbb{C})$$

*for all* $V \in \mathbb{B}$ *whenever the following conditions hold:*

- $e[E/\alpha] \in \mathbb{B}$ *for all* $E \in \mathbb{A}$*,*
- $f[E/\beta, F/\gamma] \in \mathbb{B}$ *for all* $E \in \mathbb{C}$ *and* $F \in \mathbb{B}$*, and*
- $\mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V \in \mathbb{C}$ *for all* $V \in \mathbb{B}$*.*

**Proof** Since the value accumulator $V$ will change in the tail step, we have to generalize over it. Let $V^{\mathbf{corec}}_{V} := \mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V$, where $V$ stands for a given value from $\mathbb{B}$. It suffices to show (via Corollary 6.9) that for all $V \in \mathbb{B}$, $\langle V^{\mathbf{corec}}_{V} \| \text{head}\,E\rangle$ (for all $E \in \mathbb{A}$) and $\langle V^{\mathbf{corec}}_{V} \| \text{tail}\,E'\rangle$ (for all $E' \in \mathbb{C}$). We have these two possible reductions:

$$\langle V^{\mathbf{corec}}_{V} \| \text{head}\,E\rangle \mapsto \langle V \| e[E/\alpha]\rangle$$
$$\langle V^{\mathbf{corec}}_{V} \| \text{tail}\,E'\rangle \mapsto \langle \mu\gamma.\langle V \| f[E'/\beta]\rangle \| \tilde{\mu}x.\langle V^{\mathbf{corec}}_{x} \| E'\rangle\rangle$$

Now, we note the following series of facts

1. $e[E/\alpha] \in \mathbb{B}$ by assumption because $E \in \mathbb{A}$.
2. $\langle V \| e[E/\alpha]\rangle \in \perp\!\!\!\perp$ because $V, e[E/\alpha] \in \mathbb{B}$ by assumption.
3. $f[E'/\beta, F/\gamma] \in \mathbb{B}$ for all $F \in \mathbb{B}$ by because $E' \in \mathbb{C}$.
4. $\langle V \| f[E'/\beta, F/\gamma]\rangle \in \perp\!\!\!\perp$ for all $F \in \mathbb{B}$.
5. $\mu\gamma.\langle V \| f[E'/\beta]\rangle \in \mathbb{B}$ by activation (Lemma 6.3).
6. $\langle V^{\mathbf{corec}}_{V'} \| E'\rangle \in \perp\!\!\!\perp$ for all $V' \in \mathbb{B}$ by the inductive hypothesis since $E' \in [\![\text{Stream}\,A]\!]_i$.
7. $\tilde{\mu}x.\langle V^{\mathbf{corec}}_{x} \| E'\rangle \in \mathbb{B}$ by activation (Lemma 6.3).
8. $\langle \mu\gamma.\langle V \| f[E'/\beta]\rangle \| \tilde{\mu}x.\langle V^{\mathbf{corec}}_{x} \| E'\rangle\rangle \in \perp\!\!\!\perp$.

Therefore, both $\langle V^{\mathbf{corec}}_{V} \| \text{head}\,E\rangle$ and $\langle V^{\mathbf{corec}}_{V} \| \text{tail}\,E'\rangle$ reduce to a command in $\perp\!\!\!\perp$ for any $E \in \mathbb{A}$ and $E' \in \mathbb{C}$. It follows from Property 6.4 and Corollary 6.9 that $V^{\mathbf{corec}}_{V'} \in S_{\mathbb{A}}(\mathbb{C})$. ■

**Lemma 1.21** (Stream Corecursion)**.** *For any reducibility candidate* $\mathbb{B}$*, if*

- $e[E/\alpha] \in \mathbb{B}$ *for all* $E \in [\![A]\!]$*,*
- $f[E/\beta, F/\gamma] \in \mathbb{B}$ *for all* $E \in \curlywedge_{i=0}^{\infty} [\![\text{Stream}\,A]\!]_i$ *and* $F \in \mathbb{B}$*, and*
- $V \in \mathbb{B}$*,*

*then* $\mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V \in \curlywedge_{i=0}^{\infty} [\![\text{Stream}\,A]\!]_i$.

**Proof** By induction on $i$, $\mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V' \in [\![\text{Stream}\,A]\!]_i$ for all $V' \in \mathbb{B}$:

- (0) $[\![\text{Stream}\,A]\!]_0 = \text{Neg}\{\}$ is the greatest reducibility candidate w.r.t subtyping, *i.e.,* with the most terms, so $\mathbf{corec}\{\text{head}\,\alpha \to e \mid \text{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V' \in [\![\text{Stream}\,A]\!]_0$ trivially.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

- $(i+1)$ Assume $\mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}$ **with** $V' \in [\![\mathrm{Stream}\,A]\!]_i$ for all $V' \in \mathbb{B}$. Applying Lemma 1.20 to Stream $\mathbb{A}_i$, we have (for all $V' \in \mathbb{B}$):

$$\mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V' \in S_{[\![A]\!]}([\![\mathrm{Stream}\,A]\!]_i) = [\![\mathrm{Stream}\,A]\!]_{i+1}$$

Thus, we know that in the greatest lower bound of all $[\![\mathrm{Stream}\,A]\!]_i$ in the subtype lattice (Theorem 6.10) contains this corecursor with $V' = V \in \mathbb{B}$ because it is a value (Lemma 6.7):

$$\mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V = \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$$

■

**Lemma 1.22** (Stream Choice). $\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i = \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$

**Proof** In the special case that $[\![A]\!]$ is somehow *completely* empty of values, it must be the least candidate (w.r.t. subtyping), which also makes each $[\![\mathrm{Stream}\,A]\!]_i$ the least candidate as well, forcing $\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ and $\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ to both be equal to the least candidate and thus equal to each other. Otherwise, we may assume that $[\![A]\!]$ contains at least one value.

Note from Lemma 1.15 that

$$\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i = \left(\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i\right)^{v \perp\!\!\!\perp v \perp\!\!\!\perp} = \left(\left(\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}]\!]_i\right)^{+v \perp\!\!\!\perp v \perp\!\!\!\perp}, \left(\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}]\!]_i\right)^{+v \perp\!\!\!\perp}\right)$$

We will proceed by showing there is a $V \in \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ such that $\langle V \| E\rangle \in \perp\!\!\!\perp$ forces $E \in \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$. Since we know that every $E \in \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ and $V \in \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ forms a safe command $\langle V \| E\rangle \in \perp\!\!\!\perp$, this proves the result.

First, we define the following corecursive term:

$$\mathbf{corec}_\infty[V] := \mathbf{corec}\{\mathrm{head}\,\alpha \to \alpha \to \mathrm{tail}\,\_ \to \gamma.\gamma\}\ \mathbf{with}\ V$$

and observe that $\mathbf{corec}_\infty[V] \in \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ (Lemma 1.21) for all $V \in [\![A]\!]$. In general, $\mathbf{corec}_\infty[V]$ has these reductions with head and tail:

$$\langle \mathbf{corec}_\infty[V] \| \mathrm{head}\,E\rangle \mapsto \langle V \| E\rangle \quad \langle \mathbf{corec}_\infty[V] \| \mathrm{tail}\,E\rangle \mapsto \langle \mu\gamma.\langle V \| \gamma\rangle \| \tilde{\mu}x.\langle \mathbf{corec}_\infty[x] \| E\rangle\rangle$$

The reason why this value forces covalues of $\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ into covalues of $\bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$ depends on the evaluation strategy.

*In call-by-name*, the tail reduction proceeds as:

$$\langle \mathbf{corec}_\infty[V] \| \mathrm{tail}\,E\rangle \mapsto\!\!\!\!\to \langle \mathbf{corec}_\infty[\mu\gamma.\langle V \| \gamma\rangle] \| E\rangle$$

where the value accumulator has been $\mu$-expanded. The only covalues (in call-by-name) which do not get stuck with $\mathbf{corec}_\infty$ (*i.e.,* covalues $E$ such that $\langle \mathbf{corec}_\infty[V] \| E\rangle \mapsto\!\!\!\!\to c \in \mathit{Final}$) have the form $\mathrm{tail}^n(\mathrm{head}\,E)$ with $E \in [\![A]\!]$, which is in $[\![\mathrm{Stream}\,A]\!]_{n+1} \geq \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\,A]\!]_i$.

*In call-by-value*, the tail reduction proceeds as:

$$\langle \mathbf{corec}_\infty[V] \| \mathrm{tail}\,E\rangle \mapsto\!\!\!\!\to \langle \mathbf{corec}_\infty[V] \| E\rangle$$

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Call-by-value includes another form of covalue, $\tilde{\mu}y.c$, which is not immediately stuck with $\mathbf{corec}_\infty$. We now need to show that if $V \in [\![A]\!]$ then $\langle \mathbf{corec}_\infty[V] \| E\rangle \in \perp\!\!\!\perp$, *i.e.,* $\langle \mathbf{corec}_\infty[V] \| E\rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$, forces $E \in [\![\mathrm{Stream}\, A]\!]_n$ for some $n$. Let's look at the intermediate results of this reduction sequence by abstracting out $\mathbf{corec}_\infty$ with a fresh variable $y$: $\langle \mathbf{corec}_\infty[V] \| E\rangle \mapsto\!\!\!\!\rightarrow c$ because $\langle \mathbf{corec}_\infty[V] \| E\rangle \mapsto\!\!\!\!\rightarrow c'[\mathbf{corec}_\infty[V]/y]$ for some $\langle y \| E\rangle \mapsto\!\!\!\!\rightarrow c' \not\mapsto$ and then $c'[\mathbf{corec}_\infty[V]/y] \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$. We now proceed by (strong) induction on the length of the remain reduction sequence (*i.e.,* the number of steps in $c'[\mathbf{corec}_\infty[V]/y] \mapsto\!\!\!\!\rightarrow c$) and by cases on the shape of the intermediate $c'$:

- $c' \neq \langle y \| F\rangle$. Then, $c'[\mathbf{corec}_\infty[V]/y] \in \mathit{Final}$ already. In this case, $\langle W \| F\rangle \in \perp\!\!\!\perp$ for any $W$ whatsoever by expansion (Property 6.4), and so $F \in [\![\mathrm{Stream}\, A]\!]_0 = \mathrm{Neg}\{\}$.
- $c' = \langle y \| F\rangle$. Then, $c'[\mathbf{corec}_\infty[V]/y] = \langle \mathbf{corec}_\infty[V] \| F[\mathbf{corec}_\infty[V]/y]\rangle \mapsto\!\!\!\!\rightarrow c \in \mathit{Final}$. Since $\langle y \| F\rangle \not\mapsto$, we know $F[\mathbf{corec}_\infty[V]/y]$ is not a $\tilde{\mu}$-abstraction. The only other possibilities for $F[\mathbf{corec}_\infty[V]/y]$, given the known reduction to $c$, are $\mathrm{head}\, F'$ or $\mathrm{tail}\, E'$. In the first case, we have $\langle \mathbf{corec}_\infty[V] \| \mathrm{head}\, F'\rangle \mapsto \langle V \| F'\rangle \in \perp\!\!\!\perp$ for all $V \in [\![A]\!]$; so $F' \in [\![A]\!]$ by completion of $[\![A]\!]$ and thus $\mathrm{head}\, F' \in [\![\mathrm{Stream}\, A]\!]_1$. In the second case, we have the (non-reflexive) reduction sequence $c'[\mathbf{corec}_\infty[V]/y] = \langle \mathbf{corec}_\infty[V] \| \mathrm{tail}\, E'\rangle \mapsto\!\!\!\!\rightarrow \langle \mathbf{corec}_\infty[V] \| E'\rangle \mapsto\!\!\!\!\rightarrow c$ The inductive hypothesis on the smaller reduction $\langle \mathbf{corec}_\infty[V] \| E'\rangle \mapsto\!\!\!\!\rightarrow c$ ensures $E' \in [\![\mathrm{Stream}\, A]\!]_n$ for some $n$, so that $\mathrm{head}\, E' \in [\![\mathrm{Stream}\, A]\!]_{n+1}$ by definition, and thus $E \in [\![\mathrm{Stream}\, A]\!]_{n+1}$ as well by expansion.

So in both call-by-name and call-by-value, we have $\left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^+\right)^{v\perp\!\!\!\perp} \subseteq \bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-$. De Morgan duality (Lemma 1.6) ensures $\left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^+\right)^{v\perp\!\!\!\perp} = \bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-$. Finally, because all reducibility candidates are fixed points of $\_^{v\perp\!\!\!\perp}$ (Lemma 1.6), de Morgan duality further implies:

$$\begin{aligned}
\curlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i &= \left(\left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^+\right)^{v\perp\!\!\!\perp v\perp\!\!\!\perp}, \left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^+\right)^{v\perp\!\!\!\perp}\right) \\
&= \left(\left(\bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-\right)^{v\perp\!\!\!\perp}, \bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-\right) \\
&= \left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^{-v\perp\!\!\!\perp}, \bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-\right) \\
&= \left(\bigcap_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^+, \bigcup_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i^-\right) = \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i
\end{aligned}$$

■

Now that we know that the iterative interpretations of Nat and Stream $A$ contain all the expected parts—the (de)constructors and (co)recursors—we are ready to show that they are the same as the all-at-once definition given in Fig. 10. More specifically, the iterative $\curlyvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ and $\curlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i$ correspond to the Kleene notion of (least and greatest, respectively) fixed points. Instead, the all-at-once $[\![\mathrm{Nat}]\!]$ and $[\![\mathrm{Stream}\, A]\!]$ correspond to the Knaster-Tarski fixed point definitions. These two correspond because the generating functions $N$ and $S$ are monotonic, and due to the fact that we can choose which approximation each value of $\curlyvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ and covalue of $\curlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i$ comes from (Lemmas 1.19 and 1.22).

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

**Lemma 1.23** (Monotonicity)**.** *Given reducibility candidates* $\mathbb{A}$*,* $\mathbb{B}$*, and* $\mathbb{C}$*, if* $\mathbb{B} \leq \mathbb{C}$ *then* $N(\mathbb{B}) \leq N(\mathbb{C})$ *and* $S_{\mathbb{A}}(\mathbb{B}) \leq S_{\mathbb{A}}(\mathbb{C})$*.*

**Proof** Because each of the (de)constructors, {zero}, succ($\mathbb{C}$), head($\mathbb{A}$), and succ($\mathbb{C}$) are monotonic w.r.t subtyping, as are unions, intersections, Pos, and Neg (Lemma 1.9). ■

**Lemma 6.13** ((Co)Induction Inversion)**.**

$$[\![\text{Nat}]\!] = \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \qquad\qquad [\![\text{Stream } A]\!] = \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$$

**Proof** First note that the values of $\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is closed under zero a and succ:

- zero $\in [\![\text{Nat}]\!]_1 \leq \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ by definition.
- Given $V \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$, we know $V \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ (Lemma 1.19), and thus $V \in [\![\text{Nat}]\!]_n$ for some $n$. So succ $V \in [\![\text{Nat}]\!]_{n+1} \leq \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ by definition.

Similarly, the covalues of $\bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$ is closed under head and tail:

- For all $E \in [\![A]\!]$, head $E \in [\![\text{Stream } A]\!]_1 \leq \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$ by definition.
- Given $E \in \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$, we know $E \in \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$ (Lemma 1.22), and thus $E \in [\![\text{Stream } A]\!]_n$ for some $n$. So tail $E \in [\![\text{Stream } A]\!]_{n+1} \leq \bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i$ by definition.

Because of these closure facts, we know from the definition of $\bigwedge$ and $\bigvee$, respectively, that

$$\bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \geq \bigwedge \{\mathbb{C} \mid (\text{zero} \in \mathbb{C}) \text{ and } (\forall V \in \mathbb{C}.\ \text{succ } V \in \mathbb{C})\} = [\![\text{Nat}]\!]$$

$$\bigwedge_{i=0}^{\infty} [\![\text{Stream } A]\!]_i \leq \bigvee \{\mathbb{C} \mid (\forall E \in [\![A]\!].\ \text{head } E \in \mathbb{C}) \text{ and } (\forall E \in \mathbb{C}.\ \text{tail } E \in \mathbb{C})\} = [\![\text{Stream } A]\!]$$

Going the other way, it we need to show that each approximation $[\![\text{Nat}]\!]_i$ is a subtype of the $\mathbb{C}$s that make up $[\![\text{Nat}]\!]$, and dually that each approximation $[\![\text{Stream } A]\!]_i$ is a supertype of the $\mathbb{C}$s that make up $[\![\text{Stream } A]\!]$. Suppose that $\mathbb{C}$ is any reducibility candidate such that $N(\mathbb{C}) \leq \mathbb{C}$. Then $[\![\text{Nat}]\!]_i \leq \mathbb{C}$ follows by induction on $i$:

- (0) $[\![\text{Nat}]\!]_0 = \text{Pos}\{\}$ is the least reducibility candidate w.r.t subtyping, so $[\![\text{Nat}]\!]_0 \leq \mathbb{C}$.
- $(i+1)$ Assume that $[\![\text{Nat}]\!]_i \leq \mathbb{C}$. The next approximation is $[\![\text{Nat}]\!]_{i+1} = N([\![\text{Nat}]\!]_i)$. Therefore, $[\![\text{Nat}]\!]_{i+1} = N([\![\text{Nat}]\!]_i) \leq N(\mathbb{C}) \leq \mathbb{C}$ by monotonicity of $N$ (Lemma 1.23).

Similarly, suppose that $\mathbb{C}$ is any reducibility candidate such that $S_{[\![A]\!]}(\mathbb{C}) \geq \mathbb{C}$. Then, $[\![\text{Stream } A]\!]_i \geq \mathbb{C}$ follows by induction on $i$:

- (0) $[\![\text{Stream } A]\!]_0 = \text{Neg}\{\}$ is the greatest reducibility candidate w.r.t subtyping, so $[\![\text{Stream } A]\!]_0 \geq \mathbb{C}$ trivially.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

- $(i+1)$ Assume that $[\![\text{Stream}\, A]\!]_i \geq \mathbb{C}$. The next approximation is $[\![\text{Stream}\, A]\!]_{i+1} = S_{[\![A]\!]}([\![\text{Stream}\, A]\!]_i)$. Therefore, $[\![\text{Stream}\, A]\!]_{i+1} = S_{[\![A]\!]}([\![\text{Stream}\, A]\!]_i) \geq S_{[\![A]\!]}(\mathbb{C}) \geq \mathbb{C}$ by monotonicity of $S_{[\![A]\!]}$ (Lemma 1.23).

In other words, we know that every $\mathbb{C} \geq N(\mathbb{C})$ is an upper bound of all approximations $[\![\text{Nat}]\!]_i$, and every $\mathbb{C} \leq S_{[\![A]\!]}(\mathbb{C})$ is a lower bound of all approximations $[\![\text{Stream}\, A]\!]_i$. So because $\curlyvee$ is the *least* upper bound and $\curlywedge$ is the *greatest* lower bound, we have

$$\bigcurlyvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \leq \mathbb{C} \qquad (\text{if } \mathbb{C} \geq N(\mathbb{C})) \qquad \mathbb{C} \leq \bigcurlywedge_{i=0}^{\infty} \text{Stream}\, \mathbb{A}_i \qquad (\text{if } \mathbb{C} \leq S_{[\![A]\!]}(\mathbb{C}))$$

In other words, $\curlyvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i$ is a lower bound of the $\mathbb{C}$s that make up $[\![\text{Nat}]\!]$ and $\curlywedge_{i=0}^{\infty} \text{Stream}\, \mathbb{A}_i$ is an upper bound of the $\mathbb{C}$s that make up $[\![\text{Stream}\, A]\!]$ (Lemma 1.16). Again, since $\curlywedge$ is the *greatest* lower bound and $\curlyvee$ is the *least* upper bound, we have

$$\bigcurlyvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i \leq \curlywedge\{\mathbb{C} \mid \mathbb{C} \geq N(\mathbb{C})\} = [\![\text{Nat}]\!] \qquad \bigcurlywedge_{i=0}^{\infty} \text{Stream}\, \mathbb{A}_i \geq \curlyvee\{\mathbb{C} \mid \mathbb{C} \leq S_{[\![A]\!]}(\mathbb{C})\} = [\![\text{Stream}\, A]\!]$$

■

### *1.5 Adequacy*

To conclude, we give the full proof of adequacy (Lemma 6.11) here. With the lemmas that precede in Appendix 1.4, the remaining details are now totally standard. Soundness ensures the safety of the *Cut* rule and completeness ensures that the terms of each type are included in their interpretations as reducibility candidates. The main role of adequacy is to show that the guarantees given by the premises of each typing rule are strong enough to prove their conclusion and that the notion of substitution corresponds to the interpretation of typing environments.

**Lemma 6.11** (Adequacy)**.**

1. *If* $\Gamma \vdash c$ *is derivable then* $[\![\Gamma \vdash c]\!]$ *is true.*
2. *If* $\Gamma \vdash v : A$ *is derivable then* $[\![\Gamma \vdash v : A]\!]$ *is true.*
3. *If* $\Gamma \vdash e \div A$ *is derivable then* $[\![\Gamma \vdash e \div A]\!]$ *is true.*

**Proof** By (mutual) induction on the given typing derivation for the command or (co)term:

- (*Cut*) *Inductive Hypothesis*: $[\![\Gamma \vdash v : A]\!]$ and $[\![\Gamma \vdash e \div A]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so $v[\rho] \in [\![A]\!]$ and $e[\rho] \in [\![A]\!]$ by the inductive hypothesis. Observe that $\langle v \| e \rangle[\rho] = \langle v[\rho] \| e[\rho] \rangle \in \perp\!\!\!\perp$ because all reducibility candidates are sound. In other words, $[\![\Gamma \vdash \langle v \| e \rangle]\!]$.
- (*VarR* and *VarL*) $x[\rho] \in [\![A]\!]$ for any $\rho \in [\![\Gamma, x : A]\!]$ by definition. Dually, $\alpha[\rho] \in [\![A]\!]$ for any $\rho \in [\![\Gamma, \alpha \div A]\!]$ by definition. In other words, $[\![\Gamma, x : A \vdash x : A]\!]$ and $[\![\Gamma, \alpha \div A \vdash \alpha \div A]\!]$.
- (*ActR*) *Inductive Hypothesis*: $[\![\Gamma, \alpha \div A \vdash c]\!]$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Let $\rho \in [\![\Gamma]\!]$, so that for all $E \in [\![A]\!]$, $E/\alpha, \rho \in [\![\Gamma, \alpha \div A]\!]$ by definition and $c[\rho][E/\alpha] = c[\rho, E/\alpha] \in \perp\!\!\!\perp$ by the inductive hypothesis. Thus, $(\mu\alpha.c)[\rho] = \mu\alpha.(c[\rho]) \in [\![A]\!]$ by activation (Lemma 6.3). In other words, $[\![\Gamma \vdash \mu\alpha.c : A]\!]$.

- (*ActL*) Dual to *ActR* above.
- ($\to R$) *Inductive Hypothesis*: $[\![\Gamma, x : A \vdash v : B]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so for all $W \in [\![A]\!]$, $W/x, \rho \in [\![\Gamma, x : A]\!]$ by definition and $v[\rho][W/x] = v[\rho, W/x] \in [\![B]\!]$ by the inductive hypothesis. Thus, $(\lambda x.v)[\rho] = \lambda x.(v[\rho]) \in [\![A \to B]\!]$ by Lemma 6.12. In other words $[\![\Gamma \vdash \lambda x.v : A \to B]\!]$.
- ($\to L$) *Inductive Hypothesis*: $[\![\Gamma \vdash V : A]\!]$ and $[\![\Gamma \vdash E \div B]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so $V[\rho] \in [\![A]\!]$ and $E[\rho] \in [\![B]\!]$ by the inductive hypothesis. Thus, $(V \cdot E)[\rho] = V[\rho] \cdot E[\rho] \in [\![A \to B]\!]$ by definition of $[\![A \to B]\!]$ and Lemma 6.7. In other words, $[\![\Gamma \vdash V \cdot E \div A \to B]\!]$.
- (Nat$R_{\text{zero}}$): For any substitution $\rho$, $\text{zero}[\rho] = \text{zero} \in [\![\text{Nat}]\!]$ by Lemma 6.7. In other words, $[\![\Gamma \vdash \text{zero} : \text{Nat}]\!]$.
- (Nat$R_{\text{succ}}$) *Inductive Hypothesis*: $[\![\Gamma \vdash V : \text{Nat}]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so that $V[\rho] \in [\![\text{Nat}]\!]$ by the inductive hypothesis. Thus, $(\text{succ}\, V)[\rho] = \text{succ}(V[\rho]) \in [\![\text{Nat}]\!]$ by Lemma 6.7. In other words, $[\![\Gamma \vdash \text{succ}\, V : \text{Nat}]\!]$.
- (Nat*L*) *Inductive Hypothesis*: $[\![\Gamma \vdash v : A]\!]$, $[\![\Gamma, x : \text{Nat}, y : A \vdash w : A]\!]$, and $[\![\Gamma \vdash E \div A]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so that by the inductive hypothesis:
  - $E[\rho] \in [\![A]\!]$,
  - $v[\rho] \in [\![A]\!]$, and
  - $w[\rho][V/x, W/y] = w[\rho, V/x, W/y] \in [\![A]\!]$ for all $V \in [\![\text{Nat}]\!]$ and $W \in [\![A]\!]$.

  Thus,

$$\begin{aligned} &\mathbf{rec}\{\text{zero} \to v[\rho] \mid \text{succ}\, x \to y.w[\rho]\}\ \mathbf{with}\ E[\rho] \\ &= (\mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E)[\rho] \in \bigvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i = [\![\text{Nat}]\!] \end{aligned}$$

  by Lemmas 6.13 and 1.18. In other words,

$$[\![\Gamma \vdash \mathbf{rec}\{\text{zero} \to v \mid \text{succ}\, x \to y.w\}\ \mathbf{with}\ E \div \text{Nat}]\!]$$

- (Stream*R*) *Inductive Hypothesis*: $[\![\Gamma \vdash E \div A]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so that $E[\rho] \in [\![A]\!]$ by the inductive hypothesis. Thus, $(\text{head}\, E)[\rho] = \text{head}(E[\rho]) \in [\![\text{Stream}\, A]\!]$ by Lemma 6.7. In other words, $[\![\Gamma \vdash \text{head}\, E \div \text{Stream}\, A]\!]$.
- (Stream$L_{\text{head}}$) *Inductive Hypothesis*: $[\![\Gamma \vdash E \div \text{Stream}\, A]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so that $E[\rho] \in [\![\text{Stream}\, A]\!]$ by the inductive hypothesis. Thus, $(\text{tail}\, E)[\rho] = \text{tail}(E[\rho]) \in [\![\text{Stream}\, A]\!]$ by Lemma 6.7. In other words, $[\![\Gamma \vdash \text{tail}\, E \div \text{Stream}\, A]\!]$.
- (Stream$L_{\text{tail}}$) *Inductive Hypothesis*: $[\![\Gamma, \alpha \div A \vdash e \div B]\!]$, $[\![\Gamma, \beta \div \text{Stream}\, A, \gamma \div B \vdash f \div B]\!]$, and $[\![\Gamma \vdash V : B]\!]$.
  Let $\rho \in [\![\Gamma]\!]$, so that by the inductive hypothesis:
  - $V[\rho] \in [\![B]\!]$,
  - $e[\rho][E/\alpha] = e[\rho, E/\alpha] \in [\![B]\!]$ for all $E \in [\![A]\!]$, and
  - $f[\rho][E/\beta, F/\gamma] = f[\rho, E/\beta, F/\gamma] \in [\![B]\!]$ for all $E \in [\![\text{Stream}\, A]\!]$ and $F \in [\![B]\!]$.

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press

Thus,

$$\begin{aligned}&\mathbf{corec}\{\mathrm{head}\,\alpha \to e[\rho] \mid \mathrm{tail}\,\beta \to \gamma.f[\rho]\}\ \mathbf{with}\ V[\rho]\ {}_{\infty}\\ &= (\mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V)[\rho] \in \bigwedge_{i=0} [\![\mathrm{Stream}\,A]\!]_i = [\![\mathrm{Stream}\,A]\!]\end{aligned}$$

by Lemmas 6.13 and 1.21. In other words

$$[\![\Gamma \vdash \mathbf{corec}\{\mathrm{head}\,\alpha \to e \mid \mathrm{tail}\,\beta \to \gamma.f\}\ \mathbf{with}\ V : \mathrm{Stream}\,A]\!]$$

■

https://doi.org/10.1017/S0956796822000168 Published online by Cambridge University Press