



Integrating Induction and Coinduction via Closure Operators and Proof Cycles

Liron Cohen¹  and Reuben N. S. Rowe²  

¹ Department of Computer Science, Ben-Gurion University, Beersheba, Israel
`cliron@cs.bgu.ac.il`

² Department of Computer Science, Royal Holloway University of London,
Egham, UK
`reuben.rowe@rhul.ac.uk`

Abstract. Coinductive reasoning about infinitary data structures has many applications in computer science. Nonetheless developing natural proof systems (especially ones amenable to automation) for reasoning about coinductive data remains a challenge. This paper presents a minimal, generic formal framework that uniformly captures applicable (i.e. finitary) forms of inductive and coinductive reasoning in an intuitive manner. The logic extends transitive closure logic, a general purpose logic for inductive reasoning based on the transitive closure operator, with a dual ‘co-closure’ operator that similarly captures applicable coinductive reasoning in a natural, effective manner. We develop a sound and complete non-well-founded proof system for the extended logic, whose cyclic subsystem provides the basis for an effective system for automated inductive and coinductive reasoning. To demonstrate the adequacy of the framework we show that it captures the canonical coinductive data type: streams.

1 Introduction

The principle of induction is used widely in computer science for reasoning about data types such as numbers or lists. The lesser-known principle of coinduction is used for reasoning about coinductive data types, which are data structures containing non-well-founded elements, e.g. infinite streams or trees [7, 25, 27, 32, 35, 37, 44, 46, 48]. A duality between the two principles is observed when formulating them within an algebraic, or categorical, framework [49]. However, such formulation does not account well for the way these principles are commonly used in deduction, where there is a mismatch in how they are usually applied.

Due to this tension between the abstract theory of coalgebras and its implementation in formal frameworks [41], coinductive reasoning is generally not fully and naturally incorporated into major proof assistants (e.g. Coq [7], Nuprl [20], Agda [8], Idris [9] and Dafny [36]). Even in notable exceptions such as [33, 36, 38, 44] the combination of induction and coinduction is not intuitively accounted for. The standard approach in such formalisations is to define

inductive data with constructors and coinductive data with destructors, or observations [1]. In this paper we propose an alternative approach to formally integrating induction and coinduction that clearly reveals the duality between the two principles. Our approach has the advantage that the same signature is shared for both inductive and coinductive data, making certain aspects of the relationship between the two principles more apparent. To achieve this, we extend and combine two powerful frameworks: semantically we follow the approach of transitive closure logic, a generic logic for expressing inductive structures [3, 14–16, 31, 39, 51]; for deduction, we adopt non-well-founded proof theory [2, 5, 10–12, 17–19, 23, 24, 26, 50, 55]. This combination captures the intuitive dynamics of inductive and coinductive reasoning, reflecting how these principles are understood and applied in practice.

Transitive closure (RTC) logic minimally extends first-order logic by adding a single, intuitive notion: an operator, RTC , for forming the (reflexive) transitive closures of an arbitrary formula (more precisely, of the binary relation induced by the formula). This operator alone is sufficient for capturing all finitary induction schemes within a single, unified language (unlike other systems that are *a priori* parametrized by a set of inductive definitions [12, 40, 42, 58]). Transitive closures arise as least fixed points of certain composition operators. In this paper we extend RTC logic with the semantically dual notion: an operator, RTC^{op} , for forming *greatest* fixed points of these same composition operators.¹ We call these *transitive co-closures*, and show that they are equally as intuitive. Just as transitive closure captures induction, we show that transitive co-closure facilitates coinductive definitions and reasoning.

Non-well-founded proof theory formalises the infinite-descent style of induction. It enables a separation between local steps of deductive inference and global well-foundedness arguments (i.e. induction), which are encoded in traces of formulas through possibly infinite derivations. A major benefit of these systems is that inductive invariants do not need to be explicit. On the other hand, existing approaches for combining induction and coinduction rely on making (co)invariants explicit within proofs [4, 30, 59]. In previous work, a non-well-founded proof system for RTC logic was developed [17, 18]. In this paper, we show that the meaning of the transitive co-closure operator can be captured proof-theoretically using inference rules having *the exact same structure*, with soundness now requiring infinite *ascent* (i.e. showing productivity) rather than *descent*. What obtains is a proof system in which induction and coinduction are smoothly integrated, and which very clearly highlights their similarities. Their differences are also thrown into relief, consisting in the way formulas are traced in a proof derivation. Specifically, traces of RTC formulas show that certain infinite paths *cannot* exist (induction is well-founded), while traces of RTC^{op} formulas show that other infinite paths *must* exist (coinduction is productive).

To demonstrate that our system naturally captures patterns of mixed inductive/coinductive reasoning, we formalise one of the most well-known examples of a coinductive data type: streams. In particular, we consider two illustrative

¹ The notation RTC^{op} comes from the categorical notion of the opposite (dual) category.

examples: transitivity of the lexicographic ordering on streams; and transitivity of the substream relation. Both are known to be hard to prove. Our system handles these without recourse to general fixpoint operators or algebraic structures.

The transitive (co-)closure framework is contained in the first-order mu-calculus [43], but offers several advantages. The concept of transitive (co-)closure is intuitively simpler than that of general fixed-point operators, and does not require any syntactic restrictions to ensure monotonicity. Our framework is also related, but complementary to logic programming with coinductive interpretations [52, 53] and its coalgebraic semantics [34]. Logic programs, built from Horn clauses, have a fixed intended domain (*viz.* Herbrand universes), and the semantics of mixing inductive and coinductive interpretations is subtle. Our framework, on the other hand, uses a general syntax that can freely mix closures and co-closures, and its semantics considers all first-order models. Furthermore, the notion of proof in our setting is more general than the (semantic) notion of proof in logic programming, in which, for instance, there is no analogous concept of global trace condition.

Outline. Section 2 presents the syntax and semantics of the extended logic, RTcC. Section 3 describes how streams and their properties can be expressed in RTcC. Section 4 presents non-well-founded proof systems for RTcC, showing soundness and completeness. Section 5 then illustrates how the examples of Sect. 3 are formalised in this system. Section 6 concludes with directions for future work.

2 RTcC Logic: Syntax and Semantics

Transitive closure (RTC) logic [3, 15] extends the language of first-order logic with a predicate-forming operator, RTC , for denoting the (reflexive) transitive closures of (binary) relations. In this section we extend RTC logic into what we call *transitive (co-)closure (RTcC) logic*, by adding a single *transitive co-closure* operator, RTC^{op} . Roughly speaking, whilst the RTC operator denotes the set of all pairs that are related via a *finite* chain (or path), the RTC^{op} operator gives the set of all pairs that are ‘related’ via a *possibly infinite* chain. In Sect. 3 we show that this allows capturing coinductive definitions and reasoning.

For simplicity of presentation we assume (as is standard practice) a designated equality symbol. Note also that we use the *reflexive* transitive closure; however the reflexive and non-reflexive forms are equivalent in the presence of equality.

Definition 1 (RTcC Formulas). *Let s, t and P range over the terms and predicate symbols, respectively, of a first-order signature Σ . The language $\mathcal{L}_{\text{RTcC}}$ (of formulas over Σ) is given by the following grammar:*

$$\begin{aligned} \varphi, \psi ::= & s = t \mid P(t_1, \dots, t_n) \mid \neg\varphi \mid \forall x. \varphi \mid \exists x. \varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \\ & (RTC_{x,y} \varphi)(s, t) \mid (RTC_{x,y}^{\text{op}} \varphi)(s, t) \end{aligned}$$

where the variables x and y in the formulas $(RTC_{x,y} \varphi)(s, t)$ and $(RTC_{x,y}^{\text{op}} \varphi)(s, t)$ must be distinct and are bound in the subformula φ , referred to as the body.

The semantics of formulas is an extension of the standard semantics of first-order logic. We write M and ν to denote a first-order structure over a (non-empty) domain D and a valuation of variables in D , respectively. We denote by $\nu[x_1 := d_n, \dots, x_n := d_n]$ the valuation that maps x_i to d_i for each i and behaves as ν otherwise. We write $\varphi\{t_1/x_1, \dots, t_n/x_n\}$ for the result of simultaneously substituting each t_i for the free occurrences of x_i in φ . We use $(\mathbf{d}_i)_{i \leq n}$ to denote a *non-empty* sequence of elements d_1, \dots, d_n ; and $(\mathbf{d}_i)_{i > 0}$ for a (countably) *infinite* sequence of elements d_1, d_2, \dots . We use \equiv to denote syntactic equality.

Definition 2 (Semantics). *Let M be a structure for \mathcal{L}_{RTC} , and ν a valuation in M . The satisfaction relation $M, \nu \models \varphi$ extends the standard satisfaction relation of classical first-order logic with the following clauses:*

$$\begin{aligned}
 M, \nu \models (RTC_{x,y} \varphi)(s, t) &\Leftrightarrow \\
 \exists (\mathbf{d}_i)_{i \leq n} . d_1 = \nu(s) \wedge d_n = \nu(t) \wedge \forall i < n . M, \nu[x := d_i, y := d_{i+1}] \models \varphi \\
 M, \nu \models (RTC_{x,y}^{\text{op}} \varphi)(s, t) &\Leftrightarrow \\
 \exists (\mathbf{d}_i)_{i > 0} . d_1 = \nu(s) \wedge \forall i > 0 . d_i = \nu(t) \vee M, \nu[x := d_i, y := d_{i+1}] \models \varphi
 \end{aligned}$$

Intuitively, the formula $(RTC_{x,y} \varphi)(s, t)$ asserts that there is a (possibly empty) finite φ -path from s to t . The formula $(RTC_{x,y}^{\text{op}} \varphi)(s, t)$ asserts that either there is a (possibly empty) finite φ -path from s to t , or an infinite φ -path starting at s .

We can connect these closure operators to the general theory of fixed points, with $(RTC_{x,y} \varphi)$ and $(RTC_{x,y}^{\text{op}} \varphi)$ denoting, respectively, the least and greatest fixed points of a certain operator on binary relations.

Definition 3 (Composition Operator). *Given a binary relation X , we define an operator Ψ_X on binary relations, which post-composes its input with X , by: $\Psi_X(R) = X \cup (X \circ R) = \{(a, c) \mid (a, c) \in X \vee \exists b . (a, b) \in X \wedge (b, c) \in R\}$.*

Notice that the set of all binary relations (over some given domain) forms a complete lattice under the subset ordering \subseteq . Moreover, composition operators Ψ_X are monotone w.r.t. \subseteq . Thus we have the following standard results, from the Knaster–Tarski theorem. For any binary relation X , the least fixed point $\mathbf{lfp}(\Psi_X)$ of Ψ_X is given by $\mathbf{lfp}(\Psi_X) = \bigcap \{R \mid \Psi_X(R) \subseteq R\}$, i.e. the intersection of all its prefixed points. Dually, the greatest fixed point $\mathbf{gfp}(\Psi_X)$ of Ψ_X is given by the union of all its postfixed points, i.e. $\mathbf{gfp}(\Psi_X) = \bigcup \{R \mid R \subseteq \Psi_X(R)\}$. Via the usual notion of formula definability, RTC and RTC^{op} are easily seen to be fixed point operators. For a model M and valuation ν , denote the binary relation defined by a formula φ with respect to x and y by $\llbracket \varphi \rrbracket_{x,y}^{M,\nu} = \{(a, b) \mid M, \nu[x := a, y := b] \models \varphi\}$.

Proposition 1. *The following hold.*

- (i) $M, \nu \models (RTC_{x,y} \varphi)(s, t)$ iff $\nu(s) = \nu(t)$ or $(\nu(s), \nu(t)) \in \mathbf{lfp}(\Psi_{\llbracket \varphi \rrbracket_{x,y}^{M,\nu}})$.
- (ii) $M, \nu \models (RTC_{x,y}^{\text{op}} \varphi)(s, t)$ iff $\nu(s) = \nu(t)$ or $(\nu(s), \nu(t)) \in \mathbf{gfp}(\Psi_{\llbracket \varphi \rrbracket_{x,y}^{M,\nu}})$.

Note that labelling the co-closure ‘transitive’ is justified since, for any model M , valuation ν , and formula φ , the relation $\mathbf{gfp}(\Psi_{\llbracket \varphi \rrbracket_{x,y}^{M,\nu}})$ is indeed transitive.

The RTC^{op} operator enjoys dualisations of properties governing the transitive closure operator (see, e.g., [16, Proposition 3]) *that are either symmetrical, or involve the first component*. This is because the semantics of the RTC^{op} has an embedded asymmetry between the arguments. Reasoning about closures is based on decomposition into one step and the remaining path. For RTC , this decomposition can be done in both directions, but for RTC^{op} it can only be done in one direction.

Proposition 2. *The following formulas, connecting the two operators, are valid.*

- i) $(RTC_{x,y} \varphi)(s, t) \rightarrow (RTC_{x,y}^{\text{op}} \varphi)(s, t)$
- ii) $\neg(RTC_{x,y} \neg\varphi)(s, t) \rightarrow (RTC_{x,y}^{\text{op}} \varphi)(s, t)$
- iii) $\neg(RTC_{x,y}^{\text{op}} \neg\varphi)(s, t) \rightarrow (RTC_{x,y} \varphi)(s, t)$
- iv) $((RTC_{x,y}^{\text{op}} \varphi)(s, t) \wedge \exists z. \neg(RTC_{x,y}^{\text{op}} \varphi)(s, z)) \rightarrow (RTC_{x,y} \varphi)(s, t)$
- v) $((RTC_{x,y}^{\text{op}} \varphi)(s, t) \wedge \neg(RTC_{x,y}^{\text{op}} \varphi \wedge y \neq t)(s, t)) \rightarrow (RTC_{x,y} \varphi)(s, t)$

Note that the converse of these properties do not hold in general, thus they do not provide characterisations of one operator in terms of the other. A counterexample for the converses of (ii) and (iii) can be obtained by taking φ to be $x = y$. Then, for any domain D , the formulas $(RTC_{x,y} \neg\varphi)$, $(RTC_{x,y}^{\text{op}} \varphi)$, and $(RTC_{x,y}^{\text{op}} \neg\varphi)$ all denote the full binary relation $D \times D$, while $(RTC_{x,y} \varphi)$ denotes the identity relation on D .

3 Streams in RTcC Logic

This section demonstrates the adequacy of RTcC logic for formalising and reasoning about coinductive data types. As claimed by Rutten: “streams are the best known example of a final coalgebra and offer a perfect playground for the use of coinduction, both for definitions and for proofs.” [47]. Hence, in this section and Sect. 5 we illustrate that RTcC logic naturally captures the stream data type (see, e.g., [29, 48]).

3.1 The Stream Datatype

We formalise streams as infinite lists, using a signature consisting of the standard list constructors: the constant nil and the (infix) binary function symbol ‘ $::$ ’, traditionally referred to as ‘cons’. These are axiomatized by:

$$\text{nil} = e :: \sigma \Rightarrow \quad (1) \quad e :: \sigma = e' :: \sigma' \Rightarrow e = e' \quad (2) \quad e :: \sigma = e' :: \sigma' \Rightarrow \sigma = \sigma' \quad (3)$$

Note that for simplicity of presentation we have not specified that the elements of possibly infinite lists should be any particular sort (e.g. numbers). Thus, the theory of streams we formulate here is generic in this respect. To refer specifically to streams over a particular domain, we could use a multisorted signature

containing a **Base** sort, in addition to the sort List^∞ of possibly infinite lists, with nil a constant of type List^∞ and $::$ a function of type $\text{Base} \times \text{List}^\infty \rightarrow \text{List}^\infty$. Nonetheless, we do use the following conventions for formalising streams in this section and in Sect. 5. For variables and terms ranging over **Base** we use a, b, c, \dots and e, e', \dots , respectively; and for variables and terms ranging over possibly infinite lists we use x, y, z, \dots and σ, σ', \dots , respectively.

The (graphs of) the standard head (hd) and tail (tl) functions are definable² by $\text{hd}(\sigma) = e \stackrel{\text{def}}{:=} \exists x. \sigma = e :: x$ and $\text{tl}(\sigma) = \sigma' \stackrel{\text{def}}{:=} \exists a. \sigma = a :: \sigma'$. Finite and possibly infinite lists can be defined by using the transitive closure and co-closure operators, respectively, as follows.

$$\begin{aligned} \text{List}(\sigma) &\stackrel{\text{def}}{:=} (\text{RTC}_{x,y} \text{tl}(x) = y)(\sigma, \text{nil}) \\ \text{List}^\infty(\sigma) &\stackrel{\text{def}}{:=} (\text{RTC}_{x,y}^{\text{op}} \text{tl}(x) = y)(\sigma, \text{nil}) \end{aligned}$$

Roughly speaking, these formulas assert that we can perform some number of successive tail decompositions of the term σ . For the RTC formula, this decomposition must reach the second component, nil , in a finite number of steps. For the RTC^{op} formula, on the other hand, the decomposition is not required to reach nil but, in case it does not, must be able to continue indefinitely.

To define the notion of a necessarily infinite list (i.e. a stream), we specify in the body that, at each step, the decomposition of the stream cannot actually reach nil (abbreviating $\neg(s = t)$ by $s \neq t$). Moreover, since we are using reflexive forms of the operators we must also stipulate that nil itself is not a stream.

$$\text{Stream}(\sigma) \stackrel{\text{def}}{:=} (\text{RTC}_{x,y}^{\text{op}} \text{tl}(x) = y \wedge y \neq \text{nil})(\sigma, \text{nil}) \wedge \sigma \neq \text{nil}$$

This technique—of specifying that a single step cannot reach nil and then taking nil to be the terminating case in the RTC^{op} formula—is a general method we will use in order to restrict attention to the infinite portion in the induced semantics of an RTC^{op} formula. To this end, we define the following notation.

$$\overline{\varphi}_{x,y}^{\text{inf}}(\sigma) \stackrel{\text{def}}{:=} (\text{RTC}_{x,y}^{\text{op}} (\varphi \wedge y \neq \text{nil}))(\sigma, \text{nil}) \wedge \sigma \neq \text{nil}$$

3.2 Relations and Operations on Streams

We next show that RTcC also naturally captures properties of streams. Using the RTC operator we can (inductively) define the extension relation \triangleleft on possibly infinite lists as follows:

$$\sigma \triangleleft \sigma' \stackrel{\text{def}}{:=} (\text{RTC}_{x,y} \text{tl}(x) = y)(\sigma, \sigma')$$

This asserts that σ extends σ' , i.e. that σ is obtained from σ' by prepending some finite sequence of elements to σ' . Equivalently, σ' is obtained by some finite number of tail decompositions from σ : that is, σ' is a suffix of σ .

² Although $\text{hd}(\sigma)$ and $\text{tl}(\sigma)$ could have been defined as terms using Russell's ι operator, we opted for the above definition for simplicity of the proof theory.

We next formalise some standard predicates.

$$\begin{aligned} \text{Contains}(e, \sigma) & \stackrel{\text{def}}{=} \exists x . \sigma \triangleleft x \wedge \text{hd}(x) = e \\ \text{Const}(e, \sigma) & \stackrel{\text{def}}{=} \overline{(x = e :: y)}_{x,y}^{\text{inf}}(\sigma) \\ \text{Const}_{\infty}(\sigma) & \stackrel{\text{def}}{=} \exists x . \sigma \triangleleft x \wedge \exists a . \text{Const}(a, x) \end{aligned}$$

$\text{Contains}(e, \cdot)$ defines the possibly infinite lists that contain the element denoted by e ; $\text{Const}(e, \cdot)$ defines the constant stream consisting of the element denoted by e ; and Const_{∞} defines streams that are eventually constant.

We next consider how (functional) relations on streams can be formalised in RTcC, using some illustrative examples. To capture these we need to use ordered pairs. For this, we use the notation $\langle u, v \rangle$ for $u :: (v :: \text{nil})$,³ then abbreviate $(RTC_{w,w'} \exists u, u', v, v' . w = \langle u, v \rangle \wedge w' = \langle u', v' \rangle \wedge \varphi)$ by $(RTC_{\langle u, v \rangle, \langle u', v' \rangle} \varphi)$ (and similarly for RTC^{op} formulas), and also write $\overline{\varphi}_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle}^{\text{inf}}(\langle \sigma, \sigma' \rangle)$ to stand for $(RTC_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle}^{\text{op}}(\varphi \wedge y_1 \neq \text{nil} \wedge y_2 \neq \text{nil}))(\langle \sigma, \sigma' \rangle, \langle \text{nil}, \text{nil} \rangle) \wedge \sigma \neq \text{nil} \wedge \sigma' \neq \text{nil}$.

Append and Periodicity. With ordered pairs, we can inductively define (the graph of) the function that appends a possibly infinite list to a finite list.

$$\begin{aligned} \sigma_1 \widehat{\ } \sigma_2 = \sigma_3 & \stackrel{\text{def}}{=} \\ & (RTC_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle} \exists a . x_1 = a :: y_1 \wedge x_2 = a :: y_2)(\langle \sigma_1, \sigma_3 \rangle, \langle \text{nil}, \sigma_2 \rangle) \end{aligned}$$

We remark that the formulas $\sigma \triangleleft \sigma'$ and $\exists z . z \widehat{\ } \sigma' = \sigma$ are equivalent. To define this as a *function* requires also proofs that the defined relation is total and functional. However, this is generally straightforward when the body formula is deterministic, as is the case in all the examples we present here. Other standard operations on streams, such as element-wise operations, are also definable in RTcC as (functional) relations. For example, assuming a unary function \oplus , we can coinductively define its elementwise extension to streams \oplus_{∞} as follows.

$$\oplus_{\infty}(\sigma) = \sigma' \stackrel{\text{def}}{=} \overline{(\exists a . x_1 = a :: y_1 \wedge x_2 = \oplus(a) :: y_2)}_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle}^{\text{inf}}(\langle \sigma, \sigma' \rangle)$$

As an example of mixing induction and coinduction, we can express a predicate coinductively defining the *periodic* streams using the append function.

$$\text{Periodic}(\sigma) \stackrel{\text{def}}{=} \exists z . z \neq \text{nil} \wedge \overline{(z \widehat{\ } y = x)}_{x,y}^{\text{inf}}(\sigma)$$

Lexicographic Ordering. The lexicographic order on streams extends point-wise an order on the underlying elements. Thus, we assume a binary relation symbol \leq with the standard axiomatisation of a (non-strict) partial order.

$$\Rightarrow e \leq e \quad e \leq e', e' \leq e'' \Rightarrow e \leq e'' \quad e \leq e', e' \leq e \Rightarrow e = e'$$

³ Here we use the fact that ‘ $::$ ’ behaves as a pairing function. In other languages one might need to add a function $\langle \cdot, \cdot \rangle$, and (axiomatically) restrict the semantics to structures that interpret it as a pairing function. Note that incorporating pairs is equivalent to taking $2n$ -ary operators RTC_n and RTC_n^{op} for every $n \geq 1$.

The lexicographic ordering relation \leq_ℓ is captured as follows, where we use $e < e'$ as an abbreviation for $e \leq e' \wedge e \neq e'$.

$$\begin{aligned} \sigma \leq_\ell \sigma' & \stackrel{\text{def}}{=} (RTC_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle}^{\text{op}} \psi_\ell)(\langle \sigma, \sigma' \rangle, \langle \text{nil}, \text{nil} \rangle) \\ \text{where } \psi_\ell & \equiv \exists a, b, z_1, z_2. x_1 = a :: z_1 \wedge x_2 = b :: z_2 \wedge \\ & \text{Stream}(z_1) \wedge \text{Stream}(z_2) \wedge (a < b \vee (a = b \wedge z_1 = y_1 \wedge z_2 = y_2)) \end{aligned}$$

The semantics of the RTC^{op} operator require an infinite sequence of pairs such that, until $\langle \text{nil}, \text{nil} \rangle$ is reached, each two consecutive pairs are related by ψ_ℓ . This formula states that if the heads of the lists in the first pair are equal, the next pair of lists in the infinite sequence is their two tails, thus the lexicographic relation must also hold of them. Otherwise, if the head of the first is less than that of the second, nothing is required of the tails, i.e. they may be any streams.

Substreams. We consider one stream to be a substream of another if the latter contains every element of the former in the same order (although it may contain other elements too). Equivalently, the latter is obtained by inserting some (possibly infinite) number of finite sequences of elements in between those of the former. This description makes it clearer that defining this relation involves mixing (or, rather, nesting) induction and coinduction. We formalise the substream relation, \succ using the *inductive* extension relation \triangleleft to capture the inserted finite sequences, wrapping it within a *coinductive* definition using the RTC^{op} operator.

$$\begin{aligned} \sigma \succ \sigma' & \stackrel{\text{def}}{=} \overline{\psi_{\succ}^{\text{inf}}}_{\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle}(\langle \sigma, \sigma' \rangle) \\ \text{where } \psi_{\succ} & \equiv \exists a. x_1 \triangleleft a :: y_1 \wedge x_2 = a :: y_2 \end{aligned}$$

On examination, one can observe that this relation is transitive. However, proving this is non-trivial and, unsurprisingly, involves applying both induction and coinduction. In Sect. 5, we give a proof of the transitivity of \succ in RTcC. This relation was also considered at length in [6, §5.1.3] where it is formalised in terms of *selectors*, which form streams by picking out certain elements from other streams. The treatment in [6] requires some heavy (coalgebraic) metatheory. While our proof in Sect. 5 requires some (fairly obvious) lemmas, the basic structure of the (co)inductive reasoning required is made plain by the cycles in the proof. Furthermore, the RTcC presentation seems to enable a more intuitive understanding of the nature of the coinductive definitions and principles involved.

4 Proof Theory

We now present a non-well-founded proof system for RTcC, which extends (an equivalent of) the non-well-founded proof system considered in [17, 18] for transitive closure logic (i.e. the RTC-fragment of RTcC).

$$\frac{}{\Gamma \Rightarrow \Delta, (RTC_{x,y} \varphi)(s, s)} \quad (4)$$

$$\frac{\Gamma \Rightarrow \Delta, \varphi \{s/x, r/y\} \quad \Gamma \Rightarrow \Delta, (RTC_{x,y} \varphi)(r, t)}{\Gamma \Rightarrow \Delta, (RTC_{x,y} \varphi)(s, t)} \quad (5)$$

$$\frac{\Gamma, s = t \Rightarrow \Delta \quad \Gamma, \varphi \{s/x, z/y\}, (RTC_{x,y} \varphi)(z, t) \Rightarrow \Delta}{\Gamma, (RTC_{x,y} \varphi)(s, t) \Rightarrow \Delta} (\dagger) \quad (6)$$

$$\frac{}{\Gamma \Rightarrow \Delta, (RTC_{x,y}^{\text{op}} \varphi)(s, s)} \quad (7)$$

$$\frac{\Gamma \Rightarrow \Delta, \varphi \{s/x, r/y\} \quad \Gamma \Rightarrow \Delta, (RTC_{x,y}^{\text{op}} \varphi)(r, t)}{\Gamma \Rightarrow \Delta, (RTC_{x,y}^{\text{op}} \varphi)(s, t)} \quad (8)$$

$$\frac{\Gamma, s = t \Rightarrow \Delta \quad \Gamma, \varphi \{s/x, z/y\}, (RTC_{x,y}^{\text{op}} \varphi)(z, t) \Rightarrow \Delta}{\Gamma, (RTC_{x,y}^{\text{op}} \varphi)(s, t) \Rightarrow \Delta} (\ddagger) \quad (9)$$

where: $(\dagger) z \notin \text{fv}(\Gamma, \Delta, (RTC_{x,y} \varphi)(s, t))$; and $(\ddagger) z \notin \text{fv}(\Gamma, \Delta, (RTC_{x,y}^{\text{op}} \varphi)(s, t))$.

Fig. 1. Proof rules of RTcC_G^∞

4.1 A Non-well-Founded Proof System

In non-well-founded proof systems, e.g. [2, 5, 10–12, 23, 24, 50], proofs are allowed to be infinite, i.e. non-well-founded trees, but they are subject to the restriction that every infinite path in the proof admits some infinite progress, witnessed by tracing terms or formulas. The infinitary proof system for RTcC logic is defined as an extension of $\mathcal{LK}_=$, the sequent calculus for classical first-order logic with equality and substitution [28, 56].⁴ Sequents are expressions of the form $\Gamma \Rightarrow \Delta$, for finite sets of formulas Γ and Δ . We abbreviate Γ, Δ and Γ, φ by $\Gamma \cup \Delta$ and $\Gamma \cup \{\varphi\}$, respectively, and write $\text{fv}(\Gamma)$ for the set of free variables of the formulas in Γ . A sequent $\Gamma \Rightarrow \Delta$ is valid if and only if the formula $\bigwedge_{\varphi \in \Gamma} \varphi \rightarrow \bigvee_{\psi \in \Delta} \psi$ is.

Definition 4 (RTcC_G^∞). *The proof system RTcC_G^∞ is obtained by adding to $\mathcal{LK}_=$ the proof rules given in Fig. 1.*

Rules (6), and (8) are the unfolding rules for the two operators that represent the induction and coinduction principles in the system, respectively. The proof rules for both operators have exactly the same form, and so the reader may wonder what it is, then, that distinguishes the behaviour of the two operators. The difference proceeds from the way the decomposition of the corresponding formulas is traced in the non-well-founded proof system. For induction, RTC

⁴ Unlike in the original system, here we take $\mathcal{LK}_=$ to include the substitution rule.

formulas on the left-hand side of the sequents are traced through Rule (6); for coinduction, RTC^{op} formulas on the right-hand side of sequents are traced through Rule (8).

Definition 5 (RTcC $_G^\infty$ Pre-proofs). An RTcC $_G^\infty$ pre-proof is a rooted, possibly non-well-founded (i.e. infinite) derivation tree constructed using the RTcC $_G^\infty$ proof rules. A path in a pre-proof is a possibly infinite sequence $S_0, S_1, \dots, (S_n)$ of sequents with S_0 the root of the proof, and S_{i+1} a premise of S_i for each $i < n$.

We adopt the usual proof-theoretic notions of formula occurrence and sub-occurrence, and of ancestry between formulas [13]. A formula occurrence is called a *proper* formula if it is not a sub-occurrence of any formula.

Definition 6 ((Co-)Traces). A trace (resp. co-trace) is a possibly infinite sequence $\tau_1, \tau_2, \dots, (\tau_n)$ of proper RTC (resp. RTC^{op}) formula occurrences in the left-hand (resp. right-hand) side of sequents in a pre-proof such that τ_{i+1} is an immediate ancestor of τ_i for each $i > 0$. If the trace (resp. co-trace) contains an infinite number of formula occurrences that are principal for instances of Rule (6) (resp. Rule (8)), then we say that it is infinitely progressing.

As usual in non-well-founded proof theory, we use the notion of (co-)trace to define a *global trace condition*, distinguishing certain ‘valid’ pre-proofs.

Definition 7 (RTcC $_G^\infty$ Proofs). An RTcC $_G^\infty$ proof is a pre-proof in which every infinite path has a tail followed by an infinitely progressing (co-)trace.

In general, one cannot reason effectively about infinite proofs, as found in RTcC $_G^\infty$. In order to do so our attention has to be restricted to those proof trees which are finitely representable. That is, the *regular* infinite proof trees, containing only finitely many *distinct* subtrees. They can be specified as systems of recursive equations or, alternatively, as cyclic *graphs* [22]. One way of formalising such proof graphs is as standard proof trees containing open nodes (called buds), to each of which is assigned a syntactically equal internal node of the proof (called a companion). The restriction to cyclic proofs provides the basis for an effective system for automated inductive and coinductive reasoning. The system RTcC $_G^\infty$ can naturally be restricted to a cyclic proof system for RTcC logic as follows.

Definition 8 (Cyclic Proofs). The cyclic proof system RTcC $_G^\omega$ for RTcC logic is the subsystem of RTcC $_G^\infty$ comprising of all and only the finite and regular infinite proofs (i.e. proofs that can be represented as finite, possibly cyclic, graphs).⁵

It is decidable whether a cyclic pre-proof satisfies the global trace condition, using a construction involving an inclusion between Büchi automata [10, 54]. However since this requires complementing Büchi automata (a PSPACE procedure), RTcC $_G^\omega$ is not a proof system in the Cook-Reckhow sense [21]. Notwithstanding, checking the trace condition for cyclic proofs found in practice is not prohibitive [45, 57].

⁵ Note that in [17, 18] RTcC $_G^\omega$ denoted the *full* infinitary system for the RTC-fragment.

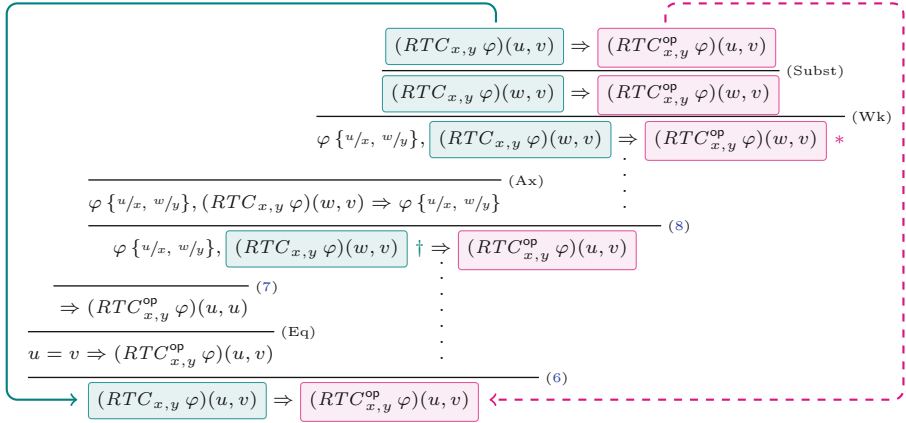


Fig. 2. Proof in RTcC_G^ω of $(RTC_{x,y} \varphi)(u, v) \Rightarrow (RTC_{x,y}^{\text{op}} \varphi)(u, v)$

Although RTcC_G^∞ is complete (cf. Theorem 2 below) RTcC_G^ω is *not*, since arithmetic can be encoded in RTcC logic and the set of RTcC_G^ω proofs is recursively enumerable.⁶ Nonetheless, RTcC_G^ω is adequate for RTcC logic in the sense that it suffices for proving the standard properties of the operators, as in, e.g., Proposition 2.

Example 1. Figure 2 demonstrates an RTcC_G^ω proof that the transitive closure is contained within the transitive co-closure. Notice that the proof has a single cycle, and thus a single infinite path. Following this path, there is both a trace (consisting of the highlighted RTC formulas, on the left-hand side of sequents) which progresses on traversing Rule (6) (marked †), and a co-trace (consisting of the highlighted RTC^{op} formulas, on the right-hand side of sequents), which progresses on traversing Rule (8) (marked *). Thus, Fig. 2 can be seen both as a proof by induction and a proof by coinduction. It exemplifies how naturally such reasoning can be captured within RTcC_G^ω .

A salient feature of non-well-founded proof systems, including this one, is that (co)induction invariants need not be mentioned explicitly, but instead are encoded in the cycles of a proof. This facilitates the automation of such reasoning, as the invariants may be interactively constructed during a proof-search process.

4.2 Soundness

To show soundness, i.e. that all derived sequents are valid, we establish that the infinitely progressing (co-)traces in proofs preclude the existence of counter-models. By local soundness of the proof rules, any given counter-model for a sequent derived by a proof identifies an infinite path in the proof consisting of

⁶ The RTC -fragment of RTcC_G^ω was shown complete for a Henkin-style semantics [17].

invalid sequents. However, the presence of a (co-)trace along this path entails a contradiction (and so conclude that no counter-models exist). From a trace, one may infer the existence of an infinitely descending chain of natural numbers. This relies on a notion of (well-founded) measure for RTC formulas, *viz.* the measure of $\phi \equiv (RTC_{x,y} \varphi)(s, t)$ with respect to a given model M and valuation ν —denoted by $\delta_\phi(M, \nu)$ —is defined to be the minimum number of φ -steps needed to connect $\nu(s)$ and $\nu(t)$ in M . Conversely, from a co-trace beginning with a formula $(RTC_{x,y}^{op} \varphi)(s, t)$ one can construct an infinite sequence of φ -steps beginning at s , *i.e.* a witness that the counter-model does in fact satisfy $(RTC_{x,y}^{op} \varphi)(s, t)$.

The key property needed for soundness of the proof system is the following strong form of local soundness for the proof rules.

Proposition 3 (Trace Local Soundness). *Let M be a model and ν a valuation that invalidate the conclusion of an instance of an $RTcC_G^\infty$ inference rule; then there exists a valuation ν' that invalidates some premise of the inference rule such that the following hold.*

1. *If (τ, τ') is a trace following the path from the conclusion to the invalid premise, then $\delta_{\tau'}(M, \nu') \leq \delta_\tau(M, \nu)$; moreover $\delta_{\tau'}(M, \nu') < \delta_\tau(M, \nu)$ if the rule is an instance of (6) and τ is the principal formula.*
2. *If (τ, τ') is a co-trace following the path from the conclusion to the invalid premise, with $\tau \equiv (RTC_{x,y}^{op} \varphi)(s, t)$ and $\tau' \equiv (RTC_{x,y}^{op} \varphi')(r, t')$, then: (a) $M, \nu[x := d, y := d'] \models \varphi$ if and only if $M, \nu'[x := d, y := d'] \models \varphi'$, for all elements d and d' in M ; and (b) $M, \nu' \models \varphi \{s/x, \tau/y\}$ if τ is the principal formula of an instance of (8), and $\nu(s) = \nu'(r)$ otherwise.*

The global soundness of the proof system then follows.

Theorem 1 (Soundness of $RTcC_G^\infty$). *Sequents derivable in $RTcC_G^\infty$ are valid.*

Proof. Take a proof deriving $\Gamma \Rightarrow \Delta$. Suppose, for contradiction, that there is a model M and valuation ν_1 invalidating $\Gamma \Rightarrow \Delta$. Then by Proposition 3 there exists an infinite path of sequents $(S_i)_{i>0}$ in the proof and an infinite sequence of valuations $(\nu_i)_{i>0}$ such that M and ν_i invalidate S_i for each $i > 0$. Since the proof must satisfy the global trace condition, this infinite path has a tail $(S_i)_{i>k}$ followed by an infinitely progressing (co-)trace $(\tau_i)_{i>0}$.

- If $(\tau_i)_{i>0}$ is a trace, Proposition 3 implies an infinitely descending chain of natural numbers: $\delta_{\tau_1}(M_{k+1}, \nu_{k+1}) \leq \delta_{\tau_2}(M_{k+2}, \nu_{k+2}) \leq \dots$
- If $(\tau_i)_{i>0}$ is a co-trace, with $\tau_1 \equiv (RTC_{x,y}^{op} \varphi)(s, t)$ and $M, \nu_{k+1} \not\models \tau_1$, then Proposition 3 entails that there is an infinite sequence of terms t_0, t_1, t_2, \dots with $s \equiv t_0$ such that $M, \nu_{k+1}[x := \nu_{k+1}(t_j), y := \nu_{k+1}(t_{j+1})] \models \varphi$ for each $j \geq 0$. That is, it follows from Definition 2 that $M, \nu_{k+1} \models (RTC_{x,y}^{op} \varphi)(s, t)$.

In both cases we have a contradiction, so conclude that $\Gamma \Rightarrow \Delta$ is valid. □

Since every $RTcC_G^\omega$ proof is also an $RTcC_G^\infty$ proof, soundness of $RTcC_G^\omega$ is an immediate corollary.

Corollary 1. *A sequent $\Gamma \Rightarrow \Delta$ is valid if there is an $RTcC_G^\omega$ proof deriving it.*

4.3 Completeness

The completeness proof for RTcC_G^∞ is obtained by extending the completeness proof of the *RTC*-fragment of RTcC_G^∞ found in [17, 18], which, in turn, follows a standard technique used in e.g. [12]. We next outline the core of the proof, full details can be found in the appendix.

Roughly speaking, for a given sequent $\Gamma \Rightarrow \Delta$ one constructs a ‘search tree’ which corresponds to an exhaustive search strategy for a cut-free proof for the sequent. Search trees are, by construction, recursive and cut-free. In case the search tree is not an RTcC_G^∞ proof (and there are no open nodes) it must contain some *untraceable* infinite branch, i.e. one that does not satisfy the global trace condition. We then collect the formulas occurring along such an untraceable branch to construct a (possibly infinite) ‘sequent’, $\Gamma_\omega \Rightarrow \Delta_\omega$ (called a *limit sequent*), and construct the Herbrand model M_ω of open terms quotiented by the equalities it contains. That is, taking \sim to be the smallest congruence on terms such that $s \sim t$ whenever $s = t \in \Gamma_\omega$, the elements of M_ω are \sim -equivalence classes and every k -ary relation symbol q is interpreted as $\{([t_1], \dots, [t_k]) \mid q(t_1, \dots, t_k) \in \Gamma_\omega\}$ (here $[t]$ denotes the \sim -equivalence class containing t). This model, together with the valuation ν_ω defined by $\nu_\omega(x) = [x]$ for all variables x , can be shown to invalidate the sequent $\Gamma \Rightarrow \Delta$. The completeness result therefore follows.

Theorem 2 (Completeness). *All valid sequents are derivable in RTcC_G^∞ .*

Proof. Given any sequent S , if some search tree for S is not an RTcC_G^∞ proof then it has an untraceable branch, and the model M_ω and valuation ν_ω constructed from the corresponding limit sequent invalidate S . Thus if S is valid, then the search tree is a recursive RTcC_G^∞ proof deriving S . \square

We obtain admissibility of cut for the full infinitary system as the search tree, by construction, is cut-free. Since the construction of the search tree does not necessarily produce RTcC_G^ω pre-proofs, we do not obtain a regular completeness result using this technique.

Corollary 2 (Cut admissibility). *Cut is admissible in RTcC_G^∞ .*

5 Proving Properties of Streams

We now demonstrate how (co)inductive reasoning about streams and their properties is formalised in the cyclic fragment of the proof system presented above. For the sake of clarity, in the derivations below we elide detailed applications of the proof rules (including the axioms for list constructors), instead indicating the principal rules involved at each step. We also elide (using ‘...’) formulas in sequents that are not relevant to the local reasoning at that point.

Transitivity of Lexicographic Ordering. Fig. 3 outlines the main structure of an RTcC_G^ω proof deriving the sequent $x \leq_\ell y, y \leq_\ell z \Rightarrow x \leq_\ell z$, where $x, y,$

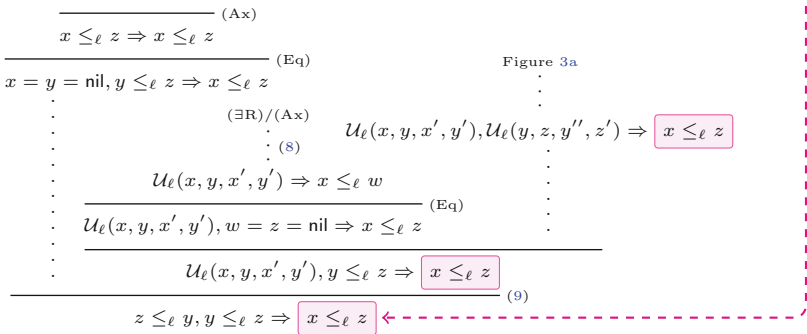
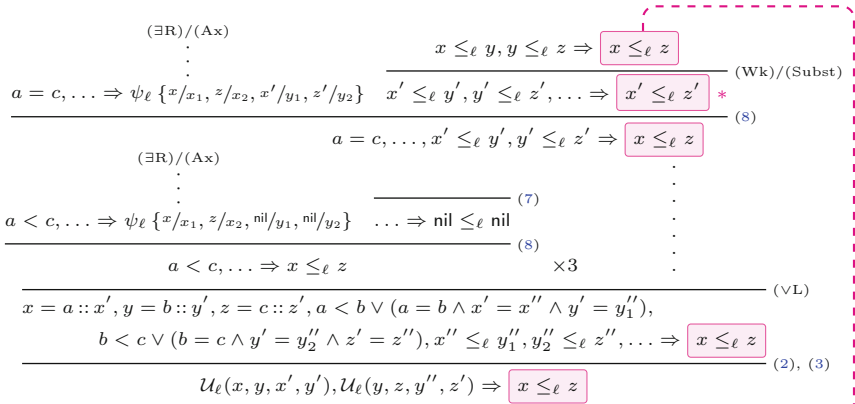


Fig. 3. High-level structure of an $RTcC_G^\omega$ proof of transitivity of \leq_ℓ .

and z are distinct variables. All other variables in Fig. 3 are freshly introduced. $\mathcal{U}_\ell(\sigma_1, \sigma_2, \sigma'_1, \sigma'_2)$ abbreviates the set $\{\psi_\ell \{ \sigma_1/x_1, \sigma_2/x_2, \sigma'_1/y_1, \sigma'_2/y_2 \}, \sigma'_1 \leq_\ell \sigma'_2\}$ (i.e. the result of unfolding the step case of the formula $\sigma_1 \leq_\ell \sigma_2$ using σ'_1 and σ'_2 as the intermediate terms).

The proof begins by unfolding the definitions of $x \leq_\ell y$ and $y \leq_\ell z$, shown in Fig. 3b. The interesting part is the sub-proof shown in Fig. 3a, when each of the lists is not nil. Here, we perform case splits on the relationship between the head elements a, b , and c . For the case $a = c$, i.e. the heads are equal, when unfolding the formula $x \leq_\ell z$ on the right-hand side, we instantiate the second components of the RTc^{op} formula to be the tails of the streams, x' and z' . In the left-hand premise we must show $\psi_\ell \{x/x_1, z/x_2, x'/y_1, z'/y_2\}$, which can be done by matching with formulas already present in the sequent. The right-hand premise must derive $x' \leq_\ell z'$, i.e. the tails are lexicographically related. This is where we apply the coinduction principle, by renaming the variables and forming a cycle in

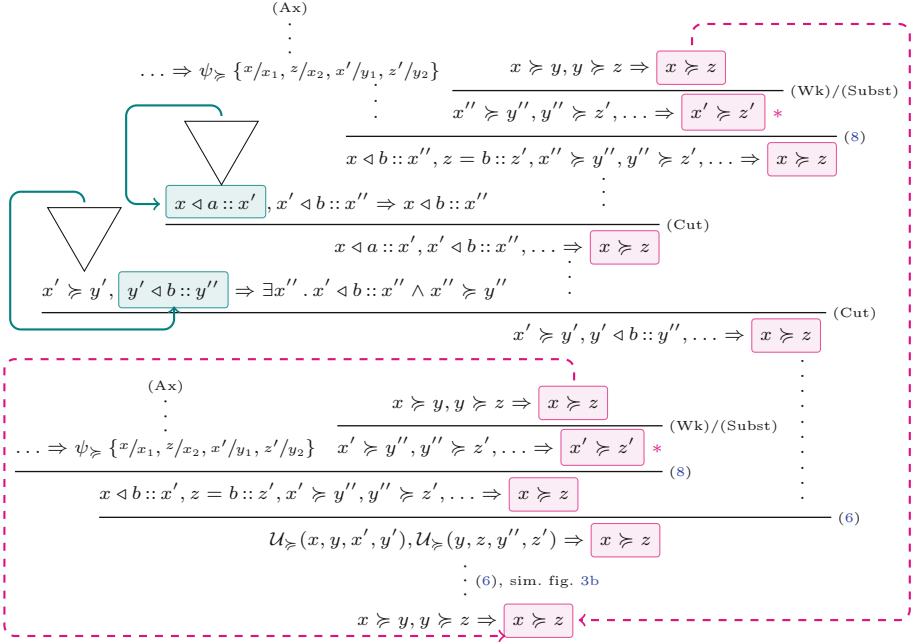


Fig. 4. High-level structure of an RTcC_G^ω proof of transitivity of \succcurlyeq .

the proof back to the root. This does indeed produce a proof, since we can form a co-trace by following the formulas $x \leq_l z, \dots, x' \leq_l z'$ on the right-hand side of sequents along this cycle. This co-trace progresses as it traverses the instance of Rule (8) each time around the cycle (marked *).

Transitivity of the Substream Relation. Fig. 4 outlines the structure of an RTcC_G^ω proof of the sequent $x \succcurlyeq y, y \succcurlyeq z \Rightarrow x \succcurlyeq z$, for distinct variables x, y , and z . As above, other variables are freshly introduced, and we use $\mathcal{U}_{\succcurlyeq}(\sigma_1, \sigma_2, \sigma'_1, \sigma'_2)$ to denote the set $\{\psi_{\succcurlyeq} \{\sigma_1/x_1, \sigma_2/x_2, \sigma'_1/y_1, \sigma'_2/y_2\}, \sigma'_1 \succcurlyeq \sigma'_2\}$ (i.e. the result of unfolding the step-case of the formula $\sigma_1 \succcurlyeq \sigma_2$ using σ'_1 and σ'_2 as the intermediate terms).

The reflexive cases are handled similarly to the previous example. Again, the work is in proving the step cases. After unfolding both $x \succcurlyeq y$ and $y \succcurlyeq z$, we obtain $x' \succcurlyeq y'$ and $y'' \succcurlyeq z'$, as part of $\mathcal{U}_{\succcurlyeq}(x, y, x', y')$ and $\mathcal{U}_{\succcurlyeq}(y, z, y'', z')$, respectively. We also have (for fresh variables a and b) that: (i) $x \triangleleft a :: x'$; (ii) $y = a :: y'$ (y' is the immediate tail of y); (iii) $y \triangleleft b :: y''$ (y'' is some tail of y); and (iv) $z = b :: z'$ (z' is the immediate tail of z). Ultimately, we are looking to obtain $x \triangleleft b :: x''$ and $x'' \succcurlyeq y''$ (for some tail x''), so that we can unfold the formula $x \succcurlyeq z$ on the right-hand side to obtain $x'' \succcurlyeq z'$ and thus be able to form a (coinductive) cycle.

The application of Rule (6) shown in Fig. 4 performs a case-split on the formula $y \triangleleft b :: y''$. The left-hand branch handles the case that y'' is, in fact, the immediate tail of y ; thus $y' = y''$ and $a = b$, and so we can substitute b and y'' in place of a and y' , respectively, and take x'' to be x' . In the right-hand branch, corresponding to the case that y'' is not the immediate tail of y , we obtain $y' \triangleleft b :: y''$ from the case-split. Then we apply two lemmas; namely: (i) if $x' \succ y'$ and $y' \triangleleft b :: y''$, then there is some x'' such that $x' \triangleleft b :: x''$ and $x'' \succ y''$; and (ii) if $x \triangleleft a :: x'$ and $x' \triangleleft b :: x''$, then $x \triangleleft b :: x''$ (a form of transitivity for the extends relation). For space reasons we do not show the structure of the sub-proofs deriving these, however, as marked in the figure, we note that they are both carried out by *induction* on the \triangleleft relation.

In summary the proof contains two (inductive) sub-proofs, each validated by infinitely progressing inductive traces, and also two overlapping outer cycles. Infinite paths following these outer cycles have co-traces consisting of the highlighted formulas in Fig. 4, which progress infinitely often as they traverse the instances of Rule (8) (marked *).

6 Conclusion and Future Work

This paper presented a new framework that extends the well-known, powerful transitive closure logic with a dual transitive co-closure operator. An infinitary proof system for the logic was developed and shown to be sound and complete. Its cyclic subsystem was shown to be powerful enough for reasoning over streams, and in particular automating combinations of inductive and coinductive arguments.

Much remains to be done to fully develop the new logic and its proof theory, and to study its implications. Although we have shown that our framework captures many interesting properties of the canonical coinductive data type, streams, a primary task for future research is to formally characterise its ability to capture finitary coinductive definitions in general. In particular, it seems plausible that RTcC is a good candidate setting in which to look for characterisations that complement and bridge existing results for coinductive data in automata theory and coalgebra. That is, it may potentially mirror (and also perhaps even replace) the role that monadic second order logic plays for (ω) -regular languages.

Another important research task is to further develop the structural proof theory of the systems RTcC_G^∞ and RTcC_G^ω in order to describe the natural process and dynamics of inductive and coinductive reasoning. This includes properties such as cut elimination, admissibility of rules, regular forms for proofs, focussing, and proof search strategies. For example, syntactic cut elimination for non-well-founded systems has been studied extensively in the context of linear logic [5, 26]. The basic approach would seem to work for RTcC, however, one expects that cut-elimination will *not* preserve regularity.

Through the proofs-as-programs paradigm (a.k.a. the Curry-Howard correspondence) our proof-theoretic synthesis of induction and coinduction has a number of applications that invite further investigation. Namely, our framework provides a general setting for verifying program correctness against specifications

of coinductive (safety) and inductive (liveness) properties. Implementing proof-search procedures can lead to automation, as well as correct-by-construction synthesis of programs operating on (co)inductive data. Finally, grounding proof assistants in our framework will provide a robust, proof-theoretic basis for mechanistic coinductive reasoning.

Acknowledgements. We are grateful to Alexandra Silva for valuable coinductive reasoning examples, and Juriaan Rot for helpful comments and pointers. We also extend thanks to the anonymous reviewers for their questions and comments.

References

1. Abel, A., Pientka, B.: Well-founded recursion with copatterns and sized types. *J. Funct. Program.* **26**, e2 (2016). <https://doi.org/10.1017/S0956796816000022>
2. Afshari, B., Leigh, G.E.: Cut-free completeness for modal mu-calculus. In: Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017), Reykjavik, Iceland, 20–23 June 2017, pp. 1–12 (2017). <https://doi.org/10.1109/LICS.2017.8005088>
3. Avron, A.: Transitive closure and the mechanization of mathematics. In: Kamareddine, F.D. (ed.) *Thirty Five Years of Automating Mathematics*, Applied Logic Series. APLS 2013, vol. 28, pp. 149–171. Springer, Netherlands (2003). https://doi.org/10.1007/978-94-017-0253-9_7
4. Baelde, D.: Least and greatest fixed points in linear logic. *ACM Trans. Comput. Log.* **13**(1), 2:1–2:44 (2012). <https://doi.org/10.1145/2071368.2071370>
5. Baelde, D., Doumane, A., Saurin, A.: Infinitary proof theory: the multiplicative additive case. In: Proceedings of the 25th EACSL Annual Conference on Computer Science Logic (CSL 2016), 29 August–1 September 2016, Marseille, France, pp. 42:1–42:17 (2016). <https://doi.org/10.4230/LIPIcs.CSL.2016.42>
6. Basold, H.: *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Ph.D. thesis, Radboud University (2018). <https://hdl.handle.net/2066/190323>
7. Bertot, Y., Casteran, P.: *Interactive Theorem Proving and Program Development*. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07964-5>
8. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_6
9. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. *J. Funct. Program.* **23**, 552–593 (2013). <https://doi.org/10.1017/S095679681300018X>
10. Brotherston, J.: Formalised inductive reasoning in the logic of bunched implications. In: Nielson, H.R., Filé, G. (eds.) *SAS 2007*. LNCS, vol. 4634, pp. 87–103. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_6
11. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), pp. 101–112 (2008). <https://doi.org/10.1145/1328438.1328453>
12. Brotherston, J., Simpson, A.: Sequent calculi for induction and infinite descent. *J. Log. Comput.* **21**(6), 1177–1216 (2010). <https://doi.org/10.1093/logcom/exq052>

13. Buss, S.R.: Handbook of proof theory. In: Studies in Logic and the Foundations of Mathematics. Elsevier Science (1998)
14. Cohen, L.: Completeness for ancestral logic via a computationally-meaningful semantics. In: Schmidt, R.A., Nalon, C. (eds.) TABLEAUX 2017. LNCS (LNAI), vol. 10501, pp. 247–260. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66902-1_15
15. Cohen, L., Avron, A.: Ancestral logic: a proof theoretical study. In: Kohlenbach, U., Barceló, P., de Queiroz, R. (eds.) WoLLIC 2014. LNCS, vol. 8652, pp. 137–151. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44145-9_10
16. Cohen, L., Avron, A.: The middle ground-ancestral logic. *Synthese*, 1–23 (2015). <https://doi.org/10.1007/s11229-015-0784-3>
17. Cohen, L., Rowe, R.N.S.: Uniform inductive reasoning in transitive closure logic via infinite descent. In: Proceedings of the 27th EACSL Annual Conference on Computer Science Logic (CSL 2018), 4–7 September 2018, Birmingham, UK, pp. 16:1–16:17 (2018). <https://doi.org/10.4230/LIPIcs.CSL.2018.16>
18. Cohen, L., Rowe, R.N.S.: Non-well-founded proof theory of transitive closure logic. *Trans. Comput. Logic* (2020, to appear). <https://arxiv.org/pdf/1802.00756.pdf>
19. Cohen, L., Rowe, R.N.S., Zohar, Y.: Towards automated reasoning in Herbrand structures. *J. Log. Comput.* **29**(5), 693–721 (2019). <https://doi.org/10.1093/logcom/exz011>
20. Constable, R.L., et al.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall Inc, Upper Saddle River (1986)
21. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. *J. Symbolic Log.* **44**(1), 36–50 (1979). <https://doi.org/10.2307/2273702>
22. Courcelle, B.: Fundamental properties of infinite trees. *Theor. Comput. Sci.* **25**, 95–169 (1983). [https://doi.org/10.1016/0304-3975\(83\)90059-2](https://doi.org/10.1016/0304-3975(83)90059-2)
23. Das, A., Pous, D.: Non-Wellfounded Proof Theory for (Kleene+Action)(Algebras+Lattices). In: Proceedings of the 27th EACSL Annual Conference on Computer Science Logic (CSL 2018), pp. 19:1–19:18 (2018). <https://doi.org/10.4230/LIPIcs.CSL.2018.19>
24. Doumane, A.: Constructive completeness for the linear-time μ -calculus. In: Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017), pp. 1–12 (2017). <https://doi.org/10.1109/LICS.2017.8005075>
25. Endrullis, J., Hansen, H., Hendriks, D., Polonsky, A., Silva, A.: A coinductive framework for infinitary rewriting and equational reasoning. In: 26th International Conference on Rewriting Techniques and Applications (RTA 2015), vol. 36, pp. 143–159 (2015). <https://doi.org/10.4230/LIPIcs.RTA.2015.143>
26. Fortier, J., Santocanale, L.: Cuts for circular proofs: semantics and cut-elimination. In: Rocca, S.R.D. (ed.) Computer Science Logic 2013 (CSL 2013). Leibniz International Proceedings in Informatics (LIPIcs), vol. 23, pp. 248–262. Dagstuhl, Germany (2013). <https://doi.org/10.4230/LIPIcs.CSL.2013.248>
27. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed. *J. Funct. Program.* **12**(6), 511–548 (2002). <https://doi.org/10.1017/S0956796802004318>
28. Gentzen, G.: Untersuchungen über das Logische Schließen. I. *Mathematische Zeitschrift* **39**(1), 176–210 (1935). <https://doi.org/10.1007/BF01201353>
29. Hansen, H.H., Kupke, C., Rutten, J.: Stream differential equations: specification formats and solution methods. In: Logical Methods in Computer Science, vol. 13(1), February 2017. [https://doi.org/10.23638/LMCS-13\(1:3\)2017](https://doi.org/10.23638/LMCS-13(1:3)2017)
30. Heath, Q., Miller, D.: A proof theory for model checking. *J. Autom. Reasoning* **63**(4), 857–885 (2019). <https://doi.org/10.1007/s10817-018-9475-3>

31. Immerman, N.: Languages that capture complexity classes. *SIAM J. Comput.* **16**(4), 760–778 (1987). <https://doi.org/10.1137/0216051>
32. Jacobs, B., Rutten, J.: A tutorial on (co) algebras and (co) induction. *Bull. Eur. Assoc. Theor. Comput. Sci.* **62**, 222–259 (1997)
33. Jeannin, J.B., Kozen, D., Silva, A.: CoCaml: functional programming with regular coinductive types. *Fundamenta Informaticae* **150**, 347–377 (2017). <https://doi.org/10.3233/FI-2017-1473>
34. Komendantskaya, E., Power, J.: Coalgebraic semantics for derivations in logic programming. In: Corradini, A., Klin, B., Cirstea, C. (eds.) *CALCO 2011*. LNCS, vol. 6859, pp. 268–282. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_19
35. Kozen, D., Silva, A.: Practical coinduction. *Math. Struct. Comput. Sci.* **27**(7), 1132–1152 (2017). <https://doi.org/10.1017/S0960129515000493>
36. Leino, R., Moskal, M.: Co-induction simply: automatic co-inductive proofs in a program verifier. Technical report MSR-TR-2013-49, Microsoft Research, July 2013. <https://www.microsoft.com/en-us/research/publication/co-induction-simply-automatic-co-inductive-proofs-in-a-program-verifier/>
37. Leroy, X., Grall, H.: Coinductive big-step operational semantics. *Inf. Comput.* **207**(2), 284–304 (2009). <https://doi.org/10.1016/j.ic.2007.12.004>
38. Lucanu, D., Roşu, G.: CIRC: a circular coinductive prover. In: Mossakowski, T., Montanari, U., Haverlaen, M. (eds.) *CALCO 2007*. LNCS, vol. 4624, pp. 372–378. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73859-6_25
39. Martin, R.M.: A homogeneous system for formal logic. *J. Symbolic Log.* **8**(1), 1–23 (1943). <https://doi.org/10.2307/2267976>
40. Martin-Löf, P.: Hauptsatz for the intuitionistic theory of iterated inductive definitions. In: Fenstad, J.E. (ed.) *Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic and the Foundations of Mathematics*, vol. 63, pp. 179–216. Elsevier (1971). [https://doi.org/10.1016/S0049-237X\(08\)70847-4](https://doi.org/10.1016/S0049-237X(08)70847-4)
41. McBride, C.: Let’s see how things unfold: reconciling the infinite with the intensional (extended abstract). In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) *CALCO 2009*. LNCS, vol. 5728, pp. 113–126. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03741-2_9
42. McDowell, R., Miller, D.: Cut-elimination for a logic with definitions and induction. *Theor. Comput. Sci.* **232**(1–2), 91–119 (2000). [https://doi.org/10.1016/S0304-3975\(99\)00171-1](https://doi.org/10.1016/S0304-3975(99)00171-1)
43. Park, D.M.R.: Finiteness is mu-ineffable. *Theor. Comput. Sci.* **3**(2), 173–181 (1976). [https://doi.org/10.1016/0304-3975\(76\)90022-0](https://doi.org/10.1016/0304-3975(76)90022-0)
44. Roşu, G., Lucanu, D.: Circular coinduction: a proof theoretical foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) *CALCO 2009*. LNCS, vol. 5728, pp. 127–144. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03741-2_10
45. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, Paris, France, 16–17 January 2017, pp. 53–65 (2017). <https://doi.org/10.1145/3018610.3018623>
46. Rutten, J.: Universal coalgebra: a theory of systems. *Theor. Comput. Sci.* **249**(1), 3–80 (2000)
47. Rutten, J.: On Streams and Coinduction (2002). <https://homepages.cwi.nl/~janr/papers/files-of-papers/CRM.pdf>
48. Rutten, J.: *The Method of Coalgebra: Exercises in Coinduction*. CWI, Amsterdam (2019)

49. Sangiorgi, D., Rutten, J.: *Advanced Topics in Bisimulation and Coinduction*, 1st edn. Cambridge University Press, Cambridge (2011)
50. Santocanale, L.: A calculus of circular proofs and its categorical semantics. In: Nielsen, M., Engberg, U. (eds.) *FoSSaCS 2002*. LNCS, vol. 2303, pp. 357–371. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_25
51. Shapiro, S.: *Foundations Without Foundationalism : A Case for Second-order Logic*. Clarendon Press, Oxford Logic Guides (1991)
52. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) *ICALP 2007*. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_42
53. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczyński, M. (eds.) *ICLP 2006*. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006). https://doi.org/10.1007/11799573_25
54. Simpson, A.: Cyclic arithmetic is equivalent to peano arithmetic. In: Esparza, J., Murawski, A.S. (eds.) *FoSSaCS 2017*. LNCS, vol. 10203, pp. 283–300. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_17
55. Sprenger, C., Dam, M.: On the structure of inductive reasoning: circular and tree-shaped proofs in the μ calculus. In: Gordon, A.D. (ed.) *FoSSaCS 2003*. LNCS, vol. 2620, pp. 425–440. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36576-1_27
56. Takeuti, G.: *Proof Theory*. Dover Books on Mathematics. Dover Publications, Incorporated, New York (2013)
57. Tellez, G., Brotherston, J.: Automatically verifying temporal properties of pointer programs with cyclic proof. In: de Moura, L. (ed.) *CADE 2017*. LNCS (LNAI), vol. 10395, pp. 491–508. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_30
58. Tiu, A.: *A Logical Framework For Reasoning About Logical Specifications*. Ph.D. thesis, Penn. State University (2004)
59. Tiu, A., Momigliano, A.: Cut elimination for a logic with induction and co-induction. *J. Appl. Log.* **10**(4), 330–367 (2012). <https://doi.org/10.1016/j.jal.2012.07.007>