TLP **17** (5–6): 906–923, 2017. © Cambridge University Press 2017
doi:10.1017/S147106841700028X First published online 22 August 2017

*Productive corecursion in logic programming**

EKATERINA KOMENDANTSKAYA and YUE LI
Heriot-Watt University, Edinburgh, Scotland, UK
(*e-mails:* ek19@hw.ac.uk, yl55@hw.ac.uk)

submitted 12 July 2017; revised 20 July 2017; accepted 21 July 2017

Abstract

Logic Programming is a Turing complete language. As a consequence, designing algorithms that decide termination and non-termination of programs or decide inductive/coinductive soundness of formulae is a challenging task. For example, the existing state-of-the-art algorithms can only semi-decide coinductive soundness of queries in logic programming for regular formulae. Another, less famous, but equally fundamental and important undecidable property is productivity. If a derivation is infinite and coinductively sound, we may ask whether the computed answer it determines actually computes an infinite formula. If it does, the infinite computation is productive. This intuition was first expressed under the name of computations at infinity in the 80s. In modern days of the Internet and stream processing, its importance lies in connection to infinite data structure processing. Recently, an algorithm was presented that semi-decides a weaker property – of productivity of logic programs. A logic program is productive if it can give rise to productive derivations. In this paper, we strengthen these recent results. We propose a method that semi-decides productivity of individual derivations for regular formulae. Thus, we at last give an algorithmic counterpart to the notion of productivity of derivations in logic programming. This is the first algorithmic solution to the problem since it was raised more than 30 years ago. We also present an implementation of this algorithm.

KEYWORDS: Horn clauses, (co)recursion, (co)induction, infinite term trees, productivity.

1 Motivation

The traditional (inductive) approach to Logic Programming (LP) is based on least fixed point semantics of logic programs, and defines, for every logic program P, the *least Herbrand model* for P, i.e., the set of all (finite) ground terms *inductively entailed* by P.

Example 1.1 (*Natural numbers*)
The program below defines the set of natural numbers:

(0) `nat(0)` ←
(1) `nat(s(X))` ← `nat(X)`

The least Herbrand model comprises the terms `nat(0)`, `nat(s(0))`, `nat(s(s(0)))`, ...

* This work has been partially supported by EPSRC grant EP/K031864/1-2.

The clauses of the above program can be viewed as inference rules $\frac{}{\texttt{nat(0)}}$ and $\frac{\texttt{nat(X)}}{\texttt{nat(s(X))}}$, and the least Herbrand model can be seen as the set obtained by the forward closure of these rules. Some approaches to LP are based on this inductive view (Heintze and Jaffar 1992) of programs.

In addition to viewing logic programs inductively, we can also view them coinductively. The *greatest complete Herbrand model* for a program P takes the backward closure of the rules derived from P's clauses, thereby producing the largest set of finite and infinite ground terms *coinductively entailed* by P. For example, the greatest complete Herbrand model for the above program is the set containing all of the finite terms in its least Herbrand model, together with the term nat(s(s(...))) representing the first limit ordinal.

As it turns out, some logic programs have no natural inductive semantics and should instead be interpreted coinductively:

Example 1.2 (*Streams of natural numbers*)
The next program comprises the clauses that define the natural numbers and the following additional one that defines streams of natural numbers:

(2) $\texttt{nats(scons(X, Y))} \leftarrow \texttt{nat(X), nats(Y)}$

No terms defined by nats are contained in the least Herbrand model for this program, but its greatest complete Herbrand model contains infinite terms representing infinite streams of natural numbers, like e.g. the infinite term $t = \texttt{nats(scons(0, scons(0, \ldots)}$.

Coinductive programs operating on infinite data structures are useful for reasoning about concurrent and non-terminating processes. For example, the program below shows a part of a description of concurrent behaviour of Internet servers (Davison 2002):

(0) $\texttt{resource([get(X)|In], [X|L])} \leftarrow \texttt{resource(In, L)}$
(1) $\texttt{resource([get(X)|In], [])} \leftarrow \texttt{signal(novalue(get(X)))}$

The two clauses above describe how a server receives and processes the input streams of data. The usual list syntax [_|_] of Prolog is used to denote the binary stream constructor scons. The first clause describes the normal behaviour of the server that reads the input data, and the second allows to raise an exception (by signalling that no value was received).

SLD-resolution (Lloyd 1988) is an algorithm that allows to semi-decide whether a given formula is in the program's least Herbrand model. In practice, SLD-resolution requires a logic program's derivations to be terminating in order for them to be inductively sound. We might dually expect a logic program's non-terminating derivations to compute terms in its greatest complete Herbrand model. However, non-termination does not play a role for coinduction dual to that played by termination for induction. In particular, the fact that a logic program admits non-terminating SLD-derivations does not, on its own, guarantee that the program's computations completely capture its greatest complete Herbrand model:

Example 1.3 (*Non-productive program*)
The following "bad" program gives rise to an infinite SLD-derivation:
(0) `bad(f(X))` ← `bad(f(X))`
Although this program does not compute any infinite terms, the infinite term `bad(f(f(...)))` is in its greatest complete Herbrand model.

The problem here actually lies in the fact that the "bad" program fails to satisfy the important property of productivity. The productivity requirement on corecursive programs should reflect the fact that an infinite computation can only be consistent with its intended coinductive semantics if it is *globally productive*, i.e., if it actually produces an infinite object in the limit. This intuition lies behind the concept of *computations at infinity* introduced in the 1980s (van Emden and Abdallah 1985; Lloyd 1988). The operational semantics of a potentially non-terminating logic program P was then taken to be the set of all infinite ground terms computable by P at infinity. For example, the infinite ground term t in Example 1.2 is computable at infinity starting with the query `nats(X)`. In modern terms, we would say that computations at infinity are *(globally) productive computations.*

However, the notion of computations at infinity does not by itself give rise to algorithms for semi-deciding coinductive entailment. Thirty years after the initial investigations into coinductive computations, coinductive LP, implemented as CoLP, was introduced (Gupta *et al.* 2007; Simon *et al.* 2007). CoLP provides practical methods for terminating infinite SLD-derivations. CoLP's coinductive proof search is based on a loop detection mechanism and unification without occurs check. CoLP observes finite fragments of SLD-derivations, checks them for unifying subgoals and terminates when loops determined by such subgoals are found.

Example 1.4 (*Productive computation by SLD-resolution*)
The query `nats(X)` to the program of Example 1.2 gives rise to an SLD-derivation with a sequence of subgoals $\texttt{nats(X)} \leadsto^{X \mapsto scons(0,Y')} \texttt{nats(Y}') \leadsto \ldots$. Observing that `nats(scons(0,Y'))` and `nats(Y')` unify (note the absence of occurs check) and thus comprise a loop, CoLP concludes that `nats(X)` has been proved and returns the answer $\texttt{X} = \texttt{scons(0,X)}$ in the form of a "circular" term indicating that this program logically entails the term t in Example 1.2.

CoLP is sound, but incomplete, relative to greatest complete Herbrand models (Gupta *et al.* 2007; Simon *et al.* 2007). But, perhaps surprisingly, it is *neither* sound *nor* complete relative to computations at infinity. CoLP is not sound because our "bad" program from Example 1.3 computes no infinite terms at infinity for the query `bad(X)`, whereas CoLP notices a loop and reports success. CoLP is not complete because not all terms computable at infinity by all programs can be inferred by CoLP. In fact, CoLP's loop detection mechanism can only terminate if the term computable at infinity is a *regular* term (Courcelle 1983; Jaffar and Stuckey 1986). Regular terms are terms that can be represented as trees that have a finite number of distinct subtrees, and can therefore be expressed in a closed finite form computed by circular unification. The "circular" term $\texttt{X} = \texttt{scons(0,X)}$ in Example 1.4 is so expressed. For irregular terms (e.g. expressing a stream of Fibonacci numbers, cf. Example 6.1), CoLP simply does not terminate.

The upshot is that the loop detection method of CoLP cannot faithfully capture the operational meaning of computations at infinity. In this paper, we propose a solution to this problem, by combining loop detection and productivity within one framework.

2 Results of this paper by means of an example

We return to our "Server" example, but this time take only the clause that describes its normal execution (without exceptions):

$\texttt{resource}([\texttt{get}(\texttt{X})|\texttt{In}], [\texttt{X}|\texttt{L}]) \leftarrow \texttt{resource}(\texttt{In}, \texttt{L})$ (*)

The second argument of `resource` is the input stream received by the server, and its first argument is the stream of successfully received and read data. We can take e.g. the query

$\texttt{resource}(\texttt{X}, \texttt{Y}), \texttt{zeros}(\texttt{Y})$, asking the server to accept as input the stream of zeros defined as $\texttt{zeros}([0|\texttt{X}]) \leftarrow \texttt{zeros}(\texttt{X})$.

Assuming a fair selection of subgoals in a derivation, we will have the following SLD-derivation

$\underline{\texttt{resource}(\texttt{X}, \texttt{Y})}, \underline{\texttt{zeros}(\texttt{Y})} \rightsquigarrow^{X \mapsto [get(X')|In], Y \mapsto [X'|L]} \rightsquigarrow \underline{\texttt{resource}(\texttt{In}, \texttt{L})}, \texttt{zeros}(\texttt{X}'|\texttt{L}) \rightsquigarrow^{X' \mapsto 0} \rightsquigarrow$
$\underline{\texttt{resource}(\texttt{In}, \texttt{L})}, \underline{\underline{\texttt{zeros}(\texttt{L})}} \rightsquigarrow \ldots$

and the substitution $X \mapsto [\texttt{get}(0)|\texttt{get}(0)|\ldots]$ will be computed at infinity. This regular computation will be processed successfully by the loop detection method of CoLP (relying on a unification algorithm without occurs check). We underlined the loops above.

There are following three cases where CoLP fails to capture the notion of productive computations:

Case 1. The coinductive definition does not contain constructors of the infinite data structure. Imagine we have the clause $\texttt{resource}(\texttt{In}, \texttt{L}) \leftarrow \texttt{resource}(\texttt{In}, \texttt{L})$ instead of (*). This new clause simply asserts a tautology: a server receives the data when it receives the data. Querying again $\texttt{resource}(\texttt{X}, \texttt{Y}), \texttt{zeros}(\texttt{Y})$, we will get an infinite looping SLD-derivation, that however will not compute an infinite ground term at infinity (the first argument will not be instantiated):

$\underline{\texttt{resource}(\texttt{X}, \texttt{Y})}, \underline{\texttt{zeros}(\texttt{Y})} \rightsquigarrow^{id} \rightsquigarrow \underline{\texttt{resource}(\texttt{X}, \texttt{Y})}, \texttt{zeros}(\texttt{Y}) \rightsquigarrow^{Y \mapsto [0|Y']} \rightsquigarrow$
$\underline{\texttt{resource}(\texttt{X}, [0|\texttt{Y}'])}, \underline{\underline{\texttt{zeros}(\texttt{Y}')}} \rightsquigarrow \ldots$

Case 2. The coinductive definition contains fresh variables that do not allow to accumulate composition of substitutions in the course of a derivation. Imagine clause (*) is replaced by $\texttt{resource}([\texttt{get}(\texttt{X})|\texttt{In}], [\texttt{X}|\texttt{L}]) \leftarrow \texttt{resource}(\texttt{Z}, \texttt{L})$ Then we would still have an infinite looping derivation, but it will not allow us to meaningfully compose the computed substitutions in the first argument:

$\underline{\texttt{resource}(\texttt{X}, \texttt{Y})}, \underline{\texttt{zeros}(\texttt{Y})} \rightsquigarrow^{X \mapsto [get(X')|In], Y \mapsto [X'|L]} \rightsquigarrow \underline{\texttt{resource}(\texttt{Z}, \texttt{L})}, \texttt{zeros}(\texttt{X}'|\texttt{L}) \rightsquigarrow^{X' \mapsto 0} \rightsquigarrow$
$\underline{\texttt{resource}(\texttt{Z}, \texttt{L})}, \underline{\underline{\texttt{zeros}(\texttt{L})}} \rightsquigarrow^{Z \mapsto [get(X'')|In'], L \mapsto [X''|L']} \rightsquigarrow \ldots$

Note that in the last step, the computed substitution for the fresh variable Z does not affect the substitution $X \mapsto [get(X')|In]$. However, the loops will still be detected as shown.

Case 3. The infinite data structure is defined via a circular unification, rather than computed by an infinite number of derivation steps. Imagine that we force our

definition to always produce an infinite stream in its first argument by defining
`resource([get(X)|In], In, [X|L]) ← resource(In, In, L)`.
We can have the following derivation (no need to give a stream of zeros as an input):

$$\underline{\texttt{resource(X, Y, [0|Z])}} \rightsquigarrow^{X \mapsto [get(X')|In], Y \mapsto In, Z \mapsto L, X' \mapsto 0} \underline{\texttt{resource(In, In, L)}} \rightsquigarrow^{In' \mapsto [get(X'')|In']} \ldots^{\infty}$$

Here, we will not have an infinite SLD-derivation, as the last step fails the occurs check. But CoLP's loop detection without occurs check will terminate successfully, as the underlined loop is found and the looping terms will unify by circular unification (denoted by ∞).

In all of the above three cases, if the three programs share the common signature of (*), their greatest complete Herbrand models will contain the term `resource([get(0)|get(0)|...], [0|0|...])`. In fact, the three looping derivations allude to this term when they succeed by the loop detection without occurs check. However, as we have seen from these examples, in neither of the three derivations, the loop detection actually guarantees that there is a way to continue the SLD-derivation lazily in order to compute this infinite ground term at infinity. Thus, in all three cases, the loop detection method is unsound relative to computations at infinity.

The question we ask is as follows: Assuming we can guarantee that none of the three cases will occur in our derivations, can the loop detection method serve as an algorithm for semi-deciding whether a derivation is productive, or equivalently, whether an infinite term is computable at infinity?

In this paper, we answer this question in the positive. Case 2 can be eliminated by a simple syntactic check disallowing fresh (or "existential") variables in the bodies of the clauses. We call the resulting programs *universal*. Case 1 is more subtle. In LP setting, unlike for example functional languages, it is not easy to identify which of the clauses form which inductive definitions by which constructors. Such properties are usually not decided until run time. Consider the following example.

Example 2.1 (*Difficulty in detection of constructor productivity in LP*)
The following program

(0) `p(s(X1), X2, Y1, Y2) ← q(X2, X2, Y1, Y2)`
(1) `q(X1, X2, s(Y1), Y2) ← p(X1, X2, Y2, Y2)`

seemingly defines, by mutual recursion, two coinductive predicates `p` and `q`, with constructor `s`. However, the SLD-derivation for a query `p(s(X), s(Y), s(Z), s(W))` will not produce an infinite term at infinity. However, it will produce loops that will be found by CoLP!

As a solution, we propose to use the notion of *observational productivity* suggested recently by Komendantskaya *et al.* (2017) or Fu and Komendantskaya (2017). Given a logic program P, a query A and an SLD-derivation for P and A, we can analyse the structure of this derivation and detect which of the unifiers used in its course are most general unifiers (mgus) and which are most general matchers (mgms). Systematic analysis of such steps is called *structural resolution* by Johann

et al. (2015). We define a logic program to be *observationally productive* if it is impossible to construct an infinite derivation only by mgms for it. For "good" coinductive programs like e.g. the one of Example 1.2, infinite SLD-derivations exist, but not infinite derivations by mgms: the derivations by mgms terminate as soon as they run out of coinductive constructors to match against, and then derivations by mgus produce further substitutions and thus produce more constructors. Examples 3.2–3.4 will make this intuition clear. In the paper by Komendantskaya *et al.* (2017), an algorithm for checking observational productivity of logic programs was introduced. In this paper, we take this algorithm as a sufficient formal check for ruling out programs giving rise to coinduction without constructors, as described in Case 1 above.

Addressing Case 3 requires modifications to the loop detection algorithm: it should be able to form circular substitutions for cases like (*), while ruling out cases like Case 3.

This paper thus establishes following two results for universal and observationally productive programs:

(1) *It shows that non-termination of derivations for such programs guarantees computation of an infinite term at infinity*. In terms of our "Server" example and its clause (*), if computations continue indefinitely, we know that the server receives and processes an infinite stream of data.
(2) *It proposes a novel loop detection algorithm that guarantees production of an infinite term at infinity*. In the "Server" example terms, if the loop detection algorithm succeeds, we know how the productive computation can proceed infinitely long.

The novelty of these results is three-fold:
– *theoretically*, it is the first time that non-terminating observationally productive derivations are proven to be globally productive (i.e. sound relative to computations at infinity);
– *practically*, it presents the first algorithm for semi-deciding computation of infinite terms at infinity since the notion was introduced in the 80s;
– *methodologically*, all proofs employ the methods of structural resolution, which extends the existing methodological machinery of LP and allows to achieve results that are not directly provable for the SLD-resolution.

The paper proceeds as follows. Section 3 gives all background definitions alongside a modified version of structural resolution that has not appeared in the literature before. This section also proves that this variant of structural resolution is sound and complete relative to SLD-resolution. This allows us to switch freely between SLD- and structural resolution throughout the paper. Section 4 proves soundness and completeness of infinite structural resolution derivations relative to computations at infinity, for universal and observationally productive programs. As a corollary, it gives conditions for soundness of infinite SLD-derivations relative to computations at infinity. Section 5 introduces the novel loop detection algorithm for structural resolution, and proves its soundness relative to computations at infinity. In Section 6, we conclude this paper. Detailed proofs can be found in the supplemental document

of the paper, henceforth referred to as Appendix. The implementation is available at `https://github.com/coalp/Productive-Corecursion` and it is discussed in Appendix A.6.

3 Background: S-resolution and observational productivity

In this section, we introduce structural resolution by means of an operational (small-step) semantics. To enable the analysis of infinite terms, we adopt the standard definitions of first-order terms as trees (Courcelle 1983; Jaffar and Stuckey 1986; Lloyd 1988). But, unlike earlier approaches (Johann *et al.* 2015), we avoid analysis of (SLD-)derivation trees in this paper and work directly with S-resolution reductions.

We write $\mathbb{N}^*$ for the set of all finite words over the set $\mathbb{N}$ of natural numbers. The length of $w \in \mathbb{N}^*$ is denoted $|w|$. The empty word ϵ has length 0; we identify $i \in \mathbb{N}$ and the word i of length 1. A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language* provided: (i) for all $w \in \mathbb{N}^*$ and all $i, j \in \mathbb{N}$, if $wj \in L$, then $w \in L$ and, for all $i < j$, $wi \in L$; and (ii) for all $w \in L$, the set of all $i \in \mathbb{N}$ such that $wi \in L$ is finite. A non-empty tree language always contains ϵ, which we call its *root*. A tree language is *finite* if it is a finite subset of $\mathbb{N}^*$, and *infinite* otherwise.

A *signature* Σ is a non-empty set of *function symbols*, each with an associated arity. To define terms over Σ, we assume a countably infinite set *Var* of *variables* disjoint from Σ, each with arity 0. If L is a non-empty tree language and Σ is a signature, then a *term* over Σ is a function $t : L \to \Sigma \cup Var$ such that, for all $w \in L$, $arity(t(w)) = |\{i \mid wi \in L\}|$. Terms are finite or infinite if their domains are finite or infinite. A term t has a depth $depth(t) = 1 + \max\{|w| \mid w \in L\}$.

Example 3.1 (*Term tree*)
Given $L = \{\epsilon, 0, 00, 01\}$, the atom `stream(scons(0,Y))` can be seen as the term tree t given by the map $t(\epsilon) =$ `stream`, $t(0) =$ `scons`, $t(00) =$ `0`, $t(01) =$ `Y`.

The set of finite (infinite) terms over a signature Σ is denoted by $\mathbf{Term}(\Sigma)$ ($\mathbf{Term}^{\infty}(\Sigma)$). The set of *all* (i.e. finite *and* infinite) terms over Σ is denoted by $\mathbf{Term}^{\omega}(\Sigma)$. Terms with no occurrences of variables are *ground*. We write $\mathbf{GTerm}(\Sigma)$ ($\mathbf{GTerm}^{\infty}(\Sigma)$, $\mathbf{GTerm}^{\omega}(\Sigma)$) for the set of finite (infinite, *all*) ground terms over Σ.

A *substitution* over Σ is a total function $\sigma : Var \to \mathbf{Term}^{\omega}(\Sigma)$. Substitutions are extended from variables to terms homomorphically. We write *id* for the identity substitution. Composition of substitutions is denoted by juxtaposition. Composition is associative, so we write $\sigma_3\sigma_2\sigma_1$ rather than $(\sigma_3\sigma_2)\sigma_1$ or $\sigma_3(\sigma_2\sigma_1)$.

A substitution σ is a *unifier* for $t, u \in \mathbf{Term}(\Sigma)$ if $\sigma(t) = \sigma(u)$, and is a *matcher* for t against u if $\sigma(t) = u$. If $t, u \in \mathbf{Term}^{\omega}(\Sigma)$, then we say that u is an *instance* of t if $\sigma(t) = u$ for some σ. A substitution σ_1 is *more general* than a substitution σ_2 if there exists a substitution σ such that $\sigma\sigma_1(\mathtt{X}) = \sigma_2(\mathtt{X})$ for every $\mathtt{X} \in Var$. A substitution σ is an *mgu* for t and u, denoted $t \sim_\sigma u$, if it is a unifier for t and u and is more general than any other such unifier. An *mgm* σ for t against u, denoted $t \prec_\sigma u$, is defined analogously. Both mgus and mgms are unique up to variable renaming if they exist. In many unification algorithms, the *occurs check* condition is imposed, so that mappings $\mathtt{X} \mapsto t[\mathtt{X}]$, where $t[\mathtt{X}]$ is a term containing $\mathtt{X}$, are disallowed. We will assume

that mgus and mgms are computed by any standard unification algorithm (Lloyd 1988) with occurs check, unless otherwise stated.

A clause C over Σ is given by $A \leftarrow B_0, \ldots, B_n$ where the head $A \in \mathbf{Term}(\Sigma)$ and the *body* $B_0, \ldots B_n$ are a list of terms in $\mathbf{Term}(\Sigma)$. Throughout the paper, we refer to standard definitions of the least and greatest complete Herbrand models and recall them in Appendix A.1.

Following Fu and Komendantskaya (2017), we distinguish several kinds of reductions for LP:

Definition 3.1 (*Different kinds of reduction in LP*)
If P is a logic program and $A_1, \ldots, A_n$ are atoms, then

- *SLD-resolution reduction*:
  $[A_1, \ldots, A_i, \ldots, A_n] \rightsquigarrow_P [\sigma(A_1), \ldots, \sigma(A_{i-1}), \sigma(B_0), \ldots, \sigma(B_m), \sigma(A_{i+1}), \ldots, \sigma(A_n)]$
  if $A \leftarrow B_0, \ldots, B_m \in P$ and $A_i \sim_\sigma A$;
- *rewriting reduction*:
  $[A_1, \ldots, A_i, \ldots, A_n] \rightarrow_P [A_1, \ldots, A_{i-1}, \sigma(B_0), \ldots, \sigma(B_m), A_{i+1}, \ldots, A_n]$
  if $A \leftarrow B_0, \ldots, B_m \in P$ and $A \prec_\sigma A_i$;
- *substitution reduction*:
  $[A_1, \ldots, A_i, \ldots, A_n] \hookrightarrow_P [\sigma(A_1), \ldots, \sigma(A_i), \ldots, \sigma(A_n)]$
  if $A \leftarrow B_0, \ldots, B_m \in P$ and $A \sim_\sigma A_i$ but not $A \prec_\sigma A_i$.
  In each of the above three cases, we will say that A_i *resolves against the clause* $A \leftarrow B_0, \ldots, B_m \in P$.

We may omit explicit mention of P as a subscript on reductions when it is clear from context. We write $\rightarrow^n$ to denote rewriting by *at most* n steps of $\rightarrow$, where n is a natural number. We use similar notations for $\rightsquigarrow$ and $\hookrightarrow$ as required. We assume, as is standard in LP, that all variables are *standardised apart* when terms are matched or unified against the program clauses.

If r is any reduction relation, we will call any (finite or infinite) sequence of r-reduction steps an *r-derivation.* An r-derivation is called an *r-refutation* if its last goal is the empty list. An SLD-resolution derivation is *fair* if either it is finite, or it is infinite and, for every atom B appearing in some goal in the SLD-derivation, (a further instantiated version of) B is chosen within a finite number of steps.

Example 3.2 (*SLD- and rewriting reductions*)
The following are SLD-resolution and rewriting derivations, respectively, with respect to the program of Example 1.2:

- $[\mathtt{nats(X)}] \rightsquigarrow [\mathtt{nat(X')}, \mathtt{nats(Y)}] \rightsquigarrow [\mathtt{nats(Y)}] \rightsquigarrow [\mathtt{nat(X'')}, \mathtt{nats(Y')}] \rightsquigarrow \ldots$
- $[\mathtt{nats(X)}]$

The reduction relation $\rightsquigarrow_P$ models traditional SLD-resolution steps (Lloyd 1988) with respect to P. Note that for this program, SLD-resolution derivations are infinite, but rewriting derivations are always finite.

The observation that, for some coinductive programs, $\rightarrow$ reductions are finite and thus can serve as measures of finite observation, has led to the following definition of

observational productivity in LP, first introduced in the paper by Komendantskaya *et al.* (2016).

Definition 3.2 (*Observational productivity*)
A program P is *observationally productive* if every rewriting derivation with respect to P is finite.

Example 3.3 (*Observational productivity*)
The program from Example 1.2 is observationally productive, whereas the program from Example 1.3 – not, as we have a rewriting derivation: $\mathtt{bad(f(X))} \to \mathtt{bad(f(X))} \to \mathtt{bad(f(X))} \to \ldots$.

Because rewriting derivations are incomplete, they can be combined with substitution reductions to achieve completeness, which is the main idea behind the *structural resolution* explored by Komendantskaya *et al.* (2016), Johann *et al.* (2015) and Fu and Komendantskaya (2017). Below, we present yet another version of combining the two kinds of reductions:

Definition 3.3 (*S-resolution reduction*)
Given a productive program P, we define *S-resolution reduction*:
$[A_1, \ldots, A_n] \to_P^n [B_1, \ldots, B_i, \ldots, B_m] \hookrightarrow_P [\theta(B_1), \ldots, \theta(B_i), \ldots, \theta(B_m)] \to_P$
$[\theta(B_1), \ldots, \theta(B_{i-1}), \theta(C_1), \ldots, \theta(C_k), \theta(B_{i+1}), \ldots, \theta(B_m)]$,
where $C \leftarrow C_1, \ldots, C_k$ is a clause in P, and $C \sim_\theta B_i$. We will denote this reduction by
$[A_1, \ldots, A_n] \rightsquigarrow_P^S [\theta(B_1), \ldots, \theta(B_{i-1}), \theta(C_1), \ldots, \theta(C_k), \theta(B_{i+1}), \ldots, \theta(B_m)]$.

Example 3.4 (*S-resolution reduction*)
Continuing Example 3.2, S-resolution derivation for that logic program is given by
$[\mathtt{nats(X)}] \hookrightarrow [\mathtt{nats(scons(X', Y))}] \to [\mathtt{nat(X')}, \mathtt{nats(Y)}] \hookrightarrow [\mathtt{nat(0)}, \mathtt{nats(Y)}]$
$\to [\mathtt{nats(Y)}] \hookrightarrow [\mathtt{nats(scons(X'', Y'))}] \to \ldots$
The initial sequences of the SLD-resolution (see Example 3.2) and S-resolution reductions shown above each compute the partial answer $\{\mathtt{X} \mapsto \mathtt{scons(0, scons(X'', Y'))}\}$ to the query $\mathtt{nats(X)}$.

Several formulations of S-resolution exist in the literature (Johann *et al.* 2015; Komendantskaya *et al.* 2016; Fu and Komendantskaya 2017), some of them are incomplete relative to SLD-resolution. However, the above definition is complete:

Theorem 3.1 (*Operational equivalence of terminating S-resolution and SLD-resolution*)

Given a logic program P and a goal A, there is an SLD-refutation for P and A iff there is an S-refutation for P and A.

Proof
Both parts of the proof proceed by induction on the length of the SLD- and S-refutations. The proofs rely on one-to-one correspondence between SLD- and S-derivations. Suppose D is an SLD-derivation, then we can construct a corresponding S-derivation D^* as follows. Some reductions in D in fact compute mgms, these reductions are modelled by rewriting reductions in D^* directly. Those reductions that involve proper mgus in D are modelled by composition of two reduction steps $\to \circ \hookrightarrow$ in D^*. □

A corollary of this theorem is inductive soundness and completeness of S-refutations relative to the least Herbrand models.

4 Soundness of infinite S-resolution relative to SLD-computations at infinity

A first attempt to give an operational semantics corresponding to greatest complete Herbrand models of logic programs was captured by the notion of *computations at infinity* for SLD-resolution (van Emden and Abdallah 1985; Lloyd 1988). Computations at infinity are usually given relative to an ultrametric on terms, constructed as follows.

We define the *truncation* of a term $t \in \mathbf{Term}^{\omega}(\Sigma)$ at depth $n \in \mathbb{N}$, denoted by $\gamma'(n,t)$. We introduce a new nullary symbol $\diamond$ to denote the leaves of truncated branches.

Definition 4.1 (*Truncation of a term*)
A *truncation* is a mapping $\gamma' : \mathbb{N} \times \mathbf{Term}^{\omega}(\Sigma) \to \mathbf{Term}(\Sigma \cup \diamond)$ where, for every $n \in \mathbb{N}$ and $t \in \mathbf{Term}^{\omega}(\Sigma)$, the term $\gamma'(n,t)$ is constructed as follows:

(a) the domain $dom(\gamma'(n,t))$ of the term $\gamma'(n,t)$ is $\{m \in dom(t) \mid |m| \leqslant n\}$;
(b)

$$\gamma'(n,t)\ (m) = \begin{cases} t(m) & \text{if } |m| < n \\ \diamond & \text{if } |m| = n \end{cases}$$

For $t, s \in \mathbf{Term}^{\omega}(\Sigma)$, we define $\gamma(s,t) = min\{n \mid \gamma'(n,s) \neq \gamma'(n,t)\}$, so that $\gamma(s,t)$ is the least depth at which t and s differ. If we further define $d(s,t) = 0$ if $s = t$ and $d(s,t) = 2^{-\gamma(s,t)}$ otherwise, then $(\mathbf{Term}^{\omega}(\Sigma), d)$ is an ultrametric space.

The definition of SLD-computable at infinity relative to a given ultrametric was first given by Lloyd (1988), we extend it here and redefine this with respect to an arbitrary infinite term, rather than a ground infinite term:

Definition 4.2 (*Formulae SLD-computable at infinity*)
The term tree $t^{\infty} \in \mathbf{Term}^{\infty}(\Sigma)$ is *SLD-computable at infinity* with respect to a program P if there exists a term tree $t \in \mathbf{Term}(\Sigma)$ and an infinite fair SLD-resolution derivation $G_0 = t \rightsquigarrow G_1 \rightsquigarrow G_2 \rightsquigarrow \ldots G_k \rightsquigarrow \ldots$ with mgus $\theta_1, \theta_2, \ldots, \theta_k, \ldots$ such that $d(t^{\infty}, \theta_k \ldots \theta_1(t)) \to 0$ as $k \to \infty$. If such a t exists, we say that t^{∞} is SLD-computable at infinity with respect to t.

Note the fairness requirement above. Defining $C_P = \{t^{\infty} \in \mathbf{GTerm}^{\infty}(\Sigma) \mid t^{\infty}$ is SLD-computable at infinity with respect to a program P by some $t \in \mathbf{Term}(\Sigma)\}$, we have that C_P is a subset of the greatest complete Herbrand model of P (van Emden and Abdallah 1985; Lloyd 1988).

Example 4.1 (*Existential variables and SLD computations at infinity*)
Consider the following program that extends Example 1.2.
(3) `p(Y)` ← `nats(X)`
Although an infinite term is SLD-computable at infinity with respect to `nats(X)`, no infinite instance of `p(Y)` is SLD-computable at infinity. Nevertheless, `p(0)` and

other instances of p(Y) are logically entailed by this program and are in its greatest complete Herbrand model.

Our Case 2 of Section 2 is also an example of the same problem.

To avoid such problems, we introduce a restriction on the shape of clauses and work only with logic programs in which only variables occurring in the heads of clauses can occur in their bodies. Formally, for each program clause $C \leftarrow C_1, \ldots, C_n$, we form the set $FV(C)$ of all free variables in C, and similarly the set $FV(C_1, \ldots, C_n)$ of all free variables in $C_1, \ldots, C_n$. We require that $FV(C_1, \ldots, C_n) \subseteq FV(C)$ and call such logic programs *universal logic programs.*

We now establish an important property – that if a program is observationally productive and universal, then it necessarily gives rise to globally productive S-resolution derivations. We call a substitution θ for a variable X *trivial* if it is a renaming or identity mapping, otherwise it is *non-trivial for* X. A substitution θ is non-trivial for a term t with variables $X_1, \ldots, X_n$ if θ is non-trivial for at least one variable $X_i \in \{X_1, \ldots, X_n\}$. We will call a composition $\theta_n \ldots \theta_1$ non-trivial for t if θ_1 is non-trivial for t and for each $1 < k \leqslant n$, θ_k is non-trivial for term $\theta_{k-1} \ldots \theta_1(t)$.

Lemma 4.1 (*Productivity lemma*)
Let P be an observationally productive and universal program and let $t \in \mathbf{Term}(\Sigma)$. Let D be an infinite S-resolution derivation given by $G_0 = t \rightsquigarrow^S G_1 \rightsquigarrow^S G_2 \rightsquigarrow^S \ldots$, then for every $G_i \in D$, there is a $G_j \in D$, with $j > i$, such that, given computed mgus $\theta_i, \ldots, \theta_1$ up to G_i and the computed mgus $\theta_j, \ldots, \theta_1$ up to G_j, $d(t^\infty, \theta_i, \ldots, \theta_1(t)) > d(t^\infty, \theta_j, \ldots, \theta_1(t))$, for some term $t^\infty \in \mathbf{Term}^\infty(\Sigma)$.

Proof Sketch
The full proof is given in Appendix A.2, and proceeds by showing that universal and productive programs have to produce non-trivial substitutions in the course of S-resolution derivations. Non-trivial computed substitutions contribute to construction of the infinite term at the limit. The construction would have been a mere adaptation of the limit term construction defined by Lloyd (1988, p. 177), had we not extended the notion of SLD-computable at infinity to all, not just ground, terms. This extension required us to redefine the limit term construction substantially, and use the properties of observationally productive S-resolution reductions. □

Note how both requirements of universality and productivity are crucial in the above lemma. For non-productive programs as in Example 1.3 or in Case 1 of Section 2, an infinite sequence of rewriting steps will not produce any substitution. Analysis of Case 2 in Section 2 explains the importance of the universality condition.

We now state the first major result: that for productive and universal logic programs, S-resolution derivations are sound and complete relative to SLD-computations at infinity. First, we extend the notion of a fair derivation to S-resolution. An S-resolution derivation is *fair* if either it is finite, or it is infinite and, for every atom B appearing in some goal in the S-resolution derivation, (a further instantiated version of) B is resolved against some program clause within a finite number of steps.

Theorem 4.1 (*Soundness and completeness of observationally productive S-resolution*)

Let P be an observationally productive and universal program, and let $t \in \mathbf{Term}(\Sigma)$.

There is an infinite fair S-resolution derivation for t iff there is a $t' \in \mathbf{Term}^{\infty}(\Sigma)$, such that t' is SLD-computable at infinity with respect to t.

Proof Sketch

The full proof is in Appendix A.3. It first proceeds by coinduction to show a one-to-one correspondence between an infinite SLD-derivation and an infinite S-derivation. The argument is very similar to the proof of Theorem 3.1. The rest of the proof uses Productivity Lemma 4.1 to show that an infinite derivation must produce an infinite term as a result. □

The practical significance of the above theorem is in setting the necessary and sufficient conditions for guaranteeing that, given an infinite fair S-resolution reduction, we are guaranteed that it will compute an infinite term at infinity. We emphasise this consequence in a corollary.

Corollary 4.1 (*Global productivity of infinite S-resolution derivations*)

Let P be an observationally productive and universal program, and let $t \in \mathbf{Term}(\Sigma)$.

If there is an infinite fair S-resolution derivation $G_0 = t \rightsquigarrow^S G_1 \rightsquigarrow^S G_2 \rightsquigarrow^S \ldots G_k \rightsquigarrow^S \ldots$ with mgus $\theta_1, \theta_2, \ldots \theta_k \ldots$, then there exists $t^{\infty} \in \mathbf{Term}^{\infty}(\Sigma)$ such that $d(t^{\infty}, \theta_k \ldots \theta_1(t)) \to 0$ as $k \to \infty$.

As a corollary of soundness of SLD-computations at infinity (Lloyd 1988) and the above result, we obtain that fair and infinite S-resolution derivations are sound relative to complete Herbrand models, given an observationally productive and universal program P. Another important corollary that follows from the construction of the proof above guarantees that, if a program is observationally productive and universal, then any infinite fair SLD-resolution derivation for it will result in computation of an infinite term at infinity.

Corollary 4.2 (*Global productivity of infinite SLD-resolution derivations*)

Let P be an observationally productive and universal program, and let $t \in \mathbf{Term}(\Sigma)$.

If there is an infinite fair SLD-resolution derivation for t, then there is a $t^{\infty} \in \mathbf{Term}^{\infty}(\Sigma)$, such that t^{∞} is SLD-computable at infinity with respect to t.

This section has established that infinite derivations for universal and observationally productive programs *"cannot go wrong"*, in the sense that they are guaranteed to be globally productive and compute infinite terms. This is the first time a result of this kind is proven in LP literature. Although sound, infinite S- or SLD-resolution derivations do not provide an implementable procedure for semi-deciding coinductive entailment. We address this problem in the next section.

5 Co-S-resolution

In this section, we embed a loop detection algorithm in S-resolution derivations, and thus obtain an algorithm of *co-S-resolution* that can semi-decide whether an

infinite term is SLD-computable at infinity for observationally productive and universal programs. To achieve this, we refine the loop detection method of co-SLD resolution (Gupta *et al.* 2007; Simon *et al.* 2007; Ancona and Dovier 2015) which we recall in Appendix A.4 for convenience.

We use $\approx$ to denote unification without occurs check, see e.g. Colmerauer (1982) for the algorithm. We use the notational style defined by Ancona and Dovier (2015) and introduce a set S_i for each predicate A_i in a goal, where S_i records the atoms from previous goals whose derivation depends on the derivation of A_i. We call S_i the *ancestors set* for A_i.

Definition 5.1 (*Algorithm of Co-S-resolution*)

- *rewriting reduction* ($G \to G'$): Let $G = [(A_1, S_1), \ldots, (A_n, S_n)]$. If $B_0 \prec_\theta A_k$ for some program clause $B_0 \leftarrow B_1, \ldots, B_m$ and some k, then let $S' = S_k \cup \{A_k\}$. Then we derive

$$G' = [(A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (\theta(B_1), S'), \ldots, (\theta(B_m), S'), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)].$$

- *substitution reduction* ($G \hookrightarrow G'$): Let $G = [(A_1, S_1), \ldots, (A_n, S_n)]$. If $B_0 \sim_\theta A_k$ but not
$B_0 \prec_\theta A_k$ for some program clause $B_0 \leftarrow B_1, \ldots, B_m$ and some k. Then we derive

$$G' = \theta\big([(A_1, S_1), \ldots, (A_n, S_n)] \big).$$

- *S-reduction* ($G \rightsquigarrow^S G'$): $G \to^n [(A_1, S_1), \ldots, (A_n, S_n)] \hookrightarrow \theta\big([(A_1, S_1), \ldots, (A_n, S_n)]\big) \to$

$$G' = \theta\big([(A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (B_1, S'), \ldots, (B_m, S'), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)] \big)$$

where $B_0 \sim_\theta A_k$ for some program clause $B_0 \leftarrow B_1, \ldots, B_m$ and some k, $S' = S_k \cup \{A_k\}$.

- *co-SLD loop detection* ($G \to_\infty G'$): Let $G = [(A_1, S_1), \ldots, (A_n, S_n)]$. If $A_k \approx_\theta B$ for some k and some $B \in S_k$. Then we derive

$$G' = \theta\big([(A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)] \big).$$

- *restricted loop detection* ($G \to_{co} G'$): Let $G = [(A_1, S_1), \ldots, (A_n, S_n)]$. If $A_k \approx_\theta B$ for some k and some $B \in S_k$, and B' is an instance of A_k, where B' is a fresh-variable variant of B. Then we derive

$$G' = [(A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)].$$

- *co-S-reduction* ($G \rightsquigarrow^S_{\text{co}} G'$): $G \rightsquigarrow^S_{\text{co}} G'$ if $G \rightsquigarrow^S G'$ or $G \to_{\text{co}} G'$.

Note that, unlike CoLP that uses no occurs check at all, our definition of co-S-reduction still relies on occurs check within S-reductions. The next difference to notice is our use of the "restricted loop detection" rule instead of the "co-SLD loop detection" rule defined by Ancona and Dovier (2015), Gupta *et al.* (2007) and Simon *et al.* (2007). The below examples explain the motivation behind the introduced restriction:

Example 5.1 (*Undesirable effect of circular unification without occurs check*)
Consider the following universal and observationally productive program that resembles the one in Case 3 of Section 2:

(0) $\mathtt{p(X, s(X))} \leftarrow \mathtt{q(X)}$
(1) $\mathtt{q(s(X))} \leftarrow \mathtt{p(X, X)}$

If we use the co-SLD loop detection rule instead of the restricted loop detection rule in the definition of co-S-reduction, we would have the following co-S-refutation: $[\big(p(X, s(X)), \emptyset\big)] \rightarrow [\big(q(X), \{p(X, s(X))\}\big)] \hookrightarrow^{X \mapsto s(X_1)} [\big(q(s(X_1)), \{p(s(X_1), s^2(X_1))\}\big)] \rightarrow \underline{[\big(p(X_1, X_1), \{q(s(X_1)), p(s(X_1), s^2(X_1))\}\big)]} \rightarrow_{\infty}^{X_1 \mapsto s(X_1)} [\,]$

The answer would be given by composition $\{X \mapsto s(X_1)\}\{X_1 \mapsto s(X_1)\} = \{X \mapsto s^{\omega}, X_1 \mapsto s^{\omega}\}$. However, as Case 3 of Section 2 explains, this derivation in CoLP style does not correspond to any SLD-computation at infinity: both SLD- and S-derivations fail at the underlined goal because the subgoal $p(X_1, X_1)$ does not unify with any program clause (recall that unification with occurs check is used in SLD- and S-derivations). The restricted loop detection rule will also fail at the underlined goal, because there is no matcher for $p(X_1, X_1)$ and $p(s(X'), s^2(X'))$.

The next example presents another case where our restriction to the loop detection is necessary:

Example 5.2 (*Ensuring precision of the answers by co-S-resolution*)
Consider the following universal and observationally productive program:
(0) $\mathtt{p(Y, s(X))} \leftarrow \mathtt{p(f(Y), X)}$

Again, if we use the co-SLD loop detection rule instead of the restricted loop detection rule, we would have the following co-S-refutation:
$[\big(p(Y, s(X)), \emptyset\big)] \rightarrow [\big(p(f(Y), X), \{p(Y, s(X))\}\big)] \hookrightarrow^{X \mapsto s(X_1)} [\big(p(f(Y), s(X_1)), \{p(Y, s^2(X_1))\}\big)]$
$\rightarrow \underline{[\big(p(f^2(Y), X_1), \{p(f(Y), s(X_1)), p(Y, s^2(X_1))\}\big)]} \rightarrow_{\infty}^{Y \mapsto f(Y), X_1 \mapsto s(X_1)} [\,]$

The answer is given by composition $\{X \mapsto s(X_1)\}\{Y \mapsto f(Y), X_1 \mapsto s(X_1)\}$ which instantiates goal $p(Y, s(X))$ to the infinite term $t_1 = p(f^{\omega}, s^{\omega})$.

Now consider the infinite fair S-derivation for the same goal:
$[p(Y, s(X))] \rightarrow [p(f(Y), X)] \hookrightarrow^{X \mapsto s(X_1)} [p(f(Y), s(X_1))] \rightarrow [p(f^2(Y), X_1)] \hookrightarrow^{X_1 \mapsto s(X_2)}$
$[p(f^2(Y), s(X_2))] \rightarrow [p(f^3(Y), X_2)] \hookrightarrow \cdots$
By the discussion of the previous section, we can construct a corresponding infinite SLD-derivation, which computes at infinity the infinite term $t_2 = p(Y, s^{\omega})$. Thus, in this case, there is an SLD-computation at infinity that corresponds to our proof by co-SLD loop detection rule, but it does not approximate $t_1 = p(f^{\omega}, s^{\omega})$.

In contrast, the restricted loop detection rule will fail at the underlined goal because neither $p(Y', s^2(X_1'))$ nor $p(f(Y'), s(X_1'))$ is an instance of $p(f^2(Y), X_1)$.

The next theorem is our main result:

Theorem 5.1 (*Soundness of co-S-resolution relative to SLD-computations at infinity*)
Let P be an observationally productive and universal logic program, and $t \in \mathbf{Term}(\Sigma)$ be an atomic goal. If there exists a co-S-refutation for P and t that involves the restricted loop detection reduction, and computes the substitution θ, then

(1) there exists an infinite fair S-derivation for P and t, and
(2) there is a term $t^{\infty} \in \mathbf{Term}^{\infty}(\Sigma)$ SLD-computed at infinity that is a variant of $\theta(t)$.

Proof Sketch
The full proof is given in Appendix A.5, it develops a method of "decircularisation", or infinite unfolding, of circular substitutions computed by the restricted loop detection of co-S-resolution. We then show how an infinite number of S-resolution steps corresponds to computation of the infinite term resulting from applying decircularisation. Similarly to related work, e.g. Ancona and Dovier (2015), we relate the use of loop detection to the existence of an infinite derivation. However, our proof does not restrict the shape of corecursion to some simple form, e.g. mutual recursion as by Ancona and Dovier (2015). This opens a possibility for future extension of this proof method. □

Similarly to any other loop detection method, co-S-resolution is incomplete relative to SLD-computations at infinity. Taking any logic program that defines irregular streams (cf. Example 6.1) will result in S-derivations with loops where subgoals do not unify.

Because the restricted loop detection is an instance of the co-SLD loop detection, and because there is a one-to-one correspondence between SLD- and S-resolution reductions, we can prove that co-S-resolution is sound relative to the greatest complete Herbrand models, by adapting the proof by Ancona and Dovier (2015), Gupta *et al.* (2007) and Simon *et al.* (2007), see Appendix A.4.

6 Conclusions, discussion, related and future work

Conclusions. We have given a computational characterisation to the *SLD-computations at infinity* (van Emden and Abdallah 1985; Lloyd 1988) introduced in the 1980s. Relying on the recently proposed notion of observational productivity of logic programs, we have shown that infinite observationally productive derivations are sound and complete relative to the SLD-computations at infinity. This paper thus confirmed the conjecture made by Komendantskaya *et al.* (2017) that the weaker notion of observational productivity of logic programs implies the much stronger notion of global productivity of individual derivations. This result only holds on extra condition – universality of logic programs in question. This fact has not been known prior to this paper.

We have introduced co-S-resolution that gives the first algorithmic characterisation of the SLD-computations at infinity. We proved that co-S-resolution is sound relative to the SLD-computations at infinity for universal and observationally productive programs. Appendix A.6 discusses its implementation. Structural resolution, seen as a method of systematic separation of SLD-resolution to mgu and mgm steps, has played an instrumental role in the proofs.

Discussion. Imposing the conditions of observational productivity and universality allowed us to study the operational properties of infinite productive SLD-derivations without introducing any additional modifications to the resolution algorithm. These

results can be directly applied in any already existing Prolog implementation: ensuring that a program satisfies these conditions ensures that all infinite computations for it are productive. The results can similarly be reused in any other inference algorithm based on resolution; this paper showed its adaptation to CoLP.

As a trade-off, both conditions exclude logic programs that may give rise to globally productive SLD derivations for certain queries. For example, a non-productive logic program that joins clauses of Examples 1.2 and 1.3 will still result in productive derivations for a query `nats(X)`, as computations calling the `bad` clause would not interfere in such derivations. The universality condition excludes following two cases of globally productive derivations:

(1) Existential variables in the clause body have no effect on the arguments in which the infinite term is computed. E.g. taking a coinductive definition $\mathtt{p(s(X),Y)} \leftarrow \mathtt{p(X,Z)}$ and a goal $\mathtt{p(X,Y)}$, an infinite term will be produced in the first argument, and the existential variable occurring in the second argument plays no role in this computation.
(2) Existential variables play a role in SLD-computations of infinite terms at infinity. In such cases, they usually depend on other variables in the coinductive definition, and their productive instantiation is guaranteed by other clauses in the program.

The famous definition of the stream of Fibonacci numbers is an example:

Example 6.1 (*Fibonacci numbers*)

(0) $\mathtt{add(0,Y,Y)} \leftarrow$
(1) $\mathtt{add(s(X),Y,s(Z))} \leftarrow \mathtt{add(X,Y,Z)}$
(2) $\mathtt{fibs(X,Y,[X|S])} \leftarrow \mathtt{add(X,Y,Z)}, \mathtt{fibs(Y,Z,S)}$

The goal $\mathtt{fibs(0,s(0),F)}$ computes the infinite substitution $\mathtt{F} \mapsto [\mathtt{0}, \mathtt{s(0)}, \mathtt{s(0)}, \mathtt{s}^2\mathtt{(0)}, \ldots]$ at infinity. The existential variable `Z` in the body of clause (2) is instantiated by the `add` predicate, as Z in fact functionally depends on X and Y (all three variables contribute to construction of the infinite term in the third argument of `fibs`).

In the future, these classes of programs may be admitted by following either of the two directions:

(1) Using program transformation methods to ensure that every logic program is transformed into observationally productive and universal form. An example of the observationally productive transformation is given by Fu and Komendantskaya (2017), and of the universal transformation – by Senni *et al.* (2008) and by Proietti and Pettorossi (1995). This approach may have drawbacks, such as changing the coinductive models of the programs.
(2) Refining the resolution algorithm and/or the loop detection method. For example, the restrictions imposed on the loop detection in Definition 5.1 can be further refined. Another solution is to use the notion of *local productivity* (Fu and Komendantskaya 2017) instead of the observational productivity. A derivation is locally productive if it computes an infinite term only at a certain argument. Both examples given in this section are locally productive. In the paper by Fu

and Komendantskaya (2017), local productivity required technical modifications to the unification algorithm (involving labelling of variables), and establishing its soundness is still an open problem.

Related Work. The related paper by Li (2017) shows that an algorithm embedding the CoLP loop detection rule into S-resolution is sound relative to greatest complete Herbrand models. However, that work does not consider conditions on which such embedding would be sound relative to SLD-computations at infinity. In fact, as this paper shows, simply embedding the CoLP loop detection rule into S-resolution does not make the resulting coinductive proofs sound relative to SLD-computations at infinity, as the three undesirable cases of Section 2 may still occur. The construction of the proof of Theorem 5.1 in this paper agrees with a similar construction for non-terminating SLDNF derivation (Shen *et al.* 2003). We show that the use of restricted loop detection corresponds to infinite S-derivation characterised by infinitely repeating subgoal variants, while Shen *et al.* (2003) show that a non-terminating SLDNF derivation admits an infinite sequence of subgoals that are either variants or increasing in size.

Future Work. Similarly to other existing loop detection methods, co-S-resolution is incomplete, as it only captures regular infinite terms. Future work will be to introduce heuristics extending our methods to irregular structures. Another direction for future work is to investigate practical applications of co-S-resolution in Internet programming and type inference in programming languages, as was done by Fu *et al.* (2016).

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S147106841700028X

References

Aczel, P. 1977. An introduction to inductive definitions. *Studies in Logic and the Foundations of Mathematics 90*, 739–782.

Ancona, D. and Dovier, A. 2015. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae 140,* 3–4, 221–246.

Colmerauer, A. 1982. *Prolog and Infinite Trees*. Academic Press.

Courcelle, B. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science 25*, 95–169.

Davison, A. 2002. Logic programming languages for the internet. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, A. C. Kakas and F. Sadri, Eds. Springer-Verlag, London, UK, 66–104.

Fu, P. and Komendantskaya, E. 2017. Operational semantics of resolution and productivity in Horn Clause logic. *Journal on Formal Aspects of Computing 29,* 3, 453–474.

Fu, P., Komendantskaya, E., Schrijvers, T. and Pond, A. 2016. Proof relevant corecursive resolution. In *FLOPS'16*. Lecture Notes in Computer Science, vol. 9613. Springer, 126–143.

Gupta, G., Bansal, A., Min, R., Simon, L. and Mallya, A. 2007. Coinductive logic programming and its applications. In *ICLP*, 27–44.

Heintze, N. and Jaffar, J. 1992. Semantic types for logic programs. In *Types in Logic Programming*, F. Pfening, ed. The MIT Press, 141–155.

Jaffar, J. and Stuckey, P. J. 1986. Semantics of infinite tree logic programming. *Theoretical Computer Science 46,* 3, 141–158.

Johann, P., Komendantskaya, E. and Komendantskiy, V. 2015. Structural resolution for logic programming. In *Proc. of Technical Communications of ICLP*.

Komendantskaya, E., Johann, P. and Schmidt, M. 2017. A productivity checker for logic programming. Logic-Based Program Synthesis and Transformation, In *Proc. of 26th International Symposium, LOPSTR 2016*, Edinburgh, UK, September 6–8, 2016, Revised Selected Papers. M. V. Hermenegildo and P. Lopez-Garcia, Eds. Springer International Publishing, 168–186.

Komendantskaya, E., Power, J. and Schmidt, M. 2016. Coalgebraic logic programming: From semantics to implementation. *Journal of Logic and Computation 26,* 2, 745–783.

Li, Y. 2017. Structural resolution with coinductive loop detection. In *Post-Proc. of CoALP-Ty'16*, E. Komendantskaya and J. Power, Eds. Open Publishing Association, to appear in.

Lloyd, J. 1988. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag.

Proietti, M. and Pettorossi, A. 1995. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science 142,* 1, 89–124.

Sangiorgi, D. 2011. *Introduction to Bisimulation and Coinduction.* Cambridge University Press.

Senni, V., Pettorossi, A. and Proietti, M. 2008. *A Folding Algorithm for Eliminating Existential Variables from Constraint Logic Programs.* Springer, Berlin, Heidelberg, 284–300.

Shen, Y.-D., You, J.-H., Yuan, L.-Y., Shen, S. S. P. and Yang, Q. 2003. A dynamic approach to characterizing termination of general logic programs. *ACM Transactions on Computational Logic 4,* 4, 417–430.

Simon, L., Bansal, A., Mallya, A. and Gupta, G. 2007. Co-logic programming: Extending logic programming with coinduction. In *Proc. of ICALP*, 472–483.

Simon, L., Mallya, A., Bansal, A. and Gupta, G. 2006. Coinductive logic programming. In *ICLP'06*, 330–345.

van Emden, M. H. and Abdallah, M. A. N. 1985. Top-down semantics of fair computations of logic programs. *Journal of Logic Programming 2,* 1, 67–75.