A Productivity Checker for Logic Programming

E. Komendantskaya[1] and P. Johann[2] and M.Schmidt[3]

[1] Heriot-Watt University, Edinburgh, Scotland, UK
[2] Appalachian State University, Boone, NC, USA
[3] University of Osnabrück, Osnabrück, Germany

Abstract. Automated analysis of recursive derivations in logic programming is known to be a hard problem. Both termination and non-termination are undecidable problems in Turing-complete languages. However, some declarative languages offer a practical work-around for this problem, by making a clear distinction between whether a program is meant to be understood inductively or coinductively. For programs meant to be understood inductively, termination must be guaranteed, whereas for programs meant to be understood coinductively, productive non-termination (or "productivity") must be ensured. In practice, such classification helps to better understand and implement some non-terminating computations.
Logic programming was one of the first declarative languages to make this distinction: in the 1980's, Lloyd and van Emden's "computations at infinity" captured the big-step operational semantics of derivations that produce infinite terms as answers. In modern terms, computations at infinity describe "global productivity" of computations in logic programming. Most programming languages featuring coinduction also provide an observational, or small-step, notion of productivity as a computational counterpart to global productivity. This kind of productivity is ensured by checking that finite initial fragments of infinite computations can always be observed to produce finite portions of their infinite answer terms.
In this paper we introduce a notion of *observational productivity* for logic programming as an algorithmic approximation of global productivity, give an effective procedure for semi-deciding observational productivity, and offer an implemented automated observational productivity checker for logic programs.

Keywords: Logic programming, corecursion, coinduction, termination, productivity.

1 Introduction

Induction is pervasive in programming and program verification. It arises in definitions of finite data (e.g., lists, trees, and other algebraic data types), in program semantics (e.g., of finite iteration and recursion), and proofs (e.g., of properties of finite data and processes). Coinduction, too, is important in these arenas, arising in definitions of infinite data (e.g., lazily defined infinite streams), in program semantics (e.g., of concurrency), and in proofs (e.g., of observational equivalence, or bisimulation, of potentially infinite processes). It is thus desirable to have good support for both induction and coinduction in systems for reasoning about programs.

Given a logic program P and a term A, SLD-resolution provides a mechanism for automatically (and inductively) inferring that $P \vdash A$ holds, i.e., that P logically entails A. The "answer" for a program P and a query $? \leftarrow A$ is a substitution σ computed from P and A by SLD-resolution. Soundness of SLD-resolution ensures that $P \vdash \sigma(A)$ holds, so we also say that P *computes* $\sigma(A)$.

Example 1 (Inductive logic program). The program P_1 codes the Peano numbers:

0. `nat(0)` $\leftarrow$
1. `nat(s(X))` $\leftarrow$ `nat(X)`

To answer the question "*Does* $P_1 \vdash$ `nat(s(X))` *hold?*", we represent it as the logic programming (LP) query ? $\leftarrow$ `nat(s(X))` and resolve it with P_1. It is standard in implementations of traditional LP to use a topmost clause selection strategy, which resolves goals against clauses in the order in which they appear in the program. Topmost clause selection gives the derivation `nat(s(X))` $\rightarrow$ `nat(X)` $\rightarrow$ `true` for P_1 and ? $\leftarrow$ `nat(s(X))`, which computes the answer $\{\mathtt{X} \mapsto \mathtt{0}\}$ in its last step. Since P_1 computes `nat(s(0))`, one answer to our question is "Yes, provided `X` is `0`."

While inductive properties of terminating computations are quite well understood [14], non-terminating LP computations are notoriously difficult to reason about, and can arise even for programs that are intended to be inductive:

Example 2 (Coinductive meaning of inductive logic program). If P_1' is obtained by reversing the order of the clauses in the program P_1 from Example 1, then the SLD-derivation for program P_1' and query ? $\leftarrow$ `nat(s(X))` does not terminate under standard topmost clause selection. Instead, it results in an attempt to compute the "answer" $\{\mathtt{X} \mapsto \mathtt{s(s(...))}\}$ by repeatedly resolving with Clause 1. Nevertheless, P_1' is still computationally meaningful, since it computes the first limit ordinal *at infinity*, in the terminology of [14].

Some programs do not admit terminating computations under *any* selection strategy:

Example 3 (Coinductive logic program). No derivation for the query ? $\leftarrow$ `stream(X)` and the program P_2 comprising the clause

0. `stream(scons(0,Y))` $\leftarrow$ `stream(Y)`

terminates with an answer, be it success or otherwise. Nevertheless, P_2 has computational meaning: it computes the infinite stream of 0s at infinity.

The importance of developing sufficient infrastructure to support coinduction in automated proving has been argued across several communities; see, e.g., [13, 17, 21]. In LP, the ability to work with non-terminating and coinductive programs depends crucially on understanding the structural properties of non-terminating SLD-derivations. To illustrate, consider the non-terminating programs P_3, P_4, and P_5:

Program	Program definition	For query ? $\leftarrow$ `p(X)`, computes the answer:
P_3	`p(X)` $\leftarrow$ `p(X)`	*id*
P_4	`p(X)` $\leftarrow$ `p(f(X))`	*id*
P_5	`p(f(X))` $\leftarrow$ `p(X)`	$\{\mathtt{X} \mapsto \mathtt{f(f...)}\}$

Programs P_3 and P_4 each loop without producing any substitutions at all; only P_5 computes an infinite term at infinity. It is of course not a coincidence that only P_5 resembles a (co)inductive data definition by pattern matching on a constructor, as is commonly used in functional programming.

When an infinite SLD-derivation computes an infinite object, and this object can be successively approximated by applying to the initial query the substitutions computed at each step of the derivation, the derivation is said to be *globally productive.* The only derivation for program P_5 and the query $? \leftarrow \mathtt{p(X)}$ is globally productive since it approximates, in the sense just described, the infinite term $\mathtt{p(f(f...))}$. That is, it computes $\mathtt{p(f(f...))}$ at infinity. Programs P_2 and P_1' similarly give rise to globally productive derivations. But no derivations for P_3 or P_4 are globally productive.

Since global productivity determines which non-terminating logic programs can be seen as defining coinductive data structures, we would like to identify exactly when a program is globally productive. But porting functional programming methods of ensuring productivity by static syntactic checks is hardly possible. Unlike pattern matching in functional programming, SLD-resolution is based on *unification*, which has very different operational properties — including termination and productivity properties — from pattern matching. For example, programs P_1, P_1', P_2, and P_5 are all terminating by term-matching SLD-resolution, i.e., resolution in which unifiers are restricted to matchers, as in term rewriting. We thus call this kind of derivations *rewriting derivations.*

Example 4 (Coinductive program defining an irrational infinite term). The program P_6 comprises the single clause

0. $\mathtt{from(X, scons(X, Y))} \leftarrow \mathtt{from(s(X), Y)}$

For P_6 and the query $? \leftarrow \mathtt{from(0, Y)}$, SLD-resolution computes at infinity the answer substitution $\{\mathtt{Y} \mapsto [\mathtt{0}, \mathtt{s(0)}, \mathtt{s(s(0))}, \ldots]\}$. Here $[t_1, t_2, \ldots]$ abbreviates $\mathtt{scons}(t_1, \mathtt{scons}(t_2, \ldots))$, and similarly in the remainder of this paper. This derivation depends crucially on unification since variables occurring in the two arguments to `from` in the clause head overlap. If we restrict to rewriting, then there are no successful derivations (terminating or non-terminating) for this choice of program and query.

Example 4 shows that any analysis of global productivity must necessarily rely on specific properties of the operational semantics of LP, rather than on program syntax alone. It has been observed in [9, 11] that one way to distinguish globally productive programs operationally is to identify those that admit infinite SLD-derivations, but whose rewriting derivations always terminate. We call this program property *observational productivity.* The programs P_1, P_1', P_2, P_5, P_6 are all observationally productive.

The key observation underlying observational productivity is that terminating rewriting derivations can be viewed as points of finite observation in infinite derivations. Consider again program P_6 and query $? \leftarrow \mathtt{from(0, Y)}$ from Example 4. Drawing rewriting derivations vertically and unification-based resolution steps horizontally, we see that each unification substitution applied to the original query effectively observes a further fragment of the stream computed at infinity:

$$\begin{tikzcd}
\mathtt{from(0, X)} \arrow[r, "{\{\mathtt{X} \mapsto [\mathtt{0}, \mathtt{X'}]\}}"] & \mathtt{from(0, [0, X'])} \arrow[d, dotted] \arrow[r, "{\{\mathtt{X'} \mapsto [\mathtt{s(0)}, \mathtt{X''}]\}}"] & \mathtt{from(0, [0, s(0), X''])} \arrow[d, dotted] \arrow[r, "\cdot"] & \ldots \\
& \mathtt{from(s(0), X')} & \mathtt{from(s(0), [s(0), X''])} \arrow[d, dotted] & \\
& & \mathtt{from(s(s(0)), X'')} &
\end{tikzcd}$$

If we compute unifiers only when rewriting derivations terminate, then the resulting derivations exhibit consumer-producer behaviour: rewriting steps consume structure (here, the constructor `scons`), and unification steps produce more structure (here, new `scons`es) for subsequent rewriting steps to consume. This style of interleaving matching and unification steps was called *structural resolution* (or S-resolution) in [9, 12].

Model-theoretic properties of S-resolution relative to least and greatest Herbrand models of programs were studied in [12]. In this paper, we provide a suitable algorithm for semi-deciding observational productivity of logic programs, and present its implementation [19]; see also Appendix B online. By definition, observational productivity of a program P is in fact a conjunction of two properties of P:

1. *universal observability*: termination of *all* rewriting derivations, and
2. *existential liveness*: existence of *at least one* non-terminating S-resolution or SLD-resolution derivation.

While the former property is universal, the latter must be existential. For example, the program P_1 defining the Peano numbers can have both inductive and coinductive meaning. When determining that a program is observationally productive, we must certify that the program actually *does* admit derivations that produce infinite data, i.e., that it actually *can* be seen as a coinductive definition. Our algorithm for semi-deciding observational productivity therefore combines two checks:

1. *guardedness checks* that semi-decide universal observability: if a program is guarded, then it is universally observable. (The converse is not true in general.)
2. *liveness invariant checks* ensuring that, if a program is guarded and exhibits an invariant in its consumption-production of constructors, then it is existentially live.

This is the first work to develop productivity checks for LP. An alternative approach to coinduction in LP, known as CoLP [7, 21], detects loops in derivations and closes them coinductively. However, loop detection was not intended as a tool for the study of productivity and, indeed, is insufficient for that purpose: programs P_3, P_4 and P_5, of which only the latter is productive, are all treated similarly by CoLP, and all give coinductive proofs via its loop detection mechanism.

Our approach also differs from the usual termination checking algorithms in term-rewriting systems (TRS) [1, 8, 22] and LP [3, 15, 16, 18, 20]. Indeed, these algorithms focus on guaranteeing termination, rather than productivity; see Section 5. And although the notion of productivity has been studied in TRS [4, 5], the actual technical analysis of productivity is rather different there because it considers infinitary properties of rewriting, whereas observational productivity relies on termination of rewriting.

The rest of this paper is organised as follows. In Section 2 we introduce a *contraction ordering* on terms that extends the more common lexicographic ordering, and argue that this extension is needed for our productivity analysis. We also recall that static guardedness checks do not work for LP. In Section 3 we employ contraction orderings in dynamic guardedness checks and present a decidable property, called $GC2$, that characterises guardedness of a single rewriting derivation, and thus certifies existential observability. In Section 4 we employ $GC2$ to develop an algorithm, called $GC3$, that analyses *consumer-producer* invariants of S-resolution derivations to certify universal observability. For universally observable programs, these invariants also serve as liveness invariant checks. We also prove that $GC3$ indeed semi-decides observational productivity. In Section 5 we discuss related work and in Section 6 we discuss our implementation and applications of the productivity checker. In Section 7 we conclude the paper.

2 Contraction Orderings on Terms

In this section, we will introduce the contraction ordering on first-order terms, on which our productivity checks will rely. We work with the standard definition of first-order logic programs. A *signature* Σ consists of a set $\mathcal{F}$ of function symbols $f, g, \ldots$ each equipped with an arity. Nullary (0-ary) function symbols are constants. We also assume a countable set *Var* of variables, and a set $\mathcal{P}$ of predicate symbols each equipped with an arity. We have the following standard definition for terms, formulas and Horn clauses:

Definition 1 (Syntax of Horn clauses and programs).

Terms $Term ::= Var \mid \mathcal{F}(Term, ..., Term)$

Atoms $At ::= \mathcal{P}(Term, ..., Term)$

(Horn) clauses $CH ::= At \leftarrow At, ..., At$

Logic programs $Prog ::= CH, ..., CH$

In what follows, we will use letters A, B with subscripts to refer to elements of At. Given a program P, we assume all clauses are indexed by natural numbers starting from 0. When we need to refer to ith clause of program P, we will use notation $P(i)$. To refer to the head of clause $P(i)$, we will use notation $head(P(i))$.

A *substitution* is a total function $\sigma : Var \rightarrow Term$. Substitutions are extended from variables to terms as usual: if $t \in Term$ and σ is a substitution, then the *application* $\sigma(t)$ is a result of applying σ to all variables in t. A substitution σ is a *unifier* for t, u if $\sigma(t) = \sigma(u)$, and is a *matcher* for t against u if $\sigma(t) = u$. A substitution σ is a *most general unifier* (*mgu*) for t and u if it is a unifier for t and u and is more general than any other such unifier. A *most general matcher* (*mgm*) σ for t against u is defined analogously.

We can view every term and atom as a tree. Following standard definitions [2, 14], such trees can be indexed by elements of a suitably defined tree language. Let $\mathbb{N}^*$ be the set of all finite words (i.e., sequences) over the set $\mathbb{N}$ of natural numbers. A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language* if the following two conditions hold: (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 tree language L is *finite* if it is a finite subset of $\mathbb{N}^*$, and *infinite* otherwise. Term trees (for terms and atoms over a given signature) are defined as mappings from a tree language L to that signature, see [2, 9, 14]. Informally speaking, every symbol occurring in a term or an atom receives an index from L.

In what follows, we will work with term tree representations of terms and atoms, and for brevity we will refer to term trees simply as *terms*. We will use notation $t(w)$ when we need to talk about the element of the term t indexed by a word $w \in L$. Note that leaf nodes are always given by variables or constants.

Example 5. Given $L = \{\epsilon, 0, 00, 01\}$, the atom $\mathtt{stream(scons(0,Y))}$ can be seen as the term tree t given by the map $t(\epsilon) = \mathtt{stream}$, $t(0) = \mathtt{scons}$, $t(00) = \mathtt{0}$, $t(01) = \mathtt{Y}$.

We can use such indexing to refer to subterms, and notation $subterm(t, w)$ will refer to the subterm of term t starting at node w. In Example 5, taking $t = \mathtt{stream(scons(0,Y))}$ gives that $subterm(t, 0)$ is $\mathtt{scons(0,Y)}$.

Two of the most popular tools for termination analysis of declarative programs are lexicographic ordering and (recursive) path ordering of terms. Informally, these can be adopted to the LP setting as follows. Suppose we have a clause $A \leftarrow B_1, \ldots, B_i, \ldots, B_n$, and want to check whether each B_i sharing the predicate with A is "smaller" than A, since this guarantees that no infinite rewriting derivation is triggered by this clause. For lexicographic ordering we will write $B_i <_l A$ and for path ordering we will write $B_i <_p A$.

Using standard orderings to prove universal observability works well for program P_2, since $\mathtt{stream(Y)} <_l \mathtt{stream(scons(0,Y))}$ and $\mathtt{stream(Y)} <_p \mathtt{stream(scons(0,Y))}$, and so any rewriting derivation for P_2 terminates. But universal observability of P_6 cannot be shown by this method. Indeed, none of the four orderings $\texttt{from(X, scons(X, Y))} <_l \texttt{from(s(X), Y)}$, $\texttt{from(s(X), Y)} <_l \texttt{from(X, scons(X, Y))}$, $\texttt{from(X, scons(X, Y))} <_p \texttt{from(s(X), Y)}$, and $\texttt{from(s(X), Y)} <_p \texttt{from(X, scons(X, Y))}$ holds because the subterms pairwise disagree on the ordering. This situation is common for LP, where some arguments hold input data and some hold output data, so that some decrease while others increase in recursive calls. Nevertheless, P_6 *is* universally observable, and we want to be able to infer this. Studying the S-resolution derivation for P_6 in Section 1, we note that universal observability of P_6 is guaranteed by contraction of `from`'s second argument. It is therefore sufficient to establish that terms get smaller in only one argument. This inspires our definition of a *contraction ordering*, which takes advantage of the tree representation of terms.

Definition 2 (Contraction, recursive contraction). *If t_1 and t_2 are terms, then t_2 is a* contraction *of t_1 (written $t_1 \rhd t_2$) if there is a leaf node $t_2(w)$ on a branch B in t_2, and there exists a branch B' in t_1 that is identical to B up to node w, but $t_1(w)$ is not a leaf. If, in addition, subterm(t_1, w) contains the symbol given by $t_2(w)$, then t_2 is a* recursive contraction *of t_1.*

We distinguish variable contractions *and* constant contractions *according as $t_2(w)$ is a variable or constant, and call subterm(t_1, w) a* reducing subterm *for $t_1 \rhd t_2$ at node w. We call subterm(t_1, w) a* recursive*,* variable*, or* constant reducing subterm *according as $t_1 \rhd t_2$ is a recursive, variable or constant contraction.*

Example 6 (Contraction orderings). We have $\texttt{from(X, scons(X, Y))} \rhd \texttt{from(s(X), Y)}$, since the leaf `Y` in the latter is "replaced" by the term `scons(X, Y)` in the former. Formally, `scons(X, Y)` is a recursive and variable reducing subterm. It can be used to certify termination of all rewriting derivations for P_6. Note that $\texttt{from(s(X), Y)} \rhd \texttt{from(X, scons(X, Y))}$ also holds, with (recursive and variable) reducing subterm `s(X)`.

The fact that $\rhd$ is not well-founded makes reasoning about termination delicate. Nevertheless, contractions emerge as precisely the additional ingredient needed to formulate our productivity check for guarded (and therefore universally observable) programs.

In general, static termination checking for LP suffers serious limitations; see, e.g., [3]. The following example illustrates this phenomenon.

Example 7 (Contraction ordering on clause terms is insufficient for termination checks). The program P_7, which is not universally observable, is given by mutual recursion:

0. $\texttt{p(s(X1), X2, Y1, Y2)} \leftarrow \texttt{q(X2, X2, Y1, Y2)}$
1. $\texttt{q(X1, X2, s(Y1), Y2)} \leftarrow \texttt{p(X1, X2, Y2, Y2)}$

No two terms from the same clause of P_7 can be related by any contraction ordering because their head symbols differ. But recursion arises for P_7 when a derivation calls its two clauses alternately, so we would like to examine rewriting derivations for queries, such as $? \leftarrow \texttt{p(s(X1), X2, s(Y1), Y2)}$ and $? \leftarrow \texttt{p(s(X1), s(X2), s(Y1), s(Y2))}$, that exhibit its recursive nature. Unfortunately, such queries are not given directly by P_7's syntax, and so are not available for static program analysis.

Since static checking for contraction ordering in clauses is not sufficient, we will define dynamic checks in the next section. The idea is to build a rewriting tree for each clause, and check whether the term trees featured in that derivation tree satisfy any contraction ordering.

3 Rewriting Trees: Guardedness Checks for Rewriting Derivations

To properly reason about rewriting derivations in LP, we need to take into account that i) in LP, unlike, e.g., in TRS, we have conjuncts of terms in the bodies of clauses, and ii) a logic program can have overlapping clauses, i.e., clauses whose heads unify. These two facts have been analysed in detail in the LP literature, usually using the notion of and-or-trees and, where optimisation has been concerned, and-or-parallel trees. We carry on this tradition and consider a variant of and-or trees for derivations. However, the trees we consider are not formed by general SLD-resolution, but rather by term matching resolution. *Rewriting trees* are so named because each of their edges represents a term matching resolution step, i.e., a matching step as in term rewriting.

Definition 3 (Rewriting tree). *Let P be a logic program with n clauses, and A be an atom. The* rewriting tree *for P and A is the possibly infinite tree T satisfying the following properties:*

- *A is the root of T*
- *Each node in T is either an and-node or an or-node*
- *Each or-node is given by $P(i)$, for some $i \in \{0, \ldots, n\}$*
- *Each and-node is an atom seen as a term tree.*
- *For every and-node A' occurring in T, if there exist exactly $k > 0$ distinct clauses $P(j_1), \ldots, P(j_k)$ in P (a clause $P(j_i)$ has the form $B_{j_i} \leftarrow B_1^{j_i}, \ldots, B_{n_{j_i}}^{j_i}$ for some n_{j_i}), such that $A' = \theta_{j_1}(B_{j_1}) = \ldots = \theta_{j_k}(B_{j_k})$, for mgms $\theta_{j_1}, \ldots, \theta_{j_k}$, then A' has exactly k children given by or-nodes $P(j_1), \ldots, P(j_k)$, such that every or-node $P(j_i)$ has n_{j_i} children given by and-nodes $\theta_{j_i}(B_1^{j_i}), \ldots, \theta_{j_i}(B_{n_{j_i}}^{j_i})$.*

When constructing rewriting trees, we assume a suitable algorithm [9] for renaming free variables in clause bodies apart. Figure 1 gives four examples of rewriting trees. If P is a program and $t_1, \ldots, t_m$ are terms, then *a rewriting reduction* is given by $[t_1, \ldots, t_i, \ldots, t_m] \rightarrow [t_1, \ldots, t_{i-1}, \sigma(B_0), \ldots \sigma(B_n), t_{i+1}, \ldots, t_m]$ for $B \leftarrow B_1, \ldots, B_n \in P$ and $\sigma(B) = t_i$. A sequence of rewriting reductions is a *rewriting derivation.* It is easy to see that a rewriting derivation for a term t corresponds to a subtree of a rewriting tree for t in which only one or-node is taken at every tree level.

Because mgms are unique up to variable renaming, given a program P and an atom A, the rewriting tree T for P and A is unique. Following the same principle as with definition of term trees, we use suitably defined finitely-branching tree languages for indexing rewriting trees; see [9] for precise definitions. When we need to talk about a node of a rewriting tree T indexed by a word $w \in L$, we will use notation $T(w)$.

We can now formally define our notion of universal observability.

Definition 4 (Universal observability). *A program P is* universally observable *if, for every atom A, the rewriting tree for A and P is finite.*

Programs P_1, P_1', P_2, P_5, P_6 are universally observable, whereas programs P_3, P_4 and P_7 are not. An exact analysis of why P_7 is not universally observable is given in Example 9.

We can now apply the contraction ordering we defined in the previous section to analyse termination properties of rewriting trees. A suitable notion of guardedness can be defined by checking for loops in rewriting trees whose terms fail to decrease by any contraction ordering. Note that our notion of a loop is more general than that used in CoLP [7, 21] since it does not require the looping terms to be unifiable.

Definition 5 (Loop in a rewriting tree). *Given a program P and an atom A, the rewriting tree T for P and A* contains a loop *at nodes w and v, denoted $loop(T, w, v)$, if w properly precedes v on some branch of T, $T(w)$ and $T(v)$ are and-nodes whose atoms have the same predicate, and the parent or-nodes of $T(w)$ and $T(v)$ are given by the same clause $P(i)$.*

Examples of loops in rewriting trees are given (underlined) in Figure 1.

If T has a loop at nodes w and v, and if t is a recursive reducing subterm for $T(w) \rhd T(v)$, then $loop(T, w, v)$ is *guarded* by $(P(i), t)$, where $P(i)$ is the clause that was resolved against to obtain $T(w)$ and $T(v)$, i.e., $P(i)$ is the parent node of $T(w)$ and $T(v)$. It is *unguarded* otherwise. A rewriting tree T is *guarded* if all of its loops are guarded, and is *unguarded* otherwise. We write $GC2(T)$ when T is guarded, and say that holds for T, or simply that $GC2(T)$ holds.

Example 8. In Figure 1, we have (underlined) loops in the third rewriting tree (for $\mathtt{q(s(X''), s(X''), s(Y'), Y'')}$ and $\mathtt{q(s(X'), s(X''), Y'', Y'')}$) and the fourth rewriting tree (for $\mathtt{q(s(X''), s(X''), s(Y'), s(Y''))}$ and $\mathtt{q(s(X''), s(X''), s(Y''), s(Y''))}$). Neither is guarded. In the former, there is a contraction on the third argument, but because $\mathtt{s(Y')}$ and $\mathtt{Y''}$ do not share a variable, it is not recursive contraction. In the latter, there is no contraction at all.

By Definition 5, each repetition of a clause and predicate in a branch of a rewriting tree triggers a check to see if the loop is guarded by some recursive reducing subterm.

Proposition 1 ($GC2$ is decidable). *GC2 is a decidable property of rewriting trees.*[4]

The proof of Proposition 1 also establishes that every guarded rewriting tree is finite.

The decidable guardedness property $GC2$ is a property of individual rewriting trees. But our goal is to decide guardedness universally, i.e., for *all* of a program's rewriting trees. The next example shows that extrapolating from existential to universal guardedness is a difficult task.

Example 9 (Existential guardedness does not imply universal guardedness). For program P_7, the rewriting trees constructed for the two clause heads $\mathtt{p(s(X'), X'', Y', Y'')}$ and $\mathtt{q(s(X'), X'', s(Y'), Y'')}$ are both guarded since neither contains any loops at all. Nevertheless, there is a rewriting tree for P_7 (the last tree in Figure 1) that is unguarded and infinite. The third tree is not guarded (due to the unguarded loop), but it is finite.

The example above shows that checking rewriting trees generated by clause heads is insufficient to detect all cases of nonterminating rewriting. Since a similar situation can

[4] All proofs are in Appendix A, and corresponding pseudocode algorithms are in Appendix B, of the version of the paper at `https://arxiv.org/abs/1608.04415`.

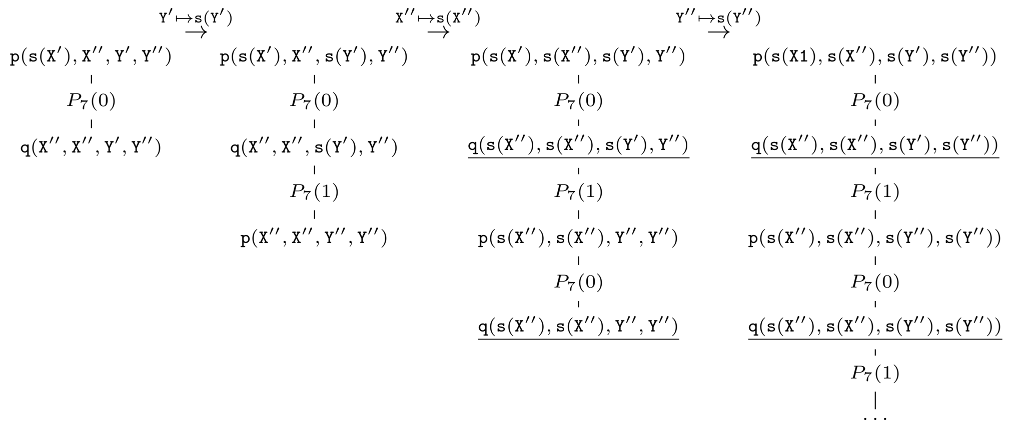
Y′↦s(Y′)
X′′↦s(X′′)
Y′′↦s(Y′′)
p(s(X′), X′′, Y′, Y′′)
P7(0)
q(X′′, X′′, Y′, Y′′)
p(s(X′), X′′, s(Y′), Y′′)
P7(0)
q(X′′, X′′, s(Y′), Y′′)
P7(1)
p(X′′, X′′, Y′′, Y′′)
p(s(X′), s(X′′), s(Y′), Y′′)
P7(0)
q(s(X′′), s(X′′), s(Y′), Y′′)
P7(1)
p(s(X′′), s(X′′), Y′′, Y′′)
P7(0)
q(s(X′′), s(X′′), Y′′, Y′′)
p(s(X1), s(X′′), s(Y′), s(Y′′))
P7(0)
q(s(X′′), s(X′′), s(Y′), s(Y′′))
P7(1)
p(s(X′′), s(X′′), s(Y′′), s(Y′′))
P7(0)
q(s(X′′), s(X′′), s(Y′′), s(Y′′))
P7(1)
...

Fig. 1. An initial fragment of the derivation tree (comprising four rewriting trees) for the program P_7 of Example 7 and the atom p(s(X′), X′′, Y′, Y′′). Its third and fourth rewriting trees each contain an unguarded loop (underlined), so both are unguarded. The fourth tree is infinite.

obtain for any finite set of rewriting trees, we see that universal observability, and hence observational productivity, of programs cannot be determined by guardedness of rewriting trees for program clauses alone. The next section addresses this problem.

4 Derivation Trees: Observational Productivity Checks

The key idea of this section is, given a program P, to identify a finite set S of rewriting trees for P such that checking guardedness of all rewriting trees in S is sufficient to guarantee guardedness of *all* rewriting trees for P. One way to identify such sets is to use the strategy of Example 9 and Figure 1: for every clause $P(i)$ of P, construct a rewriting tree for the head of $P(i)$, and, if that tree is guarded, explore what kind of mgus the leaves of that tree generate and see if applications of those mgus might give an unguarded tree. As Figure 1 shows, we may need to apply this method iteratively until we find a nonguarded rewriting tree. But we want the number of such iterations to be finite. This section shows how to do precisely this.

We start with a formal definition of rewriting tree transitions, which we have seen already in Figure 1, and see also in Figure 2 below.

Definition 6 (Rewriting tree transition). *Let P be a program and T be a rewriting tree for P and an atom A. If $T(w)$ is a leaf node of T given by an atom B, and B unifies with a clause $P(i)$ via mgu σ, we define a tree T_w as follows: we apply σ to every and-node of T, and extend the branches where required, according to Definition 3. Computation of T_w from T is denoted $T \to T_w$. The operation $T \to T_w$ is the* tree transition *for T and w.*

If a rewriting tree T is constructed for a program P and an atom A, a (finite or infinite) sequence $T \to T' \to T'' \to \ldots$ of tree transitions is an *S-resolution derivation* for P and A. For a given rewriting tree T, several different S-resolution derivations are possible from T. This gives rise to the notion of a derivation tree.

Definition 7 (Derivation tree, guarded derivation tree). *Given a logic program P and an atom A, the* derivation tree *D for P and A is defined as follows:*

– *The root of D is given by the rewriting tree for P and A.*
– *For a rewriting tree T occurring as a node of D, if there exists a transition $T \to T_w$, for some leaf node w in T, then the node T has a child given by T_w.*

A derivation tree is guarded *if each of its nodes is a guarded rewriting tree, i.e., if $GC2(T)$ holds for each of its nodes T.*

Figure 1 shows an initial fragment of the derivation tree for P_7 and $\mathtt{p(s(X'), X'', Y', Y'')}$.

Note that we now have three kinds of trees: term trees have signature symbols as nodes, rewriting trees have atoms (term trees) as nodes, and derivation trees have rewriting trees as nodes. For a given P and A, the derivation tree for P and A is unique up to renaming. We use our usual notation $D(w)$ to refer to the node of D at index $w \in L$.

Definition 8 (Existential liveness, observational productivity). *Let P be a universally observable program and let A be an atom. An S-resolution derivation for P and A is* live *if it constitutes an infinite branch of the derivation tree for P and A. The program P is* existentially live *if there exists a live S-resolution derivation for P and some atom A. P is* observationally productive *if it is universally observable and existentially live.*

To show that observational productivity is semi-decidable, we first show that universal observability is semi-decidable by means of a finite (i.e., decidable) guardedness check. We started this section by motivating the need to construct a finite set S of rewriting trees whose guardedness will guarantee guardedness for *any* rewriting tree for the given program. Our first logical step is to use derivation trees built for clause heads as generators of such a set S. Due to the properties of mgus used in forming branches of derivation trees, derivation trees constructed for clause heads generate the set of *most general* rewriting trees. The next lemma exposes this fact:

Lemma 1 (Guardedness of derivation trees implies universal observability). *Given a program P, if derivation trees for P and each $head(P(i))$ are guarded, then P is universally observable.*

Since derivation trees are infinite, in general, checking guardedness of *all* loops in *all* of their rewriting trees is not always feasible. It thus remains to define a method that extracts representative finite subtrees from such derivation trees; we call such subtrees *observation subtrees*. For this, we need only be able to detect an invariant property guaranteeing guardedness through tree transitions in the given derivation tree. To illustrate, let us check guardedness of the program P_6. Since it consists of just one clause, we take the head of that clause as the goal atom, and start constructing the infinite derivation tree D for P_6 and $\mathtt{from(X, scons(X, Y))}$ as shown in Figure 2. The first rewriting tree in the derivation tree has no loops, so we cannot identify any invariants. We make a transition to the second rewriting tree which has one loop (underlined) involving the recursive reducing subterm $\mathtt{[s(X), Y']}$. This reducing subterm is our first candidate invariant, since it is the pattern that is *consumed* from the root of the second rewriting tree to its leaf. We now need to check this pattern is added back, or *produced*, in the next tree transition. The next mgu involves substitution $\mathtt{Y'} \mapsto \mathtt{[s(s(X)), Y'']}$. Because this derivation gradually computes an infinite irrational term (rational terms are terms that can be represented as trees that have a finite number of distinct subtrees), the two terms

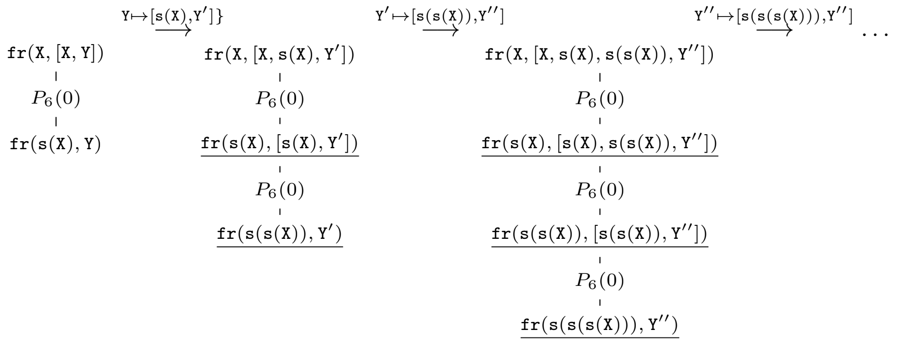

Fig. 2. An initial fragment of the infinite derivation tree D for the program P_6 from Example 4 and its clause head. It is also the observation subtree for D. We abbreviate `scons` by $[,]$, and `from` by `fr`. The guarded loops in each of D's rewriting trees are underlined.

$[\texttt{s}(\texttt{X}), \texttt{Y}']$ and $[\texttt{s}(\texttt{s}(\texttt{X})), \texttt{Y}'']$ we have identified are not unifiable. We need to be able to abstract away from their current shape and identify a common pattern, which in this case is $[\_, \_]$. Importantly, by the properties of mgus used in transitions, such most general patterns can always be extracted from clause heads themselves. Indeed, the subterm of the clause head $\texttt{from}(\texttt{X}, \texttt{scons}(\texttt{X}, \texttt{Y}))$ has the subterm $[\texttt{X}, \texttt{Y}]$ that is exactly the pattern we are looking for. Thus, our current *(coinductive) assumption* is: *given a rewriting tree T in the derivation tree D, a term of the form $[\_, \_]$ will be* consumed *by rewriting steps from its root to its leaves, and exactly a term of the form $[\_, \_]$ will be* produced *(i.e., added back) in the next tree transition.* Consumption is always finite (by the loop guardedness), and production is potentially infinite.

We now need to check that this coinductive assumption will hold for the next rewriting tree of D. The third rewriting tree indeed has guarded loops with recursive reducing subterm $[\texttt{s}(\texttt{s}(\texttt{X})), \texttt{Y}'']$, and the next mgu it gives rise to is $\texttt{Y}'' \mapsto [\texttt{s}(\texttt{s}(\texttt{s}(\texttt{X}))), \texttt{Y}'']$. Again, to abstract away the common pattern, we look for a subterm in the clause head of $P_6(0)$ that matches with both of these terms, it is the same subterm $[\texttt{X}, \texttt{Y}]$. Thus, our coinductive assumption holds again, and we conclude by coinduction that the same pattern will hold for any further rewriting tree in D. When implementing this reasoning, we take the *observation subtree* of D up to the third tree shown in Figure 2 as a sufficient set of rewriting trees to use to check guardedness of the (otherwise infinite) tree D.

The rest of this section generalises and formalises this approach. In the next definition, we introduce the notion of a *clause projection* to talk about the process of "abstracting away" a pattern from an mgu σ by matching it with a subterm t of a clause head. When t also matches with a recursive reducing subterm of a loop in a rewriting tree, we call t a *coinductive invariant.*

Definition 9 (Clause projection and coinductive invariant). *Let P be a program and A be an atom, and let D be a derivation tree for P and A in which a tree transition from T to T' is induced by an mgu σ of $P(k)$ and an atom B given by a leaf node $T(u)$.*

The clause projection *for T', denoted $\pi(T')$, is the set of all triples $(P(k), t, v)$, where t is a subterm of head$(P(k))$ at position v, such that the following conditions hold: $\sigma(B) \rhd B$ with variable reducing subterm t', and t' matches against t (i.e. $t' = \sigma'(t)$ for some σ').*

Additionally, the coinductive invariant *at T', denoted* $\mathsf{ci}(T')$*, is a subset of the* clause projection *for T' satisfying the following condition: an element $(P(k), t, v) \in \pi(T')$ is in* $\mathsf{ci}(T')$ *if T contains a loop in the branch leading from T's root to $T(u)$ that is guarded by $(P(k), t'')$ for some t'' such that t'' matches against t (i.e., $t'' = \theta(t)$ for some θ).*

Given a program P, an atom A, and a derivation tree D for P and A, the clause projection set *for D is* $\mathsf{cproj}(D) = \bigcup_T \pi(T)$*, and the* coinductive invariant set *for D is* $\mathsf{cinv}(D) = \bigcup_T \mathsf{ci}(T)$*, where these unions are taken over all rewriting trees T in D.*

Example 10 (Clause projections and coinductive invariants). Coming back to Figure 2, the mgu for the first transition is $\sigma_1 = \{\mathtt{X}' \mapsto \mathtt{s(X)}, \mathtt{Y} \mapsto \mathtt{scons(s(X), Y')}\}$ (renaming of variables in $P_6(0)$ with primes), that for the second is $\sigma_2 = \{\mathtt{X}'' \mapsto \mathtt{s(s(X))}, \mathtt{Y}' \mapsto \mathtt{scons(s(s(X)), Y'')}\}$ (renaming of variables in $P_6(0)$ with double primes), etc. Clause projections are given by $\pi(T) = \{(P_6(0), \mathtt{scons(X, Y)}, 1)\}$ for all trees T in this derivation, and thus $\mathsf{cproj}(D)$ is the finite set. Moreover, for the first rewriting tree T, $\mathsf{ci}(T) = \emptyset$, and $\mathsf{ci}(T') = \{(P_6(0),\ \mathtt{scons(X, Y)}, 1)\}$ for all trees T' except for the first one, so $\mathsf{cinv}(D) = \{(P_6(0), \mathtt{scons(X, Y)}, 1)\}$ is a finite set, too.

The clause projections for the derivation in Figure 1 are $\pi(T') = \pi(T''') = (P(1), \mathtt{s(Y1)}, 2)$, and $\pi(T'') = (P(0), \mathtt{s(X1)}, 0)$, where T', T'', T''' refer to the second, third and fourth rewriting tree of that derivation. All coinductive invariants for that derivation are empty, since none of these rewriting trees contain guarded loops.

Generally, clause projection sets are finite, since the number of subterms in the clause heads of P is finite. This property is crucial for termination of our method.

Proposition 2 (Finiteness of clause projection sets). *Given a program P, an atom A, and a derivation tree D for P and A, the clause projection set* $\mathsf{cproj}(D)$ *is finite.*

In particular, this holds for derivation trees induced by clause heads.

We terminate the construction of each branch of a derivation tree when we notice a repeating coinductive invariant. A subtree we get as a result is an observation subtree. Formally, given a derivation tree D for a program P and an atom A with a branch in which nodes $D(w)$ and $D(wv)$ are defined, if $\mathsf{ci}(D(w)) = \mathsf{ci}(D(wv)) \neq \emptyset$, then D has a *guarded transition* from $D(w)$ to $D(wv)$, denoted $D(w) \Longrightarrow D(wv)$. Every guarded transition thus identifies a repeated "consumer-producer" invariant in the derivation from $D(w)$ to $D(wv)$. This tells us that observation of this branch of D can be concluded. Imposing this condition on all branches of D gives us a general method to construct finite observation subtrees of potentially infinite derivation trees:

Definition 10 (Observation subtree of a derivation tree). *If D is a derivation tree for a program P and an atom A, the tree D' is the* observation subtree *of D if*

1) the roots of D and D' are given by the rewriting tree for P and A, and

2) if w is a node in both D and D', then the rewriting trees in D and D' at node w are the same and, for every child w' of w in D, the rewriting tree of D' at node w' exists and is the same as the rewriting tree of D at w', unless either

a) GC2 does not hold for $D(w')$, or
b) there exists a v such that $D(v) \Longrightarrow D(w)$.

In either case, $D'(w)$ is a leaf node. We say that D' is unguarded *if Condition 2a holds for at least one of D's nodes, and that D' is* guarded *otherwise.*

A branch in an observation subtree is thus truncated when it reaches an unguarded rewriting tree or its coinductive invariant repeats. The observation subtree of any derivation tree is unique. The following proposition and lemma prove the two most crucial properties of observation subtrees: that they are always finite, and that checking their guardedness is sufficient for establishing guardedness of whole derivation trees.

Proposition 3 (Finiteness of observation subtrees). *If D is a derivation tree for a program P and an atom A, then the observation subtree of D is finite.*

Lemma 2 (Guardedness of observation subtree implies guardedness of derivation tree). *If the observation subtree for a derivation tree D is guarded, then D is guarded.*

Example 11 (Finite observation subtree of an infinite derivation tree). The initial fragment D' of the infinite derivation tree D given by the three rewriting trees in Figure 2 is D's observation subtree. The third rewriting tree T'' in D is the last node in the observation tree D' because $\mathsf{ci}(T') = \mathsf{ci}(T'') = \{(P_6(0), \mathtt{scons}(\mathtt{X}, \mathtt{Y}), 1)\} \neq \emptyset$. Since D' is guarded, Lemma 2 above ensures that the whole infinite derivation tree D is guarded.

It now only remains to put the properties of the observation subtrees to practical use and, given a program P, to construct finite observation subtrees for each of its clauses. If none of these observation subtrees detects unguarded rewriting trees, we have guarantees that this program will never give rise to infinite rewriting trees. The next definition, lemmas, and theorem make this intuition precise.

Definition 11 (Guarded clause, guarded program). *Given a program P, its clause $P(i)$ is* guarded *if the observation subtree for the derivation tree for P and the atom $head(P(i))$ is guarded, and $P(i)$ is* unguarded *otherwise. A program P is* guarded *if each of its clauses $P(i)$ is guarded, and* unguarded *otherwise. We write $GC3(P(i))$ to indicate that $P(i)$ is guarded, and similarly for P.*

Lemma 3 uses Proposition 3 to show that $GC3$ is decidable.

Lemma 3 (GC3 is decidable). *$GC3$ is a decidable property of logic programs.*

Theorem 1 (Universal observability is semi-decidable). *If $GC3(P)$ holds, then P is universally observable.*

PROOF: If $GC3(P)$ holds, then the observation subtree for each $P(i)$ is guarded. Thus, by Lemma 2, the derivation tree for each $P(i)$ is guarded. But then, by Lemma 1, P is universally observable. Combining this with Lemma 3, we also obtain that universal observability is semi-decidable.

The converse of Theorem 1 does not hold: the program comprising the clause $\mathtt{p(a)} \leftarrow \mathtt{p(X)}$ is universally observable but not guarded, hence the above *semi*-decidability result.

From our check for universal observability we obtain the desired check for existential liveness, and thus for observational productivity:

Corollary 1 (Observational productivity is semi-decidable). *Let P be a guarded logic program. If there exists a clause $P(i)$ such that the derivation tree D for P and $P(i)$ has an observation subtree D' one of whose branches was truncated by Condition 2b of Definition 10, then P is existentially live. In this case, since P is also guarded and hence universally observable, P is observationally productive.*

5 Related Work: Termination Checking in TRS and LP

Because observational productivity is a combination of universal observability and existential liveness, and the former property amounts to termination of all rewriting trees, there is an intersection between this work and termination checking in TRS [1, 8, 22].

Termination checking via the transformation of LP into TRS has been studied in [20]. Here we consider termination of restricted form of SLD-resolution (given by rewriting derivations), and so a much simpler method for translating LP into TRS can be used for our purposes [6]: Given a logic program P and a clause $P(i) = A \leftarrow B_1, \ldots, B_n$ containing no existential variables, we define a rewrite rule $A \to f_i(B_1, \ldots, B_n)$ for some fresh function symbol f_i. Performing this translation for all clauses, we get a translation from P to a term rewriting system $\mathcal{T}_P$. Rewriting derivations for P can be shown operationally equivalent to term rewriting reductions for $\mathcal{T}_P$; see [6] for a proof. Therefore, for logic programs containing no existential variables, any termination method from TRS may be applied to check universal observability (but not existential liveness).

Algorithmically, our guardedness check compares directly with the method of dependency pairs due to Arts and Giesl [1, 8]. Consider again the TRS $\mathcal{T}_P$ obtained from a logic program P. The set R of dependency pairs contains, for each rewrite rule $A \to f_i(B_1, \ldots, B_n)$ in $\mathcal{T}_P$, a pair (A, B_j), $j = 1, \ldots, n$; see [6]. The method of dependency pairs consists of checking whether there exists an infinite chain of dependency pairs $(s_i, t_i)_{i=1,2,3,\ldots}$ such that $\sigma_i(t_i) \to^* \sigma_{i+1}(s_{i+1})$. If there is no such infinite chain, then $\mathcal{T}_P$ is terminating. Again this translation from LP to dependency pairs in TRS is simpler than in [15], since rewriting derivations are a restricted form of SLD-resolution. Due to the restricted syntax of $\mathcal{T}_P$ (compared to the general TRS syntax), generating the set of dependency pairs is equivalent to generating a set of rewriting trees for each clause of P and assuming $\sigma_i = \sigma_{i+1}$ (cf. $GC2$). To find infinite chains, a dependency graph is defined, in which dependency pairs are nodes, and arcs are defined whenever a substitution that allows a transition from one pair to another can be found. Finding such substitutions is the hardest part algorithmically. Note that every pair of neighbouring and-nodes in a rewriting tree corresponds to a node in a dependency graph. Generating arcs in a dependency graph is equivalent to using $GC3$ to find a representative set of substitutions. However, the way $GC3$ generates such substitutions via rewriting tree transitions differs completely from the methods approximating dependency graphs [1, 22], and relies on the properties of S-resolution, rather than recursive path orderings. This is because $GC3$ additionally generates coinductive invariants for checking existential liveness of programs.

Conceptually, observational productivity is a new property that does not amount to either termination or nontermination in LP or TRS. For instance, programs P_3 and P_4 are nonterminating (seen as LP or TRS), and $P_8 : p(X) \leftarrow q(Y)$ is terminating (seen as LP or TRS) but none of them is productive. This is why the existing powerful tools (such as AProVE) and methods [1, 8, 15, 20] that can check termination or nontermination in TRS or LP are not sufficient to serve as productivity checks. To check *termination* of rewriting trees, $GC3$ can be substituted by existing termination checkers for TRS, but none of the previous approaches can semi-decide existential liveness as $GC3$ does.

6 Implementation and Applications

We implemented the observational productivity checker in parallel Go (golang.org) [19], which allows experimentation with parallelisation of proof search [10]. Loading a logic program P, one runs a command line to initialise the $GC3$ check. The algorithm then certifies whether or not the program is guarded (and hence universally observable). If that is the case, it also checks whether $GC3$ found valid coinductive invariants, i.e. whether P is existentially live, and hence admits coinductive interpretations for some predicates. Appendix B (available online) gives further details.

In the context of S-resolution [9, 11], observational productivity of a program is a precondition for (coinductive) soundness of S-resolution derivations. This gives the first application for the productivity checker. But the notion of global productivity (as related to *computations at infinity* [14]) was first investigated in the 1980s. A program is productive, if it admits SLD- (or S-resolution) derivations that compute (or produce) an infinite term at infinity. Thus the productivity checker has more general practical significance for Prolog. In this paper we further exposed its generality by showing that productivity can be seen as a general property of logic programs, rather than property of derivations in some special dialect of Prolog.

Based on this observation, we identify three applications for productivity checks encompassing the S-resolution framework. First, in the context of CoLP [7, 21] or any other similar tool based on loop detection in SLD-derivations, one can run the observational productivity checker for a given program prior to running the usual interpreter of CoLP. If the program is certified as productive, all computations by CoLP for this program will be sound relative to computations at infinity [14]. It gives a way to characterise a subset of theorems proven by CoLP that describe the process of *production of infinite data*. For example, as explained in the introduction, CoLP will return answers for programs P_3, P_4 and P_5. But if we know that only P_5 is productive, then we also know that only CoLP's answers for P_5 will correspond to production of infinite terms at infinity. Secondly, since our productivity checker also checks *liveness* of programs, it effectively identifies which predicates may be given coinductive semantics. This knowledge can be used to type predicates as inductive or coinductive. We can use these types to mark predicates in CoLP or any other coinductive dialect of logic programming, cf. Appendix B. Finally, observational productivity is also a guarantee that a sequence of mgus approximating the infinite answer can be constructed *lazily* even if the answer is irregular. For instance, our running example of program P_6 defines an irrational term and hence cannot be handled by CoLP's loop detection. But even if we cannot form a closed-term answer for a query `from(0, X)`, the productivity checker gives us a weaker but more general certificate that lazy approximation of our infinite answer is possible.

These three classes of applications show that the presented productivity checker can be implemented and applied in any dialect of logic programming, irrespective of the fact that it initially arose from S-resolution research [9, 11].

7 Conclusions

In this paper we have introduced an observational counterpart to the classical notion of global productivity of logic programs. Using the recently introduced formalism of S-resolution, we have defined observational productivity as a combination of two program properties, namely, universal observability and existential liveness. We have introduced an algorithm for semi-deciding observational productivity for any logic program. We did not impose any restrictions on the syntax of logic programs. In particular, our algorithm handles both existential variables and non-linear recursion.

The algorithm relies on the observation that rewriting trees for productive and guarded programs must show term reduction relative to a contraction ordering from their roots to their leaves. But S-resolution derivations involving such trees can only proceed by adding term structure back in transitioning to new rewriting trees via mgus. This "producer/consumer" interaction can be formally traced by observing a derivation's coinductive invariants: these record exactly the term patterns that both reduce in the loops of rewriting trees and are added back in transitions between these trees.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(12):133 – 178, 2000.
2. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
3. D de Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19–20, Supplement 1:199–260, 1994.
4. J. Endrullis et al. Productivity of stream definitions. *Theoretical Ccomputer Science*, 411(4-5):765–782, 2010.
5. J. Endrullis et al. A coinductive framework for infinitary rewriting and equational reasoning. In *RTA*, pages 143–159, 2015.
6. P. Fu and E. Komendantskaya. Operational semantics of resolution and productivity in Horn clause logic. *Accepted, Formal Aspects of Computing*, 2016.
7. G. Gupta et al. Coinductive logic programming and its applications. In *ICLP*, pages 27–44, 2007.
8. N. N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In *RTA*, pages 249–268, 2004.
9. P. Johann et al. Structural resolution for logic programming. In *Technical Communications of ICLP*, 2015.
10. E. Komendantskaya et al. Exploiting parallelism in coalgebraic logic programming. *Electronic Notes in Theoretical Computer Science*, (33):121–148, 2014.
11. E. Komendantskaya et al. Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, 26(2):745–783, 2016.
12. E. Komendantskaya and P. Johann. Structural resolution: a framework for coinductive proof search and proof construction in Horn clause logic. *Submitted*, 2015.
13. K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM*, pages 382–398, 2014.
14. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1988.
15. M. T. Nguyen et al. Termination analysis of logic programs based on dependency graphs. In *LOPSTR*, pages 8–22, 2007.
16. F. Pfenning. *Types in Logic Programming.* The MIT Press, 1992.
17. A. Reynolds and J. Blanchette. A decision procedure for (co)datatypes in SMT solvers. In *CADE*, pages 197–213, 2015.
18. E. Rohwedder and F. Pfenning. Model and termination checking for higher-order logic programs. In *ESOP*, pages 296–310, 1996.
19. M. Schmidt. Productivity checker for LP, www.macs.hw.ac.uk/∼ek19/CoALP/, 2016.
20. P. Schneider-Kamp et al. Automated termination analysis for logic programs by term rewriting. In *LOPSTR*, pages 177–193, 2006.
21. L. Simon et al. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pages 472–483, 2007.
22. Terese. *Term Rewriting Systems.* Cambridge University Press, 2003.