Coinductive Logic Programming


Department of Computer Science
University of Texas at Dallas, Richardson, TX 75080

Abstract We extend logic programming’s semantics with the semantic dual of traditional Herbrand semantics by using greatest fixed-points in place of least fixed-points. A query then involves using coinduction to check inclusion in the greatest fixed-point that constitutes the program’s semantics. The resulting coinductive logic programming language is syntactically identical to, yet semantically subsumes logic programming with rational terms and lazy evaluation. We give the declarative and operational semantics of such a coinductive logic programming language and prove them equivalent. Our operational semantics lends itself to an elegant and efficient goal directed proof search in the presence of infinite (but rational) terms and proofs. Coinductive logic programming has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent logic programming, model checking, bisimilarity proofs, Answer Set Programming (ASP), etc. Finally, an outline of a prototype implementation realized by modifying YAP Prolog’s engine at the WAM level is also described.

But look! What was that? One of the snakes had seized hold of its own tail, and the form whirled mocking before my eyes.

—Friedrich A. Kekule, 1864

1 Introduction

The traditional semantics for logic programming (LP) is inadequate for various programming practices such as programming with infinite data structures and corecursion [3]. While such programs are theoretically interesting, their practical applications include improved modularization of programs as seen in lazy functional programming languages, rational terms, and applications to model checking as discussed in section 5. For example, we would like programs such as the following program, which describes infinite streams, to be semantically meaningful, i.e. not semantically null.

\[
\begin{align*}
\text{bit}(0). \\
\text{bit}(1). \\
\text{bitstream}([H \mid T]) & :- \text{bit}(H), \text{bitstream}(T). \\
| \leftarrow X = [0, 1, 1, 0 | X], \text{bitstream}(X).
\end{align*}
\]

We would like the above query to return a positive answer, however, aside from the \text{bit} predicate, the least fixed-point (lfp) semantics of the above program is null. The problems are two-fold. The Herbrand universe does not allow for infinite terms such as $X$ and the least Herbrand model does not allow for infinite proofs, such as the proof of \text{bitstream}($X$). These concepts are commonplace in
computer science. Recently, a sound mathematical foundation has been established for them in the field of hyperset theory [3]. Therefore, we must not exclude these concepts from logic programming, just because they break with traditional well-founded set theory. The traditional declarative semantics of LP must be extended in order to reason about infinite and cyclic structures and properties. We refer to this extension as coinductive logic programming.\(^1\) Furthermore, the operational semantics must be extended, so as to finitely represent an otherwise infinite derivation. This paper proposes such a method which is based on coinduction, and discusses its implementation and applications. The work reported in this paper is a culmination of authors’ previous work [22,8,15,9]. Our work can be thought of as developing a practical and reasonable top-down operational semantics for computing the greatest fixed-point of a logic program. The rest of the paper is organized as follows. Section 2 gives coinductive logic programs meaningful declarative and operational semantics along with the proof that both semantics are equivalent. Section 3 discusses related work. Section 4 describes an implementation of coinductive LP based on modifying YAP Prolog’s engine [19], followed by section 5 which presents applications of coinductive LP to model checking, lazy evaluation, etc. Finally, section 6 discusses future work for extending coinductive LP and its applications.

2 Syntax and Semantics

Traditionally, declarative semantics for LP has been given using the notions of Herbrand universe, Herbrand base, and minimal model [14]. Each is defined as a least fixed-point, and the set is manifested in traditional set theory. The declarative semantics of coinductive LP, on the other hand, takes the dual of each of these notions, in hyperset theory with the axiom of plenitude [3].

2.1 Induction and Coinduction

A naive attempt to prove a property of the natural numbers involves demonstrating the property for 0, 1, 2, . . . . In order for such a proof to be comprehensive, it must be infinite. However, since nobody has the time to write an infinite proof, the principle of proof by induction can be used to represent such an infinite proof in a finite form. This is precisely what coinductive LP does as well. That is, coinductive LP uses the principle of proof by coinduction for representing infinite proofs in a finite form. The difference between induction and coinduction is made more obvious later.

Following the account given in Barwise [3] and Pierce [16], we briefly review the set theoretic notions of induction and coinduction, which are defined in terms of monotonic functions on sets and least and greatest fixed-points, which exist and are unique according to Theorem 1. For the remaining discussion, it is assumed that all objects such as elements, sets, and functions are taken from the universe of hypersets with the axiom of plenitude. Details can be found in [3].

\(^1\) Note that coinductive LP defined in this paper is not at all related to inductive LP which is the common term used to refer to LP systems for learning rules. In fact, sometimes we’ll use the term inductive LP itself to refer to traditional SLD (or OLDT) resolution-based LP.
**Definition 1.** A function $\Gamma$ on sets is monotonic if $S \subseteq T$ implies $\Gamma(S) \subseteq \Gamma(T)$. Such functions are called generating functions.

Generating functions can be thought of as a definition for creating objects, such as terms and proofs. The following example demonstrates one such definition.

**Example 1.** Let $\Gamma_N$ be a function on sets: $\Gamma_N(S) = \{0\} \cup \{\text{succ}(x) \mid x \in S\}$. Obviously, $\Gamma_N$ is a monotonic function, and intuitively, it defines the set of natural numbers, as will be demonstrated below.

**Definition 2.** Let $S$ be a set.

1. $S$ is $\Gamma$-closed if $\Gamma(S) \subseteq S$; $S$ is $\Gamma$-justified if $S \subseteq \Gamma(S)$.
2. $S$ is a fixed-point of $\Gamma$ if $S$ is both $\Gamma$-closed and justified.

A set $S$ is $\Gamma$-closed when every object created by the generator $\Gamma$ is already in $S$. Similarly, a set $S$ is $\Gamma$-justified when every object in $S$ is created or justified by the generator.

One of the purposes of mathematics is to provide unambiguous means for defining concepts. Theorem 1 shows that a generating function $\Gamma$ can be used for giving a precise definition of a set of objects in terms of the least or greatest fixed-point of $\Gamma$, as these fixed-points are guaranteed to exist, and are unique.

**Theorem 1.** (Knaster-Tarski) Let $\Gamma$ be a generating function. The least fixed-point of $\Gamma$ is the intersection of all $\Gamma$-closed sets. The greatest fixed-point (gfp) of $\Gamma$ is the union of all $\Gamma$-justified sets.

Since these fixed-points always exist and are unique, it is customary to define unary operators $\mu$ and $\nu$ for manifesting either of these fixed-points.

**Definition 3.** $\mu \Gamma$ denotes the lfp of $\Gamma$, and $\nu \Gamma$ denotes the gfp.

**Example 2.** Let $\Gamma_N$ be defined as in example 1. The definition of the natural numbers $N$ can now be unambiguously invoked via theorem 1, as $N = \mu N$, which is guaranteed to exist and be unique. Note that this definition is equivalent to the standard "inductive" definition of the natural numbers, which is written: Let $N$ be the smallest set such that $0 \in N$ and if $x \in N$, then $x + 1 \in N$.

Hence what is sometimes referred to as an inductive definition, is subsumed by definition via least fixed-point. This is further generalized by creating the dual notion of a definition by greatest fixed-point, termed a coinductive definition.

**Example 3.** $\Gamma_N$ from example 1 also unambiguously defines another set, that is, $N' = \nu N = N \cup \{\omega\}$, where $\omega = \text{succ}(\omega)$, that is, $\omega = \text{succ} (\text{succ} (\text{succ} (\ldots )))$ an infinite application of succ.

**Corollary 1.** The principle of induction states that if $S$ is $\Gamma$-closed, then $\mu \Gamma \subseteq S$, and the principle of coinduction states that if $S$ is $\Gamma$-justified, then $S \subseteq \nu \Gamma$.

**Definition 4.** Let $Q(x)$ be a property. Proof by induction demonstrates that the characteristic set $S = \{x \mid Q(x)\}$ is $\Gamma$-closed, and then invokes the principle of induction to prove that every element $x$ of $\mu \Gamma$ has the property $Q(x)$.

Similarly, proof by coinduction demonstrates that the characteristic set $S$ is $\Gamma$-justified, and then invokes the principle of coinduction to prove that every element $x$ that has property $Q(x)$ is also an element of $\nu \Gamma$.
Example 4. The familiar proof by induction can be instantiated with regards to the set \( \mathcal{N} \) defined in the previous example. Let \( Q(x) \) be some property, and let \( S = \{ x \mid Q(x) \} \). In order to show that every element \( x \) in \( \mathcal{N} \) has property \( Q(x) \), by induction it is sufficient to show that \( \Gamma \mathcal{N}(S) \subseteq S \), which is equivalent to showing that \( 0 \in S \), and if \( x \in S \), then \( \text{succ}(x) \in S \).

Like proof by induction, proof by coinduction is used in many aspects of computer science, e.g., bisimilarity proofs for process algebras such as the \( \pi \)-calculus. Section 2.6 demonstrates another example of proof by coinduction: the soundness proof of the operational semantics of coinductive LP.

2.2 Syntax

A coinductive logic program \( P \) is syntactically identical to a traditional, that is, inductive logic program as demonstrated by the following account of syntax. In the following, it is important to distinguish between an idealized class of objects and the syntactic restriction of said objects. Elements of syntax are necessarily finite, while many of the semantic objects used by coinductive LP are infinite. It is assumed that there is an enumerable set of variables, an enumerable set of constants, and for all natural numbers \( n \), there are an enumerable set of function and predicate symbols of arity \( n \).

**Definition 5.** The set of terms is \( \nu \Gamma \) and the set of syntactic terms is \( \mu \Gamma \), where \( \Gamma \in \Gamma(S) \) whenever one of the following is true:

1. \( t \) is a variable.
2. \( t \) is a constant.
3. \( t = f(t_1, \ldots, t_n) \), where \( t_1, \ldots, t_n \in S \) and \( f \) is a function symbol of arity \( n \).

**Definition 6.** An atom is an expression of the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol of arity \( n \) and \( t_1, \ldots, t_n \) are terms. A syntactic atom is an atom containing only syntactic terms.

**Definition 7.** A definite clause is a logical inference rule of the form \( C \leftarrow D_1, \ldots, D_n \) where \( C, D_1, \ldots, D_n \) are atoms. A syntactic definite clause is a definite clause containing only syntactic atoms.

**Definition 8.** A definite program is a finite set of syntactic definite clauses.

Again, note that a definite program is a finite object. The following definition of a coinductive logic program marries syntax and semantics.

**Definition 9.** A coinductive logic program is a definite program with the declarative semantics defined in section 2.3.

2.3 Declarative Semantics

The declarative semantics of coinductive LP is the “across the board” dual of the traditional minimal model Herbrand semantics [14,2]. This allows the universe of terms to contain infinite terms, in addition to the traditional finite terms; and, it also allows for the model to contain ground goals that have either finite or infinite proofs.
**Definition 10.** Let $P$ be a definite program. Let $A(P)$ be the set of constants in $P$, and let $F_n(P)$ denote the set of function symbols of arity $n$ in $P$. The co-Herbrand universe of $P$, denoted $U^co(P) = \Phi_P$, where

$$\Phi_P(S) = A(P) \cup \{ f(t_1, \ldots, t_n) \mid f \in F_n(P) \land t_1, \ldots, t_n \in S \}$$

Intuitively, this is the set of terms both finite and infinite that can be constructed from the constants and functions in the program. Hence unification without occurs check has a greatest fixed-point interpretation, as rational trees are included in the co-Herbrand universe. The Herbrand universe is simply $\Phi_P$.

**Definition 11.** Let $P$ be a definite program. The co-Herbrand base (also known as the infinitary Herbrand base [13]), written $B^co(P)$, is the set of all ground atoms that can be formed from the predicate symbols in $P$ and the elements of $U^co(P)$. Also, let $G^co(P)$ be the set of ground clauses $C \leftarrow D_1, \ldots, D_n$ that are a ground instance of some clause of $P$ such that $C, D_1, \ldots, D_n \in B^co(P)$.

Finally we can give the formal definition of a coinductive logic program $P$’s semantics as the maximal co-Herbrand model of $P$.

**Definition 12.** A co-Herbrand model of a program $P$ is a fixed-point of

$$Tp(S) = \{ C \mid C \leftarrow D_1, \ldots, D_n \in G^co(P) \land D_1, \ldots, D_n \in S \}$$

The maximal co-Herbrand model of a program $P$, denoted $M^co(P)$, is the greatest fixed-point of $Tp$, which exists and is unique according to theorem 1. Hence $M^co(P)$ is taken to be the declarative semantics of a coinductive logic program $P$.

The traditional minimal model, on the other hand, is the restriction of the least fixed-point of $Tp$ to the Herbrand base. Hence coinductive LP’s declarative semantics is dual to LP’s semantics, “across the board”: Herbrand universe, Herbrand base, and the minimal Herbrand model are all taken in “coo-” form. Truth is defined in terms of inclusion in the model.

**Definition 13.** An atom $A$ is true in program $P$ iff the set of all groundings of $A$, with substitutions ranging over the $U^co(P)$, is a subset of $M^co(P)$.

**Example 5.** Let $P_1$ be the following program.

```
from(N, [N|T]) :- from(s(N), T).
\| ?- from(0, _).
```

The coinductive semantics are derived as follows. The co-Herbrand Universe is $U^co(P) = N \cup \Omega \cup L$ where $N = \{ 0, s(0), s(s(0)), \ldots \}$, $\Omega = \{ s(s(\ldots)) \}$, and $L$ is the set of all finite and infinite lists of elements in $N$, $\Omega$, $L$. Therefore the maximal co-Herbrand model $M^co(P_1) = \{ from(t, [t, s(t), s(s(s(t))), \ldots]) \mid t \in U^co(P_1) \}$, which is the meaning of the program and obviously not null, as was the case with traditional LP. Furthermore $from(0, [0, s(0), s(s(0)), \ldots]) \in M^co(P_1)$ implies that the query returns “yes”. The next example has a simpler semantics. Let $P_2$ be the following program.
\( p \vdash p \).

The maximal co-Herbrand model \( P_2 = \{ p \} \), which is not empty. Hence, the query \( \_ \leftarrow \_ \leftarrow p \) would terminate with a “yes”.

The model characterizes semantics in terms of truth, that is, the set of ground atoms that are true. This set is defined via a generator, and in section 2.6, we discuss the way in which the generator is applied in order to include an atom in the model. For example, the generator is only allowed to be applied a finite number of times for any given atom in the minimal model, while it can be applied an infinite number of times for an atom in the maximal co-Herbrand model. We characterize this by recording the application of the generator in the elements of the fixed-point itself. We call these elements “idealized proofs”.

**Definition 14.** Let \( \text{node}(A, L) \) be a constructor of a tree with root \( A \) and sub-trees \( L \), where \( A \) is an atom and \( L \) is a list of trees. The set of idealized proofs for program \( P \) is \( \nu \Sigma P \), where

\[
\Sigma P(S) = \{ \text{node}(C, [T_1, \ldots, T_n]) \mid \\
C \leftarrow D_1, \ldots, D_n \in G^\omega(P) \land \text{the root of } T_i \in S \text{ is } D_i \}
\]

Again, this is nothing more than a reformulation of the maximal co-Herbrand model, which records the applications of the generator in the elements of the fixed-point itself, as the following theorem demonstrates.

**Theorem 2.** Let \( S = \{ A \mid \exists T \in \nu \Sigma P. A \text{ is the root of } T \} \), then \( S = M^\omega(P) \).

Hence any element in the model has an idealized proof and anything that has an idealized proof is in the model. A similar theorem exists, equating the minimal model with the least fixed-point of \( \Sigma P \) restricted to finite terms, i.e., the minimal model consists of all ground atoms that have a finite idealized proof. This formulation of the declarative semantics in terms of idealized proofs will be used in section 2.6.

### 2.4 Operational Semantics

This section defines the operational semantics for coinductive logic programming. This requires some infinite tree theory. However, this section only states a few definitions and theorems without proof. Details can be found in [6].

The operational semantics given for coinductive LP is defined in a manner similar to SLD, and is therefore called co-SLD. Where SLD uses sets of syntactic atoms and syntactic term substitutions for states, co-SLD uses finite trees of syntactic atoms along with systems of equations. Of course, the traditional goals of SLD can be extracted from these trees, as the goal of a tree is simply the set of leaves of the tree. Furthermore, where SLD only allows program clauses as state transition rules, co-SLD also allows a special coinductive hypothesis rule.

**Definition 15.** A tree is rational if the cardinality of the set of all its sub-trees is finite. An object such as a term, atom, or idealized proof is said to be rational if it is modeled as a rational tree.
**Definition 16.** A substitution is a finite mapping of variables to terms. A substitution is syntactic if it only substitutes syntactic terms for variables. A substitution is said to be rational if it only substitutes rational terms for variables.

**Definition 17.** A term unification problem is a finite set of equations between terms. A unifier for a term unification problem is a substitution that satisfies every equation in the problem. σ is a most general unifier (mgu) for a term unification problem, if any other solution σ' can be defined as the composition σ'' ∘ σ.

Note that terms are possibly infinite. So it is possible for a unification problem to lack a syntactic unifier, while at the same time the problem has a solution: a rational unifier. However, objects of an operational semantics should be finite. Hence we define a standard finite representation of rational substitutions called a system of equations.

**Definition 18.** A system of equations E is a term unification problem where each equation is of the form X = t, s.t. X is a variable and t a syntactic term.

**Theorem 3.** (Courcelle) Every system of equations has a mgu that is rational.

**Theorem 4.** (Courcelle) For every rational substitution σ with domain V, there is a system of equations E, such that the most general unifier σ' of E is equal to σ when restricted to the domain V.

Without loss of generality, the previous two theorems allow for a solution to a term unification problem to be simultaneously a substitution as well as a system of equations. Note that given a substitution specified as a system of equations E, and a term A, the term E(A) denotes the result of applying the substitution E to A.

Now the operational semantics can be defined. The semantics implicitly defines a state transition system. Systems of equations are used to model part of the state of coalgebraic LP's semantics. They effectively denote the current state of unification of terms. The current state of the pending goals is modeled using a finite tree of atoms, as it is necessary to recognize cycles in the sequence of pending goals, that is, the ancestors of a goal are memo-ed in order to recognize a cycle in the proof.

**Definition 19.** A state S is a pair (T, E), where T is a finite tree with nodes labeled with syntactic atoms, and E is a system of equations.

**Definition 20.** A transition rule R of a coalgebraic logic program P is an instance of a clause in P, with variables standardized apart, i.e., consistently renamed for freshness, or R is a coalgebraic hypothesis rule of the form ν(n), where n is a natural number.

Obviously the state transition system may be nondeterministic, depending on the program, that is, it is possible for states to have more than one outgoing transition as the following definition shows.
Definition 21. A state \((T, E)\) transitions to another state \((T', E')\) by transition rule \(R\) of program \(P\) whenever:

1. \(R\) is a definite clause of the form \(p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m\) and \(E'\) is the most general unifier for \(\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E\), and \(T'\) is obtained from \(T\) according to the following case analysis of \(m\):
   (a) \(m = 0\) implies \(T'\) is obtained from \(T\) by removing a leaf labeled 
       \(p(t_1, \ldots, t_n)\) and the maximum number of its ancestors, such that the result is still a tree.
   (b) \(m > 0\) implies \(T'\) is obtained from \(T\) by adding children \(B_1, \ldots, B_m\) to a leaf labeled with \(p(t_1, \ldots, t_n)\).

2. \(R\) is of the form \(\nu(m)\), a leaf \(N\) in \(T\) is labeled with \(p(t_1, \ldots, t_n)\), the proper ancestor of \(N\) at depth \(m\) is labeled with \(p(t_1', \ldots, t_n')\), \(E'\) is the most general unifier for \(\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E\), then \(T'\) is obtained from \(T\) by removing 
   \(N\) and the maximum number of its ancestors, such that the result is still a tree.

The part of the previous definition that removes a leaf and a maximum number of its ancestors can be thought of as a successful call returning and therefore deallocating memo-ed calls on the call stack. This involves successively removing ancestor nodes of the leaf until an ancestor is reached, which still has other children, and so removing any more ancestors would cause the result to no longer be a tree, as children would be orphaned. Hence the depth of the tree is bounded by the depth of the call stack.

Definition 22. A transition sequence in program \(P\) consists of a sequence of states \(S_1, S_2, \ldots\) and a sequence of transition rules \(R_1, R_2, \ldots\), such that \(S_i\) transitions to \(S_{i+1}\) by rule \(R_i\) of program \(P\).

A transition sequence denotes the trace of an execution. Execution halts when it reaches a terminal state: either all goals have been proven or the execution path has reached a dead-end.

Definition 23. The following are two distinguished terminal states:

1. An accepting state is of the form \((\emptyset, E)\), where \(\emptyset\) denotes an empty tree.
2. A failure state is a non-accepting state lacking any outgoing edges.

Without loss of generality, we restrict queries to be single syntactic atoms. A query containing multiple atoms can be modeled by adding a new predicate with one clause to the program. Finally we can define the execution of a query as a transition sequence through the state transition system induced by the input program, with the start state consisting of the initial query.

Definition 24. A co-SLD derivation of a state \((T, E)\) in program \(P\) is a state transition sequence with the first state equal to \((T, E)\). A derivation is successful if it ends in an accepting state, and a derivation has failed if it reaches a failure state. We say that a syntactic atom \(A\) has a successful derivation in program \(P\), if \((A, \emptyset)\) has a successful derivation in \(P\).
2.5 Examples

In addition to allowing infinite terms, the operational semantics allows for an execution to succeed when it encounters the same goal again. While this is somewhat similar to tabled LP in that called atoms are recorded so as to avoid unnecessary redundant computation, the difference is that coinductive LP’s memo-ed atoms represent a coinductive hypothesis, while tabled logic programming’s table represents a list of results for each called goal in the traditional inductive semantics. Hence we call the memo-ed atoms the dynamic coinductive hypothesis. An example that demonstrates the distinction is the following program.

\[ \text{p} := \text{p}. \]

| --- |

Execution starts by checking the dynamic coinductive hypothesis for a variant of \( p \), which does not exist, so \( p \) is added to the hypothesis. Next, the body of the goal is executed. Again, the hypothesis is checked for a variant of \( p \), which is now already included, so the most recent call succeeds and then since no remaining goals exist, the original query succeeds. Hence, according to the operational semantics of coinductive LP, the query has a successful derivation, and hence returns “yes”; while traditional (tabled) LP returns “no”.

Now for a more complicated example involving function symbols. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers and \( \omega \), that is, infinity.

\[
\text{stream}([H \mid T]) := \text{number}(H), \text{stream}(T).
\]

\[
\text{number}(0).
\]

\[
\text{number}(s(N)) := \text{number}(N).
\]

| ? - \text{stream}([0, s(0), s(s(0)) \mid T]). |

The following is an execution trace, for the above query, of the memoization and unmemoization of calls by the operational semantics:

1. MEMO: \text{stream}([0, s(0), s(s(0)) \mid T])
2. MEMO: \text{number}(0)
3. UNMEMO: \text{number}(0)
4. MEMO: \text{stream}([s(0), s(s(0)) \mid T])
5. MEMO: \text{number}(s(0))
6. MEMO: \text{number}(0)
7. UNMEMO: \text{number}(0)
8. UNMEMO: \text{number}(s(0))
9. MEMO: \text{stream}([s(s(0)) \mid T])
10. MEMO: \text{number}(s(s(0)))
11. MEMO: \text{number}(s(0))
12. MEMO: \text{number}(0)
13. UNMEMO: \text{number}(0)
14. UNMEMO: \text{number}(s(0))
15. UNMEMO: \text{number}(s(s(0)))
The next goal call is \texttt{stream( T )}, which unifies with memo-ed ancestor (1), and therefore immediately succeeds. Hence the original query succeeds with
\[ T = [ 0, s(0), s(s(0)) | T ] \]
The user could force a failure here, which would cause the goal to be unified with the next matching memo-ed ancestor, if such an element exists, otherwise the goal is memo-ed and the process repeats—generating additional results (T = [0, s(0), s(s(0)) | R], R = [0 | R], etc.). Note that excluding the occurs check is necessary as such structures have a greatest fixed-point interpretation and are in the co-Herbrand Universe. We will see that this is in fact one of the benefits of coinductive LP. Traditional LP’s least Herbrand model semantics requires SLD resolution to unify with occurs check (or lack soundness), which adversely affects performance in the common case. Coinductive LP, on the other hand, has a declarative semantics that allows unification without doing occurs check in an efficient manner as seen in rational tree unification.

2.6 Correctness
We next prove the correctness of the operational semantics by demonstrating its correspondence with the declarative semantics via soundness and completeness theorems. Completeness, however, must be restricted to atoms that have a rational proof. Section 6 discusses an extension of the operational semantics, so as to improve its completeness. The soundness and completeness theorems are stated below, their proofs are relegated to Appendix I due to lack of space.

**Theorem 5.** (soundness) If atom \( A \) has a successful co-SLD derivation in program \( P \), then \( E(A) \) is true in program \( P \), where \( E \) is the resulting variable bindings for the derivation.

**Theorem 6.** (completeness) If \( A \in M^{co}(P) \) has a rational idealized proof, then \( A \) has a successful co-SLD derivation in program \( P \).

3 Related Work
Most of the work in the past has been focused on allowing for infinite data structures in LP. However, most of these stop short of including infinite proofs. Logic programming with rational trees \([4,5,13]\) allows for finite terms as well as infinite terms that are rational trees, that is, terms that have finitely many distinct subterms. Coinductive LP as defined in Section 2, on the other hand, allows for finite terms, rational infinite terms, but unlike LP with rational trees, coinductive LP also allows for irrational infinite terms. Furthermore, the declarative semantics of LP with rational trees corresponds to the minimal co-Herbrand model. On the other hand, coinductive LP’s declarative semantics is the maximal co-Herbrand model. Also, the operational semantics of LP with rational trees is simply SLD extended with rational term unification, while the operational semantics of coinductive LP corresponds to SLD only via the fact that both are implicitly defined in terms of state transition. Thus, LP with rational trees does not allow for infinite proofs while coinductive LP does. Finally, LP with rational trees can only create infinite terms via unification (without occurs check), while coinductive LP can create infinite terms via unification (without occurs check) as well as via user-defined coinductively recursive (or corecursive) predicates.
Jaffar et al’s coinductive tabling proof method [12] uses coinduction as a means of proving infinitary properties in model checking, as opposed to using it in defining the semantics of a new declarative programming language, as is the case with coinductive LP presented in this paper. Jaffar et al’s coinductive tabling proof method itself is analogous to coinductive LP’s co-SLD operational semantics described above in that both use the principle of coinduction to prove infinitary properties with some form of a finite derivation. However, Jaffar et al’s coinductive tabling proof method is not assigned any declarative, model-theoretic semantics, as is the case with coinductive logic programming presented in this paper, which has a declarative semantics, operational semantics, and a correctness proof showing the correspondence between the two. Coinductive logic programming, when extended with constraints, can be used for the same applications as Jaffar et al’s coinductive tabling proof method (see Appendix II).

Lazy functional LP (e.g., [7,11]) also allows for infinite data structures, but it encodes predicates as Boolean functions, while in comparison, coinductive LP defines predicates via Horn clauses. The difference in semantics is even more pronounced. Predicates in lazy functional LP tend to have a mostly operational semantics in terms of lazy narrowing, which means that an instance of a predicate is true when the argument terms of the corresponding predicate can be instantiated in such a way that the function evaluates to true. However, if the property is infinitary and has an infinite idealized proof, then the corresponding function will not evaluate to true because it will have an infinite evaluation. In coinductive LP, on the other hand, a predicate with an infinite idealized proof is defined as true, and the operational semantics allow for the finite derivation via the use of coinduction. Therefore, predicates in lazy functional logic programming are semantically different from those in coinductive logic programming.

4 Implementation

A prototype implementation of coinductive LP is being realized by modifying the YAP Prolog system [19]. The general operational semantics described above allows for a coinductively recursive call to terminate (coinductively succeed) if it unifies with a call that has been seen earlier. However, in the current prototype, a coinductive call terminates only if it is a variant of an ancestor call. Additionally, in the current prototype, coinductive calls can produce infinite-sized output but cannot consume infinite-sized inputs (this is primarily due to limitations in YAP Prolog implementation). Our current implementation effort is focused on fixing both these limitations.

The implementation of coinductive LP is reasonably straightforward, and is based on the machinery used in the YAP system for realizing OLDT style tabling [19]. Predicates have to be declared coinductive via the directive:

```prolog
:- coinductive p/n.
```

where p is the predicate name and n its arity. When a coinductive call is encountered for the first time, it is recorded in the memo-table that YAPTAB [19] uses for implementing standard tabled LP. The call is recorded again in the table after head unification, but this time it is saved as a solution to the tabled call. The
variables in the recorded solution are interpreted w.r.t. the environment of the coinductive call (so effectively the closure of the call is saved). When a variant call is encountered later, it is unified with the solution saved in the table and made to succeed. Note that everything recorded in the memo-table for a specific coinductive predicate $p$ will be deleted, when execution backtracks over the first call of $p$. Consider the example program:

$$\begin{align*}
:- \text{coinductive } p/1.
p(f(X)) & :- p(X). \\
| ?- p(Y).
\end{align*}$$

When the call $p(Y)$ is made, it is first copied (say as $p(A)$) in the table as a coinductive call. Next, a matching rule is found and head unification performed ($Y$ is bound to $f(X)$). Next, $p(Y)$ (i.e., $p(f(X))$) is recorded as a solution to the call $p(A)$. The variable $X$ in the solution refers to the $X$ in the rule matching the coinductive call (i.e., it points to the variable $X$ in the environment allocated on the stack). When the coinductive call $p(X)$ is encountered in the body of the rule, it is determined to be a variant of the call $p(A)$ stored in the memo-table, and unified with the solution $p(f(X))$. This results in $X$ being bound to $f(X)$, i.e., $X = f(X)$, producing the desired solution $f^a(\ldots)$.

One can see from the description above that much of the machinery for OLDT tabling present in YAP TAB can be re-used for implementing coinductive LP. However, because YAP TAB uses tries \cite{YAP}, which do not support rational trees, predicates that take rational terms as input arguments cannot be coinductively interpreted in the current implementation. Thus, coinductive member/2 and append/3 predicates will not currently work in our system. Our current implementation is, however, adequate to run constraint LP based implementations of somewhat complex applications (e.g., timed automata). Work is in progress to extend tries in YAP TAB to support rational terms.

5 Applications

Coinductive LP augments traditional logic programming with rational terms and rational proofs. These concepts generalize the notions of rational trees and lazy predicates. Coinductive LP has practical applications in concurrent LP, bisimilarity, model checking, and many other areas. Furthermore, it appears that the concept of ancestors in the co-SLD semantics can be used to give a top-down operational semantics to a restricted form of ASP programs (work is in progress). Coinductive LP also allows type inference algorithms in functional programming to be implemented directly and elegantly.

Infinite Terms and Properties: As previously stated, coinductive LP subsumes logic programming with rational trees of Jaffar et al \cite{Jaffar} and Colmerauer \cite{Colmerauer}. However, because LP with rational trees has semantics ascribed by the minimal co-Herbrand model, applying predicates to infinite trees is rather limited. Coinductive LP removes this limitation by ascribing the semantics in terms of maximal co-Herbrand model. This is demonstrated by the traditional definition of append, which, when executed with coinductive logic programming semantics, allows for calling the predicate with infinite arguments. This is illustrated below.
As an aside, note that irrational lists also make it possible to directly represent an infinite precision irrational real number as an infinite list of natural numbers.

\[
\text{append( } [], X, X ).
\]
\[
\text{append( } [H|T], Y, [H|Z] ) :- \text{ append( } T, Y, Z ).
\]

Not only can the above definition append two finite input lists, as well as split a finite list into two lists in the reverse direction, it can also append infinite lists under coinductive execution. It can even split an infinite list into two lists that when appended, equal the original infinite list. For example:

\[
| ?- Y = [4, 5, 6, Y], \text{ append([1, 2, 3], Y, Z) }.
\]
\[
\text{Answer: } Z = [1, 2, 3, Y], Y = [4, 5, 6, Y]
\]

More generally, the coinductive append has interesting algebraic properties. When the first argument is infinite, it doesn’t matter what the value of the second argument is, as the third argument is always equal to the first. However, when the second argument is infinite, the value of the third argument still depends on the value of the first. This is illustrated below:

\[
| ?- X = [1, 2, 3, X], Y = [3, 4, Y], \text{ append(X, Y, Z) }.
\]
\[
\text{Answer: } Z = [1, 2, 3, X, Y] \quad \text{and} \quad Y = [3, 4, Y]
\]
\[
| ?- Z = [1, 2, Z], \text{ append(X, Y, Z) }.
\]
\[
\text{Answers: } X = [1, 2, Z]; \\
\quad \text{and} \quad Y = [3, 4, Y]; \\
\quad \text{and} \quad Z = [1, 2, Y]
\]

**Lazy Evaluation of Logic Programs:** Coinductive LP also allows for lazy evaluation to be elegantly incorporated into Prolog. Lazy evaluation allows for manipulation of, and reasoning about, cyclic and infinite data structures and properties. In the presence of coinductive LP, if the infinite terms involved are rational, then given the goal p(X), q(X) with coinductive predicates p/1 and q/1, then p(X) can coinductively succeed and terminate, and then pass the resulting X to q(X). If X is bound to an infinite irrational term during the computation, then p and q must be executed in a coroutined manner to produce answers. That is, one of the goals must be declared the producer of X and the other the consumer of X, and the consumer goal must not be allowed to bind X.

Consider the (coinductive) lazy logic program for the sieve of Eratosthenes:

\[
:- \text{ coinductive sieve/2, filter/3, member/2.}
\]
\[
\text{primes(X)} :- \text{ generate_infinite_list(1), sieve(I,L), member(X, L).}
\]
\[
\text{sieve([H|T], [H|R]) :- filter(H, T, F), sieve(F, R).}
\]
\[
\text{filter(H, [K|T], [K|T]) :- R is K mod H, R > 0, filter(H, T, T1).}
\]
\[
\text{filter(H, [K|T], T1) :- 0 is K mod H, filter(H, T, T1).}
\]

In the above program filter/3 removes all multiples of the first element in the list, and then passes the filtered list recursively to sieve/2. If the predicate generate_infinite_list(1) binds I to a rational list (e.g., X = [2, ..., 20 | X]), then filter can be completely processed in each call to sieve/2. However, in contrast, if I is bound to an irrational infinite list as in:
:- coinductive int/2.
int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
generate_infinite_list(I) :- int(2, I).
then in the \texttt{primes/1} predicate, the calls \texttt{generate_infinite_list/1, sieve/2}
and \texttt{member/2} should be co-routined, and, likewise, in the \texttt{sieve/2} predicate,
the calls \texttt{filter/3} and the recursive call \texttt{sieve/2} must be coroutined.

\textbf{Concurrent Logic Programming:} From the discussion on lazy LP, one can
also observe that coinductive LP can be the basis of providing elegant declarative
semantics to concurrent LP [21]. Details are omitted due to lack of space.

\textbf{Model Checking and Verification:} Model checking is a popular technique
used for verifying hardware and software systems. It works by constructing a
model of the system in terms of a finite state Kripke structure and then
determining if the model satisfies various properties specified as temporal logic
formulae. The verification is performed by means of systematically searching
the state space of the Kripke structure for a counter-example that falsifies the given
property. The vast majority of properties that are to be verified can be classified
into \textit{safety} properties and \textit{liveness} properties. Intuitively, safety properties are
those which assert that ‘nothing bad will happen’ while liveness properties are
those that assert that ‘something good will eventually happen.’

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node (s0) at (0,0) [state] {s0};
\node (s1) at (2,0) [state] {s1};
\node (s2) at (2,-2) [state] {s2};
\node (s3) at (0,-2) [state] {s3};
\draw [->] (s0) edge [loop above] node {a} (s0);
\draw [->] (s0) edge node {b} (s1);
\draw [->] (s0) edge node {c} (s2);
\draw [->] (s1) edge node {d} (s3);
\draw [->] (s1) edge [loop below] node {e} (s1);
\draw [->] (s2) edge node {c} (s0);
\end{tikzpicture}
\caption{Example Automata}
\end{figure}

An important application of coinductive LP is in directly representing and
verifying properties of Kripke structures and \(\omega\)-automata (automata that accept
infinite strings). Just as automata that accept finite strings can be directly
programmed using standard LP, automata that accept infinite strings can be di-
rectly represented using coinductive LP (one merely has to drop the base case).
Consider the automata (over finite strings) shown in Figure 1.A which is repre-
sented by the logic program below.

\begin{verbatim}
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
automata([], St) :- final(St).
trans(s0, a, s1).
trans(s0, c, s3).
trans(s1, b, s2).
trans(s2, c, s3).
trans(s3, d, s0).
final(s2).
\end{verbatim}

A call to \texttt{\?- automata(X, s0)}. in a standard LP system will generate
all finite strings accepted by this automata. Now suppose we want to turn this
automata into an \(\omega\)-automata, i.e., it accepts infinite strings (an infinite string
is accepted if states designated as final state are traversed infinite number of
times), then the (coinductive) logic program that simulates this automata can be obtained by simply dropping the base case.  

\[
\text{automata}(X|T), \ St \) :- trans(St, X, NewSt), \text{automata}(T, NewSt).
\]

Under coinductive semantics, posing the query \( \text{?- automata}(X, s0) \) will yield the solutions:

\[
\begin{align*}
X & = \{a, b, c, d \mid \} \\
X & = \{a, b, e \mid \}
\end{align*}
\]

This feature of coinductive LP can be leveraged to directly verify liveness properties in model checking, multi-valued model checking, for modeling and verifying properties of timed \( \omega \)-automata, checking for bisimilarity, etc.

**Verifying Liveness Properties:** It is well known that safety properties can be verified by reachability analysis, i.e., if a counter-example to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Verification of safety properties amounts to computing least fixed-points and thus is elegantly handled by standard LP systems extended with tabling [17]. Verification of liveness properties under such tabled LP systems is however problematic. This is because counterexamples to liveness properties take the form of infinite traces, which are semantically expressed as greatest fixed-points. Tabled LP systems [17] work around this problem by transforming the temporal formula denoting the property into a semantically equivalent least fixed-point formula, which can then be executed as a tabled logic program. This transformation is quite complex as it uses a sequence of nested negations.

In contrast, coinductive LP can directly used to verify liveness properties. Coinductive LP can directly compute counterexamples using greatest fixed-point temporal formulae without requiring any transformation.  

Intuitively, a state is not live if it can be reached via an infinite loop (cycle). Liveness counterexamples can be found by (coinductively) enumerating all possible states that can be reached via infinite loops and then by determining if any of these states constitutes a valid counterexample. Consider the example of a modulo 4 counter, adapted from [20] (See Figure 1.B). For correct operation of the counter, we must verify that along every path the state \( s_{-1} \) is not reached, i.e., there is at least one infinite trace of the system along which \( s_{-1} \) never occurs. This property is naturally specified as a greatest fixed-point formula and can be verified coinductively. A simple coinductive logic program \( S_p \) to solve the problem is shown below. We compose the counter program with the negation of the property, i.e., \( N_1 \geq 0 \). Note that \( smi \) represents the state corresponding to \(-1\).

\[
\begin{align*}
: & :- \text{coinductive } s0/2, s1/2, s2/2, s3/2, smi/2. \\
& \quad \text{smi}(N,[smi|T]) :- N1 is N+1 \text{ mod 4, } s0(N1,T), N1>=0. \\
& \quad s0(N,[s0|T]) :- N1 is N+1 \text{ mod 4, } s1(N1,T), N1>=0.
\end{align*}
\]

---

2 We’ll ignore the requirement that final-designated states occur infinitely often; this can be checked by introducing a coinductive definition of \( \text{member}(s2, X) \) and checking that \( \text{member}(s2, X) \) holds for all accepting strings.

3 We assume that there are no fairness constraints on the Kripke structure, as such a scenario would require the use of alternating fixed-point temporal formulae, which are outside the scope of this paper [23].
\[ s_1(N, [s_1(T)]) \) \text{- } N_1 \text{ is } N+1 \mod 4, \ s_2(N_1, T), \ N_1 > 0. \\
\[ s_2(N, [s_2(T)]) \) \text{- } N_1 \text{ is } N+1 \mod 4, \ s_3(N_1, T), \ N_1 > 0. \\
\[ s_3(N, [s_3(T)]) \) \text{- } N_1 \text{ is } N+1 \mod 4, \ s_0(N_1, T), \ N_1 > 0. \\
The query : - sm1(-1, X), member(sm1, X) where the coinductive member checks that sm1 occurs in X infinitely often, will fail implying inclusion of the property in the model, i.e., the absence of a counterexample to the property. The benefit of our approach is that we do not have to transform the model into a form amenable to safety checking. This transformation is expensive in general and can reportedly increase the time and memory requirements 6-folds [20].

This direct approach to verifying liveness properties also applies to multi-valued model checking of the \( \mu \)-calculus [15]. Multi-valued model checking is used to model systems, whose specification has varying degrees of inconsistency or incompleteness. Earlier effort [15] verified liveness properties by computing the \( gfp \) which was found using negation based transformation described earlier. With coinduction, the \( gfp \) can be computed directly as in standard model checking as described above. We do not give details due to lack of space. Coinductive LP can also be used to check for bisimilarity. Bisimilarity is reduced to coinductively checking if two \( \omega \)-automata accept the same set of rational infinite strings.

**Verifying Properties of Timed Automata:** Timed automata are simple extensions of \( \omega \)-automata with stopwatches [1], and are easily modeled as coinductive logic programs with CLP(R) [9]. Timed automata can be modeled with coinductive logic programs together with constraints over reals for modeling clock constraints. Appendix II shows the coinductive program with CLP(R) constraints for modeling the classic train-gate-controller problem. This program can be run on our implementation of coinduction on YAP extended with CLP(R). The system can be queried to enumerate all the infinite strings that will be accepted by the automata and that meet the time constraints. Safety and liveliness properties can be checked by negating those properties, and checking that they fail for each string accepted by the automata with the help of coinductively defined \texttt{member/2}, \texttt{append/3} predicates (similar to [9]).

### 6 Conclusions and Future Work

In this paper we presented a comprehensive theory of coinductive LP, demonstrated its practical applications as well as reported on its preliminary implementation on top of YAP Prolog. Current work involves obtaining an implementation of full coinductive LP that works both for infinite outputs as well as infinite inputs. Current work [23] also involves extending coinductive logic programming to allow for finite derivations in the presence of irrational terms and proofs, that is, infinite terms and proofs that do not have finitely many distinct subtrees. Our current approach is to allow the programmer to annotate predicate definitions with pragmas, which can be used to decide at run-time when a semantic cycle in the proof search has occurred; however, in the future we intend to infer these annotations by using static analysis.

Future work also involves combining standard (tabled) LP with coinductive LP, so that traditional (i.e., inductive) LP can be mixed with the coinductive
LP [23] presented here. We are also working on incorporating coinductive reasoning in our quest of developing a single LP system that combines tabled LP, constraints, parallelism, ASP and co-routining [10]. Additionally, we are also working on applying the coinduction principle to Hereditary Harrop formulas, resolution theorem proving, and non-monotonic reasoning (ASP).

Acknowledgments: We are grateful to Vitor Santos Costa and Richardo Rocha for help with YAP, and Srividya Kona for comments.

References

Appendix I: Correctness Proof

Note: included for the convenience of reviewers.

The proofs of the soundness and correctness theorems are given below. Note that completeness is restricted to atoms that have a rational proof. Section 6 discussed an extension of the operational semantics so as to improve its completeness.

**Lemma 1.** If \((A, E_1)\) has a successful co-SLD derivation in program \(P\), with final state \((\emptyset, E_2)\), then \((A, E_3)\) has a successful co-SLD derivation in program \(P\), where \(E_2 \subseteq E_3\), with each state of the derivation of the form \((T, E_3)\) for some tree of atoms \(T\).

**Proof.** Let \((A, E_1)\) have a successful co-SLD derivation in program \(P\) ending with state \((\emptyset, E_2)\). In the sequence of states, the system of equations monotonically increases, and so the monotonicity of unification with infinite terms implies \((A, E_3)\) has a successful co-SLD derivation in program \(P\), where \(E_2 \subseteq E_3\), with each state of the derivation of the form \((T, E_3)\) for some tree of atoms \(T\).

**Lemma 2.** If \(A\) has a successful co-SLD derivation in program \(P\), which first transitions to a state \((\text{node}(A, [B_1, \ldots, B_n]), E)\) by applying clause \(A' \leftarrow B_1, \ldots, B_n\), such that \(E(A) = E(A')\), then each \((B_i, E)\) also has a successful derivation in program \(P\).

**Proof.** Let \((\text{node}(A, [B_1, \ldots, B_n]), E)\) have a successful co-SLD derivation in program \(P\), which first transitions to a state \((\text{node}(A, [B_1, \ldots, B_n]), E)\) by applying clause \(A' \leftarrow B_1, \ldots, B_n\), such that \(E(A) = E(A')\). A derivation for \((B_i, E)\) can be created by mimicking each transition that modifies the subtree rooted at \(B_i\) in the original derivation, except for the transitions which are of the form \(\nu(n)\), which no longer are correct derivations because the parent \(A\) of \(B_i\) no longer exists. In the case that \(n > 0\), instead apply the transition rule \(\nu(n - 1)\) to the corresponding leaf to which the original derivation would have applied \(\nu(n)\). Otherwise, when \(n = 0\), a coinductive transition rule cannot be applied to the corresponding leaf. Instead, mimic the transitions of the entire original derivation of \(A\).

**Lemma 3.** If \((A, E)\) has a successful co-SLD derivation in program \(P\), then \(E'(E(A))\) is true in program \(P\), where \((\emptyset, E')\) is the final state of the derivation.

**Proof.** Let \(Q\) be the set of all groundings ranging over the \(U^{co}(P)\) of all such \(E'(E(A))\). It is sufficient to prove that \(Q \subseteq M^{co}(P)\). The proof proceeds by coinduction.

Let \(A' \in S\), then \(A' = E_1(E_2(E_3(A)))\), where \(E_1\) is a grounding substitution for \(E_2(E_3(A))\) and \((A, E_3)\) has a successful derivation ending in \((\emptyset, E_2)\). By
lemma 1, \((A, E)\) has a successful derivation, where \(E = E_1 \cup E_2 \cup E_3\). This derivation must begin with an application of a program clause \(A' \leftarrow B_1, \ldots, B_n\), resulting in the state \((\text{node}(A, [B_1, \ldots, B_n]), E)\), where \(E(A) = E(A')\). By lemma 2 each state \((B_i, E)\) has a successful derivation. Let \(E'\) be a grounding substitution for the clause \(E(A' \leftarrow B_1, \ldots, B_n)\), such that \(C = E'(E(A' \leftarrow B_1, \ldots, B_n)) \in G^{\infty}(P)\), then \(C = A \leftarrow E''(B_1), \ldots, E''(B_n)\), where \(E'' = E' \cup E\). By lemma 1, each \((B_i, E'')\) has a successful derivation, and hence \(E''(B_i) \in Q\). Therefore, by the principle of coinduction, \(Q \subseteq M^{\infty}(P)\).

**Theorem 7.** (soundness) If atom \(A\) has a successful co-SLD derivation in program \(P\), then \(E(A)\) is true in program \(P\), where \(E\) is the resulting variable bindings for the derivation.

**Proof.** This follows by a straightforward instantiation of lemma 3.

The soundness of co-SLD does not require the execution of an occurs check during unification, as the declarative semantics naturally allows for infinite terms. The soundness of SLD, with regards to the minimal Herbrand model, however, requires the occurs check, which has an adverse affect on performance.

**Theorem 8.** (completeness) If \(A \in M^{\infty}(P)\) has a rational idealized proof, then \(A\) has a successful co-SLD derivation in program \(P\).

**Proof.** Let \(A \in M^{\infty}(P)\) have a rational idealized proof \(T\). The derivation is constructed by recursively applying the clause corresponding to each node encountered along a depth-first traversal of the idealized proof tree to the corresponding leaf in the current state. In order to ensure that the derivation is finite, the traversal stops at the root \(R\) of a subtree that is identical to a subtree rooted at a proper ancestor at depth \(n\). Then the derivation applies a transition rule of the form \(\nu(n)\) to the leaf corresponding to \(R\) in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation.

The fact that the set of all subtrees of \(T\) is finite in cardinality implies that the maximum depth of the traversal in the idealized proof tree is finite, and the fact that all idealized proofs are finitely branching implies that the traversal always terminates. So the constructed derivation is finite.

It remains to prove that the final state of the constructed derivation is a success state. The traversal stops going deeper in the idealized proof tree due to two cases: the traversal reaches a leaf in the idealized proof tree or the traversal encountered a subtree identical to an ancestor subtree. In either case, the derivation removes the corresponding leaf in the current state, as well as the maximal number of ancestors of the corresponding leaf such that the result is still a tree. So a leaf only remains in the state when its corresponding node in the proof tree has yet to be traversed. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty tree, which is the definition of an accept state. Therefore, \(A\) has a successful co-SLD derivation in program \(P\).
Appendix II: Timed Automata as a Coinductive Logic Program

Note: included for the convenience of reviewers.

The code for the timed automata represented in Fig 2 is given below. The predicate driver/9, that composes the 3 automata, is coinductive.

![Diagram of Timed Automata and Controller-Gate](image)

**Figure 2. Train-Controller-Gate Timed Automata**

```prolog
:- use_module(library(clpr)).
:- coinductive driver/9.

train(X,up,X,T1,T2,T3).
train(s0,approach,s1,T1,T2,T3) :-
  \{ T3 = T1 \}.
train(s1,in,s2,T1,T2,T3) :-
  \{ T1 - T2 > 2, T3 = T2 \}.
train(s2,out,s3,T1,T2,T3) :-
  \{ T3 = T2, T1 - T2 < 5 \}.

train(X,lower,X,T1,T2,T3).
train(X,down,X,T1,T2,T3).
train(X,raise,X,T1,T2,T3).

contr(s0,approach,s1,T1,T2,T1).

contr(s1,lower,s2,T1,T2,T3) :- \{ T3 = T2, T1 - T2 = 1 \}.
contr(s2,exit,s3,T1,T2,T1).
contr(s3,raise,s0,T1,T2,T1) :- \{ T1 - T2 < 1 \}.
contr(X,in,X,T1,T2,T3).
contr(X,up,X,T1,T2,T3).

contr(X,up,X,T1,T2,T3).

contr(X,down,X,T1,T2,T3).

contr(X,exit,X,T1,T2,T3).

contr(X,approach,X,T1,T2,T2).

contr(X,in,X,T1,T2,T2).

contr(X,up,X,T1,T2,T2).

contr(X,down,X,T1,T2,T2).

contr(X,exit,X,T1,T2,T2).

gate(s0,lower,s1,T1,T2,T3) :-
  \{ T3 = T1 \}.

gate(s0,lower,s1,T1,T2,T3) :-
  \{ T3 = T1 \}.

gate(s1,down,s2,T1,T2,T3) :-
  \{ T3 = T2, T1 - T2 < 1 \}.

gate(s2,raise,s3,T1,T2,T3) :-
  \{ T3 = T1 \}.

gate(s3,up,s0,T1,T2,T3) :-
  \{ T3 = T2, T1 - T2 > 1, T1 - T2 < 2 \}.

gate(X,approach,X,T1,T2,T2).

gate(X,in,X,T1,T2,T2).

gate(X,up,X,T1,T2,T2).

gate(X,exit,X,T1,T2,T2).

gate(X,exit,X,T1,T2,T2).

gate(X,exit,X,T1,T2,T2).

driver(S0,S1,S2,T,T0,T1,T2,[X|Rest],[[X,T]|R]) :-
  train(S0,X,S00,T,T0,T00),
  contr(S1,X,S10,T,T1,T10),
  gate(S2,X,S20,T,T2,T20),
  \{ TA > T \},
  driver(S0,S10,S20,T,TA,T00,T10,T20,Rest,R).
```
Given the query:

\[ \text{?- driver(s0, s0, s0, T, Ta, Tb, Tc, X, R).} \]

We obtain the following infinite lists as answers (A, B, C, .. is the time on the wall clock when the corresponding event occurs).

\[ R = [(\text{approach}, A), (\text{lower}, B), (\text{down}, C), (\text{in}, D), (\text{out}, E),
\text{(exit}, F), (\text{raise}, G), (\text{up}, H)] \mid R \]

\[ X = [\text{approach, lower, down, in, out, exit, raise, up} \mid X] \ ? ; \]

\[ R= [(\text{approach}, A), (\text{lower}, B), (\text{down}, C), (\text{in}, D), (\text{out}, E),
\text{(exit}, F), (\text{raise}, G), (\text{approach}, H), (\text{up}, I)] \mid R \]

\[ X = [\text{approach, lower, down, in, out, exit, raise, approach, up} \mid X] \ ? ; \]

\[ \text{no} \]

A call to coinductively defined sublist/2 predicate can then be used to check the safety property that the signal \text{down} occurs before \text{in} by checking that the infinite list \( Y = [\text{down, in} \mid Y] \) is coinductively contained in the infinite string \( X \) above. This ensures that the system satisfies the safety property, namely, that the gate is down before the train is in the gate area. A similar approach can be used to verify the liveness property, namely that the gate will eventually go up [9], by finding the maximum difference between the times the gate goes down and later comes up.