Co-Logic Programming: Extending Logic Programming with Coinduction

L. Simon, A. Mallya, A. Bansal, G. Gupta
Department of Computer Science, University of Texas at Dallas.

Abstract. In this paper we present the theory and practice of co-logic programming (co-LP for brevity), a paradigm that combines both inductive and coinductive logic programming. Co-LP is a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent LP, model checking, bisimilarity proofs, etc.

1 Introduction

Recently coinductive logic programming has been introduced as a means of programming with infinite data structures and supporting co-recursion [2] in logic programming [17]. Practical applications of coinductive LP include improved modularization of programs as seen in lazy functional programming languages, rational terms, and model checking. Coinductive LP allows infinite proofs and infinite data structures via a declarative semantics based on greatest fix points and an operational semantics based on the coinductive hypothesis rule that recognizes a co-recursive call and succeeds.

There are problems for which coinductive LP is better suited than traditional inductive LP (not to be confused with ILP, LP systems that deal with machine learning). Conversely, there are problems for which inductive LP is better suited than coinductive LP. But there are even more problems where both coinductive and inductive logic programming paradigms are simultaneously useful. In this paper we examine the combination of coinductive and inductive LP. We christen the new paradigm co-logic programming (co-LP for brevity). Thus, co-LP subsumes both coinductive LP and inductive LP. A combination of inductive and coinductive LP is not straightforward as cyclical nesting of inductive and coinductive definitions results in programs to which proper semantics cannot be given. Co-logic programming combines traditional and coinductive LP by allowing predicates to be optionally annotated as being coinductive; by default, unannotated predicates are interpreted as inductive. In our formulation of co-LP, coinductive predicates can call inductive predicates and vice versa, with the only exception being that no cycles are allowed through alternating calls to inductive and coinductive predicates. This results in a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. In this paper the declarative semantics for co-LP is defined, and a corresponding top-down, goal-directed operational semantics is provided in
terms of alternating SLD and co-SLD semantics and proved equivalent to the declarative semantics.

Applications of co-LP are also discussed. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent logic programming, model checking, bisimilarity proofs, Answer Set Programming (ASP), etc. (not all applications are discussed in this paper due to lack of space). Our work can be thought of as developing a practical and reasonable top-down operational semantics for computing the alternating least and greatest fixed-point of a logic program. We assume that the reader has familiarity with coinduction [2] and coinductive LP [17].

2 Coinductive Logic Programming

Coinductive LP allows one to program with infinite data structures and infinite proofs. Under Coinductive LP, programs such as the one below, which describes infinite streams of binary digits, become semantically meaningful, i.e., not semantically null.

```
bit(0).
bit(1).
bitstream([H|T]) :- bit(H), bitstream(T).
\$- X = [0, 1, 1, 0 | X], bitstream(X).
```

We would like the above query to have a finite derivation and return a positive answer; however, aside from the bit predicate, the least fixed-point (lfp) semantics of the above program is null, and its evaluation using SLD resolution lacks a finite derivation. 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 bitstream(X); yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hypersequent theory [2]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties [17].

In the coinductive LP paradigm the declarative semantics of the predicate bitstream/1 above is given in terms of infinitary Herbrand universe, infinitary Herbrand base, and maximal models (computed using greatest fixed-points). The operational semantics is given in terms of the coinductive hypothesis rule which states that during execution, if the current resolvent $R$ contains a call $C'$ that unifies with a call $C$ encountered earlier, then the call $C'$ succeeds; the new resolvent is $R\theta$ where $\theta = \text{mgu}(C, C')$ and $R'$ is obtained by deleting $C'$ from $R$. With this extension a clause such as $p([1|T]) :- p(T)$ and the query $p(Y)$ will produce an infinite answer $Y = [1|Y]$. Applications of purely coinductive logic programming to fields such as model checking, concurrent logic programming, real-time systems, etc., can also be found in [17]. Note that to implement coinductive LP, one needs to remember in a memo-table (memoize) all the calls made to coinductive predicates.
3 Motivation and Examples

Coinductive LP and inductive LP cannot be naively combined together, as this results in interleaving of least fixed point and greatest fixed point computations. Such programs cannot be given meaning easily. Consider the following program:

:- coinductive p/0. %q/0 is inductive by default.
   p :- q.
   q :- p.

For computing the result of goal ?- q., we’ll use lfp semantics, which will produce null, implying that q should fail. Given the goal ?- p. now, it should also fail, since p calls q. However, if we use gfp semantics (and the coinductive hypothesis computation rule), the goal p should succeed, which, in turn, implies that q should succeed. Thus, naively mixing coninduction and induction leads to contradictions.

We resolve this contradiction by disallowing such cyclical nesting of inductive and coinductive predicates. Thus, inductive and coinductive predicates can be used in the same program as long as the program is stratified w.r.t. inductive and coinductive predicates. That is, an inductive predicate in a given strata cannot call a coinductive predicate in a higher strata and vice versa.

Next, we illustrate co-LP via more examples.

Infinite Streams: The following example involves a combination of an inductive predicate and a coinductive predicate. By default, predicates are inductive, unless indicated otherwise. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers. Note that only the stream/1 predicate is coinductive, while the number/1 predicate is inductive.

:- coinductive stream/1.
stream( [ H | T ] ) :- number( H ), stream( T ).
number( 0 ).
number( s(N) ) :- number( N ).
| ?- stream( [ 0, s(0), s(s(0)) | T ] ).

The following is an execution trace, for the above query, of the memoization of calls by the operational semantics. Note that calls of number/1 are not memo’ed because number/1 is inductive.
MEMO: stream( [ 0, s(0), s(s(0)) | T ] )
MEMO: stream( [ s(0), s(s(0)) | T ] )
MEMO: stream( [ s(s(0)) | T ] )

The next goal call is stream(T), which unifies with the first memo’ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with the infinite solution:
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 two matching memo’ed ancestor producing T = [s(0), s(s(0)) | T] & T = [s(s(0)) | T] respectively. If no remaining memo’ed elements exist, the goal is memo’ed, and expanded using the
coinductively defined clauses, and the process repeats—generating additional results, and effectively enumerating the set of (rational) infinite lists of natural numbers that begin with the prefix \([0, s(0), s(s(0))])\.

The goal \(\text{stream}(T)\) is true whenever \(T\) is some infinite list of natural numbers. If \(\text{number}/1\) was also coinductive, then \(\text{stream}(T)\) would be true whenever \(T\) is a list containing either natural numbers or \(\omega\), i.e., infinity, which is represented as an infinite application of successors \(s(s(s(\ldots)))\). Such a term has a finite representation as \(X = s(X)\).

Note that excluding the occurs check is necessary as such structures have a greatest fixed-point interpretation and are in the co-Herbrand Universe. This is in fact one of the benefits of co-LP. Unification without occurs check is typically more efficient than unification with occurs check, and now it is even possible to define non-trivial predicates on the infinite terms that result from such unification, which are not definable in LP with rational trees. Traditional logic programming's least Herbrand model semantics requires SLD resolution to unify with occurs check (or lack soundness), which adversely affects performance in the common case. Co-LP, on the other hand, has a declarative semantics that allows unification without doing occurs check, and it also allows for non-trivial predicates to be defined on infinite terms resulting from such unification.

**List Membership:** This example illustrates that some predicates are naturally defined inductively, while other predicates are naturally defined coinductively. The \(\text{member}/2\) predicate is an example of an inherently inductive predicate.

\[
\begin{align*}
\text{member}( H, [ H | - ] ). \\
\text{member}( H, [ _ | T ] ) & : - \text{member}( H, T ).
\end{align*}
\]

If this predicate was declared to be coinductive, then \(\text{member}( X, L )\) is true whenever \(X\) is in \(L\) or whenever \(L\) is an infinite list, even if \(X\) is not in \(L\) ! The definition above, whether declared coinductive or not, states that the desired element is the last element of some prefix of the list, as the following equivalent reformulation of \(\text{member}/2\), called \(\text{membera}/2\) demonstrates, where \(\text{drop}/3\) drops a prefix ending in the desired element and returns the resulting suffix.

\[
\begin{align*}
\text{membera}( X, L ) & : - \text{drop}( X, L, _ ). \\
\text{drop}( H, [ H | T ], T ) . \\
\text{drop}( H, [ _ | T ], T1 ) & : - \text{drop}( H, T, T1 ).
\end{align*}
\]

When the predicate is inductive, this prefix must be finite, but when the predicate is declared coinductive, the prefix may be infinite. Since an infinite list has no last element, it is trivially true that the last element unifies with any other term. This explains why the above definition, when declared to be coinductive, is always true for infinite lists regardless of the presence of the desired element.

A mixture of inductive and coinductive predicates can be used to define a variation of \(\text{member}/2\), called \(\text{comember}/2\), which is true if and only if the desired element occurs an infinite number of times in the list. Hence it is false when the element does not occur in the list or when the element only occurs a finite number of times in the list. On the other
hand, if \texttt{comember/2} was declared inductive, then it would always be false. Hence coinduction is a necessary extension.

\begin{verbatim}
:- coinductive comember/2.
  comember( X, L ) :- drop( X, L, L1 ), comember( X, L1 )..
?- X = [ 1, 2, 3 | X ], comember( 2, X ).
  Answer: yes.
?- X = [ 1, 2, 3, 1, 2, 3 ], comember( 2, X ).
  Answer: no.
?- X = [ 1, 2, 3 | X ], comember( Y, X ).
  Answer: Y = 1;
    Y = 2;
    Y = 3;
\end{verbatim}

Note that \texttt{drop/3} will have to be evaluated using OLDT tabling for it not to go into an infinite loop for inputs such as $X = [1,2,3|X]$ (if $X$ is absent from the list $L$, the lfp of \texttt{drop(X,L)} is null).

**Sieve of Eratosthenes:** Co-\texttt{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 \texttt{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 coroutines 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:

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

In the above program \texttt{filter/3} removes all multiples of the first element in the list, and then passes the filtered list recursively to \texttt{sieve/2}. If the call \texttt{generate_infinite_list(I)} binds $I$ to an inductive or rational list (e.g., $X = [2, \ldots, 20]$ or $X = [2, \ldots, 20 | X]$), then filter can be completely processed in each call to \texttt{sieve/2}. However, in contrast, if $I$ is bound to an irrational infinite list as in:

\begin{verbatim}
:- coinductive int/2.
  int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
  generate_infinite_list(I) :- int(2,I).
\end{verbatim}

then in \texttt{primes/1} predicate, the calls \texttt{generate_infinite_list/1, comember/2}, and \texttt{sieve/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.
4 Syntax and Semantics

While co-logic programming is based on the very simple concept of coinduction, the previous examples show that it is an extremely elegant and powerful paradigm. Section 6 shows yet another elegant application of co-logic programming (to model checking). Next, we present the declarative and operational semantics of co-logic programming and prove that they are equivalent.

4.1 Syntax

A co-logic program $P$ is syntactically identical to a traditional, that is, inductive logic program. 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 co-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 1. A pre-program is a definite program paired with a mapping of predicate symbols to the token coinductive or the token inductive. A predicate is coinductive (resp. inductive) if the partial mapping maps the predicate to coinductive (resp. inductive). Similarly, an atom is coinductive (resp. inductive) if the underlying predicate is coinductive (resp. inductive).

Not every pre-program is a co-logic program. Co-LP programs do not allow for any pair of inductive and coinductive predicates to be mutually recursive, that is, programs must be stratified with regards to alternating induction and coinduction. An inductive predicate can call, directly or indirectly, a coinductive predicate and visa versa, but both such predicates cannot be mutually recursive.

Definition 2. In some pre-program $P$, we say that a predicate $p$ depends on a predicate $q$ if and only if $p = q$ or $P$ contains a clause $C \leftarrow D_1, \ldots, D_n$ such that $C$ contains $p$ and some $D_i$ contains $q$. The dependency graph of program $P$ has the set of its predicates as vertices, and the graph has an edge from $p$ to $q$ if and only if $p$ depends on $q$.

Co-LP programs are simply pre-programs that obey the stratification restriction.

Definition 3. A co-logic program is a pre-program such that for any strongly connected component $G$ in the dependency graph of the program, every predicate in $G$ is either mapped to coinductive or every predicate in $G$ is mapped to inductive.
4.2 Declarative Semantics

The declarative semantics of a co-logic program is a stratified interleaving of the minimal Herbrand model [1, 12] and the maximal co-Herbrand model semantics [17]. Hence, co-LP strictly contains logic programming with rational trees [10] as well as coinductive logic programming [17]. This allows the universe of terms to contain infinite terms, in addition to the traditional finite terms. Finally, co-LP also allows for the model to contain ground goals that have either finite or infinite proofs. Note that stratification is necessary because programs that cyclically interleave inductive and coinductive predicates cannot be given a meaning easily, as explained earlier.

The following definition is necessary for defining the model of a co-logic program. Intuitively, a reduced graph is derived from a dependency graph by collapsing the strongly connected components of the dependency graph into single nodes. The graph resulting from this process is acyclic.

**Definition 4.** The reduced graph for a co-logic program has vertices consisting of the strongly connected components of the dependency graph of $P$. There is an edge from $v_1$ to $v_2$ in the reduced graph if and only if some predicate in $v_1$ depends on some predicate in $v_2$. A vertex in a reduced graph is said to be coinductive (resp. inductive) if it contains only coinductive (resp. inductive) predicates.

A vertex in a reduced graph of a program $P$ is called a stratum, as the set of predicates in $P$ is stratified into a collection of mutually disjoint strata of predicates. The stratification restriction states that all vertices in the same stratum are of the same kind, i.e., every stratum is either inductive or coinductive. A stratum $v$ depends on a stratum $v'$, when there is an edge from $v$ to $v'$ in the reduced graph. When there is a path in the reduced graph from $v$ to $v'$, $v$ is said to be higher than $v'$ and $v'$ is said to be lower than $v$, and the case when $v \neq v'$ is delineated by the modifier “strictly”, as in “strictly higher” and “strictly lower”. This restriction allows for the model of a stratum $v$ to be defined in terms of the models of the strictly lower strata, upon which $v$ depends. However, before we can define the model of a stratum, we must define a few basic sets. In the description below, $\mu$ (resp. $\nu$) is the least (resp. greatest) fixed point operator.

**Definition 5.** 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 $U^{\omega}(P) = \nu \Phi_P$, 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 $\mu \Phi_P$. 
Definition 6. Let \( P \) be a definite program. The co-Herbrand base also known as the infinitary Herbrand base, written \( B^\infty(P) \), is the set of all ground atoms that can be formed from the atoms in \( P \) and the elements of \( U^\infty(P) \). Also, let \( G^\infty(P) \) 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 \( C, D_1, \ldots, D_n \in B^\infty(P) \).

Now we can define the model of a stratum, i.e., the model of a vertex in the reduced graph of a co-logic program. The model of each stratum is defined using what is effectively the same \( T_P \) monotonic operator used in defining the minimal Herbrand model [1,12], except that it is extended so that it can treat the atoms defined as true in lower strata as facts when proving atoms containing predicates in the current stratum. This is possible because co-logic programs are stratified such that the reduced graph of a program is always a DAG and every predicate in the same stratum is the same kind: inductive or coinductive.

Definition 7. The model of a stratum \( v \) of \( P \) equals \( \mu T^v_P \) if \( v \) is inductive and \( \nu T^v_P \) if \( v \) is coinductive, such that \( R \) is the union of the models of the strata that are strictly lower than \( v \) and

\[
T^v_P(S) = R \cup \left\{ q \left( \tilde{t} \right) \mid q \in v \land [q \left( \tilde{t} \right) \leftarrow \tilde{D}] \in G^\infty(P) \land \tilde{D} \in S \right\}
\]

Since any predicate resides in exactly one stratum, the definition of the model of a co-logic program is straightforward.

Definition 8. The model of a co-logic program \( P \), written \( M(P) \), is the union of the model of every stratum of \( P \).

Obviously co-LP’s semantics subsumes the minimal co-Herbrand model used as the semantics for logic programming with rational trees, as well as the maximal co-Herbrand model used as the semantics for coinductive logic programming.

Definition 9. An atom \( A \) is true in program \( P \) if and only if the set of all groundings of \( A \) with substitutions ranging over the \( U^\infty(P) \), is a subset of \( M(P) \).

Example 1. Let \( P_1 \) be the following program.

\[
\begin{align*}
\text{:- coinductive from/2.} \\
\text{from}(N, [N|T]) & \text{ :- from}(s(N), T).} \\
\text{! ?- from}(0, \_, ).
\end{align*}
\]

The model of the program, which is defined in terms of an alternating fixed-point is as follows. The co-Herbrand Universe is \( U^\infty(P_1) = N \cup \Omega \cup L \) where \( N = \{0, s(0), s(s(0)), \ldots\} \), \( \Omega = \{s(s(s(\ldots)))\} \), and \( L \) is the set of all finite and infinite lists of elements in \( N, \Omega \), and even \( L \). Therefore the model \( M(P_1) = \{\text{from}(t, [t, s(t), s(s(t)), \ldots]) \mid t \in U^\infty(P_1)\} \), which is the meaning of the program and is obviously
not null, as was the case with traditional logic programming. Furthermore, \(\text{from}(0, [0, s(0), s(s(0)), \ldots]) \in M(P_1)\) implies that the query returns “yes”. On the other hand, if the directive on the first line of the program was removed, call the resulting program \(P_1^*\), then the program’s only predicate would by default be inductive, and \(M(P_1^*) = \emptyset\). This corresponds to the traditional semantics of logic programming with infinite trees. Examples involving multiple strata of different kinds, i.e., mixing inductive and coinductive predicates, are given in sections 3 and 6.

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 4.4, we will need to talk about the manner 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 a least fixed-point, while it can be applied an infinite number of times in the greatest fixed-point. We capture this by recording the application of the generator in the elements of the fixed-point itself. We call these objects “idealized proofs.” In order to define idealized proofs, it is first necessary to define some formalisms for trees.

**Definition 10.** A path \(\pi\) is a finite sequence of positive integers \(i\). The empty path is written \(\varepsilon\), the singleton path is written \(i\) for some positive integer \(i\), and the concatenation of two paths is written \(\pi \cdot \pi'\). A tree of \(S\), also called an \(S\)-tree, is formally defined as a partial function from paths to elements of \(S\), such that the domain is non-empty and prefix-closed. A node in a tree is unambiguously denoted by a path. So a tree \(t\) is described by the paths \(\pi\) from the root \(t(\varepsilon)\) to the nodes \(t(\pi)\) of the tree, and the nodes of the tree are labeled with elements of \(S\).

A child of node \(\pi\) in tree \(t\) is any path \(\pi \cdot i\) that is in the domain of \(t\), where \(i\) is some positive integer. If \(\pi\) is in the domain of \(t\), then the subtree of \(t\) rooted at \(\pi\), written \(t \setminus \pi\), is the partial function \(t'(\pi') = t(\pi \cdot \pi')\). Also, node \((L, T_1, \ldots, T_n)\) denotes a constructor of an \(S\)-tree with root labeled \(L\) and subtrees \(T_i\), where \(L \in S\) and each \(T_i\) is an \(S\)-tree, such that \(1 \leq i \leq n\), node \((L, T_1, \ldots, T_n)(\varepsilon) = L\), and node \((L, T_1, \ldots, T_n)(i \cdot \pi) = T_i(\pi)\).

Idealized proofs are trees of ground atoms, such that a parent is deduced from the idealized proofs of its children.

**Definition 11.** The set of idealized proofs of a stratum of \(P\) equals \(\Sigma^\nu_P\) if \(\nu\) is inductive and \(\nu^\nu_P\) if \(\nu\) is coinductive, such that \(R\) is the union of the sets of idealized proofs of the strata strictly lower than \(\nu\) and

\[
\Sigma^\nu_P(S) = R \cup \{\text{node}(q \left( \hat{t} \right), T_1, \ldots, T_n) \mid q \in \nu \land T_i \in S \land [q \left( \hat{t} \right) \leftarrow D_1, \ldots, D_n] \in G^\nu(P) \land T_i(\varepsilon) = D_i\}
\]

Note that these definitions mirror the definitions defining models, with the exception that the elements of the sets record the application of the program clauses as a tree of atoms.

**Definition 12.** The set of idealized proofs generated by a co-logic program \(P\), written \(\Sigma_P\), is the union of the sets of idealized proofs of every stratum of \(P\).
Again, this is nothing more than a reformulation of $M(P)$, which records the applications of the generator in the elements of the fixed-points, as the following theorem demonstrates.

**Theorem 1.** Let $S = \{A \mid \exists T \in \Sigma_p. A \text{ is the root of } T\}$, then $S = M(P)$.

Hence any element in the model has an idealized proof and anything that has an idealized proof is in the model. This formulation of the declarative semantics in terms of idealized proofs will be used in section 4.4 in order to distinguish between the case when a query has a finite derivation, from the case when there are only infinite derivations of the query, in the operational semantics.

### 4.3 Operational Semantics

This section defines the operational semantics for co-LP. This requires some infinite tree theory. However, this section only states a few definitions and theorems without proof. The details of infinite tree theory can be found in [6].

The operational semantics given for co-LP is defined as an interleaving of SLD [12] and co-SLD [17]. 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 forest is simply the set of leaves of the forest. Furthermore, where SLD only allows program clauses as state transition rules, co-SLD also allows a special coinductive hypothesis rule for proving coinductive atoms [17]. The coinductive hypothesis rule allows us to compute the greatest fixed-point. Intuitively, it states that if, during execution, a call $C$ is encountered that unifies with an ancestor call $A$ that has been seen before, then $C$ is deleted from the resolvent and the substitution resulting from unification of $C$ and $A$ imported into the resolvent. Thus, given the clause:

\[
p : - p
\]

then the query $\mathcal{Q}$ ?- $p$. succeeds by the coinductive hypothesis rule (indeed $\{p\}$ is the gfp of this program). Likewise, given the clause:

\[
p([1|T]) : - p(T).
\]

then the query $\mathcal{Q}$ ?- $p(X)$. succeeds by the coinductive hypothesis rule with solution $X = [1 | X]$ (note that indeed $\{p([1, \ldots])\}$, where $[1, \ldots]$ denotes an infinite list of 1's, is the gfp of the program).

**Definition 13.** 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 2.** (Courcelle) Every system of equations has a mgu that is rational.

**Theorem 3.** (Courcelle) For every rational substitution $\sigma$ with domain $V$, there is a system of equations $E$, such that the most general unifier $\sigma'$ of $E$ is equal to $\sigma$ 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. 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 the part of the state involving unification. The current state of the pending goals is modeled using a forest of finite trees of atoms, as it is necessary to be able to recognize infinite proofs, for coinductive queries. However, an implementation that uses a policy of executing goals in the current resolvent from left to right (as in standard LP), only needs a single stack.

**Definition 14.** A state $S$ is a pair $(F,E)$, where $F$ is a finite multiset of finite trees, i.e., a forest of syntactic atoms, and $E$ is a system of equations.

**Definition 15.** A transition rule $R$ of a co-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 coinductive hypothesis rule of the form $\nu(\pi, \pi')$, where $\pi$ and $\pi'$ are both paths, such that $\pi$ is a proper prefix of $\pi'$.

Before we can define how a transition rule affects a state, we must define how a tree in a state is modified when an atom is proved to be true. This is called the unmemo function, and it removes memo'd atoms that are no longer necessary. Starting at a leaf of a tree, the unmemo function removes the leaf and the maximum number of its ancestors, such that the result is still a tree. This involves iteratively 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. When all nodes in a tree are removed, the tree itself is removed.

**Definition 16.** The unmemo function $\delta$ takes a tree of atoms $T$, a path $\pi$ in the domain of $T$, and returns a forest. Let $\rho(T, \pi \cdot i)$ be the partial function equal to $T$, except that it is undefined at $\pi \cdot i$. $\delta(T, \pi)$ is defined as follows:

\[
\begin{align*}
\delta(T, \pi) &= \{T\}, \text{ if } \pi \text{ has children in } T \\
\delta(T, \epsilon) &= \emptyset, \text{ if } \epsilon \text{ is a leaf in } T \\
\delta(T, \pi) &= \delta(\rho(T, \pi), \pi'), \text{ if } \pi = \pi' \cdot i \text{ is a leaf in } T
\end{align*}
\]

The intuitive explanation of the following definition is that (1) a state can be transformed by applying the coinductive hypothesis rule $\nu(\pi, \pi')$, whenever in some tree, $\pi$ is a proper ancestor of $\pi'$, such that the two atoms unify. Also, (2) a state can be transformed by applying an instance of a definite clause from the program. In either case, when a subgoal has
been proved true, the forest is pruned so as to remove unneeded memos. Also, note that the body of an inductive clause is overwritten on top of the leaf of a tree, as an inductive call need not be memo’ed, since the coinductive hypothesis rule can never be invoked on a memo’ed inductive predicate. When the leaf of the tree is also the root, this causes the old tree to be replaced with, one or more singleton trees. Coinductive subgoals, on the other hand, need to be memo’ed, in the form of a forest, so that infinite proofs can be recognized.

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 (implementations typically use backtracking to realize nondeterministic execution). We write \( S - x \) to denote the multi-set obtained by removing an occurrence of \( x \) from \( S \).

**Definition 17.** Let \( T \in F \). A state \((F, E)\) transitions to another state \(((F - T) \cup F', E')\) by transition rule \( R \) of program \( P \) whenever:

1. \( R \) is an instance of the coinductive hypothesis rule of the form \( v(\pi, \pi') \), \( p \) is a coinductive predicate, \( \pi \) is a proper prefix of \( \pi' \), which is a leaf in \( T \), \( T(\pi) = p(t_1, \ldots, t_n) \), \( T(\pi') = p(t_1, \ldots, t_n) \), \( E' \) is the most general unifier for \( \{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E \), and \( F' = \delta(T, \pi') \).
2. \( R \) is a definite clause of the form
   \[
   p(t'_1, \ldots, t'_n) \leftarrow B_1, \ldots, B_m, \pi \text{ is a leaf in } T, \ T(\pi) = p(t_1, \ldots, t_n), \ E' \text{ is the most general unifier for } \{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E, \text{ and the set of trees of atoms } F' \text{ is obtained from } T \text{ according to the following case analysis of } m \text{ and } p:
   \]
   (a) Case \( m = 0 \): \( F' = \delta(T, \pi) \).
   (b) Case \( m > 0 \) and \( p \) is coinductive: \( F' = \{T'\} \) where \( T' \) is equal to \( T \) except at \( \pi \cdot i, & T'(\pi \cdot i) = B_i \), for \( 1 \leq i \leq m \).
   (c) Case \( m > 0 \) and \( p \) is inductive: If \( \pi = \epsilon \) then \( F' = \{\text{node}(B_i) \mid 1 \leq i \leq m\} \). Otherwise, \( \pi = \pi' \cdot j \) for some positive integer \( j \). Let \( T'' \) be equal to \( T \) except at \( \pi' \cdot k \) for all \( k \), where \( T'' \) is undefined. Finally, \( F' = \{T''\} \) where \( T' \) is equal to \( T'' \) except at \( \pi' \cdot i, \) where \( T'(\pi' \cdot i) = B_i \), for \( 1 \leq i \leq m \), and \( T'(\pi' \cdot (m + k)) = T(\pi' \cdot k) \), for \( k \neq j \).

**Definition 18.** 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 is an execution trace. Execution halts when it reaches a terminal state: either all atoms have been proved or a dead-end reached.

**Definition 19.** The following are two distinguished terminal states:

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

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 20.** A derivation of a state \((F, E)\) in program \(P\) is a state transition sequence with the first state equal to \((F, E)\). 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 list of syntactic atoms \(A_1, \ldots, A_n\), also called a goal or query, has a derivation in program \(P\), if \(\{\text{node}(A_i) \mid 1 \leq i \leq n\}, \emptyset\) has a derivation in \(P\).

An implementation will use backtracking search to find successful derivations.

### 4.4 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 7 mentions an extension of the operational semantics, so as to improve its completeness. Note that proofs are relegated to Appendix I.

**Theorem 4.** (soundness) If the query \(A_1, \ldots, A_n\) has a successful derivation in program \(P\), then \(E(A_1, \ldots, A_n)\) is true in program \(P\), where \(E\) is the resulting variable bindings for the derivation.

**Theorem 5.** (completeness) Let \(A_1, \ldots, A_n \in M(P)\). If each \(A_1, \ldots, A_n\) has a rational idealized proof, then the query \(A_1, \ldots, A_n\) has a successful derivation in program \(P\).

### 5 Related Work

Most past work is focused on allowing infinite data structures in LP. However, they all stop short of including infinite proofs. LP with rational trees [4, 5, 10] allows for finite terms as well as infinite terms that are rational trees, that is, terms that have finitely many distinct subterms. Co-LP as defined in Section 4, on the other hand, allows for finite terms, rational infinite terms, but unlike LP with rational trees, co-LP also allows for irrational infinite terms. Furthermore, the declarative semantics of LP with rational trees corresponds to the minimal co-Herbrand model while co-LP's declarative semantics is the stratified alternating fixed-point co-Herbrand model, which strictly contains the minimal 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 co-LP interleaves SLD with rational term unification and co-SLD, thus containing both SLD and co-SLD. Hence, LP with rational trees does not allow for infinite proofs while co-LP does. Finally, LP with rational trees can only create infinite terms via unification (without occurs check), while
co-LP can create infinite terms via unification (without occurs check) as well as via user-defined co-recursive clauses.

Jaffar et al’s coinductive tabling proof method [9] 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 co-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 in [17], 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 formal declarative, model-theoretic semantics, as is the case with coinductive logic programming presented in this paper, which has a declarative semantics, an operational semantics, and a correctness proof showing the correspondence between the two. Co-LP, when extended with constraints, can be used (in fact, much more elegantly) for the same applications as Jaffar et al’s coinductive tabling proof method [17].

Lazy functional LP (e.g., [7, 8]) also allows for infinite data structures, but it encodes predicates as Boolean functions, while in comparison, co-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 such 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 co-LP, on the other hand, a finite derivation is possible. Therefore, predicates in lazy functional logic programming are semantically different from those in co-LP.

The Horn $\mu$-calculus of Charatonik et al. [3] and the alternation-free variant of Talbot [18] extend Horn logic with least and greatest fixed-points and provide declarative semantics in a manner similar to co-LP. The Horn $\mu$-calculus does not have the stratification restriction found in co-LP, while the alternation-free restriction does. The Horn $\mu$-calculus requires predicate symbols to be labeled with an integer priority, which is used in the definition of the language’s semantics. Co-LP does not require predicates to have a specified priority, which is semantically unambiguous thanks to the stratification restriction. Aside from respectively defining the syntax and semantics for these variants of the Horn $\mu$-calculus, both Charatonik et al. and Talbot only consider a non-Turing-complete restriction to “uniform” programs. These uniform programs are then used to model reactive systems. Hence Charatonik et al. and Talbot don’t provide a top-down, goal-directed operational semantics, let alone an efficient implementation for the full Horn $\mu$-calculus or the full alternation-free Horn $\mu$-calculus; they only consider the restricted class of uniform programs as a means of statically analyzing reactive systems, not as a general purpose programming language. Co-logic programming has a much more ambitious goal of unifying seemingly disparate logic programming concepts
into a simple and efficient general purpose declarative programming language.

Coinductive logic programming, developed by us earlier [17], does not allow the definition of strictly inductive predicates as found in traditional logic programming, let alone the mixing of inductive and coinductive predicates as found in co-LP. Hence, co-LP strictly subsumes coinductive logic programming of [17].

6 Applications
A major application of co-LP is in model checking. It is well known that reachability-based inductive techniques are not suitable for verifying liveness properties [14]. Further, it is also well known that, in general, verification of liveness properties can be reduced to verification of termination under the assumption of fairness [19] and that fairness properties can be specified in terms of alternating fixed-point temporal logic formulas [11]. In [17] we showed that co-inductive LP allows one to verify a class of liveness properties in the absence of fairness constraints. Co-LP further permits us to verify a more general class of all liveness properties that can only be verified in the presence of fairness constraints.

Essentially, a co-LP based approach demonstrates that if a model satisfies the fairness constraint then, it also satisfies the liveness property. This is achieved by composing a program $P_M$, which encodes the model, with a program $P_F$, which encodes the fairness constraint and a program $P_N$, which encodes the negation of the liveness property, to obtain a composite program $P_\mu$. We then compute the stratified alternating fixed-point of the logic program $P_\mu$ as described in section 4.2 and check for the presence of the initial state of the model in the stratified alternating fixed-point. If the alternating fixed-point contains the initial state, then that implies the presence of a valid counterexample that violates the given liveness property. On the other hand, if the alternating fixed-point is empty, then that implies that no counterexample can be constructed, which in turn implies that the model satisfies the given liveness property.

We will now illustrate our approach using a very simple example. Consider the model shown in Figure 1, consisting of four states. The system starts off in state $s_0$, enters state $s_1$, performs a finite amount of work in state $s_1$ and then exits to state $s_2$, from where it transitions back to state $s_0$, and repeats the entire loop again, an infinite number of times. The system might encounter an error, causing a transition to state $s_3$; corrective action is taken, followed by a transition back to $s_0$ (this can also happen infinitely often). The system is modeled by the Prolog code shown in Figure 1.

This simple example illustrates the power of co-LP when compared to purely inductive or coinductive LP. Note that the computation represented by the state machine in the example consists of two stratified loops, represented by recursive predicates. The outer loop (predicate state/2) is coinductive and represents an infinite computation (hence it is declared as coinductive as we are interested in its gfp). The inner loop (predicate work/0) is inductive and represents a bounded computation (we are interested in its lfp). The semantics therefore evaluates work/0 using SLD resolution and state/2 using co-SLD resolution.
The property that we would like to verify is that the computation in the state \( s_1 \) represented by \texttt{work} always terminates. In order to do so, we require the fairness property: “if the transition \texttt{enter} occurs infinitely often, then the transition \texttt{exit} also occurs infinitely often”. The stratified alternating fixed-point semantics ensures that this fairness constraint holds by computing the minimal model of the inductive program represented by the predicate \texttt{state/1} and then composing it with the coinductive program. The resulting program is then composed with the property, “the state \( s_2 \) is not present in any trace of the infinite computation,” which is the negation of the given liveness property. The negated property is represented by the predicate \texttt{absent/2}. Thus, given the program above, the user will pose the query:

\[
\text{? :- state(s0, X), absent(s2, X).}
\]

where \texttt{absent/2} is a coinductive predicate that checks that the state \( s_2 \) is not present in the (infinite) list \( X \) infinitely often (it is the negated version of the coinductive member predicate described earlier). The co-LP system will respond with a solution: \( X = [s0, s3 \mid X] \), a counterexample which states that there is an infinite path not containing \( s_2 \). One can see that this corresponds to the (infinite) behavior of the system if \texttt{error} is encountered.

7 Conclusions and Future Work

In this paper we presented a comprehensive theory of co-LP and demonstrated its practical applications. A prototype implementation of co-LP is being developed by modifying the YAP Prolog system [16]. Current work also includes extending co-LP’s operational semantics with alternating ODLT and co-SLD, so that the operational behavior of inductive predicates can be made to more closely match their declarative semantics. Current work also involves extending the operational semantics of co-LP 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.

Acknowledgments: We are grateful to Vítor Santos Costa and Ricardo Rocha for help with YAP, and Srividya Kona for comments.
References

Appendix I: Correctness Proof

This section proves the correctness of the operational semantics by demonstrating a correspondence between the declarative and operational semantics via soundness and completeness theorems. Completeness, however, must be restricted to atoms that have a rational proof, as termination cannot be guaranteed for atoms with only irrational proofs. Section 7 discusses an extension of the operational semantics, so as to improve its completeness.

**Lemma 1.** If \((A, E_1)\) has a successful derivation in program \(P\), with final state \((\emptyset, E_2)\), then \((A, E_3)\) has a successful derivation of the same length, in program \(P\), where \(E_2 \subseteq E_3\), with each state of the derivation of the form \((F, E_3)\) for some forest \(F\) of finite trees of atoms.

**Proof.** Let \((A, E_1)\) have a successful 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 derivation in program \(P\), where \(E_2 \subseteq E_3\), with each state of the derivation of the form \((F, E_3)\) for some forest \(F\) of finite trees of atoms.

**Lemma 2.** If \(A\) has a successful derivation in program \(P\), which first transitions to the second state 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 \(A\) have a successful derivation in program \(P\), which first transitions to the next state 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 (sub)tree rooted at \(B_i\) in the original derivation, except for the transitions which are of the form \(\nu(\pi, \pi')\), which are no longer correct derivations because the parent of \(B_i\), that is, \(A\), no longer exists. In the case that \(\pi = i \cdot \pi_0\) and \(\pi' = i \cdot \pi_1\) for some number \(i\) and path \(\pi_0\), instead apply the transition rule \(\nu(\pi_0, \pi_1)\) to the corresponding leaf to which the original derivation would have applied \(\nu(\pi, \pi')\). Otherwise, when \(\pi = \epsilon\), a coinductive transition rule cannot be applied to the corresponding leaf. Instead, recursively mimic the transitions of the entire original derivation of \(A\).

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

**Proof.** Since the model of \(P\) consists of the union of the models of the strata of \(P\), it is sufficient to show that if \((A, E)\) has a successful derivation in program \(P\), then all groundings of \(E'(E(A))\) are included in the model for the stratum in which \(A\) resides.

The proof proceeds by induction on the height of the strata of \(P\). Let \(Q_v\) be the set of all groundings ranging over the \(U^\omega(P)\) of all such \(E'(E(A))\) that are either in the same stratum \(v\) or some lower stratum.
We prove by induction on the height of \( v \) that \( Q_v \) is contained in the model for \( v \).

Consider the case when the stratum \( v \) is coinductive. We show that \( Q_v \subseteq \mu T_P^v \). The proof proceeds by coinduction. Let \( A' \in Q_v \), 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 \( E = E_1 \cup E_2 \cup E_3 \). The case when \( E(A) \) unifies with a fact is trivial, and the case when \( E(A) \) is in a lower stratum than \( v \) follows by induction on the height of strata. Now, consider the case where the derivation begins 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 \( C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^\text{co}(P) \), 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 the stratification restriction implies that each \( E''(B_i) \) is in a stratum equal to or lower than \( v \). Hence \( E''(B_i) \in Q_v \). Therefore, by the principle of coinduction, \( Q_v \subseteq \mu T_P^v \).

Now consider the case when the stratum \( v \) is inductive. It is sufficient to prove that \( Q_v \subseteq \mu T_P^v \). Let \( A' \in Q_v \), 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 \( E = E_1 \cup E_2 \cup E_3 \). The case when \( A' \) is in a lower stratum than \( v \) follows by induction. Now consider the case when \( A' \) does not occur in a stratum lower than \( v \), that is, \( A' \) is an inductive atom. The proof proceeds by induction on the length of the derivation \( A' \). When the derivation consists of just one transition, then \( A' \) unifies with a fact \( A \) in program \( P \), and hence \( A' \in \mu T_P^v \). Finally, consider the case when the derivation is of length \( k > 1 \). Then the derivation begins with an application of a program clause \( A'' \leftarrow B_1, \ldots, B_n \), 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 \( C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^\text{co}(P) \), 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 of length \( k' < k \), and the stratification restriction implies that each \( E''(B_i) \) is in a stratum equal to or lower than \( v \). If \( E''(B_i) \) is in a strictly lower stratum than \( v \), then by induction on strata \( E''(B_i) \in \mu T_P^v \), and if \( E''(B_i) \) is not in a lower stratum, then since it has a derivation of length \( k' < k \), by induction on the length of the derivation, \( E''(B_i) \in \mu T_P^v \). Therefore, \( A' \in \mu T_P^v \).

**Theorem 6.** (soundness) If the query \( A_1, \ldots, A_n \) has a successful derivation in program \( P \), then \( E(A_1, \ldots, A_n) \) is true in program \( P \), where \( E \) is the resulting variable bindings for the derivation.

**Proof.** If the goal \( A_1, \ldots, A_n \) has a successful derivation in program \( P \), with \( E \) as the resulting variable bindings for the derivation, then each \( E(A_i) \) independently has a successful derivation in program \( P \). By lemma 3, each \( E(A_i) \) is true in program \( P \).
Theorem 7. (completeness) Let $A_1, \ldots, A_n \in M(P)$. If each $A_1, \ldots, A_n$ has a rational idealized proof, then the query $A_1, \ldots, A_n$ has a successful derivation in program $P$.

Proof. Without loss of generality, we only consider the case when the query is a single atom, i.e., $n = 1$, as the case for $n = 0$ is trivial and the case for $n > 1$ follows by simply composing the individual derivations of each atom of the original query.

Let $A \in M(P)$ 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 coinductive root $\pi = \pi' \cdot \pi''$ of a subtree that is identical to a subtree rooted at a proper coinductive ancestor $\pi'$. Then the derivation applies a transition rule of the form $\nu(\pi', \pi)$ 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 $T$ is rational implies that the set of all subtrees of $T$ is finite in cardinality. Furthermore, the stratification restriction prevents a depth-first traversal from encountering the same subtree twice along the same path, such that the subtree has an inductive atom at its root. Only subtrees rooted at coinductive atoms can repeat in such a fashion. So the maximum depth of the traversal in the idealized proof tree is finite. Combined with the fact that all idealized proofs are finitely branching, this 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 forest, which is the definition of an accept state. Therefore, $A$ has a successful derivation in program $P$. 