

Co-Logic Programming: Extending Logic Programming with Coinduction

Luke Simon, Ajay Bansal, Ajay Mallya, Gopal Gupta

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

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 [9]. Practical applications of coinductive LP include improved modularization of programs as seen in lazy functional 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. 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] & coinductive LP [9].

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 hyperset theory [2]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties [9].

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 $\theta = mgu(C, C')$ 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 [9]. 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 coinduction 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:

$T = [0, s(0), s(s(0)) \mid T]$

The user could force a failure here, which would cause the goal to be unified with the next two matching memo'ed ancestor producing $T = [s(0), s(s(0)) \mid T]$ & $T = [s(s(0)) \mid T]$ 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 `stream(T)` is true whenever T is some infinite list of natural numbers. If `number/1` was also coinductive, then `stream(T)` would be true whenever T is a list containing either natural numbers or ω , i.e., infinity, which is represented as an infinite application of successor $s(s(s(\dots)))$. 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 `member/2` predicate is an example of an inherently inductive predicate.

```
member( H, [ H | _ ] ).
member( H, [ _ | T ] ) :- member( H, T ).
```

If this predicate was declared to be coinductive, then `member(X, L)` is true (i) whenever X is in L or (ii) 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 `member/2`, called `membera/2` shows, where `drop/3` drops a prefix ending in the desired element and returns the resulting suffix.

```
membera( X, L ) :- drop( X, L, _ ).
drop( H, [ H | T ], T ).
drop( H, [ _ | T ], T1 ) :- drop( H, T, T1 ).
```

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 `member/2`, called `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 `comember/2` was declared inductive, then it would always be false. Hence coinduction is a necessary extension.

```
:- 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;
```

Note that `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 `drop(X,L)` is null). More elaborate examples including application to model checking can be found elsewhere [10].

4 Syntax and Semantics

While co-logic programming is based on the very simple concept of co-induction, the previous examples show that it is an extremely elegant and powerful paradigm. Next, we present the declarative and operational semantics of co-logic programming and prove that they are equivalent.

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 `inductive`. A predicate is `coinductive` (resp. `inductive`) if the partial mapping maps the predicate to `coinductive` (resp. `inductive`). 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: programs must be stratified w.r.t. alternating induction and coinduction. An

inductive predicate can (indirectly) call a coinductive predicate and visa versa, but they 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 $C \leftarrow D_1, \dots, D_n$ 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 pre-programs with 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 to *inductive*.*

Declarative Semantics: The declarative semantics of a co-logic program is a stratified interleaving of the minimal Herbrand model [1, 7] and the maximal co-Herbrand model semantics [9]. Hence, co-LP strictly contains logic programming with rational trees [6] as well as coinductive logic programming [9]. 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, μ (resp. ν) 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 $U^{co}(P) = \nu\Phi_P$, where

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

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 a.k.a. the infinitary Herbrand base, written $B^{co}(P)$, is the set of all ground atoms that can be formed from the atoms in P and the elements of $U^{co}(P)$. Also, let $G^{co}(P)$ be the set of ground clauses $C \leftarrow D_1, \dots, D_n$ that are a ground instance of some clause of P such that $C, D_1, \dots, D_n \in B^{co}(P)$.

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, 7], 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 μT_P^v if v is inductive and νT_P^v if v is coinductive, such that R is the union of the models of the strata that are strictly lower than v and

$$T_P^v(S) = R \cup \left\{ q(\hat{t}) \mid q \in v \wedge [q(\hat{t}) \leftarrow \hat{D}] \in G^{co}(P) \wedge \hat{D} \in S \right\}$$

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 iff the set of all groundings of A with substitutions ranging over the $U^{co}(P)$, is a subset of $M(P)$.

Example 1. Let P_1 be the following program.

```
:- coinductive from/2.
from(N, [N|T]) :- from(s(N), T).
| ?- from(0, _).
```

The model of the program, which is defined in terms of an alternating fixed-point is as follows. The co-Herbrand Universe is $U^{co}(P_1) = N \cup \Omega \cup L$ where $N = \{0, s(0), s(s(0)), \dots\}$, $\Omega = \{s(s(s(\dots)))\}$, and L is the set of all finite and infinite lists of elements in N , Ω , and even L . Therefore the model $M(P_1) = \{from(t, [t, s(t), s(s(t)), \dots]) \mid t \in U^{co}(P_1)\}$, which is the meaning of the program and is obviously not null, as was the case with traditional logic programming. Furthermore $from(0, [0, s(0), s(s(0)), \dots]) \in M(P_1)$ 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 section 3.

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; later 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 π is a finite sequence of positive integers i . The empty path is written ϵ , 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 , 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 π from the root $t(\epsilon)$ to the nodes $t(\pi)$ of the tree, and the nodes are labeled with elements of S .*

A child of node π in tree t is any path $\pi \cdot i$ that is in the domain of t , where i is some positive integer. If π is in the domain of t , then the subtree of t rooted at π , written $t \setminus \pi$, is the partial function $t'(\pi') = t(\pi \cdot \pi')$. Also, $node(L, T_1, \dots, T_n)$ 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 $1 \leq i \leq n$, $node(L, T_1, \dots, T_n)(\epsilon) = L$, and $node(L, T_1, \dots, T_n)(i \cdot \pi) = T_i(\pi)$.

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 $\mu\Sigma_P^v$ if v is inductive and $\nu\Sigma_P^v$ if v is coinductive, such that R is the union of the sets of idealized proofs of the strata strictly lower than v and*

$$\Sigma_P^v(S) = R \cup \{node(q(\hat{t}), T_1, \dots, T_n) \mid q \in v \wedge T_i \in S \wedge [q(\hat{t}) \leftarrow D_1, \dots, D_n] \in G^{co}(P) \wedge T_i(\epsilon) = D_i\}$$

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 Σ_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 $S = \{A \mid \exists T \in \Sigma_P. A \text{ is root of } T\}$, 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 soundness and completeness proofs 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.

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 [4].

The operational semantics given for co-LP is defined as an interleaving of SLD [7] and co-SLD [9]. 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 [9]. 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: $\mathbf{p} :- \mathbf{p}$ the query $\mid ?-\mathbf{p}$. succeeds by the coinductive hypothesis rule (indeed $\{\mathbf{p}\}$ is the gfp of this program). Likewise, given the clause: $\mathbf{p}([1 \mid \mathbf{T}]) :- \mathbf{p}(\mathbf{T})$. the query $\mid ?-\mathbf{p}(\mathbf{X})$. succeeds by the coinductive hypothesis rule with solution $\mathbf{X} = [1 \mid \mathbf{X}]$ (note that indeed $\{\mathbf{p}([1, \dots])\}$, where $[1, \dots]$ 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 rational mgu.*

Theorem 3. (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 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 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 executes 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 multi-set of finite trees (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 consistently renamed for freshness, or R is a coinductive hypothesis rule of the form $\nu(\pi, \pi')$, where π and π' are paths, such that π is a proper prefix of π' .*

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'ed 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 δ takes a tree of atoms T , a path π 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:*

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

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, π is a proper ancestor of π' , 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 $\nu(\pi, \pi')$, p is a coinductive predicate, π is a proper prefix of π' , which is a leaf in T , $T(\pi) = p(t'_1, \dots, t'_n)$, $T(\pi') = p(t_1, \dots, t_n)$, E' is the most general unifier for $\{t_1 = t'_1, \dots, t_n = t'_n\} \cup E$, and $F' = \delta(T, \pi')$.
2. R is a definite clause of the form $p(t'_1, \dots, t'_n) \leftarrow B_1, \dots, B_m$, π is a leaf in T , $T(\pi) = p(t_1, \dots, t_n)$, E' is the most general unifier for $\{t_1 = t'_1, \dots, t_n = t'_n\} \cup E$, and the set of trees of atoms F' is obtained from T acc. to the following case analysis of m 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 $T' = T''$ 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, \dots and a sequence of transition rules R_1, R_2, \dots , 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: 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 $\emptyset =$ empty set.
2. A failure state is a non-accepting state with no 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, \dots, A_n , also called a goal or query, has a derivation in P , if $(\{\text{node}(A_i) \mid 1 \leq i \leq n\}, \emptyset)$ has a derivation in P .*

Equivalence: Correctness is proved by equating operational & declarative semantics via soundness and completeness theorems. Completeness is restricted to atoms with rational proofs.

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

Theorem 5. (*completeness*) *Let $A_1, \dots, A_n \in M(P)$, s.t. each A_i has a rational idealized proof; the query A_1, \dots, A_n has a successful derivation in P .*

5 Conclusions and Future Work

In this paper we presented a comprehensive theory of co-LP and demonstrated its practical applications. Note that most past efforts [3, 6, 5] do not support coinduction in its most general form and therefore are deficient. A prototype implementation of co-LP has been developed by modifying the YAP Prolog system [8]. Current work also includes extending co-LP's operational semantics with alternating OLDT 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.

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

References

1. Krzysztof R. Apt. Logic programming. Handbook of Theoretical Computer Science, 493–574. MIT Press, 1990.
2. J. Barwise, L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Pub., 1996.
3. A. Colmerauer. Equations and inequations on finite and infinite trees. *FGCS'84*.
4. B. Courcelle. Fundamental properties of infinite trees. *TCS*, pp:95–212, 1983.
5. M.Hanus. Integration of functions into LP. *J. Logic Prog.*, 19 & 20:583–628, 1994.
6. J. Jaffar, P. J. Stuckey. Semantics of infinite tree LP. *TCS*, 46(2-3):141–158, 1986.
7. J.W. Lloyd. *Foundations of LP*. Springer, 2nd. edition, 1987.
8. R. Rocha, et al. Theory and Practice of Logic Programming 5(1-2). 161-205 (2005) Tabling Engine That Can Exploit Or-Parallelism. ICLP 2001: 43-58.
9. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive Logic Programming. ICLP'06: 330-345.
10. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Co-Logic Programming: Extending Logic Programming with Coinduction. TR #UTDCS-21-06, UT Dallas, 2006.