Co-Logic Programming: Extending Logic Programming with Coinduction

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

ABSTRACT

Traditional logic programming with its minimal Herbrand model semantics is useful for declaratively defining finite data structures and properties, while *coinductive logic programming*[1] allows for logic programming with infinite data structures and properties. In this paper we present the paradigm of *co-logic programming* (co-LP for brevity). that combines both inductive and coinductive logic programming and presents its theory and applications. Co-LP allows predicates to be annotated as coinductive; by default, unannotated predicates are assumed to be inductive. Coinductive predicates can call inductive predicates and *vice versa*, the only exception being that no cycles are allowed through alternating calls to inductive and coinductive predicates. 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. 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. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent logic programming, model checking, bisimilarity proofs, Answer Set Programming (ASP), etc., some of which are discussed. An outline of a prototype implementation realized by modifying YAP Prolog's engine at the WAM level is also described.

1. INTRODUCTION

The traditional semantics for logic programming (LP) is inadequate for various programming practices such as programming with infinite data structures and *corecursion* [3]. While such programs are theoretically interesting, their practical applications include improved modularization of programs as seen in lazy functional programming languages, rational terms, and model checking. For example, we would like programs such as the following program, which describes infinite binary streams, to be 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 [3]. Therefore, we must not exclude these concepts from logic programming, just because they break with tradition. The traditional declarative semantics of LP must be extended in order to reason about infinite and cyclic structures and properties. This has indeed been done and the paradigm of coinductive logic programming defined [26] and its declarative and operational semantics given. 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.* 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 [26]. There are problems for which coinductive LP is better suited than traditional inductive LP. 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

[1]Note that coinductive LP is not at all related to *inductive* LP, the common term used to refer to LP systems for learning rules [17]. In fact, throughout this paper we use the term inductive LP to refer to traditional SLD (or OLDT) resolution-based LP.

new paradigm *co-logic programming.* However, such a combination 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 logic programming 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. 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). Finally, an outline of a prototype implementation of co-LP realized by modifying YAP Prolog's engine at the WAM level is also described.

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 programs. The rest of the paper is organized as follows. Section 2 gives co-LP programs meaningful declarative and operational semantics along with a proof that both semantics are equivalent. Section 3 discusses related work. Section 4 describes an implementation of co-LP based on modifying YAP Prolog's engine [22], followed by section 5 which presents applications of co-LP to model checking, lazy evaluation, etc. Finally, section 6 discusses future work for extending co-LP and its applications. We have tried to keep the paper self-contained, however, knowledge of coinduction [?, 26] will be helpful.

2. SYNTAX AND SEMANTICS

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

This section formally defines co-LP programs, but first it presents a standard account of set theoretic induction and coinduction in section 2.1. Next, a standard account is given for the abstract syntax of co-LP programs. Then the declarative semantics is defined in section 2.3, and the operational semantics is formally defined in section 2.4. Finally, these are demonstrated to be equivalent in section 2.6. After that, in section 4, a real world implementation of the operational semantics is described.

2.1 Induction and Coinduction

A naive attempt to prove a property of the natural numbers involves demonstrating the property for 0, 1, 2, ... In order for such a proof to be comprehensive, it must be infinite. However, an infinite proof cannot be explicitly written; the principle of proof by induction can be used to represent such an infinite proof in a finite form. This is precisely what co-LP does as well. That is, co-LP uses the principle of proof by coinduction for representing infinite proofs in a finite form. The difference between induction and coinduction is made more obvious later.

Following the account given in Barwise [3] and Pierce [18], we briefly review the set theoretic notions of induction and coinduction, which are defined in terms of monotonic functions on sets and least and greatest fixed-points, which exist and are unique according to Theorem 2.1. For the remaining discussion, it is assumed that all objects such as elements, sets, and functions are taken from the universe of hypersets with the axiom of plenitude. Details can be found in [3].

Definition 2.1 *A function Γ on sets is monotonic if $S \subseteq T$ implies $\Gamma(S) \subseteq \Gamma(T)$. Such functions are called generating functions.*

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

Example 2.1 *Let $\Gamma_{\mathcal{N}}$ be a function on sets: $\Gamma_{\mathcal{N}}(S) = \{0\} \cup \{succ(x) \mid x \in S\}$. Obviously, $\Gamma_{\mathcal{N}}$ is a monotonic function, and intuitively, it defines the set of natural numbers, as will be demonstrated below.*

Definition 2.2 *Let S be a set.*

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

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

One of the purposes of mathematics is to provide unambiguous means for defining concepts. Theorem 2.1 shows that a generating function Γ can be used for giving a precise definition of a set of objects in terms of the least or greatest fixed-point of Γ, as these fixed-points are guaranteed to exist, and are unique.

Theorem 2.1 *(Knaster-Tarski) Let Γ be a generating function. The least fixed-point of Γ is the intersection of all Γ-closed sets. The greatest fixed-point (gfp) of Γ is the union of all Γ-justified sets.*

Since these fixed-points always exist and are unique, it is customary to define unary operators μ and ν for manifesting either of these fixed-points.

Definition 2.3 *$\mu\Gamma$ denotes the lfp of Γ, and $\nu\Gamma$ denotes the gfp.*

Example 2.2 *Let $\Gamma_{\mathcal{N}}$ be defined as in example 2.1. The definition of the natural numbers N can now be unambiguously invoked via theorem 2.1, as $N = \mu\Gamma_{\mathcal{N}}$, which is guaranteed to exist and be unique. Note that this definition is equivalent to the standard "inductive" definition of the natural numbers, which is written: Let N be the smallest set such that $0 \in N$ and if $x \in N$, then $x + 1 \in N$.*

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

Example 2.3 *$\Gamma_{\mathcal{N}}$ from example 2.1 also unambiguously defines another set, that is, $\mathcal{N}' = \nu\Gamma_{\mathcal{N}} = N \cup \{\omega\}$, where $\omega = succ(\omega)$, that is, $\omega = succ(succ(succ(...)))$ an infinite application of succ.*

Corollary 2.2 *The principle of induction states that if S is Γ-closed, then $\mu\Gamma \subseteq S$, and the principle of coinduction states that if S is Γ-justified, then $S \subseteq \nu\Gamma$.*

Definition 2.4 *Let $Q(x)$ be a property. Proof by induction demonstrates that the characteristic set $S = \{x \mid Q(x)\}$ is Γ-closed, and then invokes the principle of induction to prove that every element x of $\mu\Gamma$ has the property $Q(x)$.*

Similarly, proof by coinduction demonstrates that the characteristic set S is Γ-justified, and then invokes the principle of coinduction to prove that every element x that has property $Q(x)$ is also an element of $\nu\Gamma$.

Example 2.4 *The familiar proof by induction can be instantiated with regards to the set $\mathcal{N}$ defined in the previous example. Let $Q(x)$ be some property, and let $S = \{x \mid Q(x)\}$. In order to show that every element x in $\mathcal{N}$ has property $Q(x)$, by induction it is sufficient to show that $\Gamma_{\mathcal{N}}(S) \subseteq S$, which is equivalent to showing that $0 \in S$, and if $x \in S$, then $succ(x) \in S$.*

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

2.2 Syntax

A co-LP program P is syntactically identical to a traditional, that is, inductive logic program as demonstrated by the following account of syntax. In the following, it is important to distinguish between an idealized class of objects and the syntactic restriction of said objects. Elements of syntax are necessarily finite, while many of the semantic objects used by 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 2.5 *The set of terms is $\nu\Gamma$ and the set of syntactic terms is $\mu\Gamma$, where $t \in \Gamma(S)$ whenever one of the following is true:*
1. t is a variable.
2. t is a constant.
3. $t = f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n \in S$ and f is a function symbol of arity n.

Definition 2.6 *An atom is an expression of the form $p(\widehat{t})$, where p is a predicate symbol of arity n and $\widehat{t}$ is shorthand for a sequence of terms $t_1, \ldots, t_n$. A syntactic atom is an atom containing only syntactic terms.*

Definition 2.7 *A definite clause is a logical inference rule of the form $C \leftarrow \widehat{D}$ where C is an atom and $\widehat{D}$ is shorthand for a sequence $D_1, \ldots, D_n$ of atoms. A syntactic definite clause is a definite clause containing only syntactic atoms.*

Definition 2.8 *A term, atom, or clause is said to be ground if it does not contain any variables.*

Definition 2.9 *A definite program is a finite set of syntactic definite clauses.*

Again, note that a definite program is a finite object. The following defines a program that is almost a co-LP program.

Definition 2.10 *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 said to be coinductive (resp. inductive) if the partial mapping maps the predicate to `coinductive` (resp. `inductive`). Similarly, an atom is said to be coinductive (resp. inductive) if the underlying predicate is coinductive (resp. inductive).*

Not every pre-program is a co-LP 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 2.11 *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 2.12 *A co-LP 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`.*

2.3 Declarative Semantics

The declarative semantics of a co-LP program is a stratified interleaving of the minimal co-Herbrand model [2, 15] and the maximal co-Herbrand model semantics [26]. Hence co-LP strictly contains logic programming with rational trees [13] as well as coinductive logic programming [25, 26]. 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.

The following definition is necessary for defining the model of a co-LP 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 2.13 *The reduced graph for a co-LP 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.

Definition 2.14 *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*

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

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 2.15 *Let P be a definite program. The co-Herbrand base also known as 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 $C \leftarrow D_1, \ldots, D_n$ that are a ground instance of some clause of P such that $C, D_1, \ldots, D_n \in B^{co}(P)$.*

Now we can define the model of a stratum, i.e., the model of a vertex in the reduced graph of a co-LP 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 [2, 15], 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-LP 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 2.16 *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*

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

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

Definition 2.17 *The model of a co-LP 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 2.18 *An atom A is true in program P if and only if the set of all groundings of A with substitutions ranging over the $U^{co}(P)$, is a subset of $M(P)$.*

Example 2.5 *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 $N = \{0, s(0), s(s(0)), \ldots\}$, $\Omega = \{s(s(s(\ldots)))\}$, and L is the set of all finite and infinite lists of elements in N, Ω, and even L. Therefore the model $M(P_1) = \{from(t, [t, s(t), s(s(t)), \ldots]) \mid t \in U^{co}(P_1)\}$, which is the meaning of the program and is obviously not null, as was the case with traditional logic programming. Furthermore $from(0, [0, s(0), s(s(0)), \ldots]) \in M(P_1)$ 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 2.5 and 5.

The model characterizes semantics in terms of truth, that is, the set of ground atoms that are true. This set is defined via a generator, and in section 2.6, we 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 2.19 *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, 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 π from the root $t(\epsilon)$ to the nodes $t(\pi)$ of the tree, and the nodes of the tree 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, $node(L, T_1, \ldots, T_n)$ 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 $1 \leq i \leq n$, $node(L, T_1, \ldots, T_n)(\epsilon) = L$, 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 2.20 *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*

$$\begin{aligned}\Sigma_P^v(S) = \;& R \cup \{node(q\,(\widehat{t})\,, T_1, \ldots, T_n) \mid q \in v \wedge T_i \in S \wedge \\ & [q\,(\widehat{t}) \leftarrow D_1, \ldots, D_n] \in G^{co}(P) \wedge T_i(\epsilon) = D_i\}\end{aligned}$$

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 2.21 *The set of idealized proofs generated by a co-LP 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 2.3 *Let $S = \{A \mid \exists T \in \Sigma_P . A$ 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 2.6 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.

2.4 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 [7].

The operational semantics given for co-LP is defined as an interleaving of SLD [15] and co-SLD [26]. 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 [26].

Definition 2.22 *A tree is rational if the cardinality of the set of all its subtrees is finite. An object such as a term, atom, or idealized proof is said to be rational if it is modeled as a rational tree.*

Definition 2.23 *A substitution is a finite mapping of variables to terms. A substitution is syntactic if it only substitutes syntactic terms for variables. A substitution is said to be rational if it only substitutes rational terms for variables.*

Definition 2.24 *A term unification problem is a finite set of equations between terms. A unifier for a term unification problem is a substitution that satisfies every equation in the problem. σ is a most general unifier (mgu) for a term unification problem, if any other solution σ' can be defined as the composition $\sigma'' \circ \sigma$.*

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

Definition 2.25 *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 2.4 *(Courcelle) Every system of equations has a mgu that is rational.*

Theorem 2.5 *(Courcelle) For every rational substitution σ with domain V, there is a system of equations E, such that the most general unifier σ' of E is equal to σ when restricted to the domain V.*

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

Now the operational semantics can be defined. The semantics implicitly defines a state transition system. Systems of equations are used to model 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 (see Section 4).

Definition 2.26 *A state S is a pair (F, E), where F is a finite multi-set of finite trees, i.e., a forest of syntactic atoms, and E is a system of equations.*

Definition 2.27 *A transition rule R of a co-LP 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 π and π' are both 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 2.28 *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:*

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

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 non-deterministic execution; see Section 4). We write $S - x$ to denote the multi-set obtained by removing an occurrence of x from S.

Definition 2.29 *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, $T(\pi) = p(t'_1, \ldots, t'_n)$, $T(\pi') = p(t_1, \ldots, t_n)$, 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 $p(t'_1, \ldots, t'_n) \leftarrow B_1, \ldots, B_m$, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$, E' is the most general unifier for $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E$, and the set of trees of atoms F' is obtained from T according 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 $1 \le i \le m$.

   (c) Case $m > 0$ and p is inductive: If $\pi = \epsilon$ then $F' = \{node(B_i) \mid 1 \le i \le m\}$. 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 $1 \le i \le m$, and $T'(\pi' \cdot (m + k)) = T(\pi' \cdot k)$, for $k \ne j$.

Definition 2.30 *A transition sequence in program P consists of a sequence of states $S_1 S_2, \ldots$ and a sequence of transition rules $R_1, R_2 \ldots$, such that S_i transitions to S_{i+1} by rule R_i of program P.*

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

Definition 2.31 *The following are two distinguished terminal states:*

1. An accepting state is a state of the form $(\emptyset, E)$, where $\emptyset$ denotes the 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 2.32 *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 $(\{node(A_i) \mid 1 \le i \le n\}, \emptyset)$ has a derivation in P.*

An implementation will use backtracking search in order to find a successful derivation.

2.5 Examples

In this section we illustrate co-LP via examples. In addition to allowing infinite terms, the operational semantics of co-LP allows for an execution to succeed when it encounters a subgoal that unifies with an ancestor subgoal (coinductive hypothesis rule). Note that while this is somewhat similar to tabled logic programming in that called atoms are recorded so as to avoid unnecessary redundant computation, the difference is that co-LP's memo'ed atoms represent a coinductive hypothesis, while tabled logic programming's table represents a list of results for each called goal in the traditional inductive semantics. Hence the memo'ed atoms in co-LP correspond to a *dynamic generated* coinductive hypothesis.

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

```
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. 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))].

stream(T) is true whenever T is some 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(...))). Such a term has a finite representation as ω=s(ω), which quite simply states that infinity plus one equals infinity.

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 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 member/2, called membera/2 demonstrates, 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].

Coinductive append: Co-LP subsumes logic programming with rational trees [13, 5]. This is demonstrated by the traditional definition of append, which, when executed with coinductive semantics, allows for calling the predicate with infinite arguments. This is illustrated below.

```
  append( [], X, X ).
  append( [H|T], Y, [H|Z] ) :- append( T, Y, Z ).
```

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

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

If we also allow the possibility of expanding the variant call using its definition (and apply the coinductive hypothesis rule in the variant that will arise subsequently), then we will enumerate more values for Y:

```
     Y = [4, 5, 6, 4, 5, 6, | Y]
     Y = [4, 5, 6, 4, 5, 6, 4, 5, 6, | Y]
     ....
```

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

```
| ?- X = [1, 2, 3, | X], Y = [3, 4 | Y],
        append(X, Y, Z).
      Answer:  Z = [1, 2, 3 | Z]
| ?- Z = [1, 2 | Z], append(X, Y, Z).
      Answers:  X = [], Y = [1, 2 | Z];
                X = [1], Y = [2 | Z];
                X = [1, 2], Y = Z
                ....
```

As noted earlier, more solutions (e.g., `X = [], Y = [1,2,1,2|Y` for the second query) can be generated by expanding the variant (coinductive) call to append using its definition, and applying the coinductive hypothesis rule to the subsequent variant calls.

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

```
:- 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,[],[]).
filter(H,[K|T],[K|T1]) :- R is K mod H, R > 0,
            filter(H,T,T1).
filter(H,[K|T],T1) :- 0 is K mod H,
    filter(H,T,T1).
```

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

```
:- coinductive int/2.
int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
generate_infinite_list(I) :- int(2,I).
```

then in the `primes/1` predicate, the calls `sieve/2`, `comember/2`, and `generate_infinite_list/1` should be co-routined, and, likewise, in the `sieve/2` predicate, the calls `filter/3` and the recursive call `sieve/2` must be coroutined. From the above, one can also observe that co-LP can be the basis of providing elegant declarative semantics to concurrent LP [24]. Details are omitted due to lack of space.

2.6 Correctness

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

Theorem 2.6 *(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.*

Theorem 2.7 *(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.*

3. RELATED WORK

Most of the work in the past has been focused on allowing for infinite data structures in LP. However, most of these stop short of including infinite proofs. Logic programming with rational trees [5, 6, 13] 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 2, 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. On the other hand, 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, yielding an operational semantics that strictly contains both SLD and co-SLD. Thus, 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 corecursive clauses.*

Jaffar et al's coinductive tabling proof method [12] uses coinduction as a means of proving infinitary properties in model checking, as opposed to using it in defining the semantics of a new declarative programming language, as is the case with 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 [26], 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 for the same applications as Jaffar et al's coinductive tabling proof method [26].

Lazy functional LP (e.g., [8, 11]) 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 in such a way that the function evaluates to true. However, if the property is infinitary and has an infinite idealized proof, then the corresponding function will not evaluate to true because it will have an infinite evaluation. In co-LP, on the other hand, a predicate with an infinite idealized proof is defined as true, when the predicate is coinductive, and the operational semantics allow for the finite derivation via the use of a dynamic coinductive hypothesis. Therefore, predicates in lazy functional logic programming are semantically different from those in co-LP.

The Horn μ-calculus of Charatonik et al. [4] and the alternation-free variant of Talbot [29] extend Horn logic with least and greatest fixed-points and provide declarative semantics in a manner similar to co-LP. The Horn μ-calculus does not have the stratification restriction found in co-LP, while the alternation-free restriction does. The Horn μ-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 μ-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-direction operational semantics, let alone an efficient implementation for the full Horn μ-calculus or the full alternation-free Horn μ-calculus, as 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 [26], 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.

4. IMPLEMENTATION

A prototype implementation of co-LP is being developed by modifying the YAP Prolog system [22] (in fact, we use the version of YAP extended with tabling called YAPTAB). The formal operational semantics described in section 2.4 allows for a coinductively recursive call to terminate (coinductively succeed) if it *unifies* with an ancestor call. However, in the current prototype, a coinductive call terminates only if it is a *variant* of an ancestor call. Additionally, in the current prototype, coinductive calls can produce infinite-sized output but cannot consume infinite-sized inputs (this is primarily due to limitations in the YAP implementation). Our current implementation effort is focused on fixing both these limitations.

The implementation of co-LP is reasonably straightforward, and is based on the machinery used in the YAP system for realizing OLDT style tabling [22]. The inductive predicates are realized by the syntax and semantics of normal Prolog predicates in the YAP system. A predicate is declared via a directive of the form

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

where `p` is the predicate name and `n` its arity. All other predicates are treated as inductive by default. When a coinductive call is encountered for the first time, it is recorded in the memo-table that YAPTAB [22] uses for implementing standard tabled LP. The call is recorded again in the table after head unification, but this time it is saved as a solution to the tabled call. The variables in the recorded solution are interpreted w.r.t. the environment of the coinductive call (so effectively the closure of the call is saved). When a variant call is encountered later, it is unified with the solution saved in the table and made to succeed. (In reality, a choice point is created whose first alternative corresponds to the variant call succeeding by the coinductive hypothesis as just described; the remaining alternatives in the choice-point correspond to call expansion using the matching rules as in normal Prolog execution. Allowing the expansion of the call as an alternative allows for the solutions beginning with the prefix `[0, s(0), s(s(0))]` to be reported in the `stream` example in Section 2.5 and for additional solutions to be generated in the `append` example.) Note that everything recorded in the memo-table for a specific coinductive predicate `p` will be deleted, when execution backtracks over the first call of `p`. Consider the example program:

```
:- coinductive p/1.
p(f(X)) :- p(X).
| ?- p(Y).
```

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

One can see from the description above that much of the machinery for OLDT tabling present in YAPTAB can be reused for implementing co-LP. However, because YAPTAB uses *tries* [21], which do not support rational trees, predicates that take rational terms as input arguments cannot be coinductively interpreted in the current implementation. Thus, coinductive `member/2` and `append/3` predicates will not currently work in our system. Our current implementation is, however, adequate to run constraint LP based implementations of somewhat complex applications (e.g., timed automata). Work is in progress to extend tries in YAPTAB to support rational terms.

5. APPLICATIONS

Co-LP augments traditional logic programming with infinite terms and infinite proofs. These concepts generalize the notions of rational trees and lazy predicates. Co-LP has practical applications in concurrent LP, bisimilarity, model checking, timed automata and many other areas. Furthermore, it appears that the concept of ancestors in the co-SLD semantics can be used to give a top-down operational semantics to a restricted form of ASP programs. Some of these applications have been expounded elsewhere [26], however, most of them rely solely on the coinductive LP part of co-LP so we do not reproduce them here. In this section we describe an application from model checking that needs both the traditional SLD as well as co-SLD. We show how co-LP allows us to elegantly verify liveness properties.

Model checking is a common technique used for verifying hardware and software systems. It involves constructing a model of the system, in terms of a finite state Kripke structure and then determining if the model satisfies various properties specified as temporal logic formulas. The verification is performed by means of systematically searching the state space of the Kripke structure for a counterexample to the given property. Most properties that are to be verified can be classified into

1. Safety properties—which intuitively assert that "nothing bad will happen"
2. Liveness properties—these properties assert that "something good will eventually happen."

A number of techniques for verifying safety properties have been developed over the years. Most of these techniques use some variation of reachability analysis, i.e, if a counterexample to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. This procedure consists of specifying a counterexample to the property in terms of a least fixed-point formula and then evaluating this formula over the state space of the model by means of traditional inductive tabling [20].

However, it is well known that reachability-based inductive techniques are not suitable for verifying liveness properties [19]. 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 [30] and that fairness properties can be specified in terms of alternating fixed-point temporal logic formulas [14]. In previous work [26], we introduced a technique for verifying a class of liveness properties in the absence of fairness constraints. Here, we develop a method for verifying the more general class of all liveness properties that can only be verified with the presence of fairness constraints. Therefore this method subsumes our previous work [26, 16].

Essentially, our 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_{NP}, which encodes the negation of the liveness property, to obtain a composite program P_μ. We then compute the stratified alternating fixed-point of the logic program P_μ as described in section 2.3 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 1. It consists of four states `s0`, `s1`, `s2` and `s3`. The system starts off in state `s0`, enters state `s1`, performs a finite amount of work in state `s1` and then exits to state `s2`, from where it transitions back to state `s0`, and repeats the entire loop again, an infinite

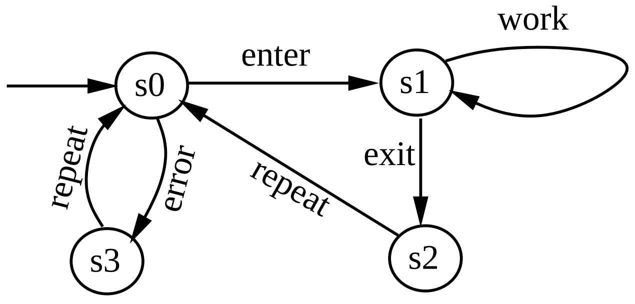

Figure 1: Example Automata

number of times. The system might encounter an error, causing a transition to state `s3`. Corrective action is taken, followed by a transition back to `s0`. The system is modeled by the following Prolog code.

```
:- coinductive state/2.

state(s0,[s0,is1|T]) :- enter, work,
state(s1,T).
state(s1,[s1|T]) :- exit, state(s2,T).
state(s2,[s2|T]) :- repeat, state(s0,T).
state(s0,[s0|T]) :- error, state(s3,T).
state(s3,[s3|T]) :- repeat, state(s0,T).
work :- state(is1),
enter.
exit.
repeat.
error.
state(is1) :- state(is1).
state(is1).
```

This simple example illustrates the power of co-LP when compared to purely inductive or coinductive logic programming. 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 `state/2`) is inductive and represents a bounded computation (we are interested in its lfp). The call graph consisting of the predicate `state/2` represents a coinductive computation, whereas the call graph consisting of the predicate `state/1` represents an inductive computation. The semantics therefore evaluates `state/1` 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 `s1` represented by the transition `work` always terminates. In order to do so, we require the fairness property: "if the transition `enter` occurs infinitely often, then the transition `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 `state/1` and then composing it with the coinductive program. The resulting program is then composed with the property, "the state `s2` 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 absent/2. Thus, given the program above, the user will pose the query:

```
| ?- state(s0,X), absent(s2,X).
```

where absent/2 is a coinductive predicate that checks that the state s2 is not present in the (infinite) list X infinitely often (it is the negated version of the coinductive comember predicate described earlier). The co-LP system will respond with a solution: `X = [s0, s3 | X]`, a counterexample which states that there is an infinite path not containing s2. One can see that this corresponds to the behavior of the system if error is encountered.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a comprehensive theory of co-LP, demonstrated its practical applications as well as reported on its preliminary implementation on top of YAP Prolog. Current work involves extending co-LP's operational semantics with alternating OLDT and co-SLD [27], 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.

We are also working on incorporating coinductive reasoning in our quest of developing a single LP system that combines tabled LP, constraints, parallelism, ASP and co-routining [10]. Additionally, we are also working on applying the coinduction principle to Hereditary Harrop formulas, resolution theorem proving, machine learning (coinductive learning), non-monotonic reasoning (ASP), and negation [28].

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

7. REFERENCES

[1] R. Alur, D.L. Dill. A theory of timed automata. *TCS* 126:183-235, 1994.

[2] Krzysztof R. Apt. Logic programming. Ch. 15. Handbook of Theoretical Computer Science, 493–574. MIT Press, 1990.

[3] J. Barwise, L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena.* CSLI Publications, 1996.

[4] Charatonik, McAllester, Niwinski, Podelski, and Walukiewicz. The horn mu-calculus. In *LICS: IEEE Symposium on Logic in Computer Science*, 1998.

[5] A. Colmerauer. Prolog and infinite trees. In Clark & Tärnlund, editors, *Logic Progr.*, pp. 231-251. 1982.

[6] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proc. FGCS-84*: pages 85–99, Tokyo, 1984.

[7] B. Courcelle. Fundamental properties of infinite trees. *TCS*, pp:95–212, 1983.

[8] E. Giovannetti, G. Levi, et al. Kernel-LEAF: A logic plus functional language. *JCSS*, 42(2):139–185, 1991.

[9] G. Gupta. Verifying Properties of Cyclic Data-structures with Tabled Unification. Internal Memo. New Mexico State University. February 2000.

[10] G. Gupta. Next Generation of Logic Programming Systems. Technical Report UTD-42-03, University of Texas, Dallas. 2003.

[11] M.Hanus. Integration of functions into LP. *J. Logic Prog.*, 19 & 20:583–628, 1994.

[12] J. Jaffar, A. E. Santosa, R. Voicu. A CLP proof method for timed automata. In *RTSS*, pages 175–186, 2004.

[13] J. Jaffar, P. J. Stuckey. Semantics of infinite tree LP. *TCS*, 46(2–3):141–158, 1986.

[14] X. Liu, C.R. Ramakrishnan, S.A. Smolka, Fully local and efficient evaluation of alternating fixed-points, *TACAS '98*, LNCS, 1384, Springer-Verlag, 1998.

[15] J.W. Lloyd. *Foundations of LP.* Springer, 2nd. edition, 1987.

[16] A. Mallya. Deductive Multi-valued Model Checking, ICLP'05, Springer LNCS 3368. pp. 297-310.

[17] S. Muggleton, editor. Inductive Logic Programming. Academic Press, 1992.

[18] B. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, 2002.

[19] A. Podelski, A. Rybalchenko, Transition Predicate Abstraction and Fair Termination, *POPL '05*, pp. 132-144, ACM Press, 2005.

[20] Y. S. Ramakrishna et al. Efficient Model Checking Using Tabled Resolution. in *Proc. CAV 1997.* pp. 143-154.

[21] I. V. Ramakrishnan et al. Efficient Access Mechanisms for Tabled Logic Programs. J. Logic Programming 38(1):31-54 (1999).

[22] 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

[23] V. Schuppan, A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. ENTCS 149(1): 79-96 (2006)

[24] E. Shapiro. Concurrent Prolog: Collected Works. MIT Press. 1987.

[25] Luke Simon. Coinductive LP. Internal memo, UT Dallas, March 2004.

[26] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive Logic Programming. Technical Report UTDCS-11-06, University of Texas, Dallas, 2006. Submitted to ICLP'06. Available at `http://www.utdallas.edu/~gupta/colp.pdf`.

[27] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Combining Tabling and co-Tabling. Internal memo, UT Dallas, 2006.

[28] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive Extensions of Theorem Proving, Machine Learning, and Hereditary Harrop Formulas. Internal memo, UT Dallas, 2006.

[29] Jean-Marc Talbot. On the alternation-free Horn μ-calculus. *LNCS*, 1955:418–435, 2000.

[30] M. Vardi, Verification of Concurrent Programs: The Automata-Theoretic Framework, *LICS '87*, pp. 167-176, IEEE, 1987.

Appendix I: Correctness Proof

Note: included for the convenience of reviewers.

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 6 discusses an extension of the operational semantics, so as to improve its completeness.

lemma 7.1 *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. $\square$

lemma 7.2 *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 π_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. $\square$

lemma 7.3 *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 $U^{co}(P)$ 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 $Q_v \subseteq \nu T_P^v$. 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 7.1, (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 $(node(A, [B_1, \ldots, B_n]), E)$, where $E(A) = E(A'')$. By lemma 7.2 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 $C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^{co}(P)$, then $C = A' \leftarrow E''(B_1), \ldots, E''(B_n)$, where $E'' = E' \cup E$. By lemma 7.1, 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, $Q_v \subseteq \nu T_P^v$.

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 7.1, (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 7.2 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 $C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^{co}(P)$, then $C = A' \leftarrow E''(B_1), \ldots, E''(B_n)$, where $E'' = E' \cup E$. By lemma 7.1, 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, $A' \in \mu T_P^v$. $\square$

Theorem 7.4 *(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.*

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 7.3, each $E(A_i)$ is true in program P. $\square$

Theorem 7.5 *(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.*

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 π'. 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. $\square$