Coinductive Logic Programming and its Applications

Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, Ajay Mallya

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

Abstract. Coinduction has recently been introduced as a powerful technique for reasoning about unfounded sets, unbounded structures, and interactive computations. Where induction corresponds to least fixed point semantics, coinduction corresponds to greatest fixed point semantics. In this paper we discuss the introduction of coinduction into logic programming. We discuss applications of coinductive logic programming to verification and model checking, lazy evaluation, concurrent logic programming and non-monotonic reasoning.

1 Introduction

Recently *coinduction* has been introduced as a technique for reasoning about unfounded sets [10], behavioral properties of programs [4, 7], and proving liveness properties in model checking [13]. Coinduction also serves as the foundation for lazy evaluation [8] and type inference [16] in functional programming as well as for interactive computing [6,25].

Coinduction is the dual of induction. Induction corresponds to well-founded structures that start from a basis which serve as the foundation for building more complex structures. For example, natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality [6]. Thus, the inductive definition of list of numbers is as follows: (i) `[]` (empty list) is a list (initiality); (ii) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (iii) nothing else is a list (minimality). Minimality implies that infinite-length lists of numbers are not members of the inductively defined set of lists of numbers. Inductive definitions correspond to least fixed point interpretations of recursive definitions.

Coinduction eliminates the initiality condition and replaces the minimality condition with maximality. Thus, the coinductive definition of a list of numbers is: (i) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (ii) the set of lists is the maximal set of such lists. There is no base case in coinductive definitions, and while this may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point interpretation of recursive definitions (recursive definitions for which gfp interpretation is intended are termed corecursive definitions). Thus, the set of lists under coinduction is the set of all infinite lists of numbers (no finite lists are contained in this set). Note, however, that if we have a recursive definition with a base case, then under coinductive interpretation, the set defined will contain both finite and infinite-sized elements, since in this case the gfp will also contain the lfp. In the context of logic programming, in the presence of coinduction, proofs may be of infinite length. A coinductive proof essentially is an infinite-length proof.

2 Coinduction and Logic Programming

Coinduction has been incorporated in logic programming in a systematic way only recently [22, 21], where an operational semantics—similar to SLD—is given for computing the greatest fixed point of a logic program. This operational semantics called co-SLD relies on a *coinductive hypothesis rule* and systematically computes elements of the gfp of a program via backtracking. The semantics is limited to only *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors.

Consider the list example above. The normal logic programming definition of a stream (list) of numbers is given as program P1 below:

```
stream([]).
stream([H|T]) :- number(H), stream(T).
```

Under SLD resolution, the query `?- stream(X)` will systematically produce all finite streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program P2:

```
stream([H|T]) :- number(H), stream(T).
```

In the program P2, the meaning of the query `?- stream(X)` is semantically null under standard logic programming. 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 `stream(X)` in program P2; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [4]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties [22, 23, 21]. In the coinductive LP paradigm the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary (or co-Herbrand) Herbrand base [12], and maximal models (computed using greatest fixed-points).*

Thus, under coinductive interpretation of P2, the query `?- stream(X)` produces all infinite sized stream as answers, e.g., `X = [1, 1, 1, ... ]`, `X = [1, 2, 1, 2, ... ]`, etc., thus, P2 is not semantically null (but proofs may be of infinite-length).

If we take a coinductive interpretation of program P1, then we get all finite and infinite stream as answers to the query `?- stream(X)`. Coinductive logic programming allows programmers to manipulate infinite structures. As a result, unification has to be necessarily extended and "occurs check" removed. Thus, unification equations such as `X = [1 | X]` are allowed in coinductive logic programming; in fact, such equations will be used to represent infinite (regular) structures in a finite manner.

The operational semantics of coinductive logic programming 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]`.

Thus, given a call during execution of a logic program, where, earlier, the candidate clauses were tried one by one via backtracking, under coinductive logic programming the trying of candidate clauses is extended with yet more alternatives: applying the coinductive hypothesis rule to check if the current call will unify with any of the earlier calls. The coinductive hypothesis rule will work for only those infinite proofs that are *regular* in nature, i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved infinite number of times (such as a circular linked list). More general implementations of coinduction are possible, but they are beyond the scope of this paper [21].

Even with regular proofs, there are many applications of coinductive logic programming, some of which are discussed next. These include model checking, concurrent logic programming, real-time systems, non-monotonic reasoning, etc. We will not focus on the implementation of coinductive LP (implementation atop YAP is available from the authors) except to note that to implement coinductive LP, one needs to remember in a memo-table (memoize) all the calls made to coinductive predicates.

Finally note that one has to be careful when using both inductive and coinductive predicates together, since careless use can result in interleaving of least fixed point and greatest fixed point computations. Such programs cannot be given meaning easily. Consider the following program where the predicate `p` is coinductive and `q` is inductive.

```
p :- q.
q :- p.
```

For computing the result of goal `?- q.`, we will 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. This contradiction is resolved by disallowing such cyclical nesting of inductive and coinductive predicates, i.e., *stratifying* inductive and coinductive predicates in a program. An inductive predicate in a given strata cannot call a coinductive predicate in a higher strata and vice versa [23, 21].

3 Examples

Next, we illustrate coinductive Logic Programming via more examples.

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

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

The following is an execution trace, for the above query, of the memoization of calls by the operational semantics. Note that calls of `number/1` are not memo'ed because `number/1` is inductive.

```
MEMO: stream([ 0, s(0), s(s(0)) | T ])
MEMO: stream([ s(0), s(s(0)) | T ])
MEMO: stream([ s(s(0)) | T ])
```

The next goal call is `stream(T)`, which unifies with the first memo'ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with the infinite solution:

```
T = [ 0, s(0), s(s(0)) | T ]
```

The user could force a failure here, which would cause the goal to be unified with the next two matching memo'ed ancestor producing `T = [s(0),s(s(0))|T]` and `T = [s(s(0))|T]` respectively. If no remaining memo'ed elements exist, the goal is memo'ed, and expanded using the coinductively defined clauses, and the process repeats—generating additional results, and effectively enumerating the set of (rational) infinite lists of natural numbers that begin with the prefix `[0,s(0),s(s(0))]`.

The goal `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(...)))`. 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 coinductive 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. Coinductive 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]` (if `X` is absent from the list `L`, the lfp of `drop(X,L)` is null).

List Append: Let us now consider the definition of standard `append` predicate.

```
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]
```

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], X = [1,2,3|X], Y = [3,4|Y]
```

The coinductive append can also be used to split infinite lists as in:

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

Note that application of the coinductive hypothesis rule will produce solutions in which `X` gets bound to an infinite list (fourth and seventh solutions above).

Sieve of Eratosthenes: Coinductive LP also allows for lazy evaluation to be elegantly incorporated into Prolog. Lazy evaluation allows for manipulation of, and reasoning about, cyclic and infinite data structures and properties. Lazy evaluation can be put to fruitful use, in situations where only a finite part of the infinite term is of interest. 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 call `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 `primes/1` predicate, the calls `generate_infinite_list/1`, `comember/2`, and `sieve/2` should be co-routined, and likewise, in the `sieve/2` predicate, the calls `filter/3` and the recursive call `sieve/2` must be coroutined.

4 Application to Model Checking and Verification

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

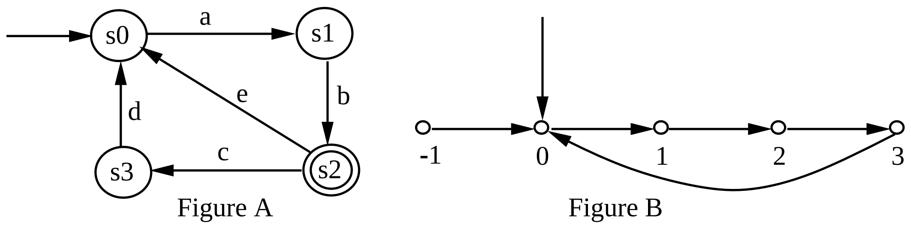


Fig. 1. Example Automata

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

```
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
automata([], St) :- final(St).
trans(s0, a, s1).                         trans(s1, b, s2).
trans(s2, c, s3).                         trans(s3, d, s0).
trans(s2, e, s0).                         final(s2).
```

A call to `?- automata(X, s0)` in a standard LP system will generate all finite strings accepted by this automata. Now suppose we want to turn this automata into an ω-automata, i.e., it accepts infinite strings (an infinite string is accepted if states designated as final state are traversed infinite number of times), then the (coinductive) logic program that simulates this automata can be obtained by simply dropping the base case (for the moment, we'll ignore the requirement that final-designated states occur infinitely often; this can be easily checked by `comember/2`).

```
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
```

Under coinductive semantics, posing the query `| ?- automata(X, s0).` will yield the solutions:

```
X = [a, b, c, d | X];
X = [a, b, e | X];
```

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

4.1 Verifying Liveness Properties

It is well known that safety properties can be verified by reachability analysis, i.e, if a counter-example to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Verification of safety properties amounts to computing least fixed-points and thus is elegantly handled by standard LP systems extended with tabling [18]. Verification of liveness properties under such tabled LP systems is however problematic. This is because counterexamples to liveness properties take the form of infinite traces, which are semantically expressed as greatest fixed-points. Tabled LP systems [18] work around this problem by transforming the temporal formula denoting the property into a semantically equivalent least fixed-point formula, which can then be executed as a tabled logic program. This transformation is quite complex as it uses a sequence of nested negations.

In contrast, coinductive LP can be directly used to verify liveness properties. Coinductive LP can directly compute counterexamples using greatest fixed-point temporal formulae without requiring any transformation. Intuitively, a state is not live if it can be reached via an infinite loop (cycle). Liveness counterexamples can be found by (coinductively) enumerating all possible states that can be reached via infinite loops and then by determining if any of these states constitutes a valid counterexample. Consider the example of a modulo 4 counter, adapted from [20] (See Figure 1.B). For correct operation of the counter, we must verify that along every path the state s_{-1} is not reached, i.e., there is at least one infinite trace of the system along which s_{-1} never occurs. This property is naturally specified as a greatest fixed-point formula and can be verified coinductively. A simple coinductive logic program S_P to solve the problem is shown below. We compose the counter program with the negation of the property, i.e., `N1 >= 0`. Note that `sm1` represents the state corresponding to `-1`.

```
:- coinductive s0/2, s1/2, s2/2, s3/2, sm1/2.
sm1(N,[sm1|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
s0(N,[s0|T]) :- N1 is N+1 mod 4, s1(N1,T), N1>=0.
s1(N,[s1|T]) :- N1 is N+1 mod 4, s2(N1,T), N1>=0.
s2(N,[s2|T]) :- N1 is N+1 mod 4, s3(N1,T), N1>=0.
s3(N,[s3|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
```

The counter is coded as a cyclic program that loops back to state s_0 via states s_1, s_2 and s_3. State s_{-1} represents a state where the counter has the value -1. The property P to be verified is whether the state s_{-1} is live. The query `:- sm1(-1,X), comember(sm1,X)` where the `comember` predicate coinductively checks that `sm1` occurs in `X` infinitely often, will fail implying inclusion of the property in the model, i.e., the absence of a counterexample to the property. The benefit of our approach is that we do not have to transform the model into a form amenable to safety checking. This transformation is expensive in general and can reportedly increase the time and memory requirements by 6-folds [20].

This direct approach to verifying liveness properties also applies to *multi-valued model checking* of the μ-calculus [13]. Multi-valued model checking is used to model systems, whose specification has varying degrees of inconsistency or incompleteness. Earlier effort [13] verified liveness properties by computing the *gfp* which was found using negation based transformation described earlier. With coinduction, the *gfp* can be computed directly as in standard model checking as described above. We do not give details due to lack of space. Coinductive LP can also be used to check for *bisimilarity.* Bisimilarity is reduced to coinductively checking if two ω-automata accept the same set of rational infinite strings.

4.2 Verifying Properties of Timed Automata

Timed automata are simple extensions of ω-automata with stopwatches [1], and are easily modeled as coinductive logic programs with CLP(R) [9]. Timed automata can be modeled with coinductive logic programs together with constraints over reals for modeling clock constraints. The coinductive logic program with CLP(R) constraints for modeling the classic train-gate-controller problem is shown below. This program runs on our implementation of coinduction on YAP [19] extended with CLP(R). The system can be queried to enumerate all the infinite strings that will be accepted by the automata and that meet the time constraints. Safety and liveness properties can be checked by negating those properties, and checking that they fail for each string accepted by the automata with the help of `comember/2` predicate.

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

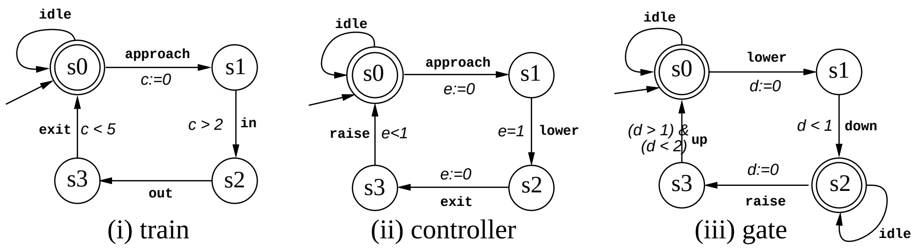


Fig. 2. Train-Controller-Gate Timed Automata

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

train(X,up,X,T1,T2,T2).              gate(s0,lower,s1,T1,T2,T3) :-
train(s0,approach,s1,T1,T2,T3) :-                {T3 = T1}.
       {T3 = T1}.                    gate(s1,down,s2,T1,T2,T3) :-
train(s1,in,s2,T1,T2,T3) :-                      {T3 = T2,
       {T1 - T2 > 2,                              T1 - T2 < 1}.
       T3 = T2}.                     gate(s2,raise,s3,T1,T2,T3) :-
train(s2,out,s3,T1,T2,T2).                       {T3 = T1}.
train(s3,exit,s0,T1,T2,T3) :-        gate(s3,up,s0,T1,T2,T3) :-
       {T3 = T2,                           {T3 = T2, T1 - T2 > 1,
       T1 - T2 < 5}.                                 T1 - T2 < 2}.
train(X,lower,X,T1,T2,T2).           gate(X,approach,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).            gate(X,in,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).           gate(X,out,X,T1,T2,T2).
                                     gate(X,exit,X,T1,T2,T2).
contr(s0,approach,s1,T1,T2,T1).
contr(s1,lower,s2,T1,T2,T3) :- {T3 = T2, T1 - T2 = 1}.
contr(s2,exit,s3,T1,T2,T1).
contr(s3,raise,s0,T1,T2,T2) :- {T1-T2 < 1}.
contr(X,in,X,T1,T2,T2).              contr(X,out,X,T1,T2,T2).
contr(X,up,X,T1,T2,T2).              contr(X,down,X,T1,T2,T2).

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

Given the query:

```
   | ?- driver(s0, s0, s0, T, Ta, Tb, Tc, X, R).
```

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

```
R = [(approach,A),(lower,B),(down,C),(in,D),(out,E),
 (exit,F),(raise,G),(up,H)|R],
X = [approach,lower,down,in,out,exit,raise,up | X] ? ;

R= [(approach,A),(lower,B),(down,C),(in,D),(out,E),
 (exit,F),(raise,G),(approach,H),(up,I)|R],
X = [approach,lower,down,in,out,exit,raise,approach,up|X] ? ;

no
```

A call to coinductively defined sublist/2 predicate (not shown here) can then be used to check the safety property that the signal `down` occurs before `in` by checking that the infinite list `Y = [down, in | Y]` is coinductively contained in the infinite string `X` above. This ensure that the system satisfies the safety property, namely, that the gate is down before the train is in the gate area. A similar approach can be used to verify the liveness property, namely that the gate will eventually go up [9], by finding the maximum difference between the times the gate goes down and later comes up.

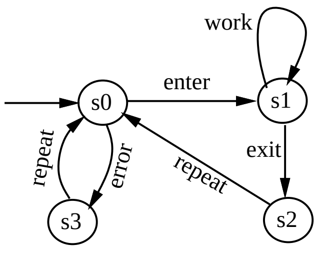
Note that from the answers above, one can see that another train can approach before the gate goes up, however, this behavior is entirely consistent as the gate will go up and will come down again, by the time the second train arrives in the gate area (see Figure 2). We can find out the minimum time that must intervene between two trains for the system to remain safe by finding he minimum value the time that elapses between two approach signals given the above constraints (answer is computed to be 7 units of time).

4.3 Verification of Nested Finite and Infinite Automata

We next illustrate application of coinductive logic programming to verification in which infinite (coinductive) and finite (inductive) automata are nested. It is well known that reachability-based inductive techniques are not suitable for verifying liveness properties [17]. 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 [24] and that fairness properties can be specified in terms of alternating fixed-point temporal logic formulas [11]. Earlier we showed that co-inductive LP allows one to verify a class of liveness properties in the absence of fairness constraints. Coinductive LP further permits us to verify a more general class of all liveness properties that can only be verified in the presence of fairness constraints.

Essentially, a coinductive LP based approach demonstrates that if a model satisfies the fairness constraint then, it also satisfies the liveness property. This is achieved by composing a program P_M, which encodes the model, with a program P_F, which encodes the fairness constraint and a program P_{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 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 (which can be programmed on our coinductive LP implementation). Consider the model shown in Figure 3, consisting of four states. 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 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` (this can also happen infinitely often). The system is modeled by the Prolog code shown in Figure 3.



```
:- coninductive 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 :- work.
work.
enter.                repeat.
exit.                 error.
```

Fig. 3. Nested Automata

This simple example illustrates the power of co-logic programming (co-LP, for brevity, that contains both inductive and coinductive LP) when compared to purely inductive or purely coinductive LP. Note that the computation represented by the state machine in the example consists of two stratified loops, represented by recursive predicates. The outer loop (predicate `state/2`) is coinductive and represents an infinite computation (hence it is declared as coinductive as we are interested in its gfp). The inner loop (predicate `work/0`) is inductive and represents a bounded computation (we are interested in its lfp). The semantics therefore evaluates `work/0` using SLD resolution and `state/2` using co-SLD resolution.

The property that we would like to verify is that the computation in the state `s1` represented by `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 (infinite) behavior of the system if `error` is encountered.

5 Applications to Non-Monotonic Reasoning

We next consider application of coinductive LP to non-monotonic reasoning, in particular its manifestation as *answer set programming* (ASP) [5, 14]. ASP has been proposed as an elegant way of introducing non-monotonic reasoning into logic programming [3]. ASP has been steadily gaining popularity since its inception due to its applications to planning, action-description, AI, etc.

We believe that if one were to add negation as failure to coinductive LP then one would obtain something resembling answer set programming. To support negation as failure in coinductive LP, we have to extend the coinductive hypothesis rule: given the goal `not(G)`, if we encounter `not(G')` during the proof, and `G` and `G'` are unifiable, then `not(G)` succeeds. Since the coinductive hypothesis rule provides a method for goal-directed execution of coinductive logic programs, it can also be used for goal-directed execution of answer set programs. As discussed earlier, coinduction is a technique for specifying unfounded set; likewise, answer set programs are also recursive specifications (containing negation as failure) for computing unfounded sets. Obviously, coinductive LP and ASP must be related.

All approaches to implementing ASP are based on bottom up execution of finitely grounded programs. If a top-down execution scheme can be designed for ASP, then ASP can be extended to include predicates over general terms. We outline how this can be achieved using our top-down implementation of coinduction. In fact, a top-down interpreter for ASP (restricted to propositions at present) can be trivially realized on top of our implementation of coinductive LP. Work is in progress to implement a top-down interpreter for ASP with general predicates [15].

5.1 A Top-Down Algorithm for Computing Answer Sets

In top-down execution of answer set programs, given a (propositional) query goal `Q`, we are interested in finding out all the answer sets that contain `Q`, one by one, via backtracking. If `Q` is not in any answer set or if there are no answer sets at all, then the query should fail. In the former case, the query `not(Q)` should succeed and should enumerate all answer sets which do not contain `Q`, one by one, via backtracking.

The top down execution algorithm is quite simply realized with the help of coinduction. The traditional Gelfond-Lifschitz (GL) method [3] starts with a candidate answer set, computes a residual program via the GL-transformation, and then finds the lfp of the residual program. The candidate answer set is an answer set, if it equals the lfp of the residual program. Intuitively, in our top down execution algorithm, the propositions in the candidate answer set are regarded as hypotheses which are treated as facts during top-down coinductive execution. A call is said to be a *positive* call if it is in the scope of even number of negations, similarly, a call is said to be a *negative* call if it is in the scope of odd number of negations.

The top down algorithm works as follows: suppose the current call (say, `p`) is a positive call, then it will be placed in a *positive coinductive hypothesis set* (PCHS), a matching rule will be found and the Prolog-style expansion done. The new resolvent will be processed left to right, except that every positive call will be continued to be placed in the positive hypothesis set, while a negative call will be placed in the *negative coinductive hypothesis set* (NCHS). If a positive call `p` is encountered again, then if `p` is in PCHS, the call immediately succeeds; if it is in NCHS, then there is an inconsistency and backtracking takes place. If a negative call (say, `not(p)`) is encountered for the first time, `p` will be placed in the NCHS. If a negative proposition `not(p)` is encountered later, then if `p` is in NCHS, `not(p)` succeeds; if `p` is in PCHS, then there is an inconsistency and backtracking takes place. Once the execution is over with success, (part of) the *potential* answer set can be found in the PCHS. The set NCHS contains propositions that are *not* in the answer set.

Essentially, the algorithm explicitly keeps track of propositions that are in the answer set (PCHS) and those that are not in the answer set (NCHS). Any time, a situation is encountered in which a proposition is both in the answer set and not in the answer set, an inconsistency is declared and backtracking ensues.

We still need one more step. ASP can specify the falsification of a goal via constraints. For example, the constraint `p :- q, not p.` restricts `q` (and `p`) to not be in the answer set (unless `p` happens to be in the answer via other rules). For such rules of the form

```
   p :- B.
```

if `not(p)` is reachable via goals in the body `B`, we need to explicitly ensure that the potential answer set does not contain a proposition that is falsified by this rule.

Given an answer set program, a rule `p :- B.` is said to be non-constraint rule (NC-rule) if `p` is reachable through calls in the body `B` through an even number of negation as failure calls, otherwise it is said to be a constraint rule (C-rule). Thus, given the ASP program:

```
p :- a, not q.                .......(i)
q :- b, not r.                .......(ii)
r :- c, not p.                .......(iii)
q :- d, not p.                .......(iv)
```

rules (i), (ii) and (iii) are C-rules, while (i) and (iv) are NC-rules. A rule can be both an NC-rule as well as a C-rule (such as rule (i)). NC-rules will be used to compute the potential answers sets, while C-rules will only be used to reject or accept potential answer sets. Rejection or acceptance of a potential answer set is accomplished as follows: For each C-rule of the form `r`$_i$ `:- B`, where `B` directly or indirectly leads to `not(r`$_i$`)`, we construct a new rule:

`chk_r`$_i$ `:- not(r`$_i$`), B.`

Next, we construct a new rule:

`nmr_check :- not(chk_r`$_1$`), not(chk_r`$_2$`), ..., not(chk_r`$_i$`), ...`

Now, the top level query, `?-Q`, is transformed into: `?- Q, nmr_check.` `Q` will be executed using only the NC-rules to generate potential answer sets, which will be subsequently either rejected or accepted by the call to `nmr_check`. If `nmr_check` succeeds, the potential answer set is an answer set. If `nmr_check` fails, the potential answer set is rejected and backtracking occurs.

For simplicity of illustration, we assume that for each NC-rule, we construct its negated version which will be expanded when a corresponding negative call is encountered (in the implementation, however, this is done implicitly). Thus, given an NC-rule for a proposition `p` of the form:

`p :- B`$_1$`.`
`p :- B`$_2$`.`
`...`
`p :- B`$_i$`.`
`...`

its negated version will be:

`not_p :- not(B`$_1$`), not(B`$_2$`), ..., not(B`$_i$`), ...`

If a call to `not(p)` is encountered, then this negated `not_p` rule will be used to expand it.

Note, finally, that the answer sets reported may be partial, because an answer set may be a union of multiple independent answer subsets, and the other subsets may not be deducible from the query goal due to the nature of the rules. The top-down algorithm for computing the (partial) answer set of an ASP program can be summarized as follows.

1. Initialize PCHS and NCHS to empty (these are maintained as global variables in our implementation of top-down ASP atop our coinductive YAP implementation). Declare every proposition in the ASP as a coinductive proposition.
2. Identify the set of C-rules and NC-rules in the program.
3. Assert `chk_r`$_i$ rule for every C-rule with `r`$_i$ as head and build the `nmr_check` rule; append the call `nmr_check` to the initial query.

4. For each NC-rule, construct its negated version.
5. For every positive call p: if p ∈ PCHS, then p succeeds coinductively and the next call in the resolvent is executed, else if p ∈ NCHS, then there is an inconsistency and backtracking ensues, else (p is not in PCHS or in NCHS) add p to PCHS and expand p using NC-rules that have p in the head (create a choice-point if there are multiple matching rules).
6. For every negative call of the form **not(p)**: if p ∈ NCHS, then **not(p)** succeeds coinductively and the next call in the resolvent is executed, else if p ∈ PCHS, then there is an inconsistency and backtracking ensues, else (p is not in PCHS or in NCHS) add p to NCHS and expand **not(p)** using the negated **not_p** rule for p.

Next, let's consider an example. Consider the following program:

```
p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).
```

After, step 1-4, we obtain the following program.

```
:- coinductive p/0, q/0, r/0.

p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).

chk_p :- not(p), not(q).
chk_q :- not(q), not(r).
chk_r :- not(r), not(p).

not_p :- q.
not_q :- r,p.
not_r :- p.

nmr_chk :- not(chk_p), not(chk_q), not(chk_r).
```

The ASP program above has {q,r} as the only answer set. Given the transformed program, the query: ?- q will produce {q} as the answer set (with p known to be not in the answer set). The query ?- r will produce the answer set {q,r} (with p known to be not in the answer set). It is easy to see why the first query ?- q will not deduce r to be in the answer set: there is nothing in the NC-rules that relates q and r.

5.2 Correctness of the Top-down Algorithm

We next outline a proof of correctness of the top-down method. First, note that the above top down method corresponds to computing the greatest fix point of the residual program rather than the least fix point. Second, we argue that the Gelfond-Lifschitz method for checking if a given set is an answer set [5] should compute the greatest fix point of the residual program instead of the least fixed point. A little thought will reflect that circular reasoning entailed by rules such as

```
  p :- p.
```

is present in ASP through rules of the form:

```
  p :- not(q).
  q :- not(p).
```

If we extend the GL method, so that instead of computing the lfp, we compute the gfp of the residual program, then the GL transformation can be modified to remove positive goals as well. Given an answer set program, whose answer set is guessed to be `A`, the modified GL transform then becomes as follows:

1. Remove all those rules whose body contains `not(p)`, where $\texttt{p} \in \texttt{A}$.
2. Remove all those rules whose body contains `p`, where $\texttt{p} \notin \texttt{A}$.
3. From body of each rule, remove all goals of the form `not(p)`, where $\texttt{p} \notin \texttt{A}$.
4. From body of each rule, remove all positive goals `p`, where $\texttt{p} \in \texttt{A}$.

After application of this transform, the residual program is a set of facts. If this set of facts is the same as `A`, then `A` is an answer set. It is easy to see that our top-down method mimics the modified GL-transformation (note, however, that our top-down algorithm can be easily modified to work with the original GL method which computes the lfp of the residual program: we merely have to disallow making inference from positive coinductive loops of the form `p :- p.`).

In the top down algorithm, whenever a positive (resp. negative) predicate is called, it is stored in the positive (resp. negative) coinductive hypothesis set. This is equivalent to removing the positive (resp. negative) predicate from the body of all the rules, since a second call to predicate will trivially succeed by the principle of coinduction. Given a predicate `p` (resp. `not(p)`) where `p` is in the positive (resp. negative) coinductive hypothesis set, if a call is made to `not(p)` (resp. `p`), then the call fails. This amounts to 'removing the rule' in GL transform [2].

The top down method described above can also be extended for deducing answers sets of programs containing first order predicates [15].

6 Conclusions

In this paper we gave an introduction to coinductiive logic programming. Practical applications of coinductive LP to verification and model checking and to non-monotonic reasoning were also discussed. Coinductive reasoning can also be included in first order theorem proving in a manner similar to coinductive LP [15].

Acknowledgments: We are grateful to Vítor Santos Costa and Ricardo Rocha.

References

1. R. Alur, D.L. Dill. A theory of timed automata. *TCS* 126:183-235, 1994.
2. A. Bansal. Towards next generation logic programming systems. Ph.D. thesis *forthcoming.*
3. C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
4. J. Barwise, L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena.* CSLI Publications, 1996.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, Logic Programming: Proc. of the Fifth International Conference and Symposium, pages 1070-1080, 1988.
6. D. Goldin, D. Keil. Interaction, Evolution and Intelligence. Proc. Congress on Evolutionary Computing. 2001.
7. J. Goguen, K. Lin. Behavioral Verification of Distributed Concurrent Systems with BOBJ. In Proc. Conference on Quality Software, IEEE Press 2003, pages 216-235.
8. A. Gordon. A Tutorial on Co-induction and Functional Programming. Springer Workshops in Computing (Functional Programming). pp. 78-95. 1995.
9. G. Gupta, E. Pontelli. Constraint-based Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium '97.* pp. 230-239.
10. B. Jacobs. Introduction to Coalgebra: Towards Mathematics of States and Observation. Draft manuscript.
11. X. Liu, C.R. Ramakrishnan, S.A. Smolka. Fully local and efficient evaluation of alternating fixed-points, *TACAS '98*, LNCS, 1384, Springer-Verlag, 1998.
12. J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd. edition, 1987.
13. A. Mallya. Multivalued Deductive Multi-valued Model Checking. Ph.d. thesis. UT Dallas. 2006.
14. W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In the Logic Programming Paradigm: a 25-Year Perspective, pages 375-398, Spring-Verlag. 1999.
15. R. Min. Coinduction in monotonic and non-monotonic reasoning. Ph.D. thesis *forthcoming.*
16. B. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, 2002.
17. A. Podelski, A. Rybalchenko. Transition Predicate Abstraction and Fair Termination, *POPL '05*, pp. 132-144, ACM Press, 2005.
18. Y. S. Ramakrishna et al. Efficient Model Checking Using Tabled Resolution. *CAV 1997.* pp. 143-154.
19. V. Santos Costa, R. Rocha. The YAP Prolog System.
20. V. Schuppan, A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. ENTCS 149(1): 79-96 (2006)
21. Luke Simon. Extending Logic Programming with Coinduction. Ph.D. Thesis. 2006.
22. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive Logic Programming. ICLP'06. Springer Verlag LNCS 4079. pp. 330-344.
23. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-Logic Programming. Proc. ICALP'07, to appear.
24. M. Vardi, Verification of Concurrent Programs: The Automata-Theoretic Framework, *LICS '87*, pp. 167-176, IEEE, 1987.
25. P. Wegner, D. Goldin. Mathematical models of interactive computing. Brown University Technical Report CS 99-13. 1999.