

Infinite Computation, Co-induction and Computational Logic

Gopal Gupta

Neda Saeedloei, Brian DeVries, Richard Min, Kyle Marple, Feliks Kluzniak

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

Abstract. We give an overview of the coinductive logic programming paradigm. We discuss its applications to modeling ω -automata, model checking, verification, non-monotonic reasoning, developing SAT solvers, etc. We also discuss future research directions.

1 Introduction

Coinduction is a technique for reasoning about unfounded sets [12], behavioral properties of programs [2], and proving liveness properties in model checking [16]. Coinduction also provides the foundation for lazy evaluation [9] and type inference [21] in functional programming as well as for interactive computing [33].

Coinduction is the dual of induction. Induction corresponds to well-founded structures that start from a basis which serves as the foundation: e.g., natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality. For example, the inductive definition of lists of numbers is as follows: (i) $[]$ (empty list) is a list (initiality); (ii) $[H|T]$ is a list if T is a list and H is some number (iteration); and, (iii) the set of lists is the smallest set satisfying (i) and (ii) (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 (*LFP*) interpretations of recursive definitions.

Coinduction eliminates the initiality condition and replaces the minimality condition with maximality. The coinductive definition of *infinite* lists of numbers is: (i) $[H|T]$ is a list if T is a list and H is some number (iteration); and, (ii) the set of lists is the largest set of lists satisfying (i) (maximality). There is no base case in a coinductive definition, and while it may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point (*GFP*) interpretation of recursive definitions: namely, the set of all infinite lists of numbers. (Note, however, that if we had a recursive definition with a base case, then under the coinductive interpretation, the set would contain both finite and infinite-sized lists.) A coinductive proof is essentially an infinite-length proof.

2 Coinduction and Logic Programming

Coinduction has been incorporated in logic programming in a systematic way only recently [30, 10, 29]. An operational semantics—similar to SLD resolution—was given for computing those answers to a query that are in the greatest fixed point of a logic program (the semantics is discussed below).

Consider the list example discussed in Sec. 1. 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 standard logic programming the query `?- stream(X).` fails, since the model of P2 does not contain any instances of `stream/1`. The problems are two-fold: (i) the Herbrand universe does not contain infinite terms; (ii) the least Herbrand model does not allow for infinite proofs, such as the proof of `stream(X)`; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [2]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties. 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 Herbrand (or co-Herbrand) base* [15], and *maximal models* (computed using greatest fixed-points).

Under the coinductive interpretation of P2, the query `?- stream(X).` should produce all infinite sized streams as answers, e.g., `X = [1, 1, 1, ...]`, `X = [1, 2, 1, 2, ...]`, etc. The model of P2 does contain instances of `stream/1` (but proofs may be of infinite length).

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

The operational semantics of coinductive logic programming is given in terms of the *coinductive hypothesis rule*: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C , 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 $Y = [1|Y]$.

In coinductive logic programming the alternative computations started by a call are not only those that begin with unifying the call and the head of a clause, but also those that begin with unifying the call and one of its ancestors in a proof tree.

Regular logic programming execution extended with the coinductive hypothesis rule is termed *co-logic programming* [29]. The coinductive hypothesis rule will work for only those infinite proofs that are *regular* (or *rational*), i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved an infinite number of times. More general implementations of coinduction are possible [29]. More complex examples of coinductive LP program can be found elsewhere [10].

Even with the restriction to rational proofs, there are many applications of coinductive logic programming, some of which are discussed next. These include model checking, modeling ω -automata, non-monotonic reasoning, etc. Implementations of co-LP have been realized: a meta-interpreter that includes both tabled and coinductive logic programming is available from the authors [14]. Recently, SWI Prolog [32] has also added support for coinduction.

The traditional model of declarative computing is based on recursion: a problem is solved by breaking it down into smaller subproblems, which are broken down further, until base cases are reached that are trivially solved. Solutions to the subproblems are then used to synthesize a solution to the problem. This model of computation is based on the theory of well-founded sets, and is not appropriate for computations that are cyclical in nature. Such cyclical computations arise when the solutions to the subproblems of a given problem are mutually interdependent. Solving the problem involves establishing the consistency of the interdependent solutions to the subproblems, and coinduction/corecursion provides the necessary framework [2].

Intuitively, a computational model in which both *LFP*-based and *GFP*-based computations can be conveniently expressed will allow one to elegantly express any computable function. As a result, logic programming extended with coinduction provides a basis for many powerful applications, in areas of verification, non-monotonic reasoning, modal logics, etc. A number of challenges remain in co-LP. These relate to:

- **Nesting of inductive and coinductive computations.** In such cases giving the proper semantics may not be simple. This is illustrated by the following program

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

where *q* has standard inductive semantics while *p* has coinductive semantics. A call to *q* should fail, and so should a call to *p* (since it calls *q*), but a call to *p* may succeed coinductively. (We will return to this issue in Sec. 6.)

- **Reporting isomorphic solutions only once.** Consider the coinductive predicate *p/1*.

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

The query *p(A)* will produce a solution *A* = [1|*A*]. However, if normal call expansion is also considered as an alternative along with the coinductive

hypothesis rule, then an infinite number of solutions will be produced, all of them isomorphic to the solution $A = [1 \mid A]$:

$$\begin{aligned} A &= [1, 1 \mid A] \\ A &= [1, 1, 1 \mid A] \\ A &= [1, 1, 1, 1 \mid A] \\ &\dots \end{aligned}$$

One could avoid this problem by memoing the cyclical solutions, however, an operational semantics that incorporates this memoing has yet to be investigated [6].

- **Efficient implementation of co-LP.** The current operational semantics of co-LP require that every coinductive call be unified with every ancestor call to the same predicate: a naive implementation may be quite inefficient.
- **Reporting all solutions.** Consider the following program where both p and q are coinductive.

$$\begin{aligned} p([a|X]) &:- q(X). \\ q([b|X]) &:- p(X). \end{aligned}$$

The query $?- p(A)$ has an infinite number of solutions, both rational and irrational. In general, systematic enumeration of even the rational solutions requires a fair execution strategy (e.g., breadth-first search). Such an execution strategy should be coupled with a mechanism that eliminates redundant isomorphic solutions.

As discussed earlier, there are many extensions and applications of coinduction and co-LP. In the rest of the paper we outline some of them. We discuss the extension of co-SLD resolution with negation (termed co-SLDNF resolution) and its use in realizing goal-directed non-monotonic reasoners and SAT solvers. We also discuss how co-LP can be extended with constraints and used for modeling complex real-time systems, for example, how timed ω -automata, timed push-down ω -automata, and timed grammars can be elegantly modeled. Co-LP can also be used for developing executable operational semantics of Π -calculus and linear-time temporal logic (LTL). Relationship between co-LP and various ω -automata (Büchi, Rabin, Streett) is also discussed.

3 Model Checking with Co-LP

Model checking is a popular technique used for verifying hardware and software systems. It is done by constructing a model of the system as a finite-state Kripke structure and then determining whether the model satisfies various properties specified as temporal logic formulae [3]. The verification is performed by means of systematically searching the state space of the Kripke structure for a counterexample that falsifies the given formula. The vast majority of properties that are to be verified can be classified into *safety* properties and *liveness* properties. Intuitively, a safety property asserts that “nothing bad will happen”, while a liveness property asserts that “something good will eventually happen”.

It is well known that safety properties can be verified by reachability analysis, i.e, if a counter-example to the postulated property exists, it can be finitely

determined by enumerating all the reachable states of the Kripke structure. In the context of Logic Programming, verification of safety properties amounts to computing elements of the *LFP* of a program, and is thus elegantly handled by standard LP systems extended with tabling [24]. Verification of liveness properties is less straightforward, because counterexamples take the form of infinite traces, which are semantically equivalent to elements of the the *GFP* of a logic program: co-LP is more suitable for directly computing such counterexamples without the expensive transformations required by some other approaches suggested in the literature [24].

Intuitively, a state is 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 determining if any of these states constitute valid counterexamples.

To demonstrate the power of coinductive logic programming, we show how an interpreter for linear temporal logic can be written very elegantly. In LTL, one checks if a temporal logic formula is true along a path. Temporal operators whose meaning is given in terms of *LFPs* are realized via tabled logic programming, while those whose meaning is given in terms of *GFPs* are realized using coinductive logic programming.

The `verify/3` predicate in the interpreter of Fig. 1 takes as input a state and an LTL formula, and produces as an answer a path for which this formula is true. To verify that an LTL formula F holds in a given state S , it has to be negated. The negated formula is then converted to negation normal form (i.e., a negation symbol can appear only next to a proposition), and then given as input to the `verify/3` predicate along with the state S . If the call to `verify/3` fails, then there is no path on which the negated formula holds, implying that the formula F holds in state S . In contrast, if `verify/3` returns a path as an answer, then that specific path is a counterexample for which the original formula F does not hold. Note that the Kripke structure is represented as a transition table using the `trans/2` predicate, while information about which proposition(s) holds in which state(s) is given by the `holds/2` predicate. The temporal operators X , F , G , U and R are represented with corresponding lower case letters, while \wedge , \vee , and \sim represent \wedge , \vee , and negation respectively. The program in Fig. 1 should be self-explanatory.

Note that while this program is elegant, its soundness and completeness depend on the execution strategy used for realizing coinduction, and on how the interleaving of coinduction and tabling (induction) is handled. The issue of interleaving of coinduction and induction is discussed in section 6.2.

4 Negation in Co-LP

As mentioned earlier, SLD resolution extended with the coinductive hypothesis rule is called co-SLD resolution. Co-SLDNF resolution further extends co-SLD resolution with negation as failure [15]. Essentially, it augments co-SLD with the negative coinductive hypothesis rule, which states that if a negated call `not(p)` is

```

verify( S, g A, Path ) :- coverify( S, g A, Path ).
verify( S, A r B, Path ) :- coverify( S, A r B, Path ).
verify( S, f A, Path ) :- tverify( S, f A, Path ).
verify( S, A u B, Path ) :- tverify( S, A u B, Path ).
verify( S, A, [ S ] ) :- proposition( A ), holds( S, A ).
verify( S, ~ A, [ S ] ) :- proposition( A ), \+ holds( S, A ).
verify( S, A ^ B, Path ) :- verify( S, A, PathA ),
                             verify( S, B, PathB ),
                             prefix( PathA, PathB, Path ).

verify( S, A v B, Path ) :- verify( S, A, Path )
                             ; verify( S, B, Path ).

verify( S, x A, [ S | P ] ) :- trans( S, S2 ), verify( S2, A, P ).

:- tabled tverify/3.
tverify( S, f A, Path ) :- verify( S, A, Path )
                           ; verify( S, x f A, Path ).
tverify( S, A u B, Path ) :- verify( S, B, Path )
                             ; verify( S, A ^ x( A u B ), Path ).

:- coinductive coverify/3.
coverify( S, g A, Path ) :- verify( S, A ^ x g A, Path ).
coverify( S, A r B, Path ) :- verify( S, A ^ B, Path ).
coverify( S, A r B, Path ) :- verify( S, B ^ x( A r B ), Path ).

prefix( Prefix, Path, Path ) :- append( Prefix, _, Path ), !.
prefix( Path, Prefix, Path ) :- append( Prefix, _, Path ), !.

```

Fig. 1. An LTL model-checker

encountered during resolution, and another call to `not(p)` has been seen before in the same computation, then `not(p)` coinductively succeeds [20].

To implement co-SLDNF resolution, the set of positive and negative calls has to be maintained in the *positive hypothesis table* (PHT) and *negative hypothesis table* (NHT) respectively. An attempt to place the same call in both tables will induce failure of the computation. The framework based on maintaining a pair of sets (corresponding to a partial interpretation of success set and failure set, resulting in a partial model [8]) provides a good basis for the operational semantics of co-SLDNF resolution [20].

One of the most interesting applications of co-SLDNF resolution is in obtaining goal-directed strategies for executing answer set programs. Answer Set Programming (ASP) [1] is a powerful paradigm for performing non-monotonic reasoning within logic programming. Current ASP implementations are restricted to grounded range-restricted function-free normal programs [1] and use essentially an evaluation strategy that is bottom-up. Co-LP with co-SLDNF resolution has allowed the development of top-down goal evaluation strategies for ASP [10], which in turn allows ASP to be extended to predicates [18]. This co-LP based method eliminates the need for grounding, allows functions, and effectively handles a large class of predicate ASP programs including possibly infinite ASP

programs. We have designed and implemented the techniques and algorithms to execute propositional and predicate ASP programs. Our techniques are applicable to the full range of propositional answer set programs, but for predicate ASP we are restricted to programs that are *call-consistent* or *order-consistent* [19].

We illustrate co-SLDNF resolution by showing how a boolean SAT solver can be readily obtained. A SAT solver attempts to find a truth assignment that satisfies a propositional formula. Our technique based on co-SLDNF resolution essentially makes assumptions about truth values of propositions, and checks that these assumptions bear out throughout the formula. A propositional formula X must satisfy the following rules:

- (1) $t(X) :- \text{not}(\text{neg}(t(X)))$.
- (2) $\text{neg}(t(X)) :- \text{not}(t(X))$.

The predicate $t/1$ is a truth-assignment (or a valuation) where X is a propositional Boolean formula to be checked for satisfiability. Clause (1) asserts that $t(X)$ is true if there is no counter-case for $\text{neg}(t(X))$ (that is, $\text{neg}(t(X))$ is false (coinductively), with the assumption that $t(X)$ is true (coinductively)). Clause (2) asserts that $\text{neg}(t(X))$ is true if there is no counter-case for $t(X)$. Next, any well-formed propositional Boolean formula constructed from a set of propositional symbols and logical connectives and in conjunctive normal form (CNF) can be translated to a query for the co-LP program as follows: first, each propositional symbol p is translated to $t(p)$. Second, any negated proposition, that is $\neg t(p)$, is translated to $\text{neg}(t(p))$. Third, the Boolean operators \wedge and \vee are translated to Prolog's conjunction ($,$) and disjunction ($;$) respectively.

The predicate $t(X)$ determines the truth-assignment of formula X (if X is true, $t(X)$ succeeds; else it fails). Note that each query is a Boolean expression whose satisfiability is to be coinductively determined. As an example, the formula $(p1 \vee p2 \vee p3) \wedge (p1 \vee \neg p3) \wedge (\neg p2 \vee \neg p4)$ will be translated into the query

$(t(p1); t(p2); t(p3)), (t(p1); \text{not}(t(p3))), (\text{not}(t(p2)); \text{not}(t(p4)))$.

Propositions that must be assigned the value true will be found in the PHT, while those that must be assigned the value false will be found in NHT. The answer printed by our prototype SAT solver based on co-LP is shown below (the `print_ans` command prints the answer):

```
?- (t(p1);t(p2);t(p3)),(t(p1); neg(t(p3))),
    (neg(t(p2)); neg(t(p4))), print_ans.
Answer:
    PHT == [t(p1)]
    NHT == [t(p2)]
yes
```

The answer indicates that this propositional formula is satisfiable if $p1$ is assigned true and $p2$ is assigned false.

5 Modeling Complex Real-time Systems

Next, we consider the use of co-LP to model ω -automata and their various extensions. As is well known, finite state automata as well as grammars can be elegantly modeled as inductive logic programs, and recognizers for them are readily

obtained (due to the relational nature of logic programming, these recognizers can also systematically generate all the words of the recognized language). Similar recognizers can be obtained for ω -automata and ω -grammars with co-LP. Moreover, co-LP can be augmented with constraint logic programming (e.g., over the real numbers [CLP(R)] [13]), allowing elegant modeling of real-time systems. We next give an overview of application of co-LP (combined with CLP(R)) to modeling complex real-time systems; in particular, we discuss how pushdown timed automata, timed grammars and timed π -calculus can be elegantly represented. The use of co-LP together with CLP(R) for modeling these system leads one naturally towards coinductive constraint programming, an area that we are beginning to investigate.

5.1 Pushdown ω -Automata and ω -Grammars

A timed automaton is an ω -automaton extended with clocks or stopwatches. A pushdown timed automaton (PTA) [4] extends a timed automaton with a stack, just as a pushdown automaton extends a finite automaton. Transitions from one state to another are made not only on the alphabet symbols of the language but also on constraints imposed on clocks (e.g., at least 2 units of time must have elapsed). Transitions may result in changing the stack by performing the push and pop operations. A PTA recognizes a language consisting of timed words, where a timed word is an infinite sequence of symbols from the alphabet, each of them paired with a time-stamp. The sequence of symbols in an infinite sequence accepted by a PTA must obey the rules of syntax laid down by the underlying untimed pushdown automaton, while the time-stamps must obey the timing constraints. Additionally, the stack must be empty whenever a final state is entered.

To model and reason about PTA (and timed grammars) we must account for the fact that: (i) the underlying language is context free, not regular; (ii) accepted strings are infinite; and (iii) clock constraints are posed over continuously flowing time. All three aspects can be elegantly handled—and PTAs naturally modeled—within co-LP. Additionally, the definite clause grammar (DCG) facility of Prolog allows one to easily obtain a parser for a grammar. Through co-LP, one can develop language processors that can act as recognizers for ω -pushdown automata and ω -grammars. Further, with incorporation of CLP(R) one can also model the timing aspects.

A PTA specification can be automatically transformed to a coinductive program extended with CLP(R) [28]. The method takes the description of a PTA and generates a coinductive constraint logic program over reals. The generated logic program models the PTA as a collection of transition rules (one rule per transition): each rule is extended with stack actions as well as clock constraints. Our coinductive constraint logic programming realization of pushdown timed automata and timed automata can be regarded as a general framework for modeling/verifying real-time systems.

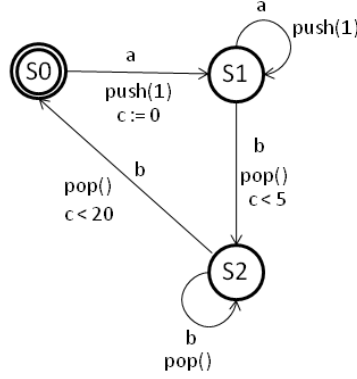


Fig. 2. A Pushdown Timed Automaton

Fig. 2 shows an example of a pushdown timed automaton. The following is an encoding of this PTA in the form of a coinductive logic program with constraints that was generated automatically by our system:

```

trans(s0, a, s1, T, Ti, To, [], [1]) :- {To = T}.
trans(s1, a, s1, T, Ti, To, C, [1 | C]) :- {To = Ti}.
trans(s1, b, s2, T, Ti, To, [1 | C], C) :- {T - Ti < 5, To = Ti}.
trans(s2, b, s2, T, Ti, To, [1 | C], C) :- {To = Ti}.
trans(s2, b, s0, T, Ti, To, [1 | C], C) :- {T - Ti < 20, To = Ti}.

:- coinductive(pta/6).
pta([ X | R], Si, T, Ti, C1, [ (X, T) | S]) :-
    trans(Si, X, So, T, Ti, To, C1, C2),
    {Ta > T}, pta(R, So, Ta, To, C2, S).
  
```

Once a timed system is modeled as a coinductive CLP(R) program, the program can be used to (i) check whether a particular timed string will be accepted or not; (ii) systematically generate all the possible timed strings that can be accepted; or, (iii) verify safety and liveness properties by posing appropriate queries.

The co-LP and CLP(R) based framework has been used to study the *generalized railroad crossing (GRC) problem* [11] and to verify its safety and utility properties by posing simple queries [28]. This approach based on coinductive CLP(R) for the GRC is considerably more elegant and simpler than other approaches (e.g., [22]).

Context Free Grammars can be extended to define languages consisting of (infinite) timed words. These extended grammars are called timed context-free ω -grammars (ω -TCFG) [27]. Such languages are useful for modeling complex real-time systems [25] that run forever.

As an example of an ω -TCFG, consider a language in which each sequence of a 's is followed by a sequence of an equal number of b 's, with each accepted string having at least two a 's and two b 's. For each pair of equinumerous sequences of a 's and b 's, the *first* symbol b must appear within 5 units of time from the *first* symbol a and the *final* symbol b must appear within 20 units of time from the

first symbol a . The grammar annotated with clock expressions is shown below: c is a clock which is reset when the first symbol a is seen.

$$\begin{aligned} S &\rightarrow R S \\ R &\rightarrow a \{c := 0\} T b \{c < 20\} \\ T &\rightarrow a T b \\ T &\rightarrow a b \{c < 5\} \end{aligned}$$

Definite Clause Grammars (DCG) [31] together with CLP(R) and coinduction can be used to develop an effective and elegant method for parsing the languages generated by ω -TCFGs. Specification of an ω -TCFG can be automatically transformed to a DCG augmented with coinduction and CLP(R) [27]. The resulting coinductive constraint logic program acts as a practical parser. Given this program, one can pose queries to check whether a timed word satisfies the timing constraints imposed by the timed grammar. Alternatively, one can generate possible legal timed words (note that in this case a CLP(R) system will output timed words in which time-stamps will be represented as variables; constraints that these time-stamps must satisfy will be output as well). Finally, one can verify properties of this timed language (e.g., checking the simple property that all the a 's are generated within 5 units of time, in any timed string that is accepted).

5.2 Timed π -calculus

The π -calculus was introduced by Milner et al. [17] with the aim of modeling concurrent/mobile processes. The π -calculus provides a conceptual framework for describing systems whose components interact with each other. It contains an algebraic language for descriptions of processes in terms of the communication actions they can perform. Theoretically, the π -calculus can model mobility, concurrency and message exchange between processes as well as infinite computation (through the '!' operation). Operational semantics of π -calculus can also be elegantly modeled with the help of coinduction. Specifically, to model the replication operator faithfully, one needs coinduction. The π -calculus can also be extended to real time, and executable operational semantics obtained for it using coinduction and constraints. The extension to time is helpful in modeling controller processes that control physical devices where a notion of real-time is important [25]. Several extensions of π -calculus with time have been proposed to overcome this problem (e.g., [5]); all these approaches discretize time rather than represent it faithfully as a continuous quantity.

For a complete encoding of the operational semantics of timed π -calculus, we must model three aspects of timed π -calculus processes: concurrency, infinite computation and time constraints/clock expressions. An executable operational semantics of π -calculus in logic programming (LP) has been developed [26, 25]. Channels are modeled as streams, *rational* infinite computations are handled by *coinduction* [29, 10] and concurrency is handled by allowing *coroutining* within logic programming computations. This operational semantics is extended with continuous real time, which we have modeled with *constraint logic programming over reals* [13]. The executable operational semantics, thus realized, automatically leads to an implementation of the timed π -calculus in the form of a coinduc-

tive coroutined constraint logic program that can be regarded as an interpreter for timed π -calculus expressions, and can be used for modeling and verification of real-time systems.

Note that there is past work on developing LP-based executable operational semantics of the π -calculus (but not timed π -calculus) [34] but it is unable to model infinite processes and infinite replication, since coinductive logic programming has been developed relatively recently [29, 10].

Note also that giving proper semantics to coinductive constraint logic programs is not straightforward, and a formal study of coinductive constraint programming must be undertaken. In our investigations thus far, to ensure soundness, we assume that all the clocks involved in a cycle are reset every time we go around the cycle [25].

6 Integrating Coinduction and Induction

We now turn our attention to the problems resulting from integrating coinductive logic programming with traditional (i.e., inductive) logic programming. As discussed earlier, the semantics of arbitrary interleaving of induction and coinduction are unclear. We begin this section with co-logic programming in its current form, along with some example applications and a discussion of its limitations, followed by a brief description of our ongoing efforts to relax these limitations and extend the expressiveness of the formalism.

6.1 Stratified Co-LP: Applications and Limitations

The current (in particular, operational) semantics of inductive and coinductive logic programming permit either a least or a greatest fixed point interpretation of a predicate: no reasonable semantics have yet been given to programs that involve mutual recursion between inductive and coinductive predicates. In practice, programs lacking this mutual recursion—known as *stratified* co-logic programs—are still capable of representing solutions to a wide range of problems.

We begin with the near-canonical example of filters over streams. Let S be an inductively defined set, such as the set of natural numbers. A *stream* over S is an infinite sequence of elements of S . A *filter* takes a stream as input and removes elements that are not in a designated subset of S .

Coinductive logic programs cannot construct finite proofs over arbitrary streams, but they can be applied to streams of the form uv^ω , where a finite suffix v is repeated ad infinitum. The following co-logic program filters a stream over the natural numbers so that only even numbers remain:

```

:- coinductive filter/2.
filter( [ H | T ], [ H | T2 ] ) :- even( H ), filter( T, T2 ).
filter( [ H | T ], T2 ) :- \+ even( H ), filter( T, T2 ).

even(0).
even( s( s( N ) ) ) :- even( N ).

```

Here, the `even` predicate is the usual inductive definition of even natural numbers, while `filter` is a coinductive definition of our desired filter. We can see that this program is stratified, as `filter` and `even` are not mutually recursive. We can make use of this program with queries such as the following:

```
?- L = [ 0, s( 0 ), s( s( 0 ) ) | L ], filter( L, L2 ).
```

`L2` will be bound to the list `L2 = [0, s(s(0)) | L2]`.

Let us also consider the case of an inductive list containing coinductive elements. Let $\mathbb{N}^+ = \mathbb{N} \cup \{\infty\}$. The following program filters finite (inductive) lists over \mathbb{N}^+ so that only even natural numbers and infinity remain (the latter is accomplished through a coinductive definition of `even`):

```
filter( [], [] ).
filter( [ H | T ], [ H | T2 ] ) :- even( H ), filter( T, T2 ).
filter( [ H | T ], T2 ) :- \+ even( H ), filter( T, T2 ).

:- coinductive even/1.
even( 0 ).
even( s( s( N ) ) ) :- even( N ).
```

In general, inductive and coinductive predicates can be mixed in co-logic programs, as long as there exists a clear, acyclic (i.e., stratified) hierarchy between them. Co-LP provides a natural paradigm for constructing definitions and proofs over such predicates.

Co-LP can also be used to construct infinite, rational proofs in a finite manner. In such cases, it is typical to augment SLD resolution with *tabling* (called SLG resolution), which memoizes calls and solutions to inductive predicates and by extension prevents construction of (rational) infinite-depth inductive proofs. The combination of coinductive and tabled logic programming is particularly useful for implementing solvers for fixed point and modal logics; for example, tabling by itself has been used to implement model checkers for the alternation-free μ -calculus [34, 24]. While the addition of coinduction permits consideration of more expressive forms of such logics, the stratification restriction must still be kept in mind: for example, co-logic programming cannot be used as a solver for the full μ -calculus, as it permits unrestricted nesting of least and greatest fixed point operators.

Part of our ongoing work is precisely describing restrictions of such logics that make them stratified. We have already discovered that the coinductive and tabled execution strategies, when considered as decision processes over proof trees (which is their use in constructing solvers for such fixed point logics), are equivalent to *stratified Büchi tree automata*, or SBTAs [6]. In SBTAs, cycles (via automaton transitions) between accepting and non-accepting states are forbidden. The class of SBTA-recognizable languages is distinguishable from the class of deterministic BTA-recognizable languages, though the two classes are not disjoint. In our equivalence proof, the non-accepting states of an SBTA correspond to calls to tabled predicates, while the coinductive calls correspond to the accepting states of the SBTA. The primary practical benefit of this equivalence is that by proving that solutions to a particular problem can or cannot be computed

by an SBTA, one can determine whether the solutions can be computed directly by the co-logic programming execution strategy. If they can, then a declarative co-logic program can be constructed from the SBTA: the automaton states and transitions are translated to predicates and clauses (respectively) in the program, with the predicates for accepting and nonaccepting states respectively declared as coinductive and tabled.

6.2 Towards Non-Stratified Co-LP

One significant hurdle to the use of co-logic programming is that existing co-logic programming systems perform no analysis to check whether a program is stratified, making the programmer responsible for ensuring the stratification of his program. According to the operational semantics of tabling and coinduction, non-stratified programs exhibit inconsistent behavior and produce non-intuitive answers, and the resulting errors are often difficult to diagnose. To ease the burden on both the programmer and the implementor of a co-logic programming system, we want to provide a consistent semantics to non-stratified programs. Because we consider execution strategies as being tree automata decision processes, we seek an acceptance condition with sufficient power to represent non-stratified programs.

To this end, we are currently focused on Rabin tree automata (RTAs) [23], again drawing a correspondence between categories of predicates and categories of states in an RTA. RTA states can be split into three categories: accepting, non-accepting, and rejecting. A tree is accepted by an RTA if, along each branch of the tree, an accepting state is encountered infinitely often without any rejecting state also being encountered infinitely often. As before, accepting states correspond to calls to coinductive predicates. In order to accommodate the other two categories, we use two categories of inductive predicates: *strongly inductive*, whose calls cannot occur infinitely often along any path in a proof and thereby forbid coinductive success when it would violate this requirement; and *weakly inductive*, whose calls cannot occur infinitely often along any proof path by themselves but still permit coinductive success [7].

By way of a brief example, let us consider the smallest non-stratified program:

```
:- coinductive p/0.
:- inductive q/0.
p :- q.
q :- p.
```

Under the existing co-logic programming semantics (using tabling for induction), the query `?- p, q.` succeeds, while `?- q, p.` fails. To resolve this inconsistency, we require that inductive predicates be declared to be either `weak_inductive` or `strong_inductive`, as shown in the following:

```
:- coinductive p/0.
:- strong_inductive q/0.
p :- q.
q :- p.
```

Both queries to this program will fail, as both proof trees contain infinite numbers of occurrences of the call to `p` and to `q`: the latter is forbidden by the `strong_inductive` declaration on `q`. A `weak_inductive` declaration would allow both queries to succeed. Finally, consider the following program:

```
:- coinductive p/0.  
:- weak_inductive q/0.  
p :- q.  
q :- q.
```

Here, only calls to `q` occur infinitely often in both proofs, causing both queries to fail (as no coinductive call occurs infinitely often).

Armed with this additional execution strategy and its correspondence to RTAs, we will use co-logic programming to declaratively implement solvers for several pertinent domains, in particular, solvers and model checkers for temporal and modal logics. While it is worth noting that we are still in the process of deriving the declarative fixed-point semantics for non-stratified co-logic programming we already regard co-logic programming as a powerful, declarative technique for solving a wide range of logical and algebraic problems.

7 Conclusions

In this paper we gave an overview of the coinductive logic programming paradigm and illustrated many of its applications. Co-LP gives an operational semantics to declarative semantics that is based on the greatest fixpoint. We believe that a combination of inductive and coinductive LP allows one to implement any desired logic programming semantics (well-founded, stable model semantics, etc.). Many problems still remain open. These include arbitrary interleaving of inductive and coinductive computations and combining constraints with coinductive LP.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. J. Barwise, L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, 1996.
3. E. M. Clarke, Jr., O. Grumberg and D. A. Peled. *Model Checking*. The MIT Press, 1999.
4. Z. Dang. Binary reachability analysis of pushdown timed automata with dense clocks. In *CAV '01*, pp. 506–518. Springer, 2001.
5. P. Degano, J. Loddo, and C. Priami. Mobile processes with local clocks. In *LOMAPS*, pp. 296–319. Springer, 1996.
6. B. DeVries et al. Semantics and Implementation of Co-Logic Programming. Forthcoming.
7. B. DeVries et al. A Co-LP Execution Strategy Derived from Rabin Tree Automata (in preparation).
8. F. Fages. Consistency of Clark’s completion and existence of stable models. *Journal of Methods of Logic in Computer Science* 1: 51–60. 1994.

9. A. Gordon. A Tutorial on Co-induction and Functional Programming. In: *Glasgow Functional Programming Workshop*, pp. 78–95. Springer, 1994.
10. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP*, pp. 27–44. Springer, 2007.
11. C. L. Heitmeyer and N. A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pp. 120–131, 1994.
12. B. Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Draft manuscript.
13. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
14. F. Kluźniak. A logic programming meta-interpreter that combines tabling and coinduction. <http://www.utdallas.edu/~gupta/meta.tar.gz>.
15. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd. edition, 1987.
16. A. Mallya. Deductive Multi-valued Model Checking. Ph.D. thesis. University of Texas at Dallas. 2006.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts i and ii. *Inf. Comput.*, 100(1):1–77, September 1992.
18. R. Min, A. Bansal, and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI09*. 2009.
19. R. Min. *Predicate Answer Set Programming with Coinduction*. Ph.D. Thesis. University of Texas at Dallas. 2009.
20. R. Min, G. Gupta. Coinductive Logic Programming with Negation. *LOPSTR 2009*, pp. 97–112. Springer LNCS 6037.
21. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
22. C. Puchol. A solution to the generalized railroad crossing problem in Esterel. Technical report, Dep. of Comp. Science, The University of Texas at Austin, 1995.
23. M. O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1, July 1969.
24. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and D. Warren. Efficient Model Checking Using Tabled Resolution. In *Proc. CAV '97*, pp. 143–154. Springer, 1997.
25. N. Saeedloei. Extending Infinite Systems with Real-time. Ph.D. Thesis. University of Texas at Dallas. Forthcoming.
26. N. Saeedloei and G. Gupta. Timed pi-calculus. University of Texas at Dallas technical report.
27. N. Saeedloei and G. Gupta. Timed definite clause omega-grammars. In *ICLP (Technical Communications)*, pp. 212–221, 2010.
28. N. Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *LATA*, pp. 536–548, 2010.
29. L. Simon. *Coinductive Logic Programming*. Ph,D thesis, University of Texas at Dallas, 2006.
30. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming. *ICLP'06*, pp. 330–344. Springer LNCS 4079.
31. L. Sterling and E. Shapiro. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*. The MIT Press, 1994.
32. J. Wielemaker. SWI-Prolog. <http://www.swi-prolog.org>.
33. P. Wegner, D. Goldin. Mathematical models of interactive computing. Brown University Technical Report CS 99-13. 1999.
34. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *VMCAI 2003*, pp. 116–131.