Coinductive Logic Programming and its Application to Planning and Model-Checking

Gopal Gupta

Applied Logic, Programming-Languages and Systems (ALPS) Lab
The University of Texas at Dallas
Outline

• Circular phenomena in computer science
• Coinduction: modeling circular phenomenon
• Coinduction in logic programming (Co-LP)
• Applications of Co-LP:
  – Model checking
    • Timed model checking
  – Planning
    • Timed planning
Circular Phenomena in Comp. Sci.

• Circularity has dogged Mathematics and Computer Science ever since Set Theory was first developed:
  – The well known Russell’s Paradox:
    • $R = \{ x \mid x$ is a set that does not contain itself$\}$
      Is $R$ contained in $R$? Yes and No
  – Liar Paradox: I am a liar
  – Hypergame paradox (Zwicker)
• All these paradoxes involve self-reference (through some type of negation)
• Russell put the blame squarely on circularity and sought to ban it from scientific discourse:
  ```
  `Whatever involves all of the collection must not be one of the collection”
  -- Russell 1908
  ```
Circular Phenomenon in Comp. Sci.

• All this changed with Kripke’s paper in 1975 who argued that circular phenomenon are far more common and circularity can’t simply be banned.

• Circularity has been banned from automated theorem proving and logic programming through the occurs check rule:
  
  An unbound variable cannot be unified with a term containing that variable: \( X = f(X) \) disallowed

• What if we allowed such unifications to proceed (as LP systems always did for efficiency reasons)?
Circularity in Computer Science

- If occurs check is removed, we will generate circular (infinite) structures:
  - $X = [1,2,3 \mid X]$
- Such structures, of course, arise in computing (circular linked lists), but banned in logic/LP.
- Subsequent LP systems did allow for such circular structures (rational terms), but they only exist as data-structures, there is no proof theory to go along with it.
  - One can hold the data-structure in memory within an LP execution, but one can’t reason about it.
Coinduction

- Circular structures are infinite structures
  \[ X = [1, 2 | X] \] is logically speaking \[ X = [1, 2, 1, 2, \ldots] \]
- Proofs about their properties are infinite-sized
- *Coinduction* is the technique for proving these properties [Aczel 1983; Moss & Barwise 1996]
  “Vicious Circles” by Moss and Barwise (1996)
- Our focus: inclusion of coinductive reasoning techniques in LP and its application to model checking and planning.
Induction vs Coinduction

• Induction is a mathematical technique for finitely reasoning about an infinite (countable) no. of things.
  – Naturals: 0, 1, 2, …
  – Lists: [], [X], [X, X], [X, X, X], …

• 3 components of an inductive definition:
  (1) Initiality, (2) iteration, (3) minimality
  – for example, the set of lists is specified as follows:
    [ ] – an empty list is a list \textbf{(initiality)} …..(i)
    [H | T] is a list if T is a list and H is an element \textbf{(iteration)} ..(ii)
    minimal set that satisfies (i) and (ii) \textbf{(minimality)}
Induction vs Coinduction

- Coinduction is a mathematical technique for (finitely) reasoning about infinite things.
  - Mathematical dual of induction
- 2 components of a coinductive definition:
  1. iteration, 2. maximality
  - for example, for a list:
    \[ [H | T] \] is a list if T is a list and H is an element (iteration).
    **Maximal** set that satisfies the specification of a list.
  - This coinductive defn. specifies all infinite sized lists
Example: Natural Numbers

- $\Gamma_N(S) = \{ 0 \} \cup \{ \text{succ}(x) \mid x \in S \}$
- $\Gamma_N$ defines 2 sets.
  - $N = \mu \Gamma_N$ (least fixed-point)
  - $N' = \nu \Gamma_N = N \cup \{ \omega \}$ (greatest fixed-point)

<table>
<thead>
<tr>
<th>Definition</th>
<th>Proof</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least fixed point</td>
<td>Induction</td>
<td>Recursion</td>
</tr>
<tr>
<td>Greatest fixed point</td>
<td>Coinduction</td>
<td>Corecursion</td>
</tr>
</tbody>
</table>

- Co-recursion: recursive definition without a base case
Infinite Objects and Properties

• Traditional logic programming (based on LFP) is unable to reason about infinite objects and/or properties
• (The glass is only half-full)
• Example: perpetual binary streams
  – traditional logic programming cannot handle

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

• Goal: Combine traditional LP with coinductive LP
Extending LP with Co-induction

• What is needed?: an operational semantics for incorporating coinduction into SLD resolution

• Declarative Semantics: across the board dual of traditional LP [Lloyd 87]:
  – greatest fixed-points
  – terms: co-Herbrand universe \( U^\omega(P) \)
  – atoms: co-Herbrand base \( B^\omega(P) \)
  – program semantics: maximal co-Herbrand model \( M^\omega(P) \).
Operational Semantics: co-SLD

- Nondeterministic state transition system
- States are pairs of
  - A finite list of goals [resolvent] (as in Prolog)
  - A set of syntactic term equations of the form $x = f(x)$ or $x = t$

- Given program $p : - p$. Then the query $?- p$. will succeed.
- Given $p( [1 \mid T]) : - p(T)$. $?- p(X)$ to succeed with $X = [1 \mid X]$.

- Transition rules
  - Definite clause rule
  - “Coinductive hypothesis rule”
    - If a coinductive goal $Q$ is called, and $Q$ unifies with an ancestor call made earlier then $Q$ succeeds.
Example: Number Stream

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

?- stream( [ 0, s( 0 ), s( s( 0 ) ) | T ] ).
1. MEMO: stream( [ 0, s( 0 ), s( s( 0 ) ) | T ] )
2. MEMO: stream( [ s( 0 ), s( s( 0 ) ) | T ] )
3. MEMO: stream( [ s( s( 0 ) ) | T ] )
4. stream(T)

Answers:
T = [ 0, s(0), s(s(0)) | T ]
T = [ 0, s(0), s(s(0)), s(0), s(s(0)) | T ]
T = [ 0, s(0), s(s(0)) | T ] ... 
T = [ 0, s(0), s(s(0)) | X ] (where X is any rational list of numbers.)
Other Examples

• Append:
  \[\text{:- coinductive append/3.}\]
  \[\text{append( \[ \], X, X ).}\]
  \[\text{append( \[ H | T \], Y, \[ H | Z \] ) :- append( T, Y, Z ).}\]
  \[\text{?- X = \[ 1, 2, 3 \ | X \], Y = \[ 3, 4 \ | Y \], append( X, Y, Z).}\]
  \[\text{Answer: Z = \[ 1, 2, 3 \ | Z \].}\]
  \[\text{?- Z = \[ 1, 2 \ | Z \], append( X, Y, Z ).}\]
  \[\text{Answer: X = \[ \], Y = \[ 1, 2 \ | Z \]; X = [1, 2 \ | X], Y = _}\]
  \[\text{X = [ 1 ], Y = [ 2 | Z ];}\]
  \[\text{X = [ 1, 2 ], Y = Z; .... ad infinitum}\]

• Co-member(X, L): is X a member of infinite list L?

• Sieve of Eratosthenes (lazy evaluation)
Co-Logic Programming

• combines both halves of logic programming:
  – traditional logic programming
  – coinductive logic programming

• syntactically identical to traditional logic programming, except predicates are labeled:
  – Inductive, or
  – coinductive

• and stratification restriction enforced where:
  – inductive and coinductive predicates cannot be mutually recursive. e.g.,
    p :- q.
    q :- p.
  Program rejected, if p coinductive & q inductive

• Preliminary implementation on top of YAP available.
Applications of Co-LP

• With Co-LP one can perform LFP as well as GFP computations elegantly
• Declarative power of LP can be harnessed for more sophisticated applications
• Two major application domains:
  – Model checking: Need to compute LFPs & GFPs
  – Planning: rules describing domain may be circular
• Using LP brings other LP-specific techniques to bear on the problem:
  – Constraints (continuous time easily included)
  – Parallelism (verification/planning done in parallel)
Application: Model Checking
Finite Automata

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).

?- automata(X,s0).
    X=[ a, b];
    X=[ a, b, e, a, b];
    X=[ a, b, e, a, b, e, a, b];
    ……
    ……
    ……

Figure A
Infinite Automata

\[
\text{automata}([X|T], \ St) :\ - \ trans(St, \ X, \ NewSt), \ \text{automata}(T, \ NewSt).
\]

\[
\begin{align*}
\text{trans}(s0,a,s1). & \quad \text{trans}(s1,b,s2). & \quad \text{trans}(s2,c,s3). \\
\text{trans}(s3,d,s0). & \quad \text{trans}(s2,3,s0). & \quad \text{final}(s2).
\end{align*}
\]

?- \text{automata}(X,s0).
\begin{align*}
X &= [a, b, c, d | X]; \\
X &= [a, b, e | X];
\end{align*}

When the same goal is seen
=> infinite cycle in the automata
=> HALT
Verifying Liveness Properties

- Verifying safety properties in LP is relatively easy: safety modeled by reachability.
- Accomplished via tabled logic programming (an efficient engine for computing LFP).
- Verifying liveness is much harder: a counterexample to liveness is an infinite trace.
- Verifying liveness is transformed into a safety check via use of negations in model checking and tabled LP.
  - Considerable overhead incurred.
- Co-LP solves the problem more elegantly:
  - Infinite traces that serve as counter-examples produced as answers.
Verifying Liveness Properties

• Consider Safety:
  – Is an unsafe state $S_u$ (wrongly) considered safe?
    i.e., is $S_u$ reachable?
  – If answer is yes, the path to $S_u$ is the counter-ex
    • Tabled LP will produce this path as an answer

• Consider Liveness, then dually
  – Is a dead state $D$ (wrongly) considered live?
  – If answer is yes, the infinite path containing $D$ is the counter example
    • Co-LP will produce this infinite path as an answer

• Liveness checking as easy as safety checking
Nested Finite and Infinite Automata

:- coinductive state/2.
state(s0, [s0,s1 | 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. enter. repeat. exit. error.
work :- work.
|?- state(s0,X), absent(s2,X).
   X=[ s0, s3 | X ]
Verification of Real-Time Systems
“Train, Controller, Gate”

Timed Automata

- ω-automata w/ time constrained transitions & stopwatches
- straightforward encoding into CLP(\(R\)) + Co-LP
  (clock to be reset in every cycle)
Verification of Real-Time Systems
“Train, Controller, Gate”

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

train(X, up, X, T1,T2,T2).
% up=idle
train(s0,approach,s1,T1,T2,T3):- {T3=T1}.
train(s1,in,s2,T1,T2,T3):-{T1-T2>2,T3=T2}.
train(s2,out,s3,T1,T2,T3). 
train(s3,exit,s0,T1,T2,T3):-{T3=T2,T1-T2<5}.

train(X,lower,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).

(i) train
Verification of Real-Time Systems
“Train, Controller, Gate”

\begin{verbatim}
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, up, X, T1, T2, T2).
contr(X, out, X, T1, T2, T2).
contr(X, down, X, T1, T2, T2).
\end{verbatim}
Verification of Real-Time Systems
“Train, Controller, Gate”

gate(s0,lower,s1,T1,T2,T3):- \{T3=T1\}.
gate(s1,down,s2,T1,T2,T3):- \{T3=T2,T1-T2<1\}.
gate(s2,raise,s3,T1,T2,T3):- \{T3=T1\}.
gate(s3,up,s0,T1,T2,T3):- \{T3=T2,T1-T2>1,T1-T2<2 \}.
gate(X,approach,X,T1,T2,T2).
gate(X,in,X,T1,T2,T2).
gate(X,out,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).
Verification of Real-Time Systems

:- coinductive driver/9.

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).

?- driver(s0,s0,s0,T,Ta,Tb,Tc,X,R).
   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] ;
\%
\% where A, B, C, ... H, I are the corresponding wall clock time of events generated.
DPP – Safety: Deadlock Free

- A solution
  - Force one philosopher to pick forks in different order than others

- Checking for deadlock
  - Bad state is not reachable
  - Implemented using Tabled LP

```prolog
:- table reach/2.
reach(Si, Sf) :- trans(_,Si,Sf).
reach(Si, Sf) :- trans(_,Si,Sfi), reach(Sfi,Sf).
?- reach([s,s,s,s,s], [w,w,w,w,w]).
no
```
DPP – Liveness: Starvation Free

- Phil. waits forever on a fork
- One potential solution
  - phil. waiting longest gets the access
  - implemented using stop watches
- Checking for starvation
  - once in bad state, is it possible to remain there forever?
  - implemented using co-LP

starved(X) :-
X = 1, driver([s, s, s, s, s], [w, _, _, _, _]);
X = 2, driver([s, s, s, s, s], [_, w, _, _, _]);
X = 3, driver([s, s, s, s, s], [_, _, w, _, _]);
X = 4, driver([s, s, s, s, s], [_, _, _, w, _]);
X = 5, driver([s, s, s, s, s], [_, _, _, _, w]).

?- starved(X).
no
Observations

• Finite and infinite automata nested within each other:
  – Not possible in standard methods for model checking?

• Continuous quantities such as time can be introduced
  – Hybrid systems can be modeled elegantly, as long as a solver/consistency-checker can be built for that domain

• Liveness checking as easy as safety checking

• Parallelism can be implicitly exploited from logic programs; (timed) model checking can be readily performed in parallel
Application: (Timed) Planning
Application: (Timed) Planning

- **Planning**: Given (i) domain \( D \), (ii) observations about initial state \( O \), (iii) a set of fluents \( g_1, \ldots, g_n \), find a set of actions \( a_1, \ldots, a_m \), such that \( D \) will entail \( g_1, \ldots, g_n \).
  - *Action description languages* (like \( A \)) describe domains (with actions and change) used for planning problems

- Planning may involve self referential rules:
  
  \[
  \text{hasball} \quad \text{if} \quad \text{receivedpass} \quad \text{&} \quad \neg \text{hasball}
  \]

- Planning & verification: two sides of the same coin
Real-Time Soccer Playing Domain

ShotTaken causes \neg HasBall, \neg ClearShot, Goal when Clock \leq 0.5
  if HasBall, ClearShot, \neg Goal

PassBall causes ClearShot resets Clock when Clock \leq 1
  if \neg ClearShot

wait causes \neg HasBall, \neg ClearShot when Clock > 0.5
  if HasBall, ClearShot

wait causes \neg HasBall, when Clock > 1 if HasBall
Real-Time Soccer Playing Domain

*PassBall* causes *ClearShot* resets *Clock* when *Clock* $\leq$ 1

if $\neg$ *ClearShot*

- holds(clearShot, res(passBall, S)) :-
  not_holds(clearShot, S), b_getval(clock, Clock), {Clock $\leq$ 1},
  {NewClock $>$ 0}, b_setval(clock, NewClock).

*wait* causes $\neg$ *HasBall*, when *Clock* $>$ 1 if *HasBall*

- not_holds(hasBall, res(wait, S)) :-
  holds(hasBall, S), b_getval(clock, Clock), {Clock $>$ 1},
  {NewClock $>$ Clock}, b_setval(clock, NewClock).

If clearShot is false in the initial state, then the query

?- holds(goal, S). % produces the solution

S = res(shotTaken, res(passBall, s0))
Other Applications & Status

• Other Applications:
  – Non monotonic reasoning: Co-LP allows goal-directed execution of Answer Set Programs
  – SAT Solvers: Goal-directed SAT solvers can be built

• Current Status:
  – A high level implementation of Co-LP on top of YAP Prolog completed.
  – Work in progress to build a low level implementation of Co-LP in an existing Prolog/CLP engine
Parallel Unified Reasoning Engine

• Lots of research in LP resulting in advances:
  – Constraints, Tabled LP, Parallelism, ASP, co-LP, coroutining
• Goal: build a system that combines them all
  build a system that run very large apps.

[Diagram showing connections between Or-Parallelism, Tabled LP, Unified LP System, Coinduction, Constraints, Coroutining, Rule selection, and Goal selection]
Conclusion

• Introducing coinduction into logic prog. allows one to compute both LFP & GFP

• GFP/LFP computations can be combined with other advanced features of LP allowing highly complex problems to be solved elegantly
  – Applications to (timed) model checking
  – Applications to (timed) planning

• Large instances of these applications can be run in parallel on a parallel logic programming system running on multicores
Related Publications


3. Coinductive Logic Programming and its Applications, ICLP’07 tutorial
