Coinductive Logic Programming and its Applications

Gopal Gupta

Luke Simon, Ajay Bansal, Ajay Mallya, Richard Min.

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

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 | 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 & Smullyan)
- 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

Circularity in Computer Science

- Following Russell's lead, Tarski proposed to ban self-referential sentences in a language
- Rather, have a hierarchy of languages
- All this changed with Kripke's paper in 1975 who showed 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

- What if we allowed such unification to proceed (as LP systems always did for efficiency reasons)?

Circularity in Computer Science

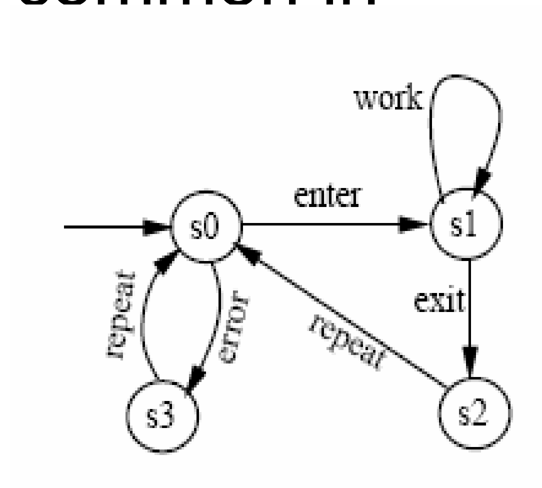
- If occurs check is removed, we'll generate circular (infinite) structures:
  - X = [1,2,3 | X]
- Such structures, of course, arise in computing (circular linked lists), but banned in logic/LP.
- Subsequent LP systems did allow for such circular structures (rational terms), but they only exist as data-structures, 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.

Circularity in Everyday Life

- Circularity arises in every day life
  - Most natural phenomenon are cyclical
    - Cyclical movement of the earth, moon, etc.
    - Our digestive system works in cycles
  - Social interactions are cyclical:
    - Conversation = (1st speaker, (2nd Speaker, Conversation)
    - Shared conventions are cyclical concepts
- Numerous other examples can be found elsewhere (Barwise & Moss 1996)

Circularity in Computer Science

- Circular phenomenon are quite common in Computer Science:
  - Circular linked lists
  - Graphs (with cycles)
  - Controllers (run forever)
  - Bisimilarity
  - Interactive systems
  - Automata over infinite strings/Kripke structures
  - Perpetual processes
- Logic/LP not equipped to model circularity



Coinduction

- Circular structures are infinite structures

  X = [1, 2 | X]   is logically speaking X = [1, 2, 1, 2, ….]
- Proofs about their properties are infinite-sized
- *Coinduction* is the technique for proving these properties
  - first proposed by Peter Aczel in the 80s
- Systematic presentation of coinduction & its application to computing, math. and set theory:

  "Vicious Circles" by Moss and Barwise (1996)
- Our focus: inclusion of coinductive reasoning techniques into LP and theorem proving

Induction vs Coinduction

- Induction is a mathematical technique for finitely reasoning about an infinite (countable) no. of things.
- Examples of inductive structures:
  - 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 (**initiality**)

    [H | T] is a list if T is a list and H is an element (**iteration**)

    nothing else is a list (**minimality**)

Induction vs Coinduction

- Coinduction is a mathematical technique for (finitely) reasoning about infinite things.
  - Mathematical dual of induction
  - If all things were finite, then coinduction would not be needed.
  - Perpetual programs, automata over infinite strings
- 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 interpretation specifies all infinite sized lists

Example: Natural Numbers

- $\Gamma_{\mathcal{N}}(S) = \{ 0 \} \cup \{ succ(x) \mid x \in S \}$
- $N = \mu\Gamma_N$
  - where $\mu\Gamma$ is least fixed-point.
- aka “inductive definition”
  - Let N be the smallest set such that
    - $0 \in N$
    - $x \in N$ implies $x + 1 \in N$
- Induction corresponds to Least Fix Point (LFP) interpretation.

Example: Natural Numbers and Infinity

- $\Gamma_N(S) = \{ 0 \} \cup \{ succ(x) \mid x \in S \}$
- Γ_N unambiguously defines another set
- $N' = \nu\Gamma_N = N \cup \{ \omega \}$
  - $\omega = succ(succ(succ(...))) = succ(\omega) = \omega + 1$
  - where $\nu\Gamma_N$ is a greatest fixed-point
- Coinduction corresponds to Greatest Fixed Point (GFP) interpretation.

Mathematical Foundations

- Duality provides a source of new mathematical tools that reflect the sophistication of tried and true techniques.

Definition	Proof	Mapping
Least fixed point	Induction	Recursion
Greatest fixed point	Coinduction	Corecursion

- Co-recursion: recursive def’n without a base case

Applications of Coinduction

- model checking
- bisimilarity proofs
- lazy evaluation in FP
- reasoning with infinite structures
- perpetual processes
- cyclic structures
- operational semantics of “coinductive logic programming”
- Type inference systems for lazy functional languages

Inductive Logic Programming

- Logic Programming
  - is actually inductive logic programming.
  - has inductive definition.
  - useful for writing programs for reasoning about finite things:
    - data structures
    - properties

Infinite Objects and Properties

- Traditional logic programming 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

Overview of Coinductive LP

- Coinductive Logic Program is

  a definite program with maximal co-Herbrand model declarative semantics.

- Declarative Semantics: across the board dual of traditional LP:
  - greatest fixed-points
  - terms: co-Herbrand universe $U^{co}(P)$
  - atoms: co-Herbrand base $B^{co}(P)$
  - program semantics: maximal co-Herbrand model $M^{co}(P)$.

Coinductive LP: An Example

- Let P_1 be the following coinductive program.

```
:- coinductive from/2.                from(x) = x cons from(x+1)
from( N, [ N | T ] ) :- from( s(N), T ).
|?- from( 0, X ).
```

- co-Herbrand Universe: $U^{co}(P_1) = N \cup \Omega \cup L$ where
  N=[0, s(0), s(s(0)), ...], Ω={ s(s(s(. . .))) }, and L is the the set of all finite and infinite lists of elements in N, Ω and L.
- co-Herbrand Model:
  $M^{co}(P_1)$={ from(t, [t, s(t), s(s(t)), ...]) | t $\in$ $U^{co}(P_1)$ }
- from(0, [0, s(0), s(s(0)), ...]) $\in$ $M^{co}(P_1)$ implies the query holds
- Without "coinductive" declaration of from, $M^{co}(P_1') = \varnothing$
  This corresponds to traditional semantics of LP with infinite trees.

Operational Semantics: co-SLD

- nondeterministic state transition system
- states are pairs of
  - a finite list of syntactic atoms [resolvent] (as in Prolog)
  - a set of syntactic term equations of the form x = f(x) or x = t
    - For a program p :- p. => the query |?- p. will succeed.
    - p([1 | T]) :- p(T). => |?- p(X) to succeed with X= [1 | X].
- transition rules
  - definite clause rule
  - "coinductive hypothesis rule"
    - if a coinductive goal Q is called,
      and Q unifies with a call made earlier (e.g., P :- Q)
      then Q succeeds.

Correctness

- Theorem (soundness). If atom A has a successful co-SLD derivation in program P, then E(A) is true in program P, where E is the resulting variable bindings for the derivation.

- Theorem (completeness). If $A \in M^{co}(P)$ has a rational proof, then A has a successful co-SLD derivation in program P.
  - Completeness only for rational/regular proofs

Implementation

- Search strategy: hypothesis-first, leftmost, depth-first
- Meta-Interpreter implementation.

```
query(Goal) :- solve([],Goal).
solve(Hypothesis, (Goal1,Goal2)) :-
      solve( Hypothesis, Goal1), solve(Hypothesis,Goal2).
solve( _ , Atom) :- builtin(Atom), Atom.
solve(Hypothesis,Atom):- member(Atom, Hypothesis).
solve(Hypothesis,Atom):- notbuiltin(Atom),
                         clause(Atom,Atoms),
                         solve([Atom|Hypothesis],Atoms).
```

- A more efficient implem. atop YAP also available

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])

Answers:

T = [0, s(0), s(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.)

Example: Append

```
:- coinductive append/3.
append( [ ], X, X ).
append( [ H | T ], Y, [ H | Z ] ) :- append( T, Y, Z ).

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

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

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

Example: Comember

```
member(H, [ H | T ]).
member(H, [ X | T ]) :- member(H, T).
    ?- L = [1,2 | L], member(3, L) succeeds.     Instead:

:- coinductive comember/2.    %drop/3 is inductive
comember(X, L) :- drop(X, L, R), comember(X, R).
drop(H, [ H | T ], T).
drop(H, [ X | T ], T1) :- drop(H, T, T1).
```

```
?- 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;
```

```
?- X = [1,2 | X], comember(3, X).
        Answer: no
```

Example: Sieve of Eratosthenes

- Lazy evaluation can be elegantly incorporated in LP

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

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
- Implementation on top of YAP available.

Application: Model Checking

- automated verification of hardware and software systems
- ω-automata
  - accept infinite strings
  - accepting state must be traversed infinitely often
- requires computation of lfp and gfp
- co-logic programming provides an elegant framework for model checking
- traditional LP works for safety property (that is based on lfp) in an elegant manner, but not for liveness .

Verification of Properties

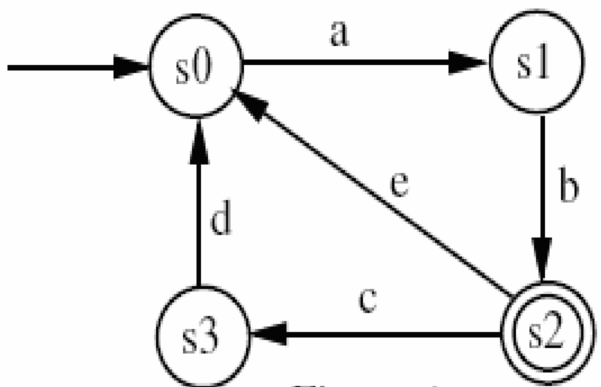


Figure A

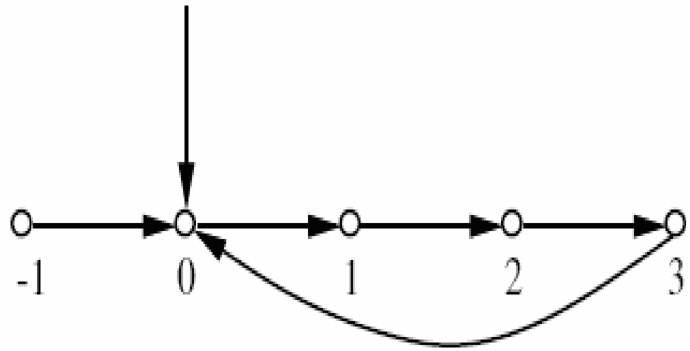


Figure B

- Types of properties: safety and liveness
- Search for counter-example

Safety versus Liveness

- Safety
  - "nothing bad will happen"
  - naturally described inductively
  - straightforward encoding in traditional LP
- liveness
  - "something good will eventually happen"
  - dual of safety
  - naturally described coinductively
  - straightforward encoding in coinductive LP

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, 3, s0).    final(s2).

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

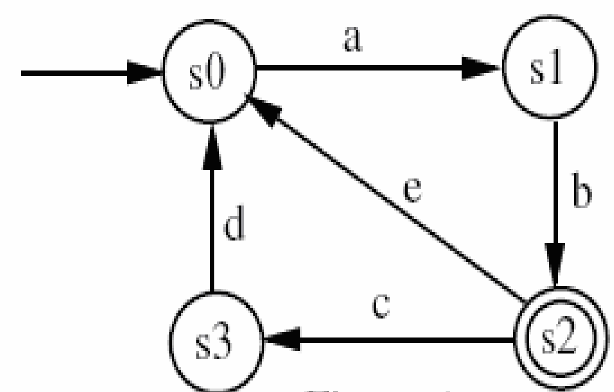


Figure A

Infinite Automata

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

```
trans(s0,a,s1).    trans(s1,b,s2).    trans(s2,c,s3).
trans(s3,d,s0).    trans(s2,3,s0).    final(s2).
```

```
?- automata(X,s0).
     X=[ a, b, c, d | X ];
     X=[ a, b, e | X ];
```

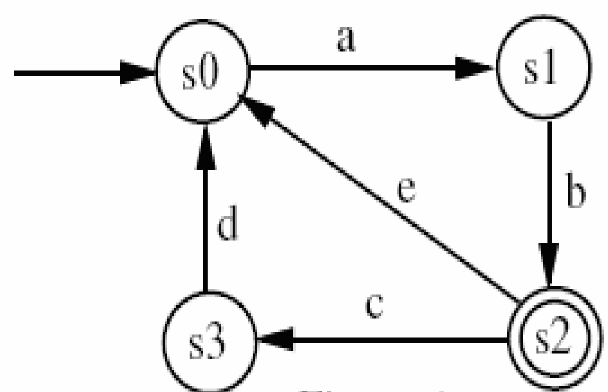


Figure A

Verifying Liveness Properties

- Verifying safety properties in LP is relatively easy: safety modeled by reachability
- Accomplished via tabled logic programming
- 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 are easily produced as answers

Verifying Liveness Properties

- Consider Safety:
  - Question: Is an unsafe state, S_u, reachable (safe)?
  - If answer is yes, the path to S_u is the counter-ex.
- Consider Liveness, then dually
  - Question: Is a state, D, that should be dead, live?
  - If answer is yes, the infinite path containing D is the counter example
    - Co-LP will produce this infinite path as the answer
- Checking for liveness is just as easy as checking for safety

Counter

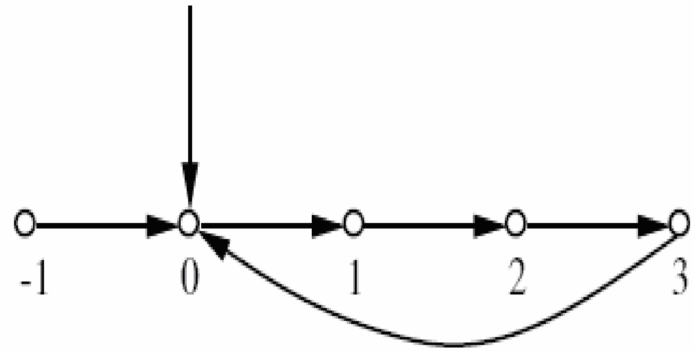
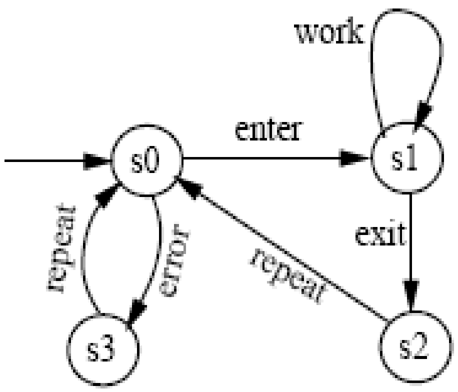


Figure B

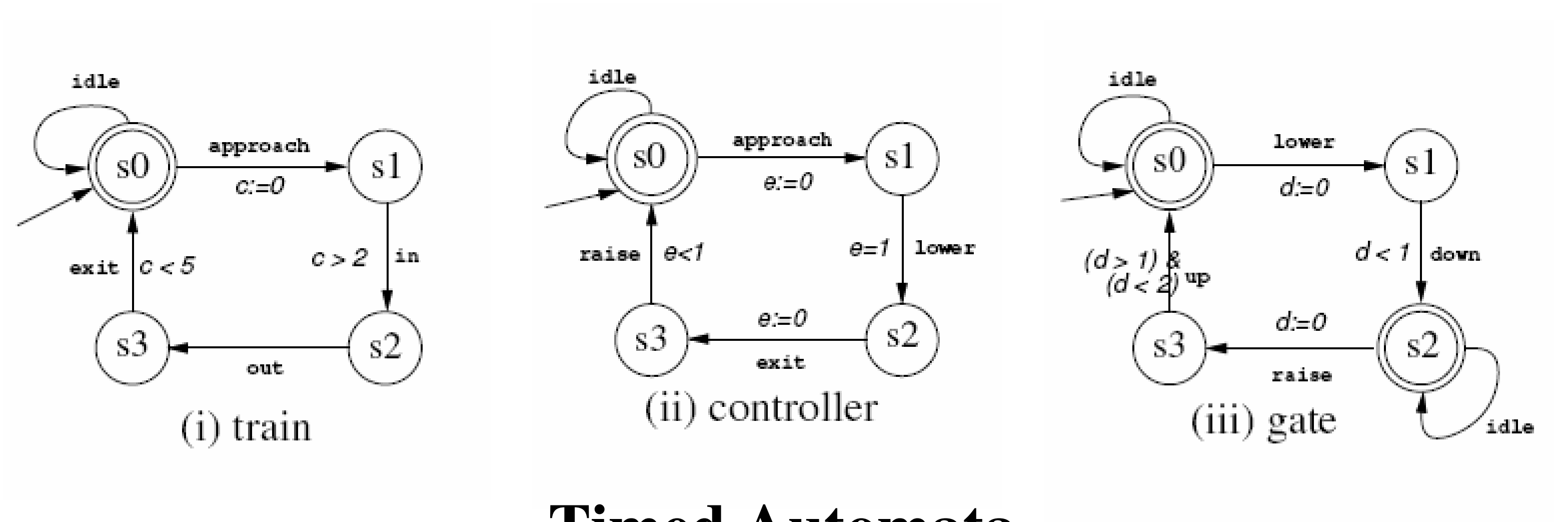
sm1(N,[sm1|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
s0(N,[s0|T]) :- N1 is N+1 mod 4, s1(N1,T), N1>=0.
s1(N,[s1|T]) :- N1 is N+1 mod 4, s2(N1,T), N1>=0.
s2(N,[s2|T]) :- N1 is N+1 mod 4, s3(N1,T), N1>=0.
s3(N,[s3|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
?- sm1(-1,X),comember(sm1,X).
No. (because sm1 does not occur in X infinitely often).

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 ]
```

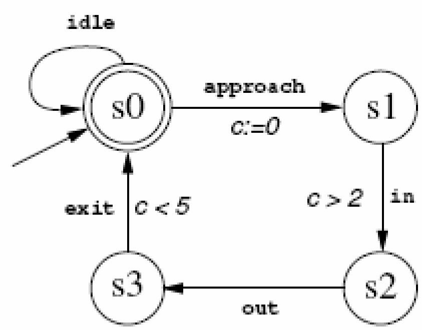
Verification of Real-Time Systems "Train, Controller, Gate"



Timed Automata

- ω-automata w/ time constrained transitions & stopwatches
- straightforward encoding into CLP(*R*) + Co-LP

Verification of Real-Time Systems "Train, Controller, Gate"

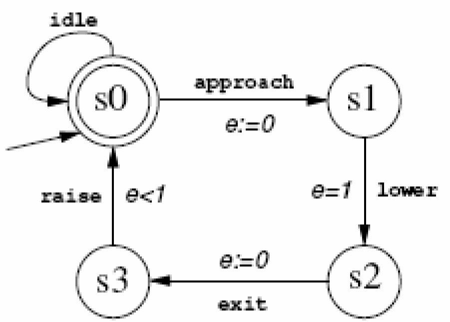


(i) train

```
:- 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).
```

Verification of Real-Time Systems "Train, Controller, Gate"



(ii) controller

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}.

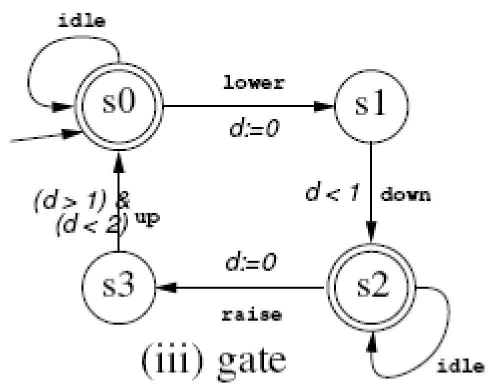
contr(X,in,X,T1,T2,T2).

contr(X,up,X,T1,T2,T2).

contr(X,out,X,T1,T2,T2).

contr(X,down,X,T1,T2,T2).

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.
```

Goal-directed execution of ASP

- Answer set programming (ASP) is a popular formalism for non monotonic reasoning
- Applications in real-world reasoning, planning, etc.
- Semantics given via lfp of a residual program obtained after “Gelfond-Lifschitz” transform
- Popular implementations: Smodels, DLV, etc.
  1. No goal-directed execution strategy available
  2. ASP limited to only finitely groundable programs
- Co-logic programming solves both these problems.
- Also provides a goal-directed method to check if a proposition is true in some model of a prop. formula

Why Goal-directed ASP?

- Most of the time, given a theory, we are interested in knowing if a *particular* goal is true or not.
- Top down goal-directed execution provides operational semantics (important for usability)
- Execution more efficient.
  - Tabled LP vs bottom up Deductive Deductive Databases
- Why check the consistency of the whole knowledgebase?
  - Inconsistency in some unrelated part will scuttle the whole system
- Most practical examples anyway add a constraint to force the answer set to contain a certain goal.
  - E.g. Zebra puzzle: :- not *satisfied*.
- Answer sets of non-finitely groundable programs computable & Constraints incorporated in Prolog style.

Negation in Co-LP

- Given a clause such as

  p :- q, not p.

  ?- p. fails coinductively when not p is encountered
- To incorporate negation in coinductive reasoning, need a negative coinductive hypothesis rule:
  - In the process of establishing not(p), if not(p) is seen again in the resolvent, then not(p) succeeds
- Also, not not p reduces to p.
- Answer set programming makes the “glass completely full” by taking into account failing computations:
  - p :- q, not p. is consistent if p = false and q = false
- However, this takes away monotonicity: q can be constrainted to false, causing q to be withdrawn, if it was established earlier.

ASP

- Consider the following program, A:

  p :- not q.   t.   r :- t, s.

  q :- not p.   s.

  A has 2 answer sets: {p, r, t, s} & {q, r, t, s}.
- Now suppose we add the following rule to A:

  h :- p, not h.   (falsify p)

  Only one answer set remains: {q, r, t, s}
- Gelfond-Lifschitz Method:
  - Given an answer set S, for each p $\in$ S, delete all rules whose body contains “not p”;
  - delete all goals of the form “not q” in remaining rules
  - Compute the least fix point, L, of the residual program
  - If S = L, then S is an answer set

Goal-directed ASP

- Consider the following program, A':

  p :- not q. t. r :- t, s.
  q :- not p, r. s. $\boxed{\text{h :- p, not h.}}$

- Separate into constraint and non-constraint rules: only 1 constraint rule in this case.
- Execute the query under co-LP, candidate answer sets will be generated.
- Keep the ones not rejected by the constraints.
- Suppose the query is ?- q. Execution: q → not p, r → not not q, r → q, r → r → t, s → s → success.

  Ans = {q, r, t, s}

- Next, we need to check that constraint rules will not reject the generated answer set.
  - (it doesn't in this case)

Goal-directed ASP

- In general, for the constraint rules of p as head, p_1:- B_1. p_2:- B_2. ... p_n :- B_n., generate rule(s) of the form:

  chk_p_1 :- not(p_1), B_1.
  chk_p_2 :- not(p_2), B_2.
  ...
  chk_p_n :- not(p), B_n.
- Generate: nmr_chk :- not(chk_p_1), ... , not(chk_p_n).
- For each pred. definition, generate its negative version:

  not_p :- not(B_1), not(B_2), ... , not(B_n).
- If you want to ask query Q, then ask ?- Q, nmr_chk.
- Execution keeps track of atoms in the answer set (PCHS) and atoms not in the answer set (NCHS).

Goal-directed ASP

- Consider the following program, P1:

  (i) p :- not q. (ii) q:- not r. (iii) r :- not p. (iv) q :- not p.

  P1 has 1 answer set: {q, r}.

- Separate into: 3 constraint rules (i, ii, iii)

  2 non-constraint rules (i, iv).

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

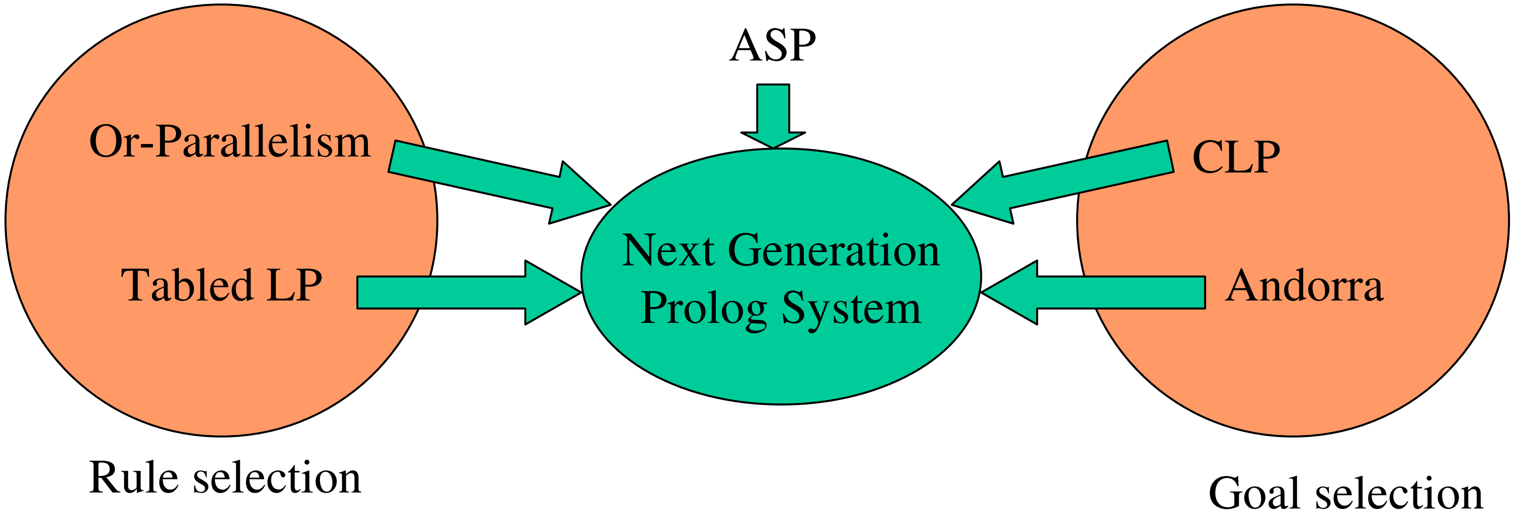
not_p :- q. not_q :- r, p. not_r :- p.

Suppose the query is ?- r.

Expand as in co-LP: r → not p → not not q → q (→ not r → fail, backtrack) → not p → success. Ans={r, q} which satisfies the constraint rules of nmr_chk.

Next Generation of LP System

- Lot of research in LP resulting in advances:
  - CLP, Tabled LP, Parallelism, Andorra, ASP, now co-LP
- However, no “one stop shop” system
- Dream: build this “one stop shop” system



Related Publications

1. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP'06* .
2. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-Logic programming: Extending logic programming with coinduction. In *ICALP'07*.
3. ICLP'07 Proceedings (this tutorial)
4. A. Bansal, R. Min, G. Gupta. Goal-directed Execution of ASP. Internal Report, UT Dallas
5. R. Min, A. Bansal, G. Gupta. Goal-directed Execution of ASP with General Predicates. Forthcoming.
6. A. Bansal, R. Min, G. Gupta. Resolution Theorem Proving with Coinduction. Internal Report, UT Dallas

Conclusion

- Circularity is a common concept in everyday life and computer science:
- Logic/LP is unable to cope with circularity
- Solution: introduce coinduction in Logic/LP
  - dual of traditional logic programming
  - operational semantics for coinduction
  - combining both halves of logic programming
- applications to verification, non monotonic reasoning, negation in LP, web services, theorem proving, propositional satisfiability.
- Acknowledgemt.: V. Santos Costa, R. Rocha, F. Silva (for help with implementation of co-LP)