Logic, Infinite Computation, Coinduction, Real-time, ....

Gopal Gupta
Neda Saeedloei, Brian DeVries, Kyle Marple, Feliks Kluzniak,
Luke Simon, Ajay Bansal, Ajay Mallya, Richard Min

Applied Logic, Programming-Languages and Systems (ALPS) Lab
The University of Texas at Dallas
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:
    - \( R = \{ x \mid x \text{ is a set that does not contain itself} \} \)
    - 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
• Kripke’s challenged this in a 1975 paper: 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 (i.e., $X = f(X)$ not allowed)

• 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:
  \[ X = [1,2,3 \mid X] \quad \quad X = f(X) \]
- 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 = (1\textsuperscript{st} speaker, (2\textsuperscript{nd} 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 directly
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
  - 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 in C/LP (and theorem proving), and its applications to verification and reasoning
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) ……(i)
    [H | T] is a list if T is a list and H is an element (iteration) ..(ii)
    minimal set that satisfies (i) and (ii) (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
  2. for example, for a list:
     - \([ H \mid T \)] 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

- \( \Gamma_N(S) = \{ 0 \} \cup \{ \text{succ}(x) | x \in S \} \)
- Inductive interpretation
  - \( N = \mu \Gamma_N \)
  - corresponds to least fix point interpretation
- Coinductive interpretation
  - \( N' = \nu \Gamma_N = N \cup \{ \omega \} \)
  - \( \omega = \text{succ}(\text{succ}(\text{succ}(\ldots))) = \text{succ}(\omega) = \omega + 1 \)
  - corresponds to greatest fixed point interpretation.
Mathematical Foundations

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

<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 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 C/LP

• (Constraint) Logic Programming
  – is actually inductive C/LP.
  – 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 $U^\omega(P)$
  - atoms: co-Herbrand base $B^\omega(P)$
  - program semantics: maximal co-Herbrand model $M^\omega(P)$. 
Operational Semantics: co-SLD Resolution

- 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 \( p \): \( p \) will succeed.
  - \( p( [ 1 | T ] ) :- p( T ) \). \( \Rightarrow \) \( ?- p(X) \) to succeed with \( X = [ 1 | X ] \).

- transition rules
  - definite clause rule
  - “coinductive hypothesis rule”
    - if a coinductive goal \( G \) is called, and \( G \) unifies with a call made earlier then \( G \) 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.
  
  \[
  \text{query(Goal)} :\text{- solve([],Goal).}
  \]
  
  \[
  \text{solve(Hypothesis, (Goal1,Goal2)) :}
  \]
  
  \[
  \text{solve(Hypothesis, Goal1), solve(Hypothesis,Goal2).}
  \]
  
  \[
  \text{solve(_, Atom) : builtin(Atom), Atom.}
  \]
  
  \[
  \text{solve(Hypothesis,Atom):- member(Atom, Hypothesis).}
  \]
  
  \[
  \text{solve(Hypothesis,Atom):- notbuiltin(Atom),}
  \]
  
  \[
  \text{clause(Atom,Atoms), solve([Atom|Hypothesis],Atoms).}
  \]

- A complete meta-interpreter available
- Implementation on top of YAP, SWI Prolog available
- Implementation within Logtalk + library of examples
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.)
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 ], 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 | X], comember(Y, X).
Answer: Y = 1;
Y = 2;
Y = 3;

?- X = [1, 2 | X], comember(3, X).
Answer: no
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
Application of Co-LP

• Co-LP allows one to compute both LFP & GFP
• Computable functions can be specified more elegantly
  – Interpreters for Modal Logics can be elegantly specified:
  – Model Checking: LTL interpreter elegantly specified
  – Timed $\omega$-automata: elegantly modeled and properties verified
  – Modeling/Verification of Cyber Physical Systems/Hybrid automata
  – Goal-directed execution of Answer Set Programs
  – Goal-directed SAT solvers (Davis-Putnam like procedure)
  – Planning under real-time constraints
  – Operational semantics of the $\pi$-calculus (incl. timed $\pi$-calculus)
    • infinite replication operator modeled with co-induction

Co-LP allows systems to be modeled naturally & elegantly
Application: Model Checking

• automated verification of hardware and software systems
• $\omega$-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.
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];
   ....
   ....
   ....
   ....
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 produced as answers
Verifying Liveness Properties

- Consider Safety:
  - Question: Is an unsafe state, \( S_u \), reachable?
  - 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 in a manner similar to safety
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 ]
An Interpreter for LTL

%--- nots have been pushed to propositions
:- tabled verify/2.

verify(S, [S], A) :- proposition(A), holds(S,A).   % p
verify(S, [S], not(A)) :- proposition(A), \+holds(S,A). % not(p)
verify(S,P, or(A,B)) :- verify(S, P, A) ; verify(S, P, B). % A or B
verify(S,P, and(A,B)) :- verify(S, P1, A), verify(S, P2, B). % A and B

(prefix(P2, P1), P=P1 ; prefix(P2,P1), P=P2)
verify(S, [S|P], x(A)) :- trans(S, S1), verify(S1, P, A). % X(A)
verify(S, P, f(A)) :- verify(S, P, A); verify(S, P, x(f(A))). % F(A)
verify(S, P, g(A)) :- coverify(S, P, g(A)). % G(A)
verify(S, P,u(A,B)) :- verify(S, P,B);
verify(S, P,and(A, x(u(A,B))))). % A u B
verify(S, r(A,B)) :- coverify(S, r(A,B)). % A r B
:- coinductive coverify/2.

coverify(S, g(A)) :- verify(S, P, and(A, x(g(A)))).
coverify(S, r(A,B)) :- verify(S, P, and(A,B)).
coverify(S, r(A,B)) :- verify(S, P, and(B, x(r(A,B))))).
Verification of Real-Time Systems
“Train, Controller, Gate”

Timed Automata

- $\omega$-automata w/ time constrained transitions & stopwatches
- straightforward encoding into CLP($R$) + Co-LP
- Assumption: no concurrent events
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).
Verification of Real-Time Systems
“Train, Controller, Gate”

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\leq1,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.

TECHNIQUE USED TO VERIFY THE GENERALIZED RAILROAD CROSSING PROBLEM
One potential solution
  - Force one philosopher to pick forks in different order than others

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

:- table reach/2.
  reach(Si, Sf) :- trans(_,Si,Sf).
  reach(Si, Sf) :- trans(_,Si,Sfi),
                reach(Sfi,Sf).

?- reach([1,1,1,1,1], [2,2,2,2,2]).
  no
DPP – Liveness: Starvation Free

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

?- starved(X).
  no
Other Applications

• Advanced $\omega$-structures can also be modeled and reasoned about: $\omega$-PTA, $\omega$-grammars
• Operational semantics of pi-calculus can be given
  – infinite replication operator modeled with co-induction;
  – can be extended with real-time through CLP(R)
• Non monotonic reasoning:
  – CoLP allows goal-directed execution of Answer Set Programs (ASP): IMPLEMENTATION AVAILABLE
  – Abductive reasoners can be elegantly implemented
  – Answer sets programming can be extended to predicates
  – ASP can be elegantly extended with constraints:
    – planning under real-time constraints become possible
Cyber-Physical Systems (CPS)

- CPS:
  - Networked/distributed Hybrid Systems
  - Discrete digital systems with
    - Inputs: continuous physical quantities
      - e.g., time, distance, acceleration, temperature, etc.
    - Outputs: control physical (analog) devices
- Elegantly modeled via co-LP extended with constraints
- Characteristics of CPS:
  - perform discrete computations (modeled via LP)
  - deal with continuous physical quantities (modeled via constraints)
  - are concurrent (modeled via LP coroutining)
  - run forever (modeled via coinduction)
CPS Example

Reactor Temperature Control System

\[
\dot{\theta} = \frac{\theta}{10} - 50
\]

\[
\dot{\theta} = \frac{\theta}{10} - 50
\]

\[
\dot{\theta} = \frac{\theta}{10} - 60
\]

\[
\dot{\theta} = \frac{\theta}{10} - 60
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]

\[
\theta = \theta_m
\]
Rod1 & Rod2

\[
\text{trans}_r1(\text{out}_1, \text{add}_1, \text{in}_1, T, Ti, To, W) :- \\
\{T - Ti >= W, To = Ti\}.
\]

\[
\text{trans}_r1(\text{in}_1, \text{remove}_1, \text{out}_1, T, Ti, To, W) :- \{To = T\}.
\]

\[
\text{trans}_r2(\text{out}_2, \text{add}_2, \text{in}_2, T, Ti, To, W) :- \\
\{T - Ti >= W, To = Ti\}.
\]

\[
\text{trans}_r2(\text{in}_2, \text{remove}_2, \text{out}_2, T, Ti, To, W) :- \{To = T\}.
\]
Controller

trans_c(norod, add1, rod1, Tetai, Tetao, T, Ti1, Ti2, To1, To2, F) :-
  (F == 1 -> Ti = Ti1; Ti = Ti2),
  {Tetai < 550, Tetao = 550, exp(e, (T - Ti)/10) = 5,
   To1 = T, To2 = Ti2}.

trans_c(rod1, remove1, norod Tetai, Tetao, T, Ti1, Ti2, To1, To2, F) :-
  {Tetai > 510 Tetao = 510, exp(e, (T - Ti1)/10) = 5,
   To1 = T, To2 = Ti2}.

trans_c(norod, add2, rod2, Tetai, Tetao, T, Ti1, Ti2, To1, To2, F) :-
  (F == 1 -> Ti = Ti1; Ti = Ti2),
  {Tetai < 550, Tetao = 550, exp(e, (T - Ti)/10) = 5,
   To1 = Ti1, To2 = T}.

trans_c(rod2, remove2, norod, Tetai, Tetao, T, Ti1, Ti2, To1, To2, F) :-
  {Tetai > 510, Tetao = 510, exp(e, (T - Ti2)/10) = 9/5,
   To1 = Ti1, To2 = T}.

trans_c(norod, _, shutdown, Tetai, Tetao, T, Ti1, Ti2, To1, To2, F) :-
  (F == 1 -> Ti = Ti1; Ti = Ti2),
  {Tetai < 550 Tetao = 550, exp(e, (T - Ti)/10) = 5,
   To1 = Ti1, To2 = Ti2}. 
:- coinductive(contr/7).
contr(X, Si, T, Tetai, Ti1, Ti2, Fi) :-
  (H = add1; H = remove1; H = add2; H = remove2; H = shutdown),
  \{Ta > T\},
  freeze(X, contr(Xs, So, Ta, Tetao, To1, To2, Fo)),
  trans_c(Si, H, So, Tetai, Tetao, T, Ti1, Ti2, To1, To2, Fi),
  ((H=add1; H=remove1) -> Fo = 1; Fo = 2),
  ((H=add1; H=remove1; H=add2; H=remove2) -> X = [ (H, T) | Xs]; X = [ (H, T) ] ).

:- coinductive(rod1/6).
rod1([ (H, T)| Xs], Si1, Si2, Ti1, Ti2, W) :-
  H = add1 ->
  freeze(Xs,rod1(Xs, So1, Si2, To1, Ti2, W));
  H = remove1 ->
  freeze(Xs,rod1(Xs, So1, Si2, To1, Ti2, W));
  freeze(Xs,rod2(Xs, Si1, So2, Ti1, To2, W));
  rod2(Xs, So1, Si2, To1, Ti2, W)),
  trans_r1(Si1, H, So1, T, Ti1, To1, W);
  H = shutdown -> \{T - Ti1 < A, T - Ti2 < A\}.

:- coinductive(rod2/6).
rod2([ (H, T)| Xs], Si1, Si2, Ti1, Ti2, W) :-
  H = add2 ->
  freeze(Xs,rod2(Xs, Si1, So2, Ti1, To2, W));
  H = remove2 ->
  freeze(Xs,rod1(Xs, Si1, So2, Ti1, To2, W));
  freeze(Xs,rod2(Xs, Si1, So2, Ti1, To2, W));
  rod2(Xs, Si1, So2, Ti1, To2, W)),
  trans_r2(Si2, H, So2, T, Ti2, To2, W);
  H = shutdown -> \{T - Ti1 < A, T - Ti2 < A\}. 
Controller || Rod1 || Rod2

\[
\text{main}(S, T, W) :- \{ T - Tr1 = W, T - Tr2 = W \}, \\
\text{freeze}(S, (\text{rod1}(S, s0, s0, Tr1, Tr2, W); \\
\quad \text{rod2}(S, s0, s0, Tr1, Tr2, W))), \\
\text{contr}(S, s0, T, 510, Tc1, Tc2, 1).
\]

- With more elegant modeling with LP, we were able to improve the bounds on \( W \) compared to previous work
- HyTech determines \( W < 20.44 \) to prevent shutdown
- Subsequently, using linear hybrid automata with clock translation, HyTech improves to \( W < 37.8 \)
- Using our LP method, we refine it to \( W < 38.06 \)
Related Publications

3. G. Gupta et al. Co-LP and its applications, ICLP’07 (tutorial)
4. G. Gupta et al. Infinite computation, coinduction and computational logic. CALCO’11
5. A. Bansal, R. Min, G. Gupta. Goal-directed Execution of ASP. Internal Report, UT Dallas
7. R. Min, G. Gupta. Towards Predicate ASP, AIAI’09
9. N. Saeedloei, G. Gupta, Timed π-Calculus
10. N. Saeedloei, G. Gupta. Modeling/verification of CPS with coinductive coroutined CLP(R)
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, propositional satisfiability, hybrid systems, cyberphysical systems
- Metainterpreter available:
  
  http://www.utdallas.edu/~gupta/meta.tar.gz
Conclusion (cont’d)

• Computation can be classified into two types:
  – Well-founded,
    • Based on computing elements in the LFP
    • Implemented w/ recursion (start from a call, end in base case)
  – Consistency-based
    • Based on computing elements in the GFP (but not LFP)
    • Implemented via co-recursion (look for consistency)

• Combining the two allows one to compute any computable function elegantly:
  – Implementations of modal logics (LTL, etc.)
  – Complex reasoning systems (NM reasoners)

• Combining them is challenging
Motivation
Motivation
Conclusions: Future Work

• Design execution strategies that enumerate all rational infinite solutions while avoiding redundant solutions

\[
p([a|X]) :- p(X).
p([b|X]) :- p(X).
\]

-- If \( X = [a|X] \) is reported, then avoid \( X = [a, a | X], \ X = [a,a,a|X], \) etc.
-- A fair depth first search strategy that will produce
\[
X = [a,b|X]
\]

• Combining induction (tabling) and co-induction:
  – Stratified co-LP: equivalent to *stratified Büchi tree automata* (SBTAs)
  – Non-stratified co-LP: inspired by *Rabin automata*; 3 class of predicates (i) coinductive, (ii) weakly coinductive and (iii) strongly coinductive
QUESTIONS?