Microsoft Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
Logic, Coinduction, and Infinite Computation

Gopal Gupta
Luke Simon, Ajay Mallya, Ajay Bansal
Feliks Kluzniak, Richard Min, Neda Saeedloei, Kyle Marple, Brian DeVries, Elmer Salazar, Zhuo Chen, Farhad Shakerin

Applied Logic, Programming-Languages and Systems (ALPS) Lab
The University of Texas at Dallas
UT Dallas: A Brief History

- Founded in 1969; less than 50 years old
- #21 in Times of London ranking of Universities younger than 50
- #1 in US among Universities younger than 50
- 28,000 students: CS the largest department with 3,500+ students.
- Focus on computing, engineering, tech, science & management

CS @ UT Dallas

- 4th largest CS department in US (largest at UT Dallas)
- ~3,550 students (2,400 BS, 1,000 MS, 150+ PhD)
- ~1,000 CS graduates produced last year ( > 1% of US output)
- 53 T/T faculty, 1 Res, 38 Senior Lecturers, 12+ part-time lec.
- $9 Million in annual research expenditures (37th in US)
- 21st in LinkedIn placement ranking;
- Ranked #8 in NLP, #9 in SW Engg nationally (csrankings.org)
- 100s of UT Dallas CS alumni work at Microsoft
Prelude #1

- Teaching Logic Prog. to 1st year students in early 90s
- Involved in a project dealing with teaching logic, functional, and imperative programming to 1st year CS students
- 2 course sequence equally split in LP, FP, and C, including data-structures
- Question: What is Prolog’s and ML’s counterpart of circular linked lists that we covered in C?
- LP systems of the time did allow circular structures to be created and unified
  - No occurs check
- At the time I thought that tabled LP was what was needed; posed the problem to many PhD students incl. Luke & Ajay
Prelude #2

- In mid 90s Enrico Pontelli and I worked on modeling timed automata with LP and CLP(R)
- Timed automata are $\omega$-automata that accept infinite strings composed of finite strings repeated infinitely often
- We compromised by considering only one run of the automata around a cycle for verifying properties
- The question of how to handle this elegantly lingered
Prelude #3

- Luke, interested in functional programming and process algebra, stumbles upon a book that discusses coinduction:
  
  Vicious Circles
  
  by
  
  Jon Barwise
  
  Larry Moss

- Luke tries programming $\infty$-streams in LP, doesn’t see much in it, writes a technical memo, and files it away
- Luke, Mallya, Bansal continue to discuss coinduction
- A little later they discuss the ideas with me, we connect the dots and the solution to manipulating circular lists & $\omega$-automata found; many more applications developed
Prelude #4

- We develop the idea of coinductive logic programming and submit the paper to ICLP 2006
- The paper is rejected 😞
- We lodge a protest, PC Chairs get it reviewed again, and the paper is accepted
- In 2016, the paper got the 10 year Test of Time award at ICLP 2016.

The team of Simon, Mallya, Bansal won many awards:
- Mallya: best student paper award at ICLP’05
- Simon, Mallya, Bansal: best paper award ECOWS and then the test of time award
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:
  ```quote```
  Whatever involves all of the collection must not be one of the collection
  ```quote```
  -- 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 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:
  \[ X = [1,2,3 | X] \quad X = f(X) \]
- 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 cannot 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
  – Jack will eat if Jill eats, and Jill will eat if Jack eats
• 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 developed by Peter Aczel in the 80s
  - Relates to coalgebras/category theory
- Infinity: rational and irrational
- Systematic presentation of coinduction & its application to computing, math and set theory:
  “Vicious Circles” by Moss and Barwise (1996)
- Our initial focus: inclusion of coinductive reasoning techniques in 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) ......(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
  - 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

- $\Gamma_N(S) = \{ 0 \} \cup \{ \text{succ}(x) \mid x \in S \}$
- $N = \mu \Gamma_N$
  - where $\mu \Gamma$ is least fixed-point (inductive)
- $\Gamma_N$ unambiguously defines another set
- $N' = \nu \Gamma_N = N \cup \{ \omega \}$
  - $\omega = \text{succ}(\text{succ}(\text{succ}(\ldots))) = \text{succ}(\omega) = \omega + 1$
  - where $\nu \Gamma_N$ is a greatest fixed-point (coinductive)
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 tech.</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
- Reasoning with infinite structures
- Perpetual processes
- Cyclic structures
- Operational semantics of “coinductive logic programming”
- Type inference systems for lazy functional languages
- Common sense reasoning
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

\[
\text{bit}(0).
\]
\[
\text{bit}(1).
\]
\[
\text{bitstream}( [ H \mid T ] ) :- \text{bit}( H ), \text{bitstream}( T ).
\]
\[
\text{?- } X = [ 0, 1, 1, 0 \mid X ], \text{bitstream}( X ).
\]
- 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.
  
  ```prolog
  :- coinductive from/2.
  from(x) = x cons from(x+1)
  from( N, [ N | T ] ) :- from( s(N), T ).
  ?- from( 0, X ).
  ```
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 $x = t$

- transition rules
  - definite clause rule
  - "coinductive hypothesis rule"
    - if a coinductive goal $G$ is called, and $G$ unifies with an ancestor call then $G$ succeeds.
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 $p : - p$. => the query $\leftarrow p$. will succeed.
    - $p( [1 | T ] ) : - p( T )$. => $\leftarrow p( X )$ 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 an ancestor call then $G$ succeeds.
Coinductive LP vs Tabled LP

- Coinductive LP is the dual of Tabled LP
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 $p ::- p. =>$ the query $\text{?- p.}$ will succeed.
    - $p([[1|T]]) ::- p(T). =>$ $\text{?- p(X)}$ 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 an ancestor call then
        $G$ succeeds.
Coinductive LP vs Tabled LP

- Coinductive LP is the dual of Tabled LP
- Tabled LP: computes LFP of a set of recursive predicates
- Coinductive LP: computes GFP, in contrast
- Queries for which Tabled LP fails, Coinductive LP should succeed, and *vice versa*
- Given:
  \[ p \leftarrow p. \]
  Query: \[ ?- p \]
  fails under Tabled LP, succeeds under Coinductive LP
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.
  
  ```prolog
  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 meta-interpreter available on my homepage
- 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 | T]
T = [s(0), s(0) | T]  
T = [0, s(0) | T]
T = [s(s(0)) | T]  
T = [0, s(0), s(s(0)) | T]

........
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: 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

\[
\begin{align*}
\text{member}(H, [H | T]). \\
\text{member}(H, [X | T]) & :\text{member}(H, T). \\
?- L = [1,2 | L], \text{member}(3, L). & \text{ succeeds. Instead:} \\
:- \text{coinductive comember/2}. & \%\text{drop/3 is inductive} \\
\text{comember}(X, L) & :\text{drop}(X, L, R), \text{comember}(X, R). \\
\text{drop}(H, [H | T], T). \\
\text{drop}(H, [X | T], T1) & :\text{drop}(H, T, T1). \\
\end{align*}
\]

?- X=[1, 2, 3 | X], \text{comember}(2, X). \text{Answer: yes.} \\
?- X=[1, 2, 3, 1, 2, 3], \text{comember}(2, X). \text{Answer: no.} \\
?- X=[1, 2, 3 | X], \text{comember}(Y, X). \\
\text{Answer: Y = 1;} \\
\text{Y = 2;} \\
\text{Y = 3;} \\
?- X = [1,2 | X], \text{comember}(3, X). \text{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.,
    - \( p :\neg q \).
    - \( q :\neg p \).
  Program rejected, if \( p \) coinductive & \( q \) inductive
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
The Nature of Computation

- Computation can be classified into two types:
  - Well-founded,
    - Based on computing elements of 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)
      - We consider only rational infinite elements
    - Implemented via co-recursion (look for consistency)
- Combining the two allows one to express any computable function elegantly:
  - Implementations of modal logics (LTL, etc.)
  - Complex reasoning systems (Nonmonotonic logics)
LFP vs GFP
LFP vs GFP
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

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

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

?- \text{automata}(X,s0).
\quad X=[ a, b, c, d | X ];
\quad X=[ a, b, e | X ];
\quad \ldots;
\quad \ldots;

Figure A
An Interpreter for LTL

% Negation Normal Form: 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);
% A u B
verify(S, P, and(A, x(u(A, 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)))).

Real-time extension of LTL easily obtained
Verification of Real-Time Systems
“Train, Controller, Gate”

(i) train

(ii) controller

(iii) gate

Timed Automata

- $\omega$-automata w/ time constrained transitions & stopwatches
- straightforward encoding into CLP($R$) + Co-LP
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”

(i) train

(ii) controller

(iii) gate

Timed Automata

• $\omega$-automata w/ time constrained transitions & stopwatches
• straightforward encoding into $\text{CLP}(R) + \text{Co-LP}$
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”

\[
\begin{align*}
\text{contr}(s0, \text{approach}, s1, T1, T2, T1). \\
\text{contr}(s1, \text{lower}, s2, T1, T2, T3): - \{T3=T2, T1-T2=1\}. \\
\text{contr}(s2, \text{exit}, s3, T1, T2, T1). \\
\text{contr}(s3, \text{raise}, s0, T1, T2, T2): -\{T1-T2<1\}. \\
\text{contr}(X, \text{in}, X, T1, T2, T2). \\
\text{contr}(X, \text{up}, X, T1, T2, T2). \\
\text{contr}(X, \text{out}, X, T1, T2, T2). \\
\text{contr}(X, \text{down}, X, T1, T2, T2). \\
\end{align*}
\]
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,T,A,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) \mid R \},
   X=[approach, lower, down, in, out, exit, raise, up \mid X] ;
   R=\{(approach,A),(lower,B),(down,C),(in,D),(out,E),(exit,F),(raise,G),
        (approach,H),(up,I)\mid R \},
   X=[approach,lower,down,in,out,exit,raise,approach,up \mid X] ;

% where A, B, C, ... H, I are the corresponding wall clock time of events generated.
DPP – Safety: Deadlock Free

- 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) :-
  X=1, str_driver([1,1,1,1], [2,_,_,_,_,_]);
  X=2, str_driver([1,1,1,1], [_,2,_,_,_,_]);
  X=3, str_driver([1,1,1,1], [_,_,2,_,_,_]);
  X=4, str_driver([1,1,1,1], [_,_,_,2,_,_]);
  X=5, str_driver([1,1,1,1], [_,_,_,_,2,_,_]).

?- starved(X).
no
Other Applications

- Advanced $\omega$-structures can also be modeled (Saeedloei)
  - $\omega$-PDA, $\omega$-grammars, Cyber physical systems
  - Operational semantics of $\pi$-calculus elegantly given (v operator)
- Coinductive Constraint Logic Programming (Saeedloei)
- Coinduction can be extended to reasoning with negation:
  - $\text{co-SLDNF}$ resolution (Min)
- Goal-directed execution strategies for answer set programming designed (Bansal, Min, Marple)
- Top-down, query driven predicate answer set programming systems built (Marple, Salazar)
- Elegant proof-theoretic foundations of negation in LP can be developed (Salazar)
Other Applications

- Advanced $\omega$-structures can also be modeled (Saeedloei)
  - $\omega$-PDA, $\omega$-grammars, Cyber physical systems
  - Operational semantics of $\pi$-calculus elegantly given
- Coinductive Constraint Logic Programming (Saeedloei)
- Coinduction can be extended to reasoning with negation:
  - co-SLDNF resolution (Min, Bansal, Marple)
- Goal-directed extended strategies for answer set programming (Bansal, Min, Marple)
- Temporal query driven predicate answer set programming systems built (Marple, Salazar)
- Elegant proof-theoretic foundations of negation in LP can be developed (Salazar)
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
Rod1 & Rod2

\[
\text{trans}_r1(\text{out1}, \text{add1}, \text{in1}, T, Ti, To, W) \\
\quad : - \\
\quad \{T - Ti \geq W, To = Ti\}.
\]

\[
\text{trans}_r1(\text{in1}, \text{remove1}, \text{out1}, T, Ti, To, W) : - \{To = T\}.
\]

\[
\text{trans}_r2(\text{out2}, \text{add2}, \text{in2}, T, Ti, To, W) \\
\quad : - \\
\quad \{T - Ti \geq W, To = Ti\}.
\]

\[
\text{trans}_r2(\text{in2}, \text{remove2}, \text{out2}, T, Ti, To, W) : - \{To = T\}.
\]
Controller

\[
\text{trans}_c(\text{norod}, \text{add1}, \text{rod1}, \text{Tetai}, \text{Tetao}, \text{T}, \text{Ti1}, \text{Ti2}, \text{To1}, \text{To2}, \text{F}) :- \\
(F == 1 \rightarrow \text{Ti} = \text{Ti1}; \text{Ti} = \text{Ti2}), \\
\{\text{Tetai} < 550, \text{Tetao} = 550, \exp(e, (\text{T} - \text{Ti1})/10) = 5, \\
\text{To1} = \text{T}, \text{To2} = \text{Ti2}\}.
\]

\[
\text{trans}_c(\text{rod1}, \text{remove1}, \text{norod} \text{Tetai}, \text{Tetao}, \text{T}, \text{Ti1}, \text{Ti2}, \text{To1}, \text{To2}, \text{F}) :- \\
\{\text{Tetai} > 510 \text{Tetao} = 510, \exp(e, (\text{T} - \text{Ti1})/10) = 5, \\
\text{To1} = \text{T}, \text{To2} = \text{Ti2}\}.
\]

\[
\text{trans}_c(\text{norod, add2}, \text{rod2}, \text{Tetai}, \text{Tetao}, \text{T}, \text{Ti1}, \text{Ti2}, \text{To1}, \text{To2}, \text{F}) :- \\
(F == 1 \rightarrow \text{Ti} = \text{Ti1}; \text{Ti} = \text{Ti2}), \\
\{\text{Tetai} < 550, \text{Tetao} = 550, \exp(e, (\text{T} - \text{Ti1})/10) = 5, \\
\text{To1} = \text{Ti1}, \text{To2} = \text{T}\}.
\]

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

\[
\text{trans}_c(\text{norod, _, shutdown} \text{Tetai}, \text{Tetao}, \text{T}, \text{Ti1}, \text{Ti2}, \text{To1}, \text{To2}, \text{F}) :- \\
(F == 1 \rightarrow \text{Ti} = \text{Ti1}; \text{Ti} = \text{Ti2}), \\
\{\text{Tetai} < 550 \text{Tetao} = 550, \exp(e, (\text{T} - \text{Ti})/10) = 5, \\
\text{To1} = \text{Ti1}, \text{To2} = \text{Ti2}\}.
\]
Controller || Rod1 || Rod2

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

- With this elegant modeling, 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

4. G. Gupta et al. Infinite computation, coinduction and computational logic. *CALCO’11*
5. R. Min, A. Bansal, G. Gupta. Co-LP with negation, *LOPSTR 2009*
6. R. Min, G. Gupta. Towards Predicate ASP, *AIAI’09*
8. N. Saeedloei, G. Gupta, Timed τ-Calculus
9. N. Saeedloei, G. Gupta. Modeling/verification of CPS with coinductive coroutined CLP(R)
10. K. Marple, A. Bansal, R. Min, G. Gupta. Goal-directed Execution of ASP. *PPDP’12*
Applications of CoLP

- Type inference for coinductive types in OO-languages; inspired Featherweight Java
  - ECOOP best paper award winner (David Ancona & Giovanni Lagorio)
- Further work on coLP by Ancona and Dovier including application of coinductive subtyping to abstract compilation
- Significant work by the group of Ekaterina Komendantskaya at Dundee on coalgebraic logic programming and extending coLP
- CoCAML: Extending OCAML with coinduction (Kozen et al)
- Hirohisa Seki has developed fold/unfold transformations for coLP, and used it for developing branching-time model checking
- Coinductive semantics for CHR: Remy Haemmerlé at IMDEA
- Actionscript Bytecode Verification using coLP (Hamlen et al)
Coinduction and AI

• Coinduction is crucial to automating common sense reasoning.
• Given a situation we model it using statements in logic
• These statements may have multiple models, each representing a possible world. Consider
  
  \[
  \text{jack\_eats} \leftarrow \text{jill\_eats}.
  
  \text{jill\_eats} \leftarrow \text{jack\_eats}.
  \]
  \[
  \text{jack\_eats} \leftrightarrow \text{jill\_eats}
  \]

• As long as we are in the propositional world, no problem ....
  
  • Two models: both eat or neither eats
  
• Suppose we generalize it: \[
\text{eats}(X) \leftarrow \text{eats}(Y).
\]
  \[
\text{eats}(Y) \leftarrow \text{eats}(X).
\]

• We generally want to give operational semantics to these rules, for ease of programming (and to have a proof trace for a query)
  
  • but no good induction-based operational semantics;
  
  • Coinduction based corecursive semantics will have to be used

• Introduce negation, and life becomes even more complex
Coinduction and AI

- Consider the jury decision task in a murder trial where individual A has stabbed individual B; suppose we want to automate this task.
- Various hypotheses can be constructed:
  - A was afraid of B (prior altercation), and took a knife to the bar in self defense.
  - A is a revengeful person, who took a knife to the bar to attack B.
- Many scenarios can be constructed based on common sense knowledge (A was known to be troublemaker, so 2nd option likely).
- Each scenario has information which is mutually consistent, and will produce a guilty or not guilty verdict.
- Each scenario is a solution to a set of corecursive equations; the moment a fact is established, it becomes inductive/recursive, and we have firm basis to produce a judgement.
- Much of common sense reasoning is similar to above.
Common Sense Reasoning

- Coinductive LP has been used to give operational semantics to predicate *answer set programming (ASP)*, an extension of logic programming that allows negation as failure.
- ASP allows simulation of various mechanisms used in common sense reasoning:
  - Default reasoning, non monotonic reasoning, abductive reasoning, counterfactual reasoning, preferences, etc.
- A query-driven Predicate ASP system was thought to be impossible to build: coinductive LP made it possible.
- Realized in the s(ASP) system, freely available from my home page.
- The s(ASP) system has been used to simulate expert knowledge based on common sense reasoning:
  - Used in developing a system for treating heart failure (outperforms doctors)
  - Used in developing natural language question answering system that makes use of common sense reasoning to answer questions; (can easily outperform machine learning-based systems)
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
- Coinductive LP is a powerful concept:
  - applications to verification, non monotonic reasoning, negation in LP, type inference, hybrid systems, cyberphysical systems, common sense reasoning
- Metainterpreter for coinductive LP available:
  http://www.utdallas.edu/~gupta/meta.tar.gz
- s(ASP) system: utdallas.edu/~gupta
THANK YOU
http://utdallas.edu/~gupta
QUESTIONS?