Coinduction and Declarative Programming

Horst Reichel

December 2012

Structure of the lecture:

Structure of the lecture:

- What is the meaning of coinduction?

Structure of the lecture:

- What is the meaning of coinduction?
- Some applications of coinduction.

Structure of the lecture:

- What is the meaning of coinduction?
- Some applications of coinduction.
- Coinduction and logic programming.

Structure of the lecture:

- What is the meaning of coinduction?
- Some applications of coinduction.
- Coinduction and logic programming.
- Coinduction and constraint programming.

Structure of the lecture:

- What is the meaning of coinduction?
- Some applications of coinduction.
- Coinduction and logic programming.
- Coinduction and constraint programming.

The notion of INDUCTION is well known, but what is the meaning of COINDUCTION?

The notion of INDUCTION is well known, but what is the meaning of COINDUCTION?

Can the pattern AUTHOR and COAUTHOR be used to derive the meaning of coinduction?

The notion of INDUCTION is well known, but what is the meaning of COINDUCTION?

Can the pattern AUTHOR and COAUTHOR be used to derive the meaning of coinduction?

NO!

The notion of INDUCTION is well known, but what is the meaning of COINDUCTION?

Can the pattern AUTHOR and COAUTHOR be used to derive the meaning of coinduction?

NO!

In our case the prefix CO means that both concepts are dual in the sense of category theory.

The notion of INDUCTION is well known, but what is the meaning of COINDUCTION?

Can the pattern AUTHOR and COAUTHOR be used to derive the meaning of coinduction?

NO!

In our case the prefix CO means that both concepts are dual in the sense of category theory.

Intuitively Coinduction is a mathematical technique for (finitely) reasoning about infinite things.

In order to understand the duality of induction and coinduction we start with the Fixed Point Theorem of Knaster and Tarski.

In order to understand the duality of induction and coinduction we start with the Fixed Point Theorem of Knaster and Tarski.

Fixed Point Theorem (Knaster, Tarski):
If $g : 2^S \to 2^S$ is monotonic with respect to the subset ordering $\subseteq$ then g

1. has a least fixed point with respect to $\subseteq$ given by $\bigcap\{X \subseteq S \mid g(X) \subseteq X\}$
2. has a greatest fixed point with respect to $\subseteq$ given by $\bigcup\{X \subseteq S \mid X \subseteq g(X)\}$.

Let denote

$$PRE_g = \{X \subseteq S \mid g(X) \subseteq X\}$$

respectively

$$POST_g = \{X \subseteq S \mid X \subseteq g(X)\}$$

the set of prefixed respectively postfixed points of the monoton mapping $g : 2^S \to 2^S$.

Let denote

$$PRE_g = \{X \subseteq S \mid g(X) \subseteq X\}$$

respectively

$$POST_g = \{X \subseteq S \mid X \subseteq g(X)\}$$

the set of prefixed respectively postfixed points of the monoton mapping $g : 2^S \to 2^S$.

The least fixed point $lfp(g)$ is the inductively defined prefixed point, and the greatest fixed point $gfp(g)$ is the coinductively defined postfixed point of the monoton mapping $g : 2^S \to 2^S$.

Let denote

$$PRE_g = \{X \subseteq S \mid g(X) \subseteq X\}$$

respectively

$$POST_g = \{X \subseteq S \mid X \subseteq g(X)\}$$

the set of prefixed respectively postfixed points of the monoton mapping $g : 2^S \to 2^S$.

The least fixed point $lfp(g)$ is the inductively defined prefixed point, and the greatest fixed point $gfp(g)$ is the coinductively defined postfixed point of the monoton mapping $g : 2^S \to 2^S$.

One should remember that the μ–calculus uses least fixed points and greatest fixed points to express liveness respectively safety properties.

As next we generalize the monotonic map $g : 2^S \to 2^S$ to an endofunctor of the category of sets.

As next we generalize the monotonic map $g : 2^S \to 2^S$ to an endofunctor of the category of sets.

A type functor $\mathcal{T} : \mathcal{S}et \to \mathcal{S}et$ is a uniform construction

1. that assigns a set $\mathcal{T}(X)$ to a given set X,
2. and assigns a mapping $\mathcal{T}(f) : \mathcal{T}(A) \to \mathcal{T}(B)$ to a given mapping $f : A \to B$,
3. which is compatible with the composition of mappings, i.e., $\mathcal{T}(f; g) = \mathcal{T}(f); \mathcal{T}(g)$ for $f : A \to B, g : B \to C$.

As next we generalize the monotonic map $g : 2^S \to 2^S$ to an endofunctor of the category of sets.

A type functor $\mathcal{T} : \mathcal{S}et \to \mathcal{S}et$ is a uniform construction

1. that assigns a set $\mathcal{T}(X)$ to a given set X,
2. and assigns a mapping $\mathcal{T}(f) : \mathcal{T}(A) \to \mathcal{T}(B)$ to a given mapping $f : A \to B$,
3. which is compatible with the composition of mappings, i.e., $\mathcal{T}(f; g) = \mathcal{T}(f); \mathcal{T}(g)$ for $f : A \to B, g : B \to C$.

Prefixed points and postfixed points will become $\mathcal{T}-$ algebras and $\mathcal{T}-$coalgebras respectively.

For a given type functor $\mathcal{T} : \mathcal{S}et \to \mathcal{S}et$

- a $\mathcal{T}$– algebra $\mathbf{A} = (A, \alpha)$ is given by a set A and a mapping $\alpha : \mathcal{T}(A) \to A$
- a $\mathcal{T}$– coalgebra $\mathbf{S} = (S, \sigma)$ is given by a set S and a mapping $\sigma : S \to \mathcal{T}(S)$.

For a given type functor $\mathcal{T} : \mathcal{S}et \to \mathcal{S}et$

- a $\mathcal{T}$– algebra $\mathbf{A} = (A, \alpha)$ is given by a set A and a mapping $\alpha : \mathcal{T}(A) \to A$
- a $\mathcal{T}$– coalgebra $\mathbf{S} = (S, \sigma)$ is given by a set S and a mapping $\sigma : S \to \mathcal{T}(S)$.

For a $\mathbf{A} = (A, \alpha)$ A is called the *carrier set* and α the *structure map*, and for a $\mathcal{T}$– coalgebra $\mathbf{S} = (S, \sigma)$ the set S is called the *universe* and σ the *structure map*.

Examples:

1. The signature, or the type of a *group* G is given by a constant $e \in G$, by a unary operation $\_^{-1} : G \to G$ and a binary operation $\_ \cdot \_ : G \times G \to G$. This ranked alphabet can be represented by the type functor $\mathcal{T}_0 : \mathcal{S}et \to \mathcal{S}et$ with

$$\mathcal{T}_0(G) = 1 + G + G \times G$$

where $1 = \{0\}$ represents the empty product of sets.

Examples:

1. The signature, or the type of a *group* G is given by a constant $e \in G$, by a unary operation $\_^{-1} : G \to G$ and a binary operation $\_ \cdot \_ : G \times G \to G$. This ranked alphabet can be represented by the type functor $\mathcal{T}_0 : \mathcal{S}et \to \mathcal{S}et$ with

$$\mathcal{T}_0(G) = 1 + G + G \times G$$

where $1 = \{0\}$ represents the empty product of sets.

Let be $i_1 : \{0\} \to 1 + G + G \times G$, $i_2 : G \to 1 + G + G \times G$, $i_3 : G \times G \to 1 + G + G \times G$ the canonical injections of the summ and let (G, α) be a $\mathcal{T}_0$–algebra, then $i_1; \alpha$, $i_2; \alpha$, $i_3; \alpha$ are the constant, unary and binary operation, respectively.

2. Let be In, Out two fixed sets. Now we can define a type functor $\mathcal{T}_1 : \mathcal{S}et \to \mathcal{S}et$ by

$$\mathcal{T}_1(S) = S^{In} \times Out.$$

If (S, σ) is a $\mathcal{T}_1$–coalgebra and $p_1 : S^{In} \times Out \to S^{In}$, $p_2 : S^{In} \times Out \to Out$ are the canonical projections of the product we get two mappings

$$\sigma_1 = \sigma; p_1 : S \to S^{In}$$
$$\sigma_2 = \sigma; p_2 : S \to Out.$$

It is easy to see that a $\mathcal{T}_1$–coalgebra represents a *Moore automaton*. The mapping σ_1 is equivalent (by currying) to a mapping $\sigma_1' : S \times In \to S$ known as transition function.

Whereas the structure mappings of pre- and postfixed points are set inclusions, the structure mappings of $\mathcal{T}$ –algebras and $\mathcal{T}$–coalgebras are arbitrary mappings.

Whereas the structure mappings of pre- and postfixed points are set inclusions, the structure mappings of $\mathcal{T}$ –algebras and $\mathcal{T}$–coalgebras are arbitrary mappings.

Now we can generalize least and greatest fixed points to $\mathcal{T}$–algebras and $\mathcal{T}$–coalgebras.

Whereas the structure mappings of pre- and postfixed points are set inclusions, the structure mappings of $\mathcal{T}$ –algebras and $\mathcal{T}$–coalgebras are arbitrary mappings.

Now we can generalize least and greatest fixed points to $\mathcal{T}$–algebras and $\mathcal{T}$–coalgebras.

The resulting notions are initial and final (terminal) $\mathcal{T}$–algebras and $\mathcal{T}$–coalgebras respectively.

A $\mathcal{T}$–algebra $\mathbf{I} = (I, \iota)$ is called initial if for for any $\mathcal{T}$–algebra $\mathbf{A} = (A,\ \mathit{alpha})$ there exists exactly one $\mathcal{T}$–homomorphism $h : \mathbf{I} \to \mathbf{A}$.

A $\mathcal{T}$–algebra $\mathbf{I} = (I, \iota)$ is called initial if for for any $\mathcal{T}$–algebra $\mathbf{A} = (A,\ \mathit{alpha})$ there exists exactly one $\mathcal{T}$–homomorphism $h : \mathbf{I} \to \mathbf{A}$.

Dualy, a $\mathcal{T}$–coalgebra $\mathbf{F} = (F, \varphi)$ is called final if for any $\mathcal{T}$–coalgebra $\mathbf{S} = (S, \sigma)$ there exists exactly one $\mathcal{T}$–homomorphism $h : \mathbf{S} \to \mathbf{F}$.

A $\mathcal{T}$–algebra $\mathbf{I} = (I, \iota)$ is called initial if for for any $\mathcal{T}$–algebra $\mathbf{A} = (A,\ \mathit{alpha})$ there exists exactly one $\mathcal{T}$–homomorphism $h : \mathbf{I} \to \mathbf{A}$.

Dualy, a $\mathcal{T}$–coalgebra $\mathbf{F} = (F, \varphi)$ is called final if for any $\mathcal{T}$–coalgebra $\mathbf{S} = (S, \sigma)$ there exists exactly one $\mathcal{T}$–homomorphism $h : \mathbf{S} \to \mathbf{F}$.

For a $\mathcal{T}$–homomorphism it is required, that the mapping is compatible with the corresponding structure map of the $\mathcal{T}$–algebras or $\mathcal{T}$–coalgebras respectively.

Induction in terms of initial $\mathcal{T}$–algebras is intensively used in the approach of abstract data types.

Induction in terms of initial $\mathcal{T}$–algebras is intensively used in the approach of abstract data types.

Coinduction in terms of final $\mathcal{T}$–coalgebras was manly applied to reason about the behavior of state based systems.

Induction in terms of initial $\mathcal{T}$–algebras is intensively used in the approach of abstract data types.

Coinduction in terms of final $\mathcal{T}$–coalgebras was manly applied to reason about the behavior of state based systems.

The unique final $\mathcal{T}$–coalgebra represents the behavior that can be observed for elements of $\mathcal{T}$–coalgebras.

Induction in terms of initial $\mathcal{T}$–algebras is intensively used in the approach of abstract data types.

Coinduction in terms of final $\mathcal{T}$–coalgebras was manly applied to reason about the behavior of state based systems.

The unique final $\mathcal{T}$–coalgebra represents the behavior that can be observed for elements of $\mathcal{T}$–coalgebras.

The unique $\mathcal{T}$–homomorphism

$$h : \mathbf{S} \to \mathbf{F}$$

identifies elements in S if and only if they have equal observable behavior.

Induction in terms of initial $\mathcal{T}$–algebras is intensively used in the approach of abstract data types.

Coinduction in terms of final $\mathcal{T}$–coalgebras was manly applied to reason about the behavior of state based systems.

The unique final $\mathcal{T}$–coalgebra represents the behavior that can be observed for elements of $\mathcal{T}$–coalgebras.

The unique $\mathcal{T}$–homomorphism

$$h : \mathbf{S} \to \mathbf{F}$$

identifies elements in S if and only if they have equal observable behavior. Thus, the congruence relation of the unique homomorphism to the final coalgebra is the greatest (strong) bisimulation in the domain coalgebra.

In case of Moore automata a final coalgebra (unique up to isomorphisms) $\mathbf{F} = (F, \varphi)$ is given by $F = Out^{In^*}$, $\varphi_2(f) = f(\epsilon)$ and $\varphi_1'(f, a) = f(aw)$ for each $f \in F$.

In case of Moore automata a final coalgebra (unique up to isomorphisms) $\mathbf{F} = (F, \varphi)$ is given by $F = Out^{In^*}$, $\varphi_2(f) = f(\epsilon)$ and $\varphi_1'(f, a) = f(aw)$ for each $f \in F$.

The unique homomorphism $h; \mathbf{S} \to \mathbf{F}$ assignes to a state $s \in S$ the mapping $\lambda w \in In^*.\sigma_2(\sigma_1^*(s, w))$ where $\sigma_1^* : S \times In \to S$ is defined by $\sigma_1^*(s, \epsilon) = s$ and $\sigma_1^*(s, wa) = \sigma_1'(\sigma_1^*(s, w), a)$.

In the last years coinduction has found many applications:

In the last years coinduction has found many applications:

- model checking

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages
- semantics of programming languages

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages
- semantics of programming languages
- perpetual processes

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages
- semantics of programming languages
- perpetual processes
- cyclic structures

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages
- semantics of programming languages
- perpetual processes
- cyclic structures
- unification of modal logics

In the last years coinduction has found many applications:

- model checking
- bisimilarity proofs
- lazy evaluation in functional programming languages
- semantics of programming languages
- perpetual processes
- cyclic structures
- unification of modal logics
- computable analysis

The declarative semantics of Prolog can be defined by a least fixed point and least Herbrand model, which is an initial model in the corresponding category of models.

The declarative semantics of Prolog can be defined by a least fixed point and least Herbrand model, which is an initial model in the corresponding category of models.

Does this mean that only induction is of interest in logic programming?

The declarative semantics of Prolog can be defined by a least fixed point and least Herbrand model, which is an initial model in the corresponding category of models.

Does this mean that only induction is of interest in logic programming?

On the other side, most Prolog implementations know infinite terms. But they don't offer any support to reason about infinite terms.

On the ICLP conference in 2006 Gopal Gupta presented a talk on *Co–Logic Programming: Extending Logic Programming with Coinduction*.

On the ICLP conference in 2006 Gopal Gupta presented a talk on *Co–Logic Programming: Extending Logic Programming with Coinduction*.

This extension has been developed by Luke Simon, a student of G. Gupta, in his PhD thesis *Extending Logic Programming with Coinduction*.

On the ICLP conference in 2006 Gopal Gupta presented a talk on *Co–Logic Programming: Extending Logic Programming with Coinduction*.

This extension has been developed by Luke Simon, a student of G. Gupta, in his PhD thesis *Extending Logic Programming with Coinduction*.

In the following we will sketch this extension:

Now let be given a program P, Π the ranked alphabet of predicate symbols in P and F the ranked alphabet of function symbols in P.

Now let be given a program P, Π the ranked alphabet of predicate symbols in P and F the ranked alphabet of function symbols in P..

Let $HU_F^\infty = TU_{F,\emptyset}^\infty$ be the infinitary Herbrand universe and $HB_{\Pi,F}^\infty = TB_{\Pi,F,\emptyset}^\infty$ the infinitary Herbrand base associated with P. $ground(P)$ denotes the set of all ground instances of clauses of P.

Now let be given a program P, Π the ranked alphabet of predicate symbols in P and F the ranked alphabet of function symbols in P..

Let $HU_F^\infty = TU_{F,\emptyset}^\infty$ be the infinitary Herbrand universe and $HB_{\Pi,F}^\infty = TB_{\Pi,F,\emptyset}^\infty$ the infinitary Herbrand base associated with P. $ground(P)$ denotes the set of all ground instances of clauses of P.

The consequence operator

$$T_P(X) = \{A \mid A \leftarrow B_1, \ldots, B_n \in ground(P), B_1, \ldots, B_n \in X\}$$

is a monotonic mapping.

We know

- $I \subseteq HB_{\Pi,F} \subseteq HB^{\infty}_{\Pi,F}$ is a model of P if and only if $T_P(I) \subseteq I$, i.e. if I is a pre–fixed point of the consequence operator.
- The inductive semantics of P is given by the least fixed point $lfp(T_P)$ of the consequence operator.

We know

- $I \subseteq HB_{\Pi,F} \subseteq HB^{\infty}_{\Pi,F}$ is a model of P if and only if $T_P(I) \subseteq I$, i.e. if I is a pre–fixed point of the consequence operator.
- The inductive semantics of P is given by the least fixed point $lfp(T_P)$ of the consequence operator.

The coinductive semantics of a program P can now be defined as follows:

We know

- $I \subseteq HB_{\Pi,F} \subseteq HB^{\infty}_{\Pi,F}$ is a model of P if and only if $T_P(I) \subseteq I$, i.e. if I is a pre–fixed point of the consequence operator.
- The inductive semantics of P is given by the least fixed point $lfp(T_P)$ of the consequence operator.

The coinductive semantics of a program P can now be defined as follows:

- $I \subseteq HB^{\infty}_{\Pi,F}$ is a co–model if $I \subseteq T_P(I)$, i.e. if it is a post–fixed point of the consequence operator.

We know

- $I \subseteq HB_{\Pi,F} \subseteq HB^{\infty}_{\Pi,F}$ is a model of P if and only if $T_P(I) \subseteq I$, i.e. if I is a pre–fixed point of the consequence operator.
- The inductive semantics of P is given by the least fixed point $lfp(T_P)$ of the consequence operator.

The coinductive semantics of a program P can now be defined as follows:

- $I \subseteq HB^{\infty}_{\Pi,F}$ is a co–model if $I \subseteq T_P(I)$, i.e. if it is a post–fixed point of the consequence operator.
- The coinductive semantics of a program is given by the greatest fixed point $gfp(T_P)$ of the consequence operator.

To illustrate the differences between the traditional (inductive) and the coinductive semantics of a logical programm we consider the following example:

To illustrate the differences between the traditional (inductive) and the coinductive semantics of a logical programm we consider the following example:

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

To illustrate the differences between the traditional (inductive) and the coinductive semantics of a logical programm we consider the following example:

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

In the least fixed point semantics the question leads to `false`, since the least model assigns the empty set to the predicate `stream` and the set $\{0, 1, 2, 3, \ldots\}$ of natural numbers to the predicate `number`.

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

Now the answer becomes `true`

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

Now the answer becomes `true` since

$$\{0,1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 2,0,1,2,0,1,2,0,1,2,\ldots\}$$

defines a co–model of the program above

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

Now the answer becomes `true` since

$$\{0,1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 2,0,1,2,0,1,2,0,1,2,\ldots\}$$

defines a co–model of the program above which assigns the set $\{0,1,2,3,\ldots\} \cup \{\omega\}$ to the predicate `number`

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

Now the answer becomes `true` since

$$\{0,1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 2,0,1,2,0,1,2,0,1,2,\ldots\}$$

defines a co–model of the program above which assigns the set $\{0,1,2,3,\ldots\} \cup \{\omega\}$ to the predicate `number` the three-element set above to the predicate `stream`

What happens in the greatest fixed point semantics with the program

```
stream([H|T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).

?- stream([0, s(0), s(s(0))|T]).
```

Now the answer becomes `true` since

$$\{0,1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 1,2,0,1,2,0,1,2,0,1,2,\ldots,\\ 2,0,1,2,0,1,2,0,1,2,\ldots\}$$

defines a co–model of the program above which assigns the set $\{0,1,2,3,\ldots\} \cup \{\omega\}$ to the predicate `number` the three-element set above to the predicate `stream` and $gfp(T_P)$ is the union of all co–models.

The operational semantics of coinductive logic programming is given in terms of the coinductive hypothesis rule: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new resolvent is $R\theta$ where $\theta = mgu(C, C')$ and R' is obtained by C' from R.

The operational semantics of coinductive logic programming is given in terms of the coinductive hypothesis rule: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new resolvent is $R\theta$ where $\theta = mgu(C, C')$ and R' is obtained by C' from R.

The unification does not apply the *occurs check* in order to allow infinite (regular) terms.

The operational semantics of coinductive logic programming is given in terms of the coinductive hypothesis rule: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new resolvent is $R\theta$ where $\theta = mgu(C, C')$ and R' is obtained by C' from R.

The unification does not apply the *occurs check* in order to allow infinite (regular) terms.

Recently SWI–Prolog has added support for coinduction.

The operational semantics of coinductive logic programming is given in terms of the coinductive hypothesis rule: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new resolvent is $R\theta$ where $\theta = mgu(C, C')$ and R' is obtained by C' from R.

The unification does not apply the *occurs check* in order to allow infinite (regular) terms.

Recently SWI–Prolog has added support for coinduction.

In February 2012 Ronald de Haan extended the functional logic programming language CURRY with generalized circular coinduction.

This extension of Logic Programming was only a first step of the Gupta's group in Dallas.

This extension of Logic Programming was only a first step of the Gupta's group in Dallas.

In 2009 R. Min, also a student of G. Gupta, applied coinduction to *Predicate Answer Set Programming* in his PhD thesis. The results have been presented in a talk in 2009.

This extension of Logic Programming was only a first step of the Gupta's group in Dallas.

In 2009 R. Min, also a student of G. Gupta, applied coinduction to *Predicate Answer Set Programming* in his PhD thesis. The results have been presented in a talk in 2009.

The group of G. Gupta has also extended CLP (Constrained Logic Programming) by coinduction. This has bee done by N. Saeedloein in his PhD thesis *Modeling and Verification of Real–Time and Cyber–Physical Systems* in 2011 and has been presented on FLOPS 2012.

This extension of Logic Programming was only a first step of the Gupta's group in Dallas.

In 2009 R. Min, also a student of G. Gupta, applied coinduction to *Predicate Answer Set Programming* in his PhD thesis. The results have been presented in a talk in 2009.

The group of G. Gupta has also extended CLP (Constrained Logic Programming) by coinduction. This has bee done by N. Saeedloein in his PhD thesis *Modeling and Verification of Real–Time and Cyber–Physical Systems* in 2011 and has been presented on FLOPS 2012.

Gupta's group is not only very active to make coinduction applicable in declarative programming. The group has also made available interesting applications of coinduction.

One interesting application is the operational semantics of timed π–calculus with continuous real numbers.

One interesting application is the operational semantics of timed π–calculus with continuous real numbers.In the CALCO 2011 paper this application is described as follows:

One interesting application is the operational semantics of timed π–calculus with continuous real numbers.In the CALCO 2011 paper this application is described as follows:

For a complete encoding of the operational semantics of timed π–calculus, we must model three aspects of timed π–calculus processes: concurrency, infinite computation and time constraints/clock expressions. An executable operational semantics of π–calculus in logic programming has been developed (in N. Saeedloein's PhD thesis). Channels are modeled as streams, rational infinite computations are handled by by coinduction and concurrency is handled by allowing coroutining within logic programming computations. This operational semantics is extended with continuous real time, which we have modeled with constraint logic programming over reals. The executable operational semantics, thus realized, automatically leads to an implementation of the timed π–calculus in the form of a coinductive coroutined constraint logic program the can be regarded as an interpreter for timed π–calculus expressions, and can be used for modeling and verification of real–time systems.

Recently coinduction has also been investigated in connection with CHR (Constraint Handling Rules) by Remy Haemmerlé.

Recently coinduction has also been investigated in connection with CHR (Constraint Handling Rules) by Remy Haemmerlé.

He addresses the problem of defining a fixpoint semantics for Constraint Handling Rules that captures the behavior of both simplification and propagation rules in a sound and complete way with respect to their declarative semantics.

Recently coinduction has also been investigated in connection with CHR (Constraint Handling Rules) by Remy Haemmerlé.

He addresses the problem of defining a fixpoint semantics for Constraint Handling Rules that captures the behavior of both simplification and propagation rules in a sound and complete way with respect to their declarative semantics.

It turns out that semantics of simplification rules can be characterized by a least fixpoint over the transition system generated by the abstract operational semantics of CHR and that the semantics of propagation rules can be characterized by a greatest fixpoint.

With this talk we want to show that coinduction is of growing interest also for declarative programming.

With this talk we want to show that coinduction is of growing interest also for declarative programming.

Probably, what we have presented is only the beginning, since there any many open problems.

With this talk we want to show that coinduction is of growing interest also for declarative programming.

Probably, what we have presented is only the beginning, since there any many open problems.

One problem is the mutual recursive use of induction and coinduction.

With this talk we want to show that coinduction is of growing interest also for declarative programming.

Probably, what we have presented is only the beginning, since there any many open problems.

One problem is the mutual recursive use of induction and coinduction. From μ–calculus we know that the combination of liveness and safety properties leads to interesting temporal properties. But at the moment in logic programming induction and coinduction cannot be combined as freely as in μ–calculus.

With this talk we want to show that coinduction is of growing interest also for declarative programming.

Probably, what we have presented is only the beginning, since there any many open problems.

One problem is the mutual recursive use of induction and coinduction. From μ–calculus we know that the combination of liveness and safety properties leads to interesting temporal properties. But at the moment in logic programming induction and coinduction cannot be combined as freely as in μ–calculus.

Also new applications of combined inductive and coinductive logic programming is an interesting field of research.

Literature:

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming, ICPL'06 pp. 330-344, Springer LNCS 4079

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming, ICPL'06 pp. 330-344, Springer LNCS 4079

R. Min, A. Bansal and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI09*. 2009.

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming, ICPL'06 pp. 330-344, Springer LNCS 4079

R. Min, A. Bansal and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI09*. 2009.

N. Saeedloei. Extending Infinite Systems with Real–time. PhD thesis, University of Texas at Dallas. 2011.

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming, ICPL'06 pp. 330-344, Springer LNCS 4079

R. Min, A. Bansal and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI09*. 2009.

N. Saeedloei. Extending Infinite Systems with Real–time. PhD thesis, University of Texas at Dallas. 2011.

N. Saeedloei and G, Gupta. Verifying complex continuous real–time systems with coinductive CLP(R). In *LATA*, pp. 536-548, 2010.

Literature:

B. Jacobs. *Introduction to Coalgebra; Towards Mathematics of States and Observations.* Draft manuscript.

L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming, ICPL'06 pp. 330-344, Springer LNCS 4079

R. Min, A. Bansal and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI09*. 2009.

N. Saeedloei. Extending Infinite Systems with Real–time. PhD thesis, University of Texas at Dallas. 2011.

N. Saeedloei and G, Gupta. Verifying complex continuous real–time systems with coinductive CLP(R). In *LATA*, pp. 536-548, 2010.

G. Gupta, N. Saeedloei, B. W. DeVries, R. Min, K. Marple and F. Kluzniak. Infinite Computation, Co–induction and Computational Logic. In CALCO 2011, pp. 40-54. 2011

R. Haemmerlé. (Co)–Inductive Semantics for Constraint Handling Rules. in TPLP, vol.11, nr.4-5, pp 593-609. 2011