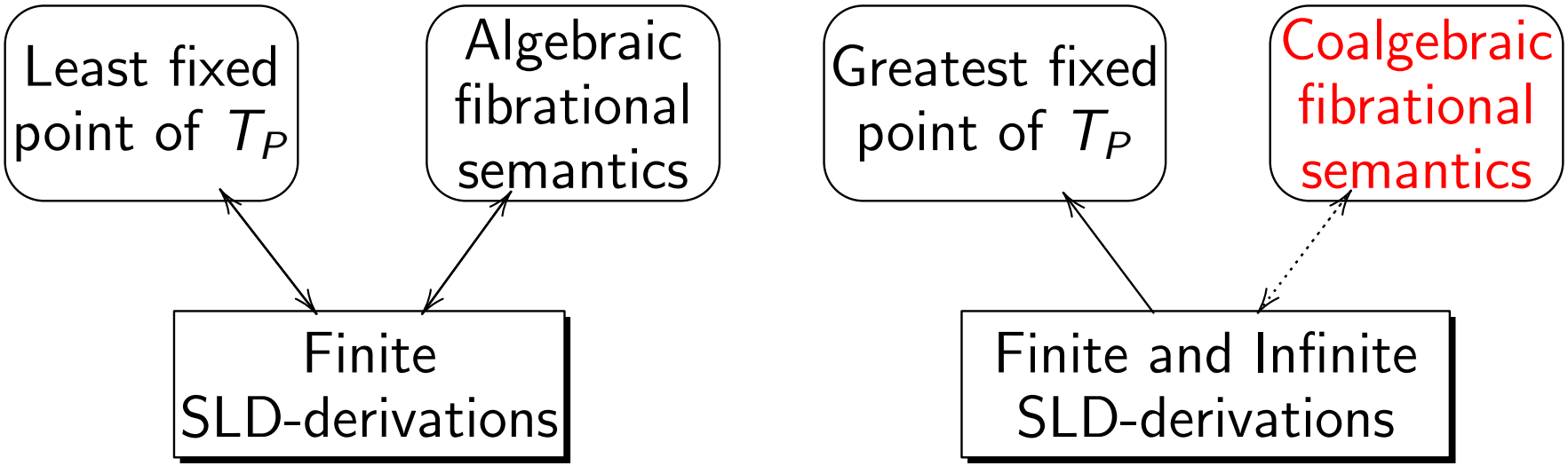
Coalgebraic Semantics in Logic Programming

Katya Komendantskaya

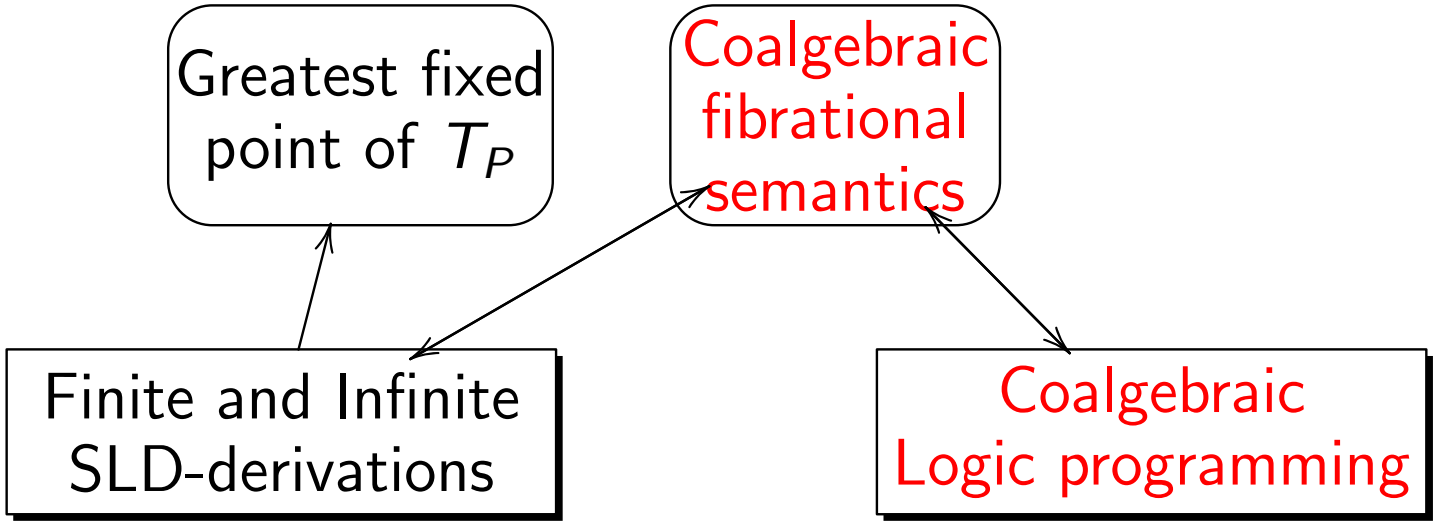
School of Computing, University of Dundee, UK

CSL'11,
13 September 2011

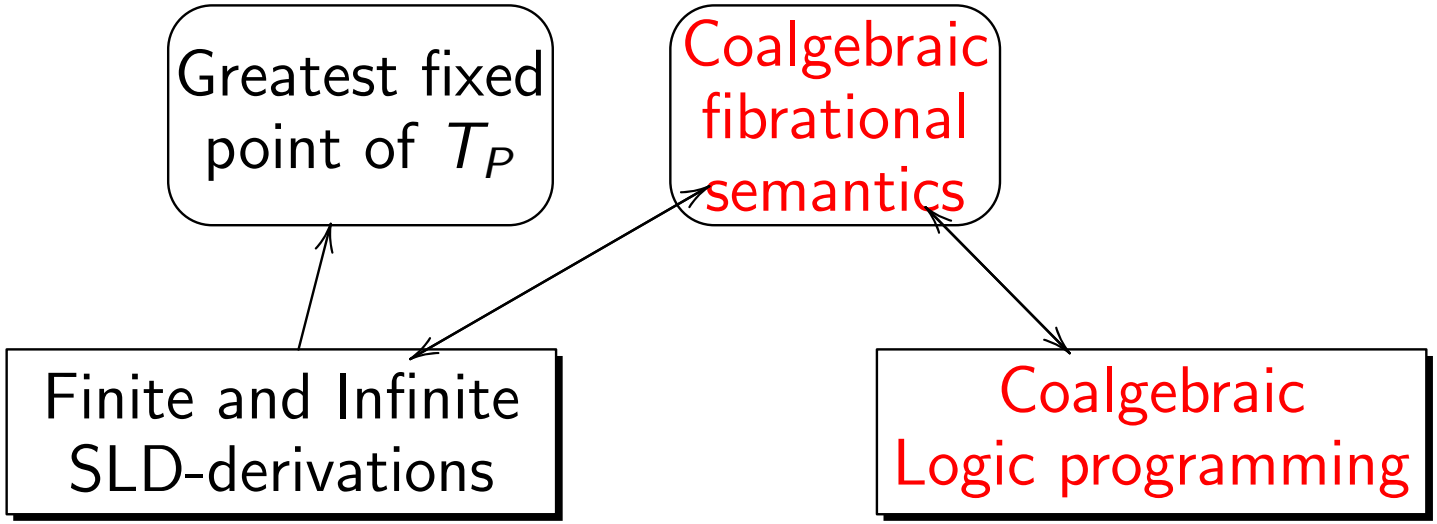
Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



In what follows, you can use intuitions coming from Process Calculi (concurrent processes and their semantics), or Transition systems. There will be some discussion of linearity and branching as well.

Outline

Outline

Outline

Outline

Recursion and Corecursion in Logic Programming

Example

```
nat(0)          ←
nat(s(x))       ←  nat(x)
list(nil)       ←
list(cons x y)  ←  nat(x), list(y)
```

Recursion and Corecursion in Logic Programming

Example

$$\begin{aligned} \texttt{bit(0)} &\leftarrow \\ \texttt{bit(1)} &\leftarrow \\ \texttt{stream(cons (x,y))} &\leftarrow \texttt{bit(x),stream(y)} \end{aligned}$$

SLD-resolution (+ unification and backtracking) behind LP derivations.

Example

$\texttt{nat(0)} \leftarrow$

$\texttt{nat(s(x))} \leftarrow \texttt{nat(x)}$

$\texttt{list(nil)} \leftarrow$

$\texttt{list(cons x y)} \leftarrow \texttt{nat(x)}, \texttt{list(y)}$

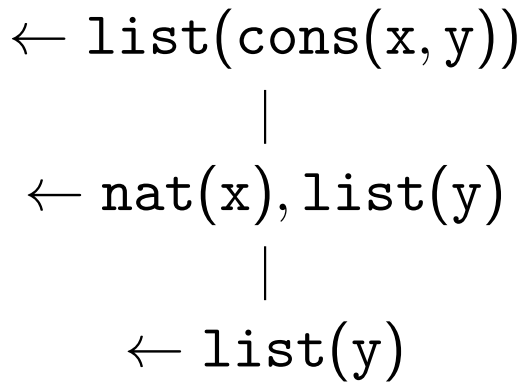
$\leftarrow \texttt{list(cons(x, y))}$

$\leftarrow \texttt{nat(x)}, \texttt{list(y)}$

SLD-resolution (+ unification) is behind LP derivations.

Example

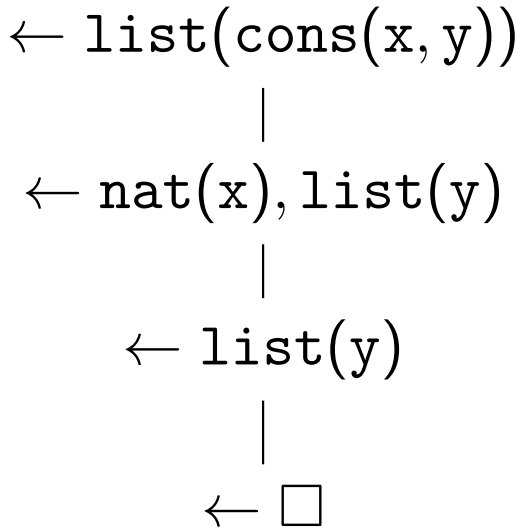
```
nat(0) ←
nat(s(x)) ← nat(x)
list(nil) ←
list(cons x y) ← nat(x),
                  list(y)
```



SLD-resolution (+ unification) is behind LP derivations.

Example

```
nat(0) ←
nat(s(x)) ← nat(x)
list(nil) ←
list(cons x y) ← nat(x),
                 list(y)
```



The answer is x/O, y/nil, but we can get more substitutions by backtracking. We can backtrack infinitely many times, but each time computation will terminate.

Things go wrong

Example

```
bit(0) ←
bit(1) ←
stream(scons x y) ←

            bit(x), stream(y)
```

Things go wrong

Example

```
bit(0) ←
bit(1) ←
stream(scons x y) ←

            bit(x), stream(y)
```

No answer, as derivation never terminates.

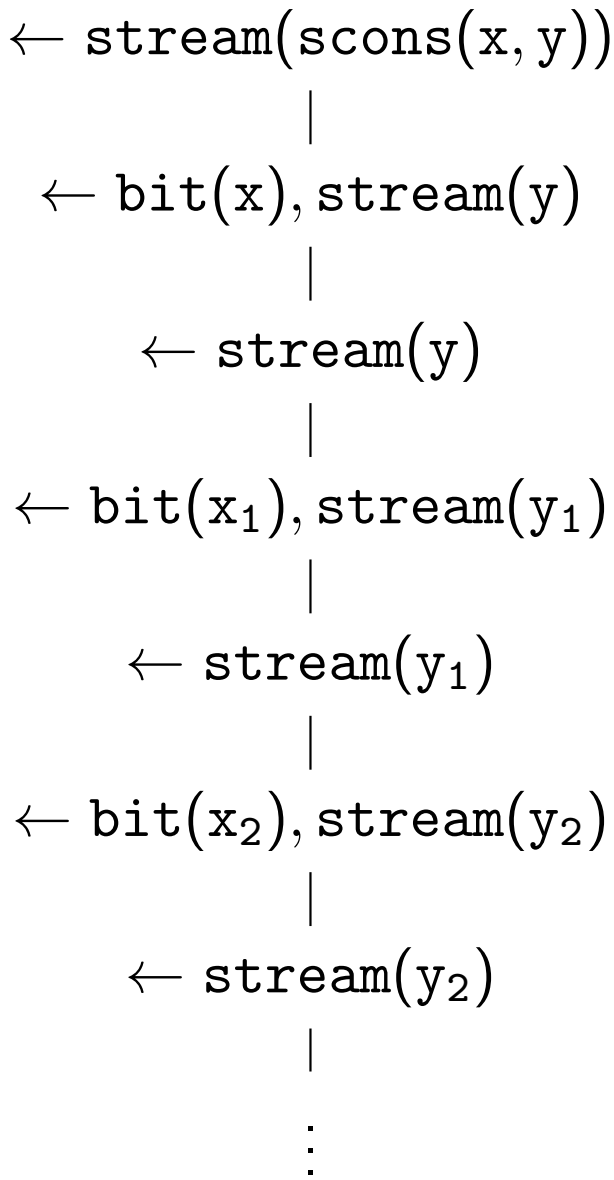
Things go wrong

Example

```
bit(0) ←
bit(1) ←
stream(scons x y) ←

                bit(x), stream(y)
```

No answer, as derivation never terminates.
Semantics may go wrong as well.



Coalgebraic Analysis of derivations in Logic Programs

Given a variable-free logic program P, let At be the set of all atoms appearing in P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(\mathrm{At}))$ sends an atom A to the set of bodies of those clauses in P with head A, each body being viewed as the set of atoms that appear in it.

Coalgebraic Analysis of derivations in Logic Programs

Taking $p : \mathrm{At} \longrightarrow P_f P_f(\mathrm{At})$, the corresponding $C(P_f P_f)$-coalgebra where $C(P_f P_f)$ is the cofree comonad on $P_f P_f$ is given as follows: $C(P_f P_f)(\mathrm{At})$ is given by a limit of the form

$$\ldots \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At} \times P_f P_f(\mathrm{At})) \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At}) \longrightarrow \mathrm{At}.$$

This chain has length ω.
We inductively define the objects $\mathrm{At}_0 = \mathrm{At}$ and $\mathrm{At}_{n+1} = \mathrm{At} \times P_f P_f \mathrm{At}_n$, and the cone

$$\begin{aligned} p_0 &= id : \mathrm{At} \longrightarrow \mathrm{At}(= \mathrm{At}_0) \\ p_{n+1} &= \langle id, P_f P_f(p_n) \circ p \rangle : \mathrm{At} \longrightarrow \mathrm{At} \times P_f P_f \mathrm{At}_n (= \mathrm{At}_{n+1}) \end{aligned}$$

and the limit determines the required coalgebra $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$.

Lawvere theories and the first-order signature Σ

A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these are called *constants*.

Given a signature Σ, construct the Lawvere theory $\mathcal{L}_\Sigma$:

- Define the set $\mathrm{ob}(\mathcal{L}_\Sigma)$ to be the set of natural numbers.
- For each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
- Define $\mathrm{ob}(\mathcal{L}_\Sigma)(n, m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
- Define composition in $\mathcal{L}_\Sigma$ by substitution.

Example of Lawvere theory generated by a LP

Example

The constants `0` and `nil` are modelled by maps from 0 to 1 in $\mathcal{L}_\Sigma$, `s` is modelled by a map from 1 to 1, and `cons` is modelled by a map from 2 to 1. The term `s(0)` is therefore modelled by the map from 0 to 1 given by the composite of the maps modelling `s` and `0`; similarly for the term `s(nil)`, although the latter does not make semantic sense.

We use Lawvere Theory $\mathcal{L}_\Sigma$ intead of set At

Some modifications are needed:

- we need to extend *Set* to *Poset*,
- natural transformations to *lax natural transformations*, and
- replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability).

We use Lawvere Theory $\mathcal{L}_\Sigma$ intead of set At

Some modifications are needed:

- we need to extend *Set* to *Poset*,
- natural transformations to *lax natural transformations*, and
- replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability).

Then $p : At \longrightarrow P_c P_f At$ gives a $Lax(\mathcal{L}_\Sigma^{op}, P_c P_f)$-coalgebra structure on At; and p determines a $Lax(\mathcal{L}_\Sigma^{op}, C(P_c P_f))$-coalgebra structure $\bar{p} : At \longrightarrow C(P_c P_f)(At)$.

Coinductive trees and forests

I will use a convenient structure (and also graphical representation) to illustrate the colagebraic model just defined. And that is of a

Coinductive tree

... and corresponding notion coinductive forest (a set of coinductive trees).

Coinductive trees and forests

I will use a convenient structure (and also graphical representation) to illustrate the colagebraic model just defined. And that is of a

Coinductive tree

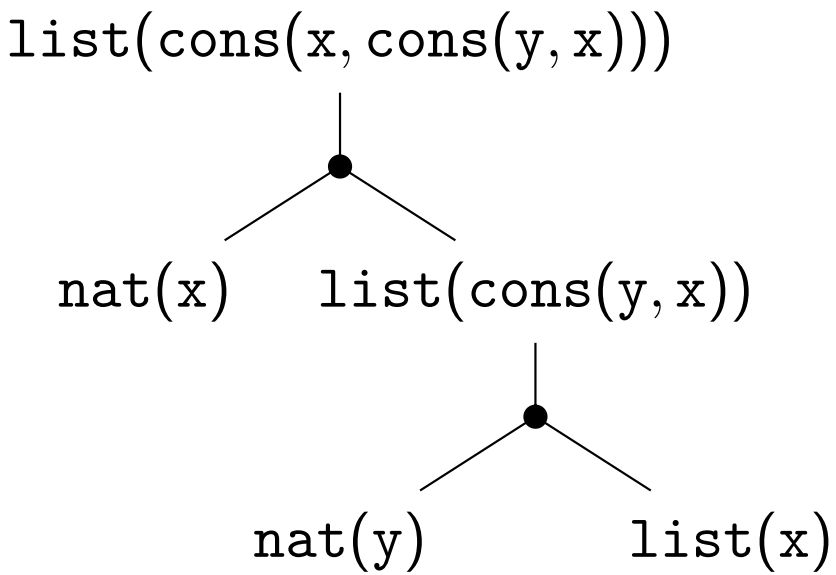
... and corresponding notion coinductive forest (a set of coinductive trees).

We also use these constructions in the proofs of adequacy, soundness, and completeness.

Examples of first-order coinductive trees determined by the semantics:

$A(x, y) \in At(2)$

Then apply At to the map $(s, s) : 1 \to 2$ in $\mathcal{L}_\Sigma$



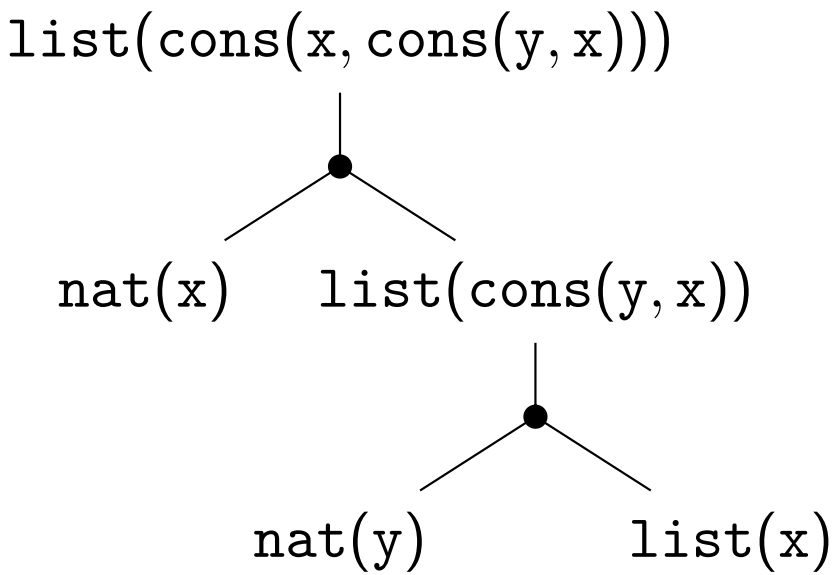
Examples of first-order coinductive trees determined by the semantics:

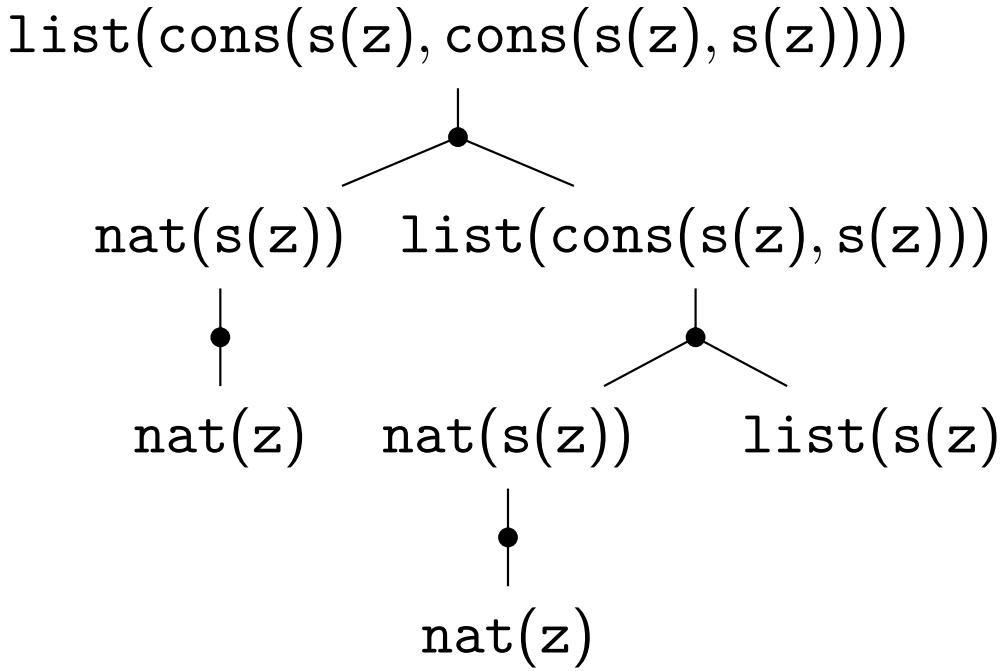
$A(x, y) \in At(2)$

Then apply At to the map $(s, s) : 1 \to 2$ in $\mathcal{L}_\Sigma$

$A(z) \in At(1)$

$At((s, s))(A(x, y))$ is an element of $P_c P_f At(1)$.

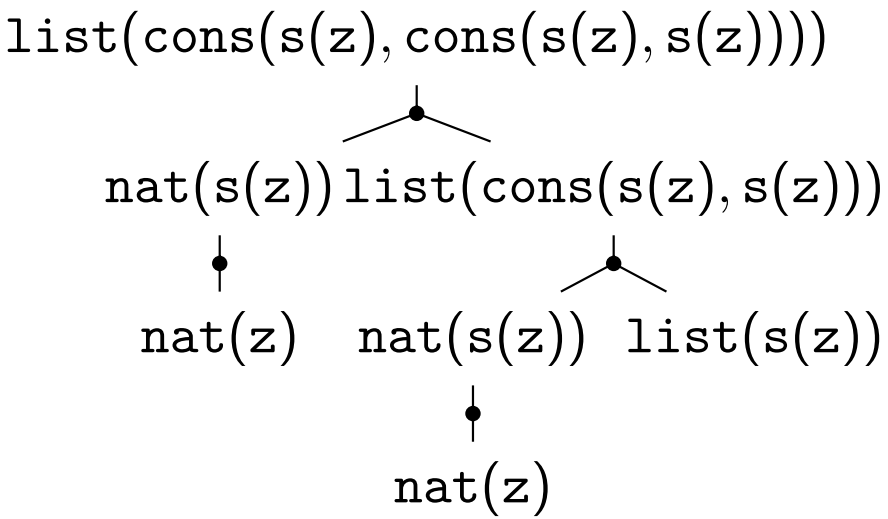




Examples of first-order coinductive trees determined by the semantics:

$A(z) \in At(1)$

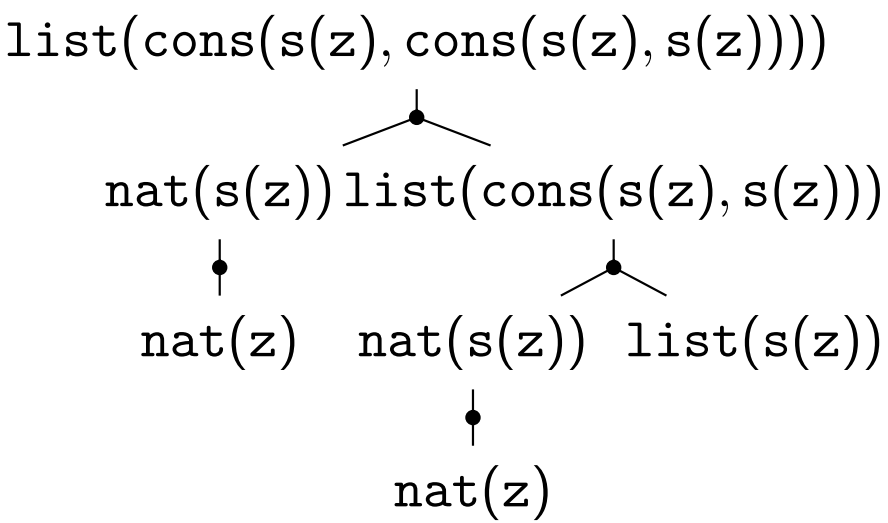
Then apply At to the map $O : 0 \to 1$ in $\mathcal{L}_\Sigma$.



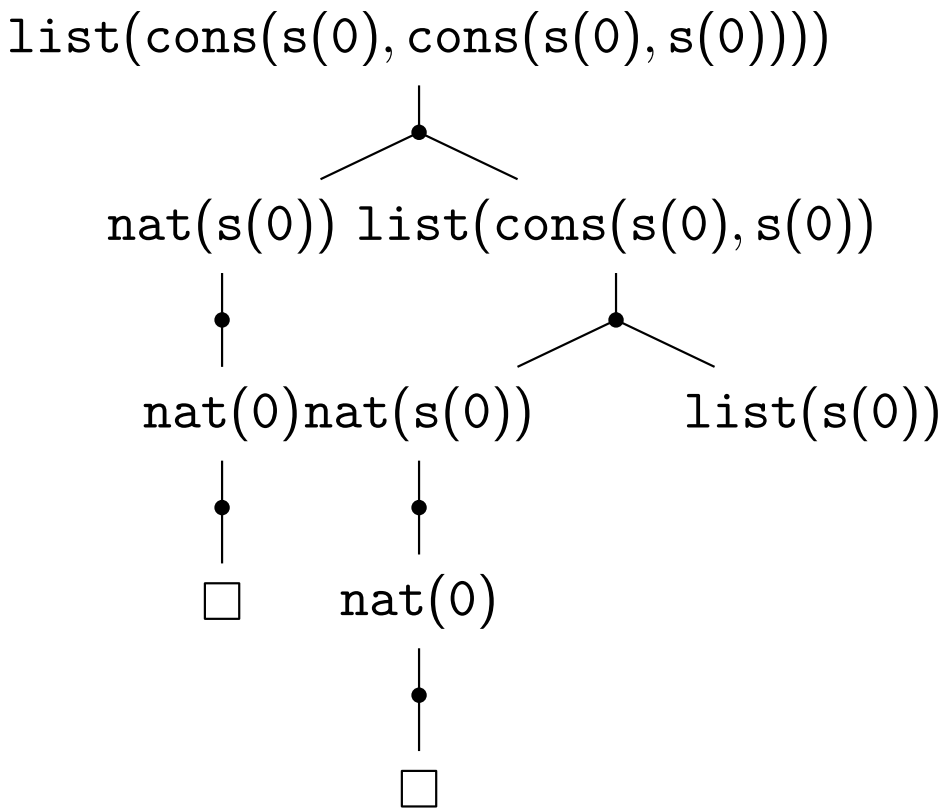
Examples of first-order coinductive trees determined by the semantics:

$A(z) \in At(1)$

Then apply At to the map $O : 0 \to 1$ in $\mathcal{L}_\Sigma$.



$pAt(0)At((s,s))(A(x,y))$



Adequacy

For any logic program P and for any atom A generated by the predicate symbols of P and k distinct variables $x_1, \ldots, x_k$, $\bar{p}(k)(A)$ expresses precisely the same information as that given by a coinductive forest F for the goal A. That is, the following holds:

- $p_n(k)(A)$ is isomorphic to the coinductive forest of depth n and breadth k.
- F has the finite depth n if and only if $\bar{p}(k)(A) = p_n(k)(A)$.
- F has infinite depth if and only if $\bar{p}(k)(A)$ is given by the element of the limit of the infinite chain.

Adequacy

Proof.

For every atomic formula A:

- $p_0(k)(A) = A$
- $p_1(k)(A) = (A, \{\{B^1\theta, \ldots, B^m\theta\}$, such that $B \leftarrow B^1, \ldots, B^m$ is a clause in P with $B\theta = A$ and $B^1\theta, \ldots, B^m\theta$ have variables among $x_1, \ldots, x_k.\})$
- $p_2(k)(A) = (A, \{\{(B^1\theta, \{\{C_1^1\theta_1\theta, \ldots, C_1^{m_1}\theta_1\theta\}$ such that $C \leftarrow C_1^1, \ldots, C_1^{m_1}$ is a clause in P with $C\theta_1 = B^1\}), \ldots$ and $C_1^1\theta_1\theta, \ldots, C_1^{m_1}\theta_1\theta$ have variables among $x_1, \ldots, x_k.\}\})$

The limit of the sequence is precisely (the extension of) the structure described by Proposition **??**. For each atomic formula A, $p_0(k)(A)$ corresponds to the root of a coinductive derivation tree, and, more generally, each $p_n(k)(A)$ corresponds to the coinductive forest of breadth k, as far as depth n. □

Soundness and completeness of SLD-resolution relative to coinductive derivation trees.

Let P be a logic program, and G be a goal.

1. Soundness. If there is an SLD-refutation for G in P with computed answer θ, then there exists a coinductive derivation tree for $G\theta$ that contains a success subtree.
2. Completeness. If a coinductive derivation tree for $G\theta$ contains a success subtree, then there exists an SLD-refutation for G in P, with computed answer λ such that there exists substitution σ such that $\lambda = \sigma\theta$.

Soundness and completeness of SLD-resolution relative to coinductive derivation trees.

Let P be a logic program, and G be a goal.

1. Soundness. If there is an SLD-refutation for G in P with computed answer θ, then there exists a coinductive derivation tree for $G\theta$ that contains a success subtree.
2. Completeness. If a coinductive derivation tree for $G\theta$ contains a success subtree, then there exists an SLD-refutation for G in P, with computed answer λ such that there exists substitution σ such that $\lambda = \sigma\theta$.

Corollary

Given a logic program P, SLD-refutations in P are sound and complete with respect to the $Lax(\mathcal{L}_\Sigma^{op}, P_c P_f)$-coalgebra determined by P.

The Theory of Observables, Observational equivalence

One of the main purposes of giving a semantics to logic programs is its ability to observe equal behaviors of logic programs and distinguish logic programs with different computational behavior. Therefore, the choice of observables and semantic models is closely related to the choice of equivalence relation defined over logic programs.

Definition

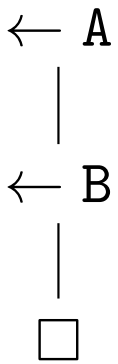
Let P_1 and P_2 be ground logic programs. Then we define $P_1 \approx P_2$ if and only if, for any goal G, the following four conditions hold:

1. G has a refutation in P_1 if and only if G has a refutation in P_2;
2. G has the same set of computed answers in P_1 and P_2.
3. G has the same set of (correct) partial answers in P_1 and P_2.
4. G has the same set of call patterns in P_1 and P_2.

Example of different behavior of model-theoretically "equal" programs

Example

$$A \leftarrow B$$
$$B \leftarrow$$
$$B \leftarrow B$$



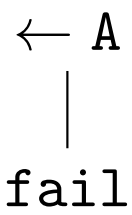
Example

$$A \leftarrow B$$
$$B \leftarrow B$$
$$B \leftarrow$$

Example of different behavior of model-theoretically "equal" programs

Example

$$A \leftarrow false, B$$
$$B \leftarrow B$$
$$B \leftarrow$$



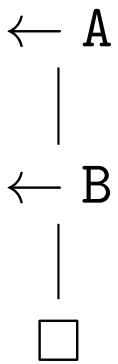
Example

$$A \leftarrow B, false$$
$$B \leftarrow B$$
$$B \leftarrow$$

Example of different programs with identical behavior

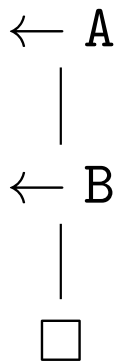
Example

$$A \leftarrow B, \mathit{false}, C, D$$
$$B \leftarrow$$



Example

$$A \leftarrow B, \mathit{false}$$
$$B \leftarrow$$



Correctness of coalgebraic semantics relative to observational semantics for sequential programs

Theorem

For logic programs P_1 and P_2, if coinductive tree for P_1 is equal to the coinductive tree for P_2, then $P_1 \approx P_2$.

Full abstraction result?

The converse of the Theorem does not hold. That is, there can be observationally equivalent programs that have different and-or parallel trees.

Example

Consider two logic programs, P_1 and P_2, whose clauses are exactly the same, with the exception of one clause: P_1 contains
$A \leftarrow B_1, \ldots, B_i, \texttt{false}, \ldots, B_n$; and P_2 contains the clause
$A \leftarrow B_1, \ldots B_i, \texttt{false}$ instead.

Concurrent programming

The failure of full abstraction result signposts a more fundamental mismatch between the coalgebraic (comonadic) semantics (akin process calculi, and concurrent processes), and traditional sequential method of SLD-resolution in logic programs.

Concurrent programming

The failure of full abstraction result signposts a more fundamental mismatch between the coalgebraic (comonadic) semantics (akin process calculi, and concurrent processes), and traditional sequential method of SLD-resolution in logic programs.

Solution?

Concurrent programming

The failure of full abstraction result signposts a more fundamental mismatch between the coalgebraic (comonadic) semantics (akin process calculi, and concurrent processes), and traditional sequential method of SLD-resolution in logic programs.

Solution?

... Derivations by coinductive trees instead of SLD-resolution.

Coinductive derivation for the goal `stream(x)`

$$\texttt{stream(x)} \xrightarrow{\theta_1}$$

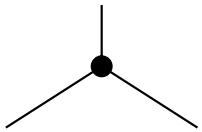
Coinductive derivation for the goal `stream(x)`

$\xrightarrow{\theta_1}$

`stream(x)`

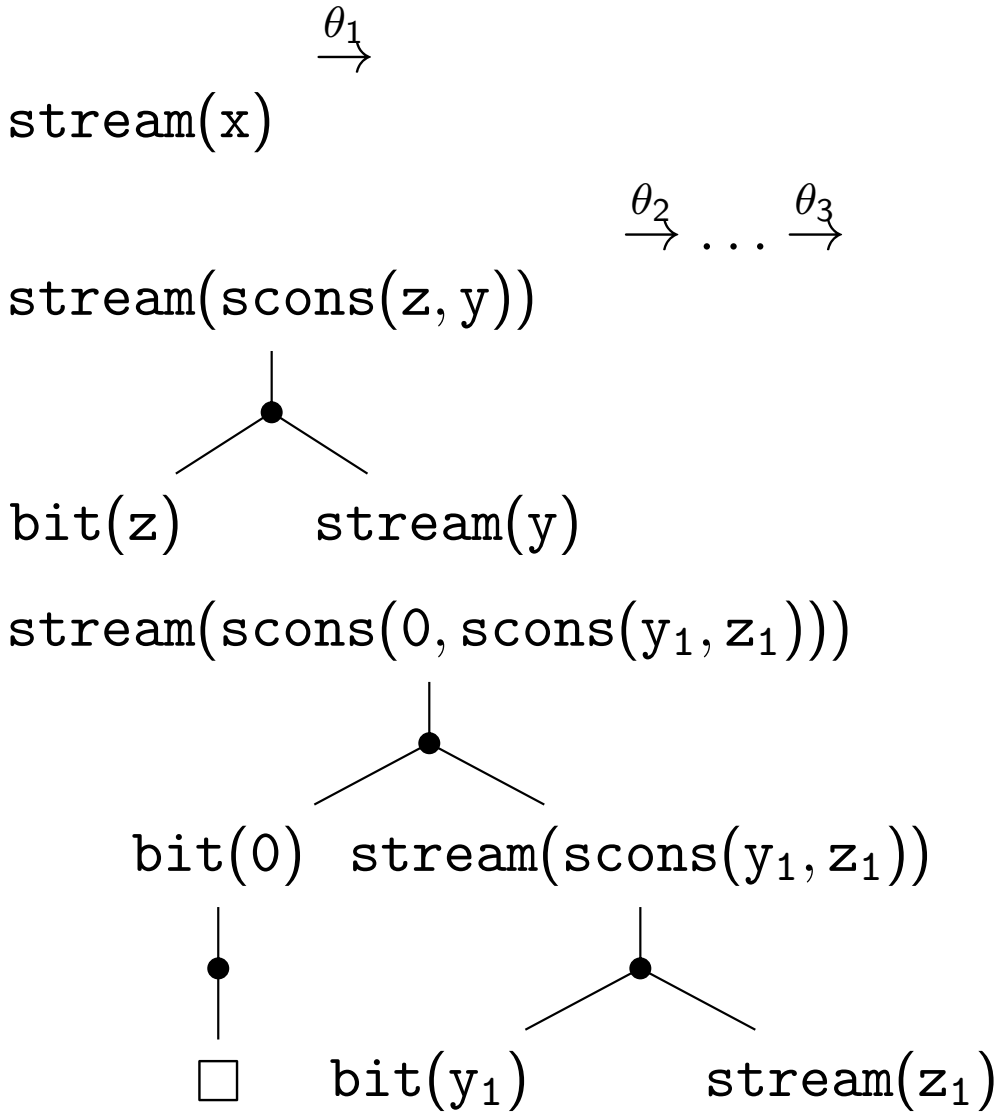
$\xrightarrow{\theta_2} \ldots \xrightarrow{\theta_3}$

`stream(scons(z, y))`



`bit(z)` `stream(y)`

Coinductive derivation for the goal `stream(x)`



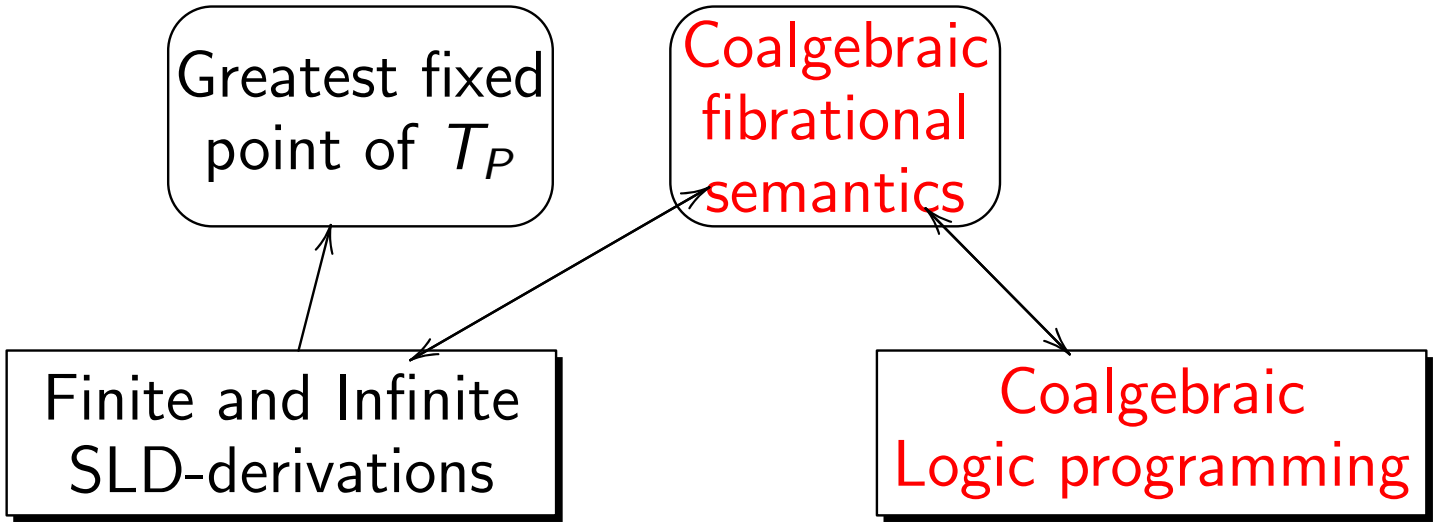
Answers for x: $cons(z, y)$ and $cons(0, cons(y_1, z_1))$. It's a different (corecursive) approach to what a "terminating derivation" is.

Main theorems:

Coinductive derivations yeild the following results:

- Soundness and completeness relative to the Coalgebraic semantics
- Correcteness and full abstraction result relative to the theory of observables.

Conclusions



Important note: all these results were related to the theory of observables and tested for observational equivalence.

Applications:

- Concurrent programming
- Automated Proofs for propositions/statements about infinite structures (e.g.streams)
- Type inference (recursive/corecursive types)
- Cyclic proofs
- Statistical analysis of automated proofs in Machine learning.
- More?..

Thank you!

Questions?