Coalgebraic Logic Programming

Katya Komendantskaya

School of Computing, University of Dundee, UK

08 October 2014

Outline

Outline

1. Background: Horn-Clause Logic
2. Recursion and Corecursion in LP and beyond

Outline

1. Background: Horn-Clause Logic
2. Recursion and Corecursion in LP and beyond
3. Coalgebraic Logic Programming

Syntax of Horn-clause Logic

First-order signature Σ

- constants;
- function symbols;
- predicates;

- variables;
- connectives $\wedge, \vee, \neg$;
- quantifiers $\forall, \exists$

Standard definition of first-order term and formula. Atom is a formula containing no connectives or quantifiers; literal is an atom or a negation of an atom.

A clause is a formula $\forall x_1, \ldots x_n (L_1 \vee \ldots \vee L_m)$,

where each L is a literal; and $x_1, \ldots x_n$ – are all variables occurring in $L_1, \ldots L_m$.

Syntax of Horn-clause Logic

Notation for clauses

$\forall x_1, \ldots x_n (A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_k)$
is denoted by
$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_k$

Horn Clauses

- a definite clause $A \leftarrow B_1, \ldots, B_k$ or
- a goal $\leftarrow B_1, \ldots, B_k$

A (definite) logic program is a finite set of definite clauses

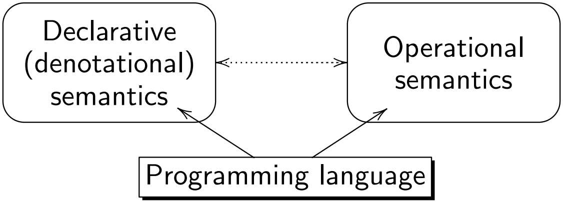
... Gives us a Turing-complete programming language.

Example: lists of natural numbers

Example

$$
\begin{aligned}
\texttt{nat(0)} &\leftarrow \\
\texttt{nat(s(x))} &\leftarrow \texttt{nat(x)} \\
\texttt{list(nil)} &\leftarrow \\
\texttt{list(cons(x,y))} &\leftarrow \texttt{nat(x), list(y)}
\end{aligned}
$$

The Semantics game:



Herbrand models for Logic Programs

[70s-80s: Apt, van Emden, Kowalski]

Herbrand Universe and Herbrand Base for a program P

- U_P – the set of all ground terms formed from Σ
- B_P – the set of all ground atoms formed from Σ

Herbrand interpretation for P

- Domain of interpretation is U_P
- Constants in P are assigned themselves in U_P
- If $f^n \in P$, it is assigned the mapping $(U_P)^n \to U_P$ defined by $(t_1, \ldots, t_n) \to f(t_1, \ldots, t_n)$.
- If $Q^n \in P$, it is assigned a mapping $(U_P)^n \to \{true, false\}$.

Herbrand interpretation is often identified with a subset of the Herbrand base – the set of all ground atoms that are true under the interpretation.

Herbrand Models

Herbrand model for P is an Herbrand interpretation for atoms over Σ_P which is a model for P.

The model intersection property

given a set of non-empty Herbrand models $\{M_i\}$ for P, $\bigcap_i M_i$ is a model for P, also known as the *least Herbrand model* of P, denoted by M_P.

The set of all Herbrand interpretations for a program P forms a complete lattice under the partial order of set inclusion; the top element of this lattice is B_P and the bottom element is $\emptyset$.

Fixed point semantics

If HI is a Herbrand interpretation for P, define
$T_P(HI) = \{t \in B_P \mid$
$t \leftarrow t_1, \ldots, t_n$ is a ground instance of a clause in P and
$t_1, \ldots, t_n \subseteq HI\}$.

Some properties; making use of Knaster-Tarski and Kleene theorems

T_P is monotonic and continuous
$M_P = lfp(T_P) = T_P \uparrow \omega$
$gfp(T_P) = T_P \downarrow \alpha$, α may be greater than ω

With standard definitions:

$T_P \uparrow 0 = \bot$ $T_P \uparrow \alpha = T_P(T_P \uparrow (\alpha - 1))$	$T \downarrow 0 = \top$ $T_p \downarrow \alpha = T_P(T_P \downarrow (\alpha - 1))$	α is a successor ordinal
$T_P \uparrow \alpha = lub\{T_P \uparrow \beta \mid \beta < \alpha\}$	$T_P \downarrow \alpha = glb\{T_P \downarrow \beta \mid \beta < \alpha\}$	α is a limit ordinal

Example:

For **NatList**:

$T_P \uparrow \omega =$

$\{nat(0), nat(s(0)), nat(s(s(0))), \ldots)$

$list(cons(0, nil)), list(cons(s(0), nil)), list(cons(s(s(0)), nil)), \ldots$

$list(cons(0, cons(0, nil))), list(cons(s(0), cons(0, nil))),$

$list(cons(s(s(0)), cons(0, nil))), \ldots$

$\vdots$

$\}$

“Operational Semantics” (?)

SLD resolution + Unification

SLD-resolution + unification in LP derivations.

Program **NatList**:

Example

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

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

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

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

$\texttt{list(y)}$

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

SLD-resolution + unification in LP derivations.

Example

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

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

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

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

$\texttt{list(y)}$

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

$$|$$

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

SLD-resolution (+ unification) in LP derivations.

Example

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

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

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

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

$\mathtt{list(y)}$

$$\begin{array}{c} \leftarrow \mathtt{list}(\mathrm{cons}(\mathrm{x},\mathrm{y})) \\ | \\ \leftarrow \mathtt{nat}(\mathrm{x}), \mathtt{list}(\mathrm{y}) \\ | \\ \leftarrow \mathtt{list}(\mathrm{y}) \end{array}$$

SLD-resolution (+ unification) in LP derivations.

Example

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

$$\leftarrow \mathtt{list(cons(x,y))}$$
$$|$$
$$\leftarrow \mathtt{nat(x), list(y)}$$
$$|$$
$$\leftarrow \mathtt{list(y)}$$
$$|$$
$$\leftarrow \Box$$

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.

Soundness and Completeness

[70s-80s: Apt, van Emden, Kowalski]

Theorem (Soundness and Completeness of Derivations)

Soundness. *Given a logic program P, and an atom A, if there is a refutation for P and $\leftarrow A$, then there is a grounding substitution θ, such that $\theta(A) \in M_P$.*
Completeness. *Given a logic program P, and an atom $A \in M_P$, there is a refutation for A.*

Outline

Corecursion in LP?

Program **Stream**:

Example

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

Corecursion in LP?

Program **Stream**:

Example

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

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

No answer, as derivation never terminates.

Corecursion in LP?

Program **Stream**:

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: *gfp*(T_P) will contain an infinite term corresponding to stream: so completeness fails.

$$\leftarrow \mathtt{stream(scons(x,y))}$$
$$|$$
$$\leftarrow \mathtt{bit(x)}, \mathtt{stream(y)}$$
$$|$$
$$\leftarrow \mathtt{stream(y)}$$
$$|$$
$$\leftarrow \mathtt{bit(x_1)}, \mathtt{stream(y_1)}$$
$$|$$
$$\leftarrow \mathtt{stream(y_1)}$$
$$|$$
$$\leftarrow \mathtt{bit(x_2)}, \mathtt{stream(y_2)}$$
$$|$$
$$\leftarrow \mathtt{stream(y_2)}$$
$$|$$
$$\vdots$$

A quick look at Greatest Fixed point semantics

[80s: Abdallah, van Emden, Lloyd]

- Extend U'_P to contain infinite terms;
- Extend B'_P to contain infinite atoms;
- Amend T'_P and M'_P accordingly.

$$\boxed{\begin{array}{l} M'_P = lfp(T'_P) = T'_P \uparrow \omega \\ gfp(T'_P) = T'_P \downarrow \omega \end{array}}$$

Instead of SLD-resolution: – "computation of an infinite atom at infinity".
Soundness, but not completeness...

Computation at infinity...

Program **BitList**:

Example

```
bit(0) ←
bit(1) ←
bitlist(scons(x, y)) ←

     bitlist(x), bitlist(y)

list(nil) ←
```

$$\leftarrow \mathtt{bitlist}(\mathtt{scons}(x, y))$$
$$|$$
$$\leftarrow \mathtt{bit}(x), \mathtt{bitlist}(y)$$
$$|$$
$$\leftarrow \mathtt{bitlist}(y)$$
$$|$$
$$\leftarrow \mathtt{bit}(x_1), \mathtt{bitlist}(y_1)$$
$$|$$
$$\leftarrow \mathtt{bitlist}(y_1)$$
$$|$$
$$\leftarrow \mathtt{bit}(x_2), \mathtt{bitlist}(y_2)$$
$$|$$
$$\leftarrow \mathtt{bitlist}(y_2)$$
$$|$$
$$\vdots$$

At infinity, converges to `bitlist(scons(0,scons(0,scons(0, ... )`

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **BitList**...

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **BitList**...
- Or may be it is a recursive program with coinductive interpretation? (again, **BitList**)

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **BitList**...
- Or may be it is a recursive program with coinductive interpretation? (again, **BitList**)
- Or may be it is just some bad loop without particular computational meaning:

$$badstream(scons(x, y)) \leftarrow badstream(scons(x, y))$$

Problems...

This "operational semantics" does not give us any formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **BitList**...
- Or may be it is a recursive program with coinductive interpretation? (again, **BitList**)
- Or may be it is just some bad loop without particular computational meaning:

$$badstream(scons(x, y)) \leftarrow badstream(scons(x, y))$$

What kind of semantic support for (co)recursion is possible?

Looking for Inspiration

Lets take a (hopefully useful) detour into typeful functional languages

Inductive Types and Recursive Functions

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

Recursive functions have arguments of inductive types.

```
Fixpoint length (A:Type) (l: list A) : nat :=
match l with
 | nil => O
 | cons _ l' => S (length l')
end.
```

Termination

Universal Termination

A recursive function is terminating, if it terminates for all possible (legal) inputs.

"Easy" to reason about, as legal input is defined by constructors; checking for structural recursion is one elegant way to decide termination.

```
Fixpoint length (A:Type) (l:  list A) : nat :=
match l with
 | nil => O
 | cons _ l' => S (length l')
end.
```

Coinductive Types and Corecursive Functions

```
CoInductive stream (A:Set) :  Set :=
SCons:  A -> stream A -> stream A.
```

Corecursive functions have outputs of coinductive types. (Type of input arguments is not important.)

```
CoFixpoint map (s:Stream A) : Stream B :=
SCons (f (hd s)) (map (tl s)).
```

Productivity

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

Productivity

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

The element of the stream at position n can be found by:

Definition

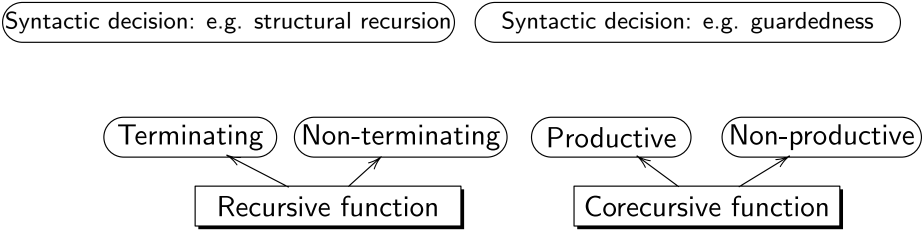
$$\begin{cases} \texttt{nth}\ 0\ (\texttt{SCons a tl}) = \texttt{a} \\ \texttt{nth}\ (\texttt{S n})\ (\texttt{SCons a tl}) = \texttt{nth n tl} \end{cases}$$

A given stream s is productive if we can be sure that the computation of `nth n s` is guaranteed to terminate, whatever the value of `n` is.

We call a function *productive*, if, for any given input, it outputs a productive value.

```
CoFixpoint map (s:Stream A) : Stream B :=
SCons (f (hd s)) (map (tl s)).
```

To notice:



- The role of inductive and coinductive types in definition of recursive and corecursive functions
- The role of constructors and (co)-pattern matching

Termination in LP

- no types to make distinction between types and functions, recursion and corecursion, “constructors” and “just function symbols”;
- Searching strategies/clause order impact termination;
- No general agreement in the literature on what “termination” is;

Termination in LP

- no types to make distinction between types and functions, recursion and corecursion, "constructors" and "just function symbols";
- Searching strategies/clause order impact termination;
- No general agreement in the literature on what "termination" is;
- Termination is often an existential property

Example

The query `bitstream(scons(0,1))?` will terminate, whereas `bitstream(scons(0,y))?` – would not.

Termination in LP

- no types to make distinction between types and functions, recursion and corecursion, "constructors" and "just function symbols";
- Searching strategies/clause order impact termination;
- No general agreement in the literature on what "termination" is;
- Termination is often an existential property

Example

The query `bitstream(scons(0,1))?` will terminate, whereas `bitstream(scons(0,y))?` – would not.

- "Input"/"output" of "functions" are not defined in advance: "lack of directionality" impacts termination

Example

$\texttt{add(0,y,y)} \leftarrow$

$\texttt{add(s(x),y,s(z))} \leftarrow \texttt{add(x,y,z)}$

Recursion and Corecursion in Logic Programming

1. in the 70s-80s: Apt, van Emden, Kowalski: study of recursion and least Herbrand model semantics of LP.

Recursion and Corecursion in Logic Programming

1. in the 70s-80s: Apt, van Emden, Kowalski: study of recursion and least Herbrand model semantics of LP.
2. in the 80s: Abdallah, van Emden, Lloyd: "perpetual" computations in LP and the greatest fixed point semantics of LP: incomplete, no finite procedure for computations given.

Recursion and Corecursion in Logic Programming

1. in the 70s-80s: Apt, van Emden, Kowalski: study of recursion and least Herbrand model semantics of LP.
2. in the 80s: Abdallah, van Emden, Lloyd: "perpetual" computations in LP and the greatest fixed point semantics of LP: incomplete, no finite procedure for computations given.
3. Many papers on problem-specific/existential termination analysis ...

No general coherent notion of termination/productivity matching that of FP!

Recursion and Corecursion in Logic Programming

1. in the 70s-80s: Apt, van Emden, Kowalski: study of recursion and least Herbrand model semantics of LP.
2. in the 80s: Abdallah, van Emden, Lloyd: "perpetual" computations in LP and the greatest fixed point semantics of LP: incomplete, no finite procedure for computations given.
3. Many papers on problem-specific/existential termination analysis ...

No general coherent notion of termination/productivity matching that of FP!

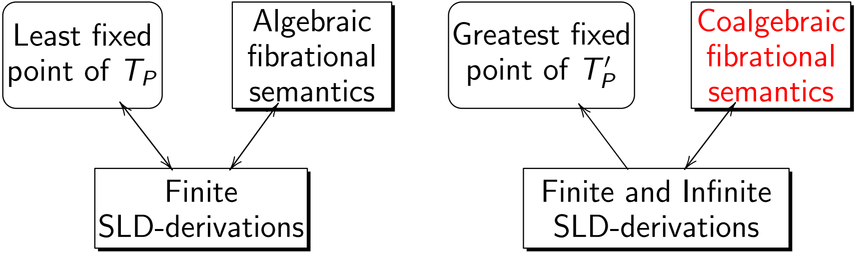
May be we need a new operational semantics?

4. Our work, from 2010-14, – coalgebraic semantics for LP – a solution?

Outline

Coalgebraic Logic programming...

An independent discovery



Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 1: Logic programs as coalgebras

Definition

For a functor F, a *coalgebra* is a pair (U, c) consisting of a set U and a function $c : U \to F(U)$.

1. Let At be the set of all atoms appearing in a program P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(At))$ sends an atom A to the set of bodies of those clauses in P with head A.

Example

$T \leftarrow Q, R$
$T \leftarrow S$
$p(T) = \{\{Q, R\}, \{S\}\}$

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 2: Derivations as Comonads

In general, if $U : H\text{-}coalg \longrightarrow C$ has a right adjoint G, the composite functor $UG : C \longrightarrow C$ possesses the canonical structure of a *comonad* $C(H)$, called the *cofree* comonad on H. One can form a *coalgebra* for a comonad $C(H)$.

- Taking $p : At \longrightarrow P_f P_f(At)$, 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: $C(P_f P_f)(At)$ is given by a limit of the form

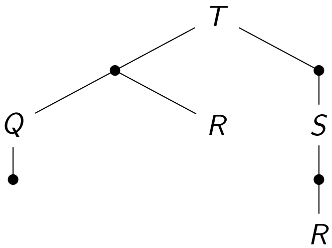
  $$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

  This gives a "tree-like" structure: we call them $\& V$-trees.

Example

Example

$T \leftarrow Q, R$
$T \leftarrow S$
$Q \leftarrow$
$S \leftarrow R$



This models and-or parallel trees known in LP [AMAST 2010]

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

Definition

A *Lawvere theory* consists of a small category L with strictly associative finite products, and a strict finite-product preserving identity-on-objects functor $I : \mathbb{N}^{op} \to L$.

- Take *Lawvere Theory* $\mathcal{L}_\Sigma$ to model the terms over Σ
  - $\mathtt{ob}(\mathcal{L}_\Sigma)$ is $\mathbb{N}$.
  - For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
  - $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
  - composition in $\mathcal{L}_\Sigma$ is first-order substitution.

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

Definition

A *Lawvere theory* consists of a small category L with strictly associative finite products, and a strict finite-product preserving identity-on-objects functor $I : \mathbb{N}^{op} \to L$.

- Take *Lawvere Theory* $\mathcal{L}_\Sigma$ to model the terms over Σ
  - $\mathtt{ob}(\mathcal{L}_\Sigma)$ is $\mathbb{N}$.
  - For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
  - $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
  - composition in $\mathcal{L}_\Sigma$ is first-order substitution.
- take the functor $At : \mathcal{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ and n variables.

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

Definition

A *Lawvere theory* consists of a small category L with strictly associative finite products, and a strict finite-product preserving identity-on-objects functor $I : \mathbb{N}^{op} \to L$.

- Take *Lawvere Theory* $\mathcal{L}_\Sigma$ to model the terms over Σ
  - $\mathtt{ob}(\mathcal{L}_\Sigma)$ is $\mathbb{N}$.
  - For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
  - $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
  - composition in $\mathcal{L}_\Sigma$ is first-order substitution.
- take the functor $At : \mathcal{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ and n variables.
- model a program P by the $[\mathcal{L}_\Sigma^{op}, P_f P_f]$-coalgebra $p : At \longrightarrow P_f P_f At$ on the category $[\mathcal{L}_\Sigma^{op}, Set]$.

Examples

Program **Stream**: "fibers" given by term arities. Take the fiber of 1.
$\&V$-trees:

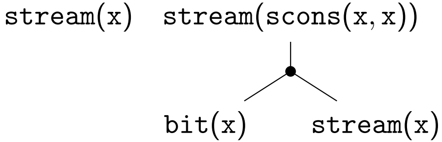
Examples

Program **Stream**: "fibers" given by term arities. Take the fiber of 1.
&V-trees:

$\mathtt{stream(x)}$

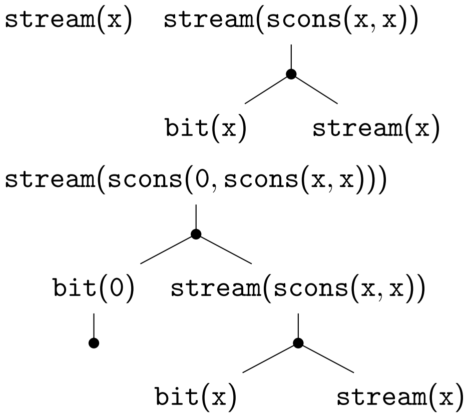
Examples

Program **Stream**: "fibers" given by term arities. Take the fiber of 1.
& V-trees:



Examples

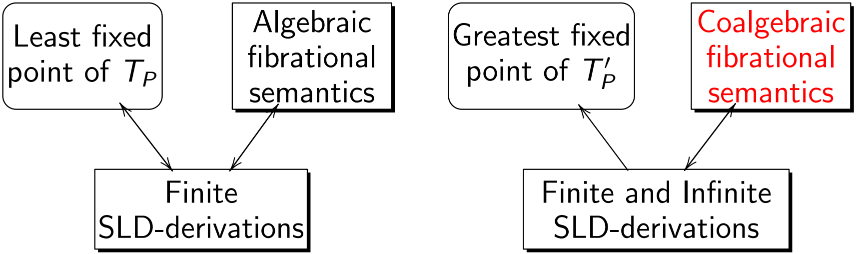
Program **Stream**: "fibers" given by term arities. Take the fiber of 1.
&V-trees:



Coalgebraic Logic programming...

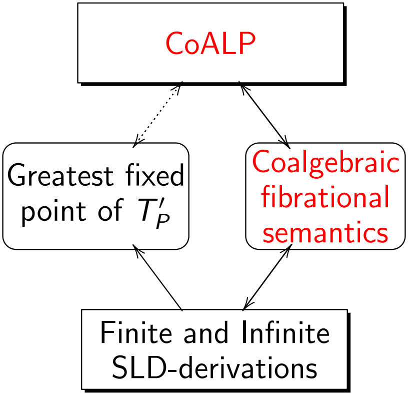
[CSL 2011, JLC 2014]

There is more structure in this fibrational coalgebraic semantics than in SLD-resolution!!!



Coalgebraic Logic programming...

[CSL 2011, JLC 2014]



Computationally essential:

1. for coinductive **Stream**, the $\& V$-trees are finite!!! – both in depth and in breadth;
2. each tree gives only a partial computation – it is not like eager SLD-trees we have seen earlier;
3. the effect of fibers is best modelled by restricting unification to term-matching (note resemblance to the pattern-matching in Functional setting).

Computationally essential:

1. for coinductive **Stream**, the $\& V$-trees are finite!!! – both in depth and in breadth;
2. each tree gives only a partial computation – it is not like eager SLD-trees we have seen earlier;
3. the effect of fibers is best modelled by restricting unification to term-matching (note resemblance to the pattern-matching in Functional setting).

1. $\Rightarrow$ gives hope for a formalism to describe termination and productivity
2. $\Rightarrow$ hints there may be different tiers of computation...

Lazy Corecursion in CoALP: Coinductive trees

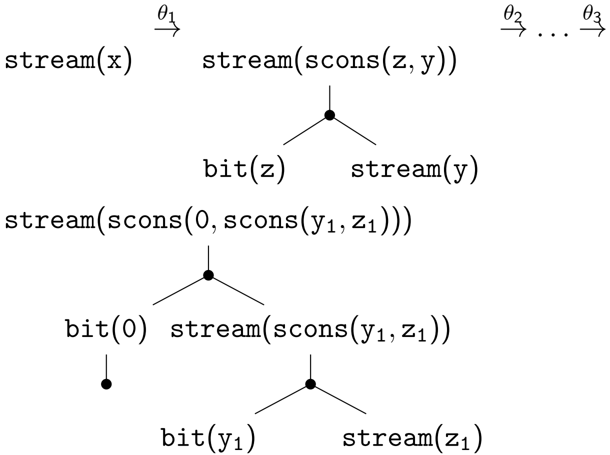
$$\texttt{stream(x)} \xrightarrow{\theta_1}$$

Lazy Corecursion in CoALP: Coinductive trees

$$\texttt{stream(x)} \xrightarrow{\theta_1} \texttt{stream(scons(z,y))} \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_3}$$

$$\texttt{bit(z)} \qquad \texttt{stream(y)}$$

Lazy Corecursion in CoALP



The above would correspond to one-branch of SLD-derivations we have seen! The main driving force: separation of layers of computations into different Tiers.

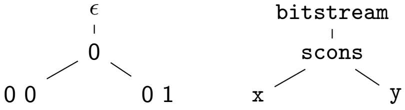
CoALP: the three-tier calculus of trees

Inspired by Fibrational Coalgebraic Semantics, a new three-tier Calculus of Horn-Clause Logic

1. Tier 1: term-trees;
2. Tier 2: coinductive trees;
3. Tier 3: derivation trees.

Tier-1: Term-trees

Take a "tree-language" $\mathbb{N}^*$ – a set of all finite words of $\mathbb{N}$.
Given an $L \in \mathbb{N}^*$, a term tree is a map $L \to \Sigma$, satisfying term arity restrictions.
Example:



Operation: – first-order substitution
Calculus: – first-order unification.

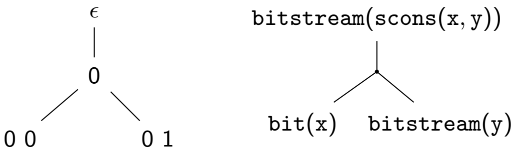
Notation:

Term(Σ)	*finite* term trees over Σ
Term$^\infty(\Sigma)$	*infinite* term trees over Σ
Term$^\omega(\Sigma)$	*finite and infinite* term trees over Σ

Tier-2: Coinductive trees

Given an $L \in \mathbb{N}^*$, a coinductive tree is a map $L \rightarrow \mathbf{Term}(\Sigma_P)$, with &$\vee$-tree structure.

Example:



Operation: – coinductive tree substitution
Calculus: – coinductive derivations.

Notation:

$\mathbf{CTree}(\mathbf{Term}(\Sigma_P))$	all *finite* coinductive trees over $\mathbf{Term}(\Sigma_P)$
$\mathbf{CTree}^{\infty}(\mathbf{Term}(\Sigma_P))$	all *infinite* coinductive trees over $\mathbf{Term}(\Sigma_P)$
$\mathbf{CTree}^{\omega}(\mathbf{Term}(\Sigma_P))$	all *finite and infinite* coinductive trees over $\mathbf{Term}(\Sigma_P)$

Tier-3: Derivation trees

Given an $L \in \mathbb{N}^*$, a coinductive derivation is a map $L \to \mathbf{CTree}(\mathbf{Term}(\Sigma_P))$.

$\xrightarrow{\sigma_\epsilon}$ ϵ `bitstream(x)` $\xrightarrow{\sigma_0}$ 0 `bitstream([x1|y])` $\overset{\sigma_{01}}{\downarrow}$ $\xrightarrow{\sigma_{00}}$ 00 `bitstream([0|y])`

`bit(x1)` `bitstream(y)`

`bit(0)` `bitstream(y)`

01 `bitstream([x1|[x2|y1]])`

`bit(x1)` `bitstream([x2|y1])`

`bit(x2)` `bitstream(y1)`

Tier-3 notation

$\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$	all *finite* coinductive derivations over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$
$\mathbf{CDer}^{\infty}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$	all *infinite* coinductive trees over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$
$\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$	all *finite and infinite* coinductive trees over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\mathbf{Term}(\Sigma_P))$.

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\mathbf{Term}(\Sigma_P))$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$,
  E.g. **bit**.

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\mathbf{Term}(\Sigma_P))$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$,
  E.g. **bit**.
- inductive LPs all derivations for which are in $\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$;
  E.g. **NatList**.

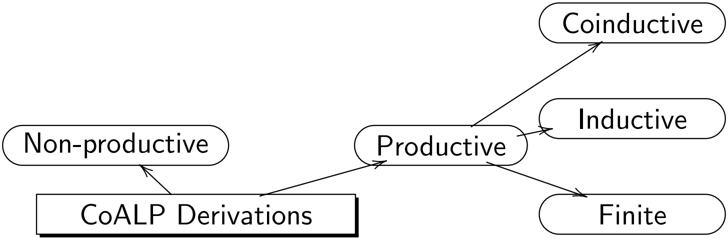
Theory of Productivity for LP

A first-order logic program P is *productive* if

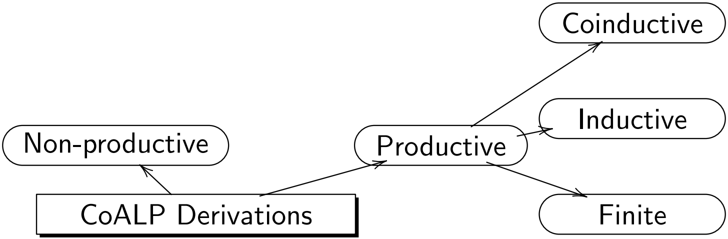
for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\mathbf{Term}(\Sigma_P))$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$,
  E.g. **bit**.
- inductive LPs all derivations for which are in
  $\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P)))$;
  E.g. **NatList**.
- coinductive LPs all derivations for which are in
  $\mathbf{CDer}^{\infty}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P))))$
  E.g. **Stream**.

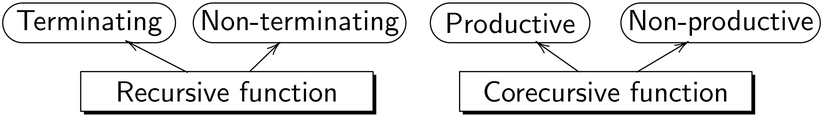
Theory of Productivity in LP



Theory of Productivity in LP



Compare with Typed FP:



Conclusions

1. We have seen (a history of development of) declarative and operational semantics of LP;
2. Understanding of recursion/corecursion, termination/productivity is the key issue for operational semantics;
3. Fibrational algebraic/coalgebraic semantics is a convenient way to give an operational semantics to LP;
4. It gave rise to a new, better structured, 3-Tier Calculus for Horn Clause Logic (= CoALP);
5. It allowed to formulate a coherent theory of terminaion and productivity for LP;
6. ...made possible more precision in soundness and completeness results.

Current and future work

1. Verify Guardedness conditions

Current and future work

1. Verify Guardedness conditions
2. Soundness and completeness of the 3-Tier calculus of CoALP relative to the $gfp(T'_P)$.

Current and future work

1. Verify Guardedness conditions
2. Soundness and completeness of the 3-Tier calculus of CoALP relative to the $gfp(T'_P)$.
3. Extensions, implementation, applications: CoALP for type inference in functional languages
4. Relation to Type Theory; e.g. Session Types.

... join us, there is a lot more to it!

Thank you!

Download your copy of CoALP today:

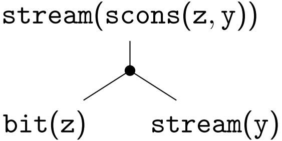
CoALP webpage: http://staff.computing.dundee.ac.uk/katya/CoALP/

CoALP authors and contributors:

- John Power
- Martin Schmidt
- Jonathan Heras
- Vladimir Komendantskiy
- Patty Johann
- Andrew Pond

Deciding Productivity: Guardedness

- Tier 1. Measures of reduction on term trees:
  `stream(y)` is a reduction of `stream(scons(x,y))`
- Tier 2. Reduction on coinductive tree loops:



- Tier 3. Discovery of derivation loops.

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Execution	Eager	Eager	Eager	Lazy
Corecursion				
Mode of execution				
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Productivity & Guardedness
Mode of execution				
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Productivity & Guardedness
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Productivity & Guardedness
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics	lfp	lfp	gfp (restricted)	lfp & gfp
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Productivity & Guardedness
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics	lfp	lfp	gfp (restricted)	lfp & gfp
Operational semantics	transitions; states: lists of formulae	transitions; states: lists of formulae	transitions; states: lists of formulae	Coalgebraic

Solution - 1 [Gupta, Simon et al., 2007 - 2008]

If a formula repeatedly appears as a resolvent (modulo α-conversion), then conclude the proof.

Example

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

$\texttt{bit(1)} \leftarrow$

$\texttt{stream(scons (X, Y))} \leftarrow$

$\qquad\qquad \texttt{bit(X), stream(Y)}$

$$\leftarrow \texttt{stream}(\texttt{X})$$

$$|$$

$$\leftarrow \texttt{bit}(\texttt{X1}), \texttt{stream}(\texttt{X})$$

$$|$$

$$\leftarrow \texttt{stream}(\texttt{X})$$

$$|$$

$$\Box^{c}$$

Solution - 1 [Gupta, Simon et al., 2007 - 2008]

If a formula repeatedly appears as a resolvent (modulo α-conversion), then conclude the proof.

Example

```
bit(0) ←
bit(1) ←
stream(scons (X, Y)) ←

          bit(X), stream(Y)
```

$\leftarrow \texttt{stream(X)}$

$|$

$\leftarrow \texttt{bit(X1)}, \texttt{stream(X)}$

$|$

$\leftarrow \texttt{stream(X)}$

$|$

$\square^c$

The answer is: $X/cons(0, X)$.
Requires programs to be regular, in order to be sound and complete

Deciding Termination: Structural Recursion

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Deciding Termination: Structural Recursion

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Example

```
Fixpoint length (A:Type) (l:  list A) : nat :=
match l with
 | nil => O
 | cons _ l' => S (length l')
end.
```

Deciding Productivity: Guardedness

The guardedness condition insures that

- * each corecursive call is made under at least one constructor;
- ** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Deciding Productivity: Guardedness

The guardedness condition insures that

- * each corecursive call is made under at least one constructor;
- ** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Example

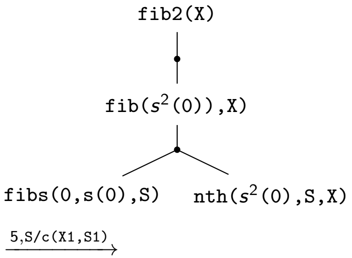
```
CoFixpoint map (s:Stream A) : Stream B :=
SCons (f (hd s)) (map (tl s)).
```

Stream of Fibonacci numbers:

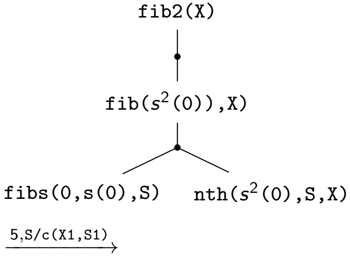
Falls into infinite loops in Prolog and CoLP.

```
1.  add(0,Y,Y).
2.  add(s(X),Y,s(Z)) :- add(X,Y,Z).
3.  fibs(X,Y,cons(X,S)) :- add(X,Y,Z), fibs(Y,Z,S).
4.  nth(0,cons(X,S),X).
5.  nth(s(N),cons(X,S),Y) :- nth(N,S,Y).
6.  fib(N,X) :- fibs(0,s(0),S), nth(N,S,X).
7.  fib2(X) :- fib(s(s(0)),X).
```

Examples of derivations with Fib: lazy step 1

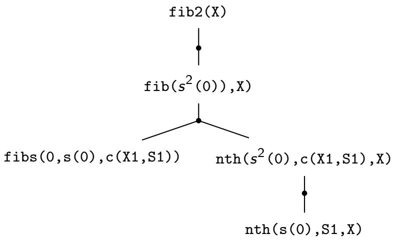


Examples of derivations with Fib: lazy step 1



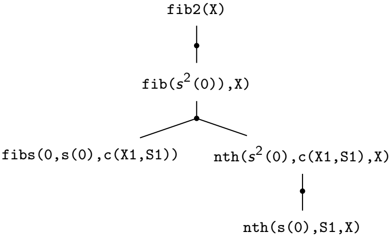

```
1.  add(0,Y,Y).
2.  add(s(X),Y,s(Z)) :-
add(X,Y,Z).
3.  fibs(X,Y,cons(X,S)) :-
add(X,Y,Z), fibs(Y,Z,S).
4.  nth(0,cons(X,S),X).
5.  nth(s(N),cons(X,S),Y) :-
nth(N,S,Y).
6.  fib(N,X) :- fibs(0,s(0),S),
nth(N,S,X).
7.  fib2(X) :- fib(s(s(0)),X).
```

Examples of derivations with Fib: lazy step 2



$\xrightarrow{5,S1/c(X2,S2)}$

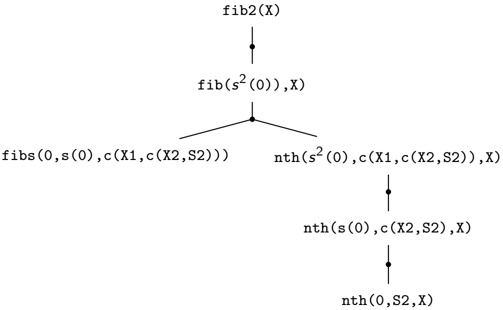
Examples of derivations with Fib: lazy step 2



$\xrightarrow{5,S1/c(X2,S2)}$

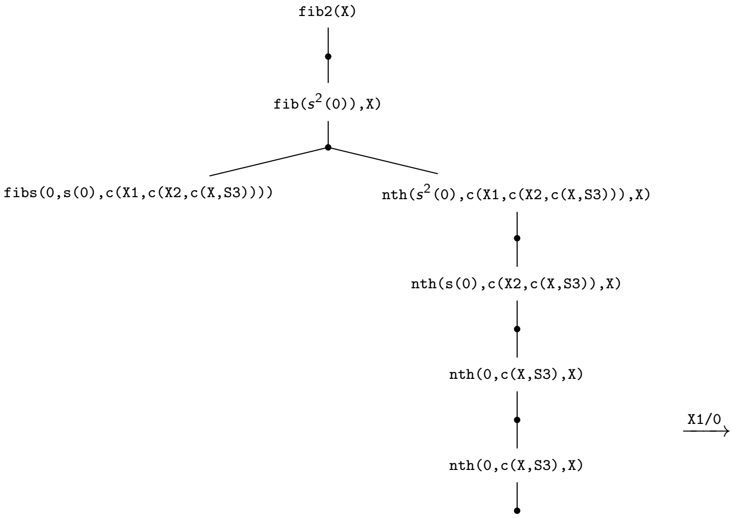
```
1.  add(0,Y,Y).
2.  add(s(X),Y,s(Z)) :-
add(X,Y,Z).
3.  fibs(X,Y,cons(X,S)) :-
add(X,Y,Z), fibs(Y,Z,S).
4.  nth(0,cons(X,S),X).
5.  nth(s(N),cons(X,S),Y) :-
nth(N,S,Y).
6.  fib(N,X) :- fibs(0,s(0),S),
nth(N,S,X).
7.  fib2(X) :- fib(s(s(0)),X).
```

Examples of derivations with Fib: lazy step 3

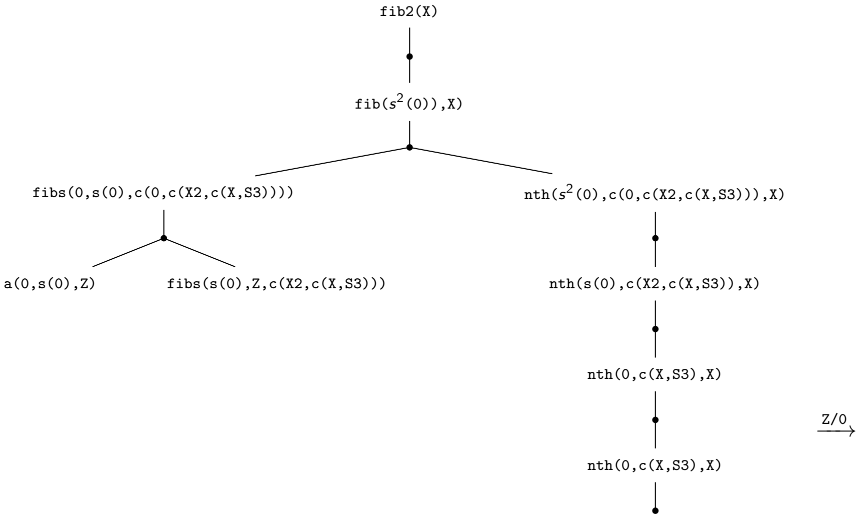


$\xrightarrow{\texttt{S2/c(X,S3)}}$

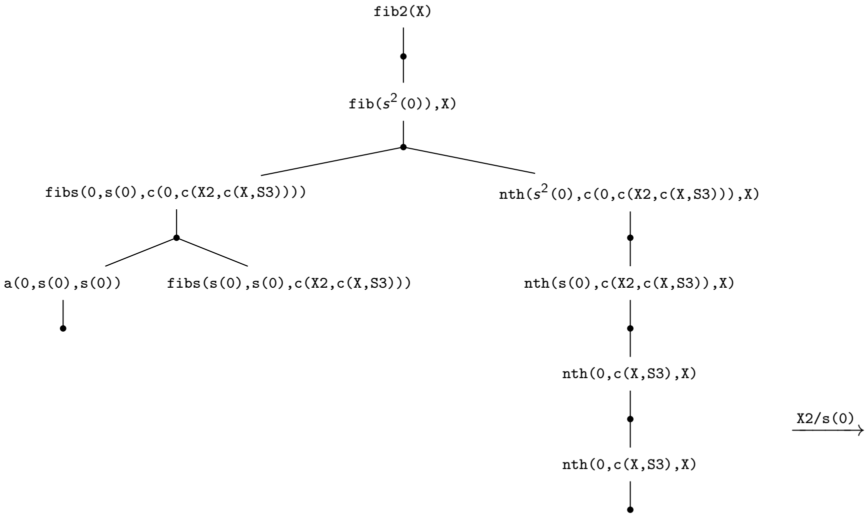
Examples of derivations with Fib: lazy step 4



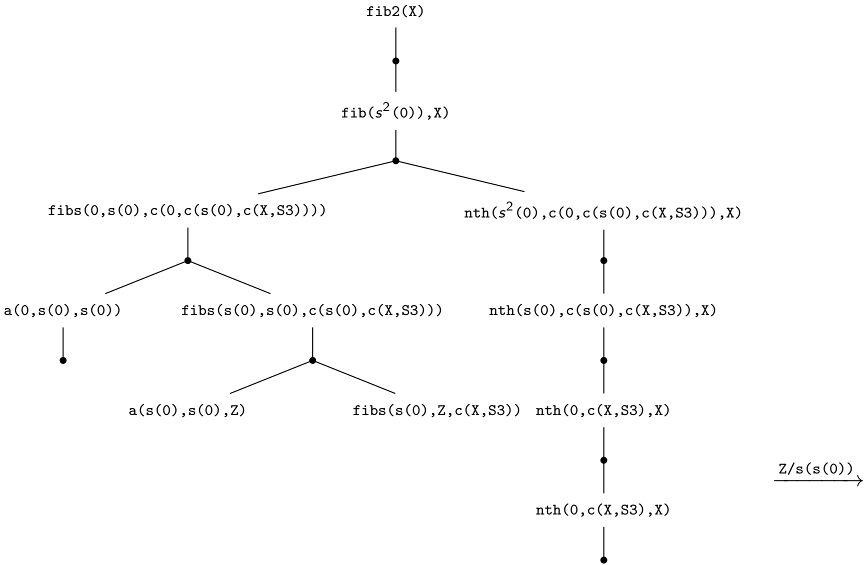
Examples of derivations with Fib: lazy step 5



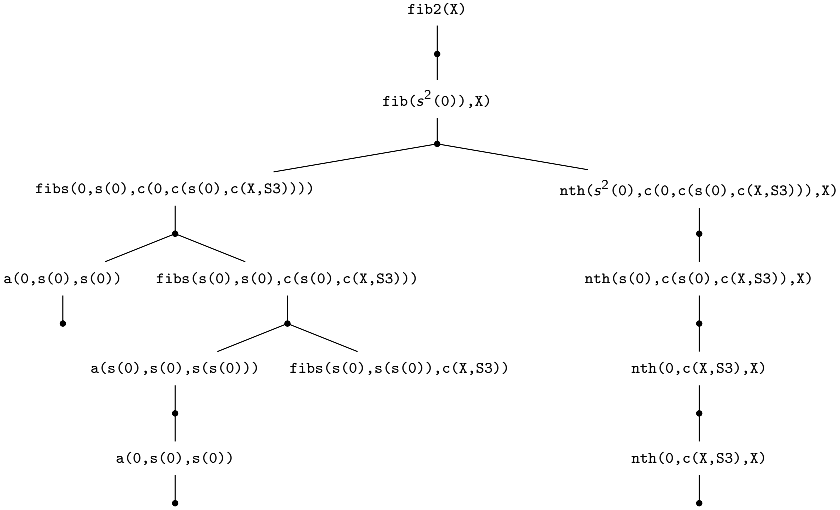
Examples of derivations with Fib: lazy step 6



Examples of derivations with Fib: lazy step 7



Examples of derivations with Fib: lazy step 8



$\xrightarrow{\texttt{X/s(0)}}$

Examples of derivations with Fib: lazy step 9