Coalgebraic Logic Programming

Katya Komendantskaya, joint work with J. Power, M. Schmidt, J. Heras, V. Komendantsky

School of Computing, University of Dundee, UK

3 July 2014
Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types
Outline

1 Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2 Coalgebraic Logic Programming
Outline

1 Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2 Coalgebraic Logic Programming

3 Parallelism
Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2. Coalgebraic Logic Programming

3. Parallelism

4. Future directions: Applications to type inference
Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2. Coalgebraic Logic Programming

3. Parallelism

4. Future directions: Applications to type inference

5. Appendix: LP in Type inference
Today’s talk...

...continuation of Thanos’es talk of yesterday:

- about logic programming (LP);
- about first order (= in Thanos’es terms infinite) language for LP;
- about how much we can merge methods of FP and LP...
- may be you will see some references to a possible game semantics.
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  =>  0
| _ ::  l’ => S (length l’)
end.
CoInductive types and Corecursive Functions


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)).
Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow \leftrightarrow$ types; proofs $\rightarrow \leftrightarrow$ programs): non-terminating proofs can lead to inconsistency.
Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\leftrightarrow$ types; proofs $\leftrightarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.
Productive Values

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

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**

\[
\begin{align*}
\text{nth } 0 \ (S\text{Cons } a \ tl) &= a \\
\text{nth } (S \ n) \ (S\text{Cons } a \ tl) &= \text{nth } n \ tl
\end{align*}
\]

A given stream \( s \) is productive if we can be sure that the computation of the list \( \text{nth } n \ s \) is guaranteed to terminate, whatever the value of \( n \) is.

We call a function **productive at the input value** \( i \), if it outputs a productive value at \( i \).
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.
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 => 0
| _ :: 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)).
To notice:

- The role of types in definition of (co)recursive functions;
- The role of constructors and (co)-pattern matching;
Example

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

Example

bit(0) ←
bit(1) ←
stream(cons (x,y)) ← bit(x), stream(y)
SLD-resolution (+ unification and backtracking) behind LP derivations.

Example

\[
\begin{align*}
\text{nat}(0) & \leftarrow \\
\text{nat}(s(x)) & \leftarrow \text{nat}(x) \\
\text{list}(\text{nil}) & \leftarrow \\
\text{list}(\text{cons } x \ y) & \leftarrow \text{nat}(x), \\
& \quad \text{list}(y) \\
\end{align*}
\]

\[
\begin{align*}
\leftarrow \text{list}(\text{cons}(x, y)) & \\
\quad \leftarrow \text{nat}(x), \text{list}(y)
\end{align*}
\]
SLD-resolution (+ unification) is behind LP derivations.

Example

\[
\begin{align*}
nat(0) & \leftarrow \\
nat(s(x)) & \leftarrow nat(x) \\
list(nil) & \leftarrow \\
list(cons \; x \; y) & \leftarrow nat(x), \\
& \quad list(y)
\end{align*}
\]

\[
\begin{align*}
list(cons(x, y)) & \leftarrow \\
nat(x), list(y) & \leftarrow \\
list(y) & \leftarrow
\end{align*}
\]
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 \(x/O, y/\text{nil}\), but we can get more substitutions by backtracking. We can backtrack infinitely many times, but each time computation will terminate.
SLD-resolution (+ unification) is behind LP derivations.

Example

\[
\begin{align*}
nat(0) & \leftarrow \\
nat(s(x)) & \leftarrow nat(x) \\
list(nil) & \leftarrow \\
list(\text{cons } x \ y) & \leftarrow nat(x), \\
\text{list}(y) & \leftarrow \square
\end{align*}
\]

\[\leftarrow \text{list}(\text{cons}(x, y)) \quad \mid \quad \leftarrow \text{nat}(x), \text{list}(y) \quad \mid \quad \leftarrow \text{list}(y) \quad \mid \quad \leftarrow \square \]

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.

Nice, clean semantics: least Herbrand model exists, sound&complete, etc.: see Thanos’es Viva of yesterday.
Corecursion in LP?

Example

\[
\begin{align*}
\text{bit}(0) & \leftarrow \\
\text{bit}(1) & \leftarrow \\
\text{stream}(\text{scons}(x, y)) & \leftarrow \\
\text{bit}(x), \text{stream}(y) & \leftarrow \\
\end{align*}
\]
Corecursion in LP?

Example

\[
\begin{align*}
\text{bit}(0) & \leftarrow \\
\text{bit}(1) & \leftarrow \\
\text{stream}(\text{scons}(x, y)) & \leftarrow \\
\quad \text{bit}(x), \text{stream}(y)
\end{align*}
\]

No answer, as derivation never terminates.
Corecursion in LP?

Example

\[
\begin{align*}
\text{bit}(0) & \leftarrow \\
\text{bit}(1) & \leftarrow \\
\text{stream}(\text{scons}(x, y)) & \leftarrow \\
& \quad \text{bit}(x), \text{stream}(y) \\
& \quad \text{stream}(\text{scons}(x, y)) \\
& \quad \text{bit}(x_1), \text{stream}(y_1) \\
& \quad \text{stream}(y_1) \\
& \quad \text{bit}(x_2), \text{stream}(y_2) \\
& \quad \text{stream}(y_2) \\
& \quad \vdots
\end{align*}
\]

No answer, as derivation never terminates.
Semantics may go wrong as well: least Herbrand models will contain an infinite term corresponding to \text{stream}: so completeness fails.
It can be worse....

Example

\[
\begin{align*}
\text{bit}(0) & \leftarrow \\
\text{bit}(1) & \leftarrow \\
\text{list}(\text{cons}(x, y)) & \leftarrow \\
& \quad \text{bit}(x), \text{list}(y) \\
\text{list}(\text{nil}) & \leftarrow
\end{align*}
\]
It can be worse....

Example

\[
\begin{align*}
\text{bit}(0) & \leftarrow \\
\text{bit}(1) & \leftarrow \\
\text{list}(\text{cons}(x, y)) & \leftarrow \\
& \quad \text{bit}(x), \text{list}(y) \\
\text{list}(\text{nil}) & \leftarrow
\end{align*}
\]

No answer, as derivation never terminates.
It can be worse....

Example

\[
\text{bit(0)} \leftarrow \\
\text{bit(1)} \leftarrow \\
\text{list(cons(x, y))} \leftarrow \\
\quad \text{bit(x), list(y)}
\]

\[
\text{list(nil)} \leftarrow \\
\quad \text{list(y)}
\]

No answer, as derivation never terminates.
Semantics goes wrong: this time, soundness!
To notice:

- Distinction between (co)inductive type, (co)recursive function over (co)inductive type and a proof by (co)induction is erased.
- Without types guarding (co)recursion, things get messy:
  - ...not ”just” termination, but also semantics
- We do not have a formalism to speak about termination and productivity, or generally, recursion/corecursion.

Note aside: LP has instances of dependent types, mixed induction/coinduction, recursion/corecursion....
Outline

1. **Recursion and Corecursion**
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2. **Coalgebraic Logic Programming**

3. Parallelism

4. **Future directions**: Applications to type inference

5. **Appendix**: LP in Type inference
CoALP: what is it about?

- syntactically – first-order logic programming;
- operationally – lazy (co)recursion;
- inspired by coalgebraic fibrational semantics;
- explores the tree-structure of partial proofs – ”coinductive trees”;
- uses lazy guarded corecursion using measures of corecursive steps given by coinductive trees (cf. ”clocked corecursion”);
- parallel...
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

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 $p : At \rightarrow P_f(P_f(At))$ sends an atom $A$ to the set of bodies of those clauses in $P$ with head $A$. 
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

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 $p : At \rightarrow P_f(P_f(At))$ sends an atom $A$ to the set of bodies of those clauses in $P$ with head $A$.

2. Taking $p : At \rightarrow P_f P_f(At)$, 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

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

This gives a “tree-like” structure: we call them $\& V$-trees.
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

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

2. Taking \( p : At \rightarrow P_fP_f(At) \), the corresponding \( C(P_fP_f) \)-coalgebra where \( C(P_fP_f) \) is the cofree comonad on \( P_fP_f \) is given as follows: \( C(P_fP_f)(At) \) is given by a limit of the form

\[
\ldots \rightarrow At \times P_fP_f(At \times P_fP_f(At)) \rightarrow At \times P_fP_f(At) \rightarrow At.
\]

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

3. For first order extension: Take Lawvere Theory \( L_\Sigma \) to model the signature \( \Sigma \) (objects are natural numbers, arrows – term arities, composition = substitution), and take \( L_\Sigma \rightarrow Set \) – to model (predicates in) \( At \).
Examples

Program Stream: fibers are term arities. Take the fiber of 1. & V-trees:
Examples

Program Stream: fibers are term arities. Take the fiber of 1. \& V-trees:

\text{stream}(x)
Examples

Program Stream: fibers are term arities. Take the fiber of 1. & \( V \)-trees:

\[
\text{stream}(x) \quad \text{stream}(\text{scons}(x, x))
\]

\[
\text{bit}(x) \quad \text{stream}(x)
\]
Examples

Program Stream: fibers are term arities. Take the fiber of 1. \& V-trees:

\[
\text{stream}(x) \quad \text{stream}(\text{scons}(x, x))
\]

\[
\text{bit}(x) \quad \text{stream}(x)
\]

\[
\text{stream}(\text{scons}(0, \text{scons}(x, x)))
\]

\[
\text{bit}(0) \quad \text{stream}(\text{scons}(x, x))
\]

\[
\text{bit}(x) \quad \text{stream}(x)
\]
Computationally essential:

1. for coinductive Stream program, 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 program, 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 laziness involved...
Lazy Corecursion in CoALP: Coinductive trees

\[ \theta_1 \rightarrow \]

\( \text{stream}(x) \)
Lazy Corecursion in CoALP: Coinductive trees

\[ \theta_1 \rightarrow \text{stream}(x) \rightarrow \text{stream}(\text{scons}(z, y)) \]

\[ \theta_2 \rightarrow \text{bit}(z) \rightarrow \text{stream}(y) \]

Note that transitions \( \theta \) may be determined in a number of ways:
- Using mgus;
- Non-deterministically;
- In a distributed/parallel manner.
Lazy Corecursion in CoALP: Coinductive trees

\[
\begin{align*}
\theta_1 & \rightarrow \\
\text{stream}(x) & \rightarrow \text{stream}(\text{scons}(z,y)) \\
& \quad \quad \quad \text{bit}(z) \quad \text{stream}(y)
\end{align*}
\]

Note that transitions \( \theta \) may be determined in a number of ways:

- using mgus;
- non-deterministically;
- in a distributed/parallel manner.
Lazy Corecursion in CoALP

\[ \theta_1 \xrightarrow{} \quad \text{stream}(x) \quad \text{stream}(\text{scons}(z, y)) \]

\[ \quad \text{bit}(z) \quad \text{stream}(y) \]

\[ \text{stream}(\text{scons}(0, \text{scons}(y_1, z_1))) \]

\[ \quad \text{bit}(0) \quad \text{stream}(\text{scons}(y_1, z_1)) \]

\[ \quad \quad \text{bit}(y_1) \quad \text{stream}(z_1) \]

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


- Sound and complete with respect to the coalgebraic semantics;
- Finite computations are sound and complete with respect to the least Herbrand model semantics (so, we can do as much as standard Prolog for sure).
- Adequacy result for observational semantics.
Properties:

- Sound and complete with respect to the coalgebraic semantics;
- Finite computations are sound and complete with respect to the least Herbrand model semantics (so, we can do as much as standard Prolog for sure).
- Adequacy result for observational semantics.

What does it tell us beyond LP?
CoALP: the three-dimensional calculus of trees

1. Dimension 1: term-trees;
2. Dimension 2: coinductive trees;
Dimension-1: Term-trees

Take a tree-language $\mathbb{N}^*$. Given an $L \in \mathbb{N}^*$, a term tree is a map $L \rightarrow \Sigma$, satisfying term arity restrictions.

Example:

0. 

1. bitstream

```
0 0 0 0 1
```

Notation:

<table>
<thead>
<tr>
<th>$\text{Term}(\Sigma)$</th>
<th>finite term trees over $\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Term}_\infty(\Sigma)$</td>
<td>infinite term trees over $\Sigma$</td>
</tr>
<tr>
<td>$\text{Term}^\omega(\Sigma)$</td>
<td>finite and infinite term trees over $\Sigma$</td>
</tr>
</tbody>
</table>
Given an $L \in \mathbb{N}^*$, a coinductive tree is a map $L \rightarrow \text{Term}(\Sigma P)$, satisfying coinductive tree construction for $P$.

Example:
0. \[
\begin{array}{c}
\varepsilon \\
0 \\
0 0
\end{array}
\]
2. \[
\begin{array}{c}
\text{bitstream}(\text{scons}(X,Y)) \\
\text{bit}(X) \\
\text{bitstream}(Y)
\end{array}
\]

Notation:

\[
\begin{array}{|c|c|}
\hline
\text{CTree} & \text{all finite coinductive trees over } \text{Term}(\Sigma P) \\
\text{CTree}^\infty & \text{all infinite coinductive trees over } \text{Term}(\Sigma P) \\
\text{CTree}^\omega & \text{all finite and infinite coinductive trees over } \text{Term}(\Sigma P) \\
\hline
\end{array}
\]
Dimension-3: Derivation trees

Given an \( L \in \mathbb{N}^* \), a coinductive derivation is a map \( L \rightarrow \text{CTree}(\text{Term}(\Sigma_P), P) \), satisfying the mgu requirement.

\[
\begin{align*}
\sigma \xi & \quad \sigma \epsilon & \quad \sigma 0 \\
\epsilon \text{ bitstream}(X) & \quad 0 \text{ bitstream}([X1|Y]) & \quad 00 \text{ bitstream}([0|Y]) \\
& \quad \text{ bitstream}(Y) & \\
& \quad \text{ bitstream}(Y) \\
01 \text{ bitstream}([X1|[X2|Y1]]) & \\
& \quad \text{ bitstream}([X2|Y1]) \\
& \quad \text{ bitstream}(Y1)
\end{align*}
\]
### Dimension-3 notation

<table>
<thead>
<tr>
<th>CDer((\text{CTree}(\text{Term}(\Sigma P), P)))</th>
<th>all finite coinductive derivations over ((\text{CTree}(\text{Term}(\Sigma P), P)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDer(^\infty)(\text{CTree}(\text{Term}(\Sigma P), P))</td>
<td>all infinite coinductive trees over ((\text{CTree}(\text{Term}(\Sigma P), P)))</td>
</tr>
<tr>
<td>CDer(^\omega)(\text{CTree}(\text{Term}(\Sigma P), P))</td>
<td>all finite and infinite coinductive trees over ((\text{CTree}(\text{Term}(\Sigma P), P)))</td>
</tr>
</tbody>
</table>
A first-order logic program $P$ is *productive* if for any term $t \in \text{Term}(\Sigma_P)$, the coinductive tree $CT$ with the root $t$ belongs to $CTree(\text{Term}(\Sigma_P), P)$. 
A first-order logic program $P$ is *productive* if

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

In the class of Productive LPs, we can further distinguish *finite LP* that give rise to derivations in $CDer(CTree(\text{Term}(\Sigma_P), P))$, E.g. *bit*. 
A first-order logic program $P$ is \textit{productive} if

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

- In the class of Productive LPs, we can further distinguish \textit{finite LP} that give rise to derivations in $CDer(CTree(\text{Term}(\Sigma_P), P))$, E.g. \textit{bit}.
- \textit{Inductive LPs} all derivations for which are in $CDer^{\omega}(CTree(\text{Term}(\Sigma_P), P))$; E.g. \textit{ListNat}.
A first-order logic program $P$ is *productive* if
for any term $t \in \text{Term}(\Sigma_P)$, the coinductive tree $CT$ with the root $t$
belongs to $CTree(\text{Term}(\Sigma_P), P)$.

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

- *inductive LPs* all derivations for which are in
  $CDer^\omega(CTree(\text{Term}(\Sigma_P), P))$;
  E.g. *ListNat*.

- *coinductive LPs* all derivations for which are in
  $CDer^\infty(CTree(\text{Term}(\Sigma_P), P))$
  E.g. *Stream*. 
Theory of Productivity in LP

Non-productive

(Coinductive) Derivations

Productive

Coinductive

Inductive

Finite

Compare with:

Recursion

Terminating

Non-terminating

Corecursion

Productive

Non-productive
Theory of Productivity in LP

Compare with:

Terminating → Non-terminating
Recursion

Productive

Non-productive

(Coinductive) Derivations

Coinductive → Inductive
Finite

Recursion

Corecursion
Deciding Productivity: Guardedness

- **Dimension 1.** Measures of reduction on term trees:
  \( \text{stream}(y) \) is a reduction of \( \text{stream}(\text{scons}(x,y)) \)

- **Dimension 2.** Reduction on coinductive tree loops:

  \[
  \begin{aligned}
  \text{stream}(\text{scons}(z,y)) \\
  \quad \\
  \quad \\
  \text{bit}(z) \quad \text{stream}(y)
  \end{aligned}
  \]

- **Dimension 3.** Discovery of derivation loops.
Example of guardedness issues

\[ p(s(X1), X2, Y1, Y2) \leftarrow q(X2, X2, Y1, Y2) \]
\[ q(X1, X2, s(Y1), Y2) \leftarrow p(X1, X2, Y2, Y2) \]
Example of guardedness issues

\[
\begin{align*}
p(s(X1),X2,Y1,Y2) & \leftarrow q(X2,X2,Y1,Y2) \\
q(X1,X2,s(Y1),Y2) & \leftarrow p(X1,X2,Y2,Y2)
\end{align*}
\]

\[
\begin{align*}
p(s(X1),X2,Y1,Y2) \\
\downarrow \\
q(X2,X2,Y1,Y2)
\end{align*}
\]
Example of guardedness issues

\[ p(s(X_1), X_2, Y_1, Y_2) \leftarrow q(X_2, X_2, Y_1, Y_2) \]
\[ q(X_1, X_2, s(Y_1), Y_2) \leftarrow p(X_1, X_2, Y_2, Y_2) \]
Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

- Imposing guardedness conditions, to ensure every coinductive tree is finite.
  To ban programs that are not guarded by constructors:
  \[
  \text{stream(scons x,y) } \leftarrow \text{stream(scons x,y)}
  \]
Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

- Imposing guardedness conditions, to ensure every coinductive tree is finite.
  To ban programs that are not guarded by constructors:
  \[
  \text{stream(scons } x,y) \leftarrow \text{stream(scons } x,y)
  \]
  Unlike termination checks in Coq/Agda cannot be done fully statically (no types to help!), and needs some proof search in Dimension 3.
Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

- Imposing guardedness conditions, to ensure every coinductive tree is finite.
  - To ban programs that are not guarded by constructors:
    stream(scons x, y) ← stream(scons x, y)
  - Unlike termination checks in Coq/Agda cannot be done fully statically (no types to help!), and needs some proof search in Dimension 3.

- Determining when it is safe to make a coinductive conclusion (and finding a right coinductive hypothesis).
  (Again, the troubles come form un-typed setting.)
Stream of Fibonacci numbers:

Falls into infinite loops in Prolog and Prolog-like version of CoLP [Gupta et al. 2007] [Both are eager...] Those powerful SAT/SMT solvers would not do it either.

1. \( \text{add}(0, Y, Y) \).
2. \( \text{add}(s(X), Y, s(Z)) \) :- \( \text{add}(X, Y, Z) \).
3. \( \text{fibs}(X, Y, \text{cons}(X, S)) \) :- \( \text{add}(X, Y, Z) \), \( \text{fibs}(Y, Z, S) \).
4. \( \text{nth}(0, \text{cons}(X, S), X) \).
5. \( \text{nth}(s(N), \text{cons}(X, S), Y) \) :- \( \text{nth}(N, S, Y) \).
6. \( \text{fib}(N, X) \) :- \( \text{fibs}(0, s(0), S) \), \( \text{nth}(N, S, X) \).
7. \( \text{fib2}(X) \) :- \( \text{fib}(s(s(0)), X) \).
Examples of derivations with Fib: lazy step 1

1. \texttt{add(0,Y,Y)}.
2. \texttt{add(s(X),Y,s(Z)) :- add(X,Y,Z)}.
3. \texttt{fibs(X,Y,cons(X,S)) :- add(X,Y,Z), fibs(Y,Z,S)}.
4. \texttt{nth(0,cons(X,S),X)}.
5. \texttt{nth(s(N),cons(X,S),Y) :- nth(N,S,Y)}.
6. \texttt{fib(N,X) :- fibs(0,s(0),S), nth(N,S,X)}.
7. \texttt{fib2(X) :- fib(s(s(0)),X)}.
Examples of derivations with Fib: lazy step 1

1. \( \text{add}(0, Y, Y) \).
2. \( \text{add}(s(X), Y, s(Z)) := \text{add}(X, Y, Z) \).
3. \( \text{fibs}(X, Y, \text{cons}(X, S)) := \text{add}(X, Y, Z), \text{fibs}(Y, Z, S) \).
4. \( \text{nth}(0, \text{cons}(X, S), X) \).
5. \( \text{nth}(s(N), \text{cons}(X, S), Y) := \text{nth}(N, S, Y) \).
6. \( \text{fib}(N, X) := \text{fibs}(0, s(0), S), \text{nth}(N, S, X) \).
7. \( \text{fib2}(X) := \text{fib}(s(s(0)), X) \).
Examples of derivations with Fib: lazy step 2

\[
\text{fib2}(X) \\
\text{fib}(s^2(0)), X) \\
\text{fibs}(0, s(0), c(X_1, S_1)) \\
\text{nth}(s^2(0), c(X_1, S_1), X) \\
\text{nth}(s(0), S_1, X) \\
5, S_1/c(X_2, S_2)
\]
Examples of derivations with Fib: lazy step 2

```
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,cons(X,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

\[
\begin{align*}
\text{fib2}(X) & \quad \text{fib}(s^2(0), X) \\
\text{fibs}(0, s(0), c(X_1, c(X_2, S_2))) & \quad \text{nth}(s^2(0), c(X_1, c(X_2, S_2)), X) \\
& \quad \text{nth}(s(0), c(X_2, S_2), X) \\
& \quad \text{nth}(0, S_2, X) \\
S_2/c(X, S_3) & \quad \rightarrow
\end{align*}
\]
Examples of derivations with Fib: lazy step 4

```
fib2(X)

fib(s^2(0)),X)
```

```
fibs(0,s(0),c(X1,c(X2,c(X,S3))))
```

```
nth(s^2(0),c(X1,c(X2,c(X,S3))),X)
```

```
nth(s(0),c(X2,c(X,S3)),X)

nth(0,c(X,S3),X)
```

```
nth(0,c(X,S3),X)
```

```
X1/0
```

Katya (Dundee)

CoALP for Type Inference

Lyon’14 40 / 56
Examples of derivations with Fib: lazy step 5

```
fib2(X)

fib(s^2(0)),X)

fibs(0,s(0),c(0,c(X2,c(X,S3))))

a(0,s(0),Z)
fibs(s(0),Z,c(X2,c(X,S3)))

nth(s^2(0),c(0,c(X2,c(X,S3))),X)
nth(s(0),c(X2,c(X,S3)),X)
nth(0,c(X,S3),X)
nth(0,c(X,S3),X)
```

\[ Z/0 \]
Examples of derivations with Fib: lazy step 6
Examples of derivations with Fib: lazy step 7

\[
\begin{align*}
&\text{fib2}(X) \\
&\text{fib}(s^2(0), X) \\
&\text{fibs}(0, s(0), c(0, c(s(0), c(X, S3)))) \\
&\text{nth}(s^2(0), c(0, c(s(0), c(X, S3))), X) \\
&\text{a}(0, s(0), s(0)) \\
&\text{fibs}(s(0), s(0), c(s(0), c(X, S3))) \\
&\text{nth}(s(0), c(s(0), c(X, S3)), X) \\
&\text{a}(s(0), s(0), Z) \\
&\text{fibs}(s(0), Z, c(X, S3)) \\
&\text{nth}(0, c(X, S3), X) \\
&\text{Z}/s(s(0)) \\
&\text{nth}(0, c(X, S3), X)
\end{align*}
\]
Examples of derivations with Fib: lazy step 8

\[ \text{fib2}(X) \]
\[ \text{fib}(s^2(0)), X) \]
\[ \text{fibs}(0, s(0), c(0, c(s(0), c(X, S3)))) \]
\[ \text{a}(0, s(0), s(0)) \]
\[ \text{fibs}(s(0), s(0), c(s(0), c(X, S3))) \]
\[ \text{a}(s(0), s(0), s(s(0))) \]
\[ \text{fibs}(s(0), s(s(0)), c(X, S3)) \]
\[ \text{a}(0, s(0), s(0)) \]
\[ X/s(0) \]
\[ \text{nth}(s^2(0), c(0, c(s(0), c(X, S3))), X) \]
\[ \text{nth}(s(0), c(s(0), c(X, S3)), X) \]
\[ \text{nth}(0, c(X, S3), X) \]
\[ \text{nth}(0, c(X, S3), X) \]
Examples of derivations with Fib: lazy step 9

```
fib2(s(0))

fib(s^2(0)), s(0))

fibs(0, s(0), c(0, c(s(0), c(s(0), S3))))

fibs(s(0), s(0), c(s(0), c(s(0), S3)))

fibs(s(s(0)), s(0), c(s(0), S3))

fibs(s(0), s(s(0)), c(s(0), S3))

nth(s^2(0), c(0, c(s(0), c(s(0), S3))), s(0))

nth(s(0), c(s(0), c(s(0), S3)), s(0))

nth(0, c(s(0), S3), s(0))

nth(0, c(s(0), S3), s(0))
```

Katya (Dundee)
## Logic Programming dialects, compared

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Parallel Prolog</th>
<th>Co-LP</th>
<th>CoALP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fib example</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Eager</td>
<td>Eager</td>
<td>Eager</td>
<td>Lazy</td>
</tr>
<tr>
<td><strong>Corecursion</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mode of execution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Declarative semantics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Operational semantics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Prolog**: Full name
- **Parallel Prolog**: Parallel Logic Programming
- **Co-LP**: Co-Logic Programming
- **CoALP**: CoALP for Type Inference

---

Katya (Dundee)
## Logic Programming dialects, compared

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Parallel Prolog</th>
<th>Co-LP</th>
<th>CoALP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib example</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Execution</td>
<td>Eager</td>
<td>Eager</td>
<td>Eager</td>
<td>Lazy</td>
</tr>
<tr>
<td>Corecursion</td>
<td>No</td>
<td>No</td>
<td>by Regular Loop detection</td>
<td>Guardedness by constructors</td>
</tr>
<tr>
<td>Mode of execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declarative semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operational semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Katya (Dundee)
## Logic Programming dialects, compared

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Parallel Prolog</th>
<th>Co-LP</th>
<th>CoALP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib example</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Eager</td>
<td>Eager</td>
<td>Eager</td>
<td>Lazy</td>
</tr>
<tr>
<td>Corecursion</td>
<td>No</td>
<td>No</td>
<td>by Regular Loop detection</td>
<td>Guardedness by constructors</td>
</tr>
<tr>
<td>Mode of execution</td>
<td>Sequential</td>
<td>Parallel</td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>Declarative semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operational semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Logic Programming dialects, compared

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Parallel Prolog</th>
<th>Co-LP</th>
<th>CoALP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fib example</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Eager</td>
<td>Eager</td>
<td>Eager</td>
<td>Lazy</td>
</tr>
<tr>
<td><strong>Corecursion</strong></td>
<td>No</td>
<td>No</td>
<td>by Regular Loop detection</td>
<td>Guardedness by constructors</td>
</tr>
<tr>
<td><strong>Mode of execution</strong></td>
<td>Sequential</td>
<td>Parallel</td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td><strong>Declarative semantics</strong></td>
<td>Ifp</td>
<td>Ifp</td>
<td>gfp (restricted)</td>
<td>coalgebraic</td>
</tr>
<tr>
<td><strong>Operational semantics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logic Programming dialects, compared

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Parallel Prolog</th>
<th>Co-LP</th>
<th>CoALP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib example</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Eager</td>
<td>Eager</td>
<td>Eager</td>
<td>Lazy</td>
</tr>
<tr>
<td><strong>Corecursion</strong></td>
<td>No</td>
<td>No</td>
<td>by Regular Loop detection</td>
<td>Guardedness by constructors</td>
</tr>
<tr>
<td><strong>Mode of execution</strong></td>
<td>Sequential</td>
<td>Parallel</td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td><strong>Declarative semantics</strong></td>
<td>Ifp</td>
<td>Ifp</td>
<td>gfp (restricted)</td>
<td>coalgebraic</td>
</tr>
<tr>
<td><strong>Operational semantics</strong></td>
<td>transitions; states: lists of formulae</td>
<td>transitions; states: lists of formulae</td>
<td>transitions; states: lists of formulae</td>
<td>transitions; states: coinductive trees</td>
</tr>
</tbody>
</table>
Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

2. Coalgebraic Logic Programming

3. Parallelism

4. Future directions: Applications to type inference

5. Appendix: LP in Type inference
Parallelising CoALP

Komendantskaya, Schmidt, Heras: *Exploiting Parallelism in Coalgebraic Logic Programming*, ENTCS, 2014

1. bit(0).
2. bit(1).
3. btree(empty).
4. btree(tree(L,X,R)) ← btree(L), bit(X), btree(R).
Parallel CoALP

threads (t) and expand threads (e)

Katya (Dundee)  CoALP for Type Inference  Lyon'14
Directions we are exploring

Haskell implementation is nearly finished. Current task: to find a “right” language to try CoALP-based type inference

- Using CoALP in Hume: for analysis of stream-based networks and/or for type inference;
- Type-inference in Haskell;
- SSReflect: overloading in canonical structures currently requires the use of back-tracking in LP-like algorithm. It could be parallel CoALP execution instead;
- CoALP for global type analysis in object-oriented languages: CoLP is already used for that.
- Formal Verification of CoALP-based type inference
The end

- A paper on implementing lazy guarded corecursion in CoALP using Haskell is in preparation...
- CoALP webpage has various prototype implementations to play with... http://staff.computing.dundee.ac.uk/katya/CoALP/

We will be happy to apply CoALP for TI (or other purposes) in *YOUR* language!
Milner, 1978

“A theory of Type Polymorphism in Programming”
“A theory of Type Polymorphism in Programming”
An elegant match between polymorphic $\lambda$-calculus and type inference by means of Robinson’s unification/resolution algorithm.
Trend in type inference:

improvement in **expressiveness** of the underlying type system, e.g., in terms of

- *Dependent Types*,
- *Type Classes* [Wadler&Blott 89],
- *Generalised Algebraic Types* (GADTs) [Peyton Jones & al, 2006]
- *Dependent Type Classes* [Sozeau & al 08] and
- *Canonical Structures* [Gonthier& al 11].

Milner-style decidable type inference does not always suffice (e.g. the *principal type* may no longer exist), and TI requires additional inference algorithms.
Trend in type inference:

improvement in **expressiveness** of the underlying type system, e.g., in terms of

- *Dependent Types*,
- *Type Classes* [Wadler&Blott 89],
- *Generalised Algebraic Types* (GADTs) [Peyton Jones & al, 2006]
- *Dependent Type Classes* [Sozeau & al 08] and
- *Canonical Structures* [Gonthier& al 11].

Milner-style decidable type inference does not always suffice (e.g. the *principal type* may no longer exist), and TI requires additional inference algorithms.

Implementations of new type inference algorithms include a variety of first-order decision procedures, notably Unification and Logic Programming (LP) [Peyton Jones & al, 2006], Constraint LP [Odersky Sulzmann, Vytiniotis & many more 1999-], LP embedded into interactive tactics (*Coq’s eauto* Sozeau & al. 08], and LP supplemented by rewriting [Gonthier & al, 11].
Motivation: type inference with Polymorphic types

List Length in Haskell

\[
\begin{align*}
\text{length :: } & [a] \rightarrow \text{Integer} \\
\text{length } [] & = 0 \\
\text{length } (x:xs) & = 1 + \text{length } xs
\end{align*}
\]

Logic program for type inference

\[
\begin{align*}
\text{cons}(X) & \leftarrow X = Y \rightarrow \text{list}(Y) \rightarrow \text{list}(Y). \\
\text{plus}(X) & \leftarrow X = \text{int} \rightarrow \text{int} \rightarrow \text{int}. \\
\text{nil}(X) & \leftarrow X = \text{list}(Y). \\
\text{length}(X) & \leftarrow (X = Y \rightarrow Z) \& \text{nil}(Y) \& Z = \text{int} \& \text{cons}(W) \& \\
& \quad (W = W1 \rightarrow W2 \rightarrow Y) \& \text{plus}(U) \& \\
& \quad (U = \text{int} \rightarrow Z \rightarrow Z) \& W2 = Y.
\end{align*}
\]

Query: length(X)?
Answer (any existing PROLOG version): \(X = \text{list}(\_\_\_) \rightarrow \text{int}\).
Trend to do more by type-inference:

... session types,
... writing contracts by means of types:

Example


\[
\begin{align*}
\text{f } xs &= \text{head (reverse (True : xs))} \\
\text{g } xs &= \text{head (reverse } xs) \\
\end{align*}
\]

Both \( f \) and \( g \) are well typed and "‘can’t go wrong’" in Milner’s sense, but \( g \) will crash for empty list, and \( f \) will never crash.

Contract:

\[
\text{reverse } \in (xs : \text{CF}) \rightarrow \{ys | \text{null } xs \leftrightarrow \text{null } ys\}
\]

Requires strong first-order type inference engines: Z3, Vampire, E...
Could it get any better?

- Clear trend on Type theory side: increase in type expressiveness (dependent types, GADTs, type classes, session types, etc etc)
Could it get any better?

- Clear trend on Type theory side: increase in type expressiveness (dependent types, GADTs, type classes, session types, etc etc)
- Chaotic use of type-inference engines, also known in the literature as “using off-the-shelf” first order TPs.
Could it get any better?

- Clear trend on Type theory side: increase in type expressiveness (dependent types, GADTs, type classes, session types, etc etc)
- Chaotic use of type-inference engines, also known in the literature as “using off-the-shelf” first order TPs.
- Would it pay-off to get more conceptually elegant on type inference side? – especially bearing in mind the big emphasis on type inference in more expressive type systems.
Could it get any better?

- Clear trend on Type theory side: increase in type expressiveness (dependent types, GADTs, type classes, session types, etc, etc)
- Chaotic use of type-inference engines, also known in the literature as “using off-the-shelf” first order TPs.
- Would it pay-off to get more conceptually elegant on type inference side? – especially bearing in mind the big emphasis on type inference in more expressive type systems.
- Would our ”Coalgebraic Logic programming” grow to become a type-inference specific theorem prover (with stronger theoretical background and motivation than state-of-the-art SAT/SMT-solvers)?