Productive Corecursion in Logic Programming

Yue Li

School of Mathematical and Computer Sciences
Heriot-Watt University
Joint research with Ekaterina Komendantskaya

19 May 2017
Outline

1 Motivation
   - Overview of motivation
   - Background knowledge for understanding motivation
   - Problem description

2 Productive Corecursion
   - Loop detection rule review
   - Productivity guarantee

3 Conclusion

4 Future Work & Implementation
Motivation

- Overview of motivation
- Background knowledge for understanding motivation
- Problem description

Productive Corecursion

- Loop detection rule review
- Productivity guarantee

Conclusion

Future Work & Implementation
non-terminating SLD derivation

(Jaffar & Stuckey 86)

greatest fixed point semantics
non-terminating SLD derivation

(Jaffar & Stuckey 86)

greatest fixed point semantics

(van Emden & Abdallah 85)

productivity
Motivation

Overview of motivation

non-terminating SLD derivation

CoLP

implementation

(Jaffar & Stuckey 86)

greatest fixed point semantics

(van Emden & Abdallah 85)

productivity

Y. Li (Heriot-Watt)
Motivation

Overview of motivation

- **non-terminating SLD derivation**
  - CoLP (Gupta et al. 07)
  - greatest fixed point semantics (van Emden & Abdallah 85)
  - productivity

Our Work

- productivity (Jaffar & Stuckey 86)

Implementation

Y. Li (Heriot-Watt)

✓: sound
Motivation

Overview of motivation

non-terminating SLD derivation

CoLP

implementation

(Jaffar & Stuckey 86)

greatest fixed point semantics

(van Emden & Abdallah 85)

productivity

✓: sound

✗: not sound
Motivation

Overview of motivation

non-terminating SLD derivation

CoLP (Gupta et al. 07)

implementation (Jaffar & Stuckey 86)

Our Work

implementation

greatest fixed point semantics (van Emden & Abdallah 85)

productivity

✓: sound
✗: not sound

Y. Li (Heriot-Watt)
Productive Corecursion in LP
19 May 2017 9 / 34
Definition (Syntax of definite clause logic) (Lloyd 87)

Term := Constant | Variable | Functor (List of Terms)

Definite clause := Term ← Set of Terms

Goal clause := List of Terms

Program := Set of Definite clauses
Definition (Syntax of definite clause logic) (Lloyd 87)

Term := Constant | Variable | Functor (<List of Terms>)
Definition (Syntax of definite clause logic) (Lloyd 87)

Term := Constant | Variable | Functor (⟨List of Terms⟩)
Definite clause := Term ← Set of Terms
Definition (Syntax of definite clause logic) (Lloyd 87)

Term := Constant | Variable | Functor (<List of Terms>)
Definite clause := Term ← Set of Terms
Goal clause := List of Terms
Motivation

Definition (Syntax of definite clause logic) (Lloyd 87)

Term := Constant | Variable | Functor (<List of Terms>)
Definite clause := Term ← Set of Terms
Goal clause := List of Terms
Program := Set of Definite clauses
Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

Δ

the least fixed point is the smallest set closed forward under the program.

Δ

the greatest fixed point is the largest set closed backward under the program.

Example

nat(0)

nat(s(X))

←

nat(X)

The least fixed point is

{nat(0), nat(s(0)), nat(s(s(0))), . . .}

The greatest fixed point is

{nat(0), nat(s(0)), nat(s(s(0))), . . .} ∪

{nat(s(s(. . .)))}

Formulae computed by non-terminating derivations are in greatest fixed points. (Jaffar & Stuckey 86; van Emden & Abdallah 85)
Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

the least fixed point is the smallest set closed forward under the program.

Example

\[
\text{nat}(0) \\
\text{nat}(s(X)) \\
\rightarrow \text{nat}(X)
\]

The least fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), ...\}.

The greatest fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), ...\} \cup \{nat(s(s(. . . )))\}.

Formulae computed by non-terminating derivations are in greatest fixed points. (Jaffar & Stuckey 86; van Emden & Abdallah 85)
**Definition (Fixed point semantics) (Lloyd 87)**

Given a logic program,

- **the least fixed point** is the smallest set closed forward under the program.
- **the greatest fixed point** is the largest set closed backward under the program.
Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

the least fixed point  is the smallest set closed forward under the program.

the greatest fixed point  is the largest set closed backward under the program.

Example
**Definition (Fixed point semantics) (Lloyd 87)**

Given a logic program,

- **the least fixed point** is the smallest set closed forward under the program.
- **the greatest fixed point** is the largest set closed backward under the program.

**Example**

```
nat(0)
```

The least fixed point is

```
{nat(0), nat(s(0)), nat(s(s(0))), ...}
```

The greatest fixed point is

```
{nat(0), nat(s(0)), nat(s(s(0))), ...} \cup \{nat(s(s(...)))\}
```

Formulae computed by non-terminating derivations are in greatest fixed points. (Jaffar & Stuckey 86; van Emden & Abdallah 85)
**Definition (Fixed point semantics) (Lloyd 87)**

Given a logic program,

- **the least fixed point** is the smallest set closed forward under the program.
- **the greatest fixed point** is the largest set closed backward under the program.

**Example**

```
nat(0)  
nat(s(X)) ← nat(X)
```
Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

the least fixed point  is the smallest set closed forward under the program.

the greatest fixed point  is the largest set closed backward under the program.

Example

nat(0)
nat(s(X)) ← nat(X)

The least fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), \ldots \}.
Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

the least fixed point is the smallest set closed forward under the program.

the greatest fixed point is the largest set closed backward under the program.

Example

\[
\begin{align*}
nat(0) \\
nat(s(X)) &\leftarrow nat(X)
\end{align*}
\]

The least fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), \ldots \}.

The greatest fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), \ldots \} \cup \{nat(s(s(\ldots ))))\}.  

Definition (Fixed point semantics) (Lloyd 87)

Given a logic program,

the least fixed point is the smallest set closed forward under the program.

the greatest fixed point is the largest set closed backward under the program.

Example

nat(0)
nat(s(X)) ← nat(X)

The least fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), \ldots \}.
The greatest fixed point is \{nat(0), nat(s(0)), nat(s(s(0))), \ldots \} \cup \{nat(s(s(\ldots))))\}.

Formulae computed by non-terminating derivations are in greatest fixed points. (Jaffar & Stuckey 86; van Emden & Abdallah 85)
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16;
FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example
nat(X) ← nat(X)

has non-productive derivation:
nat(X)
nat(X)
nat(X)
...

Example
nat(s(X)) ← nat(X)

computes the first limit ordinal
nat(s(s(. . . ))):
nat(X)
nat(X^2)
nat(X^3)
...

X ↦→ s(X^2)
X^2 ↦→ s(X^3)
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16; 
FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16;
FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example

\[ \text{nat}(X) \leftarrow \text{nat}(X) \]
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16;
FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example

nat(X) ← nat(X)
has non-productive derivation:
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16; FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example

\[ \text{nat}(X) \leftarrow \text{nat}(X) \]

has non-productive derivation:

\[ \begin{array}{c}
\text{nat}(X) \\
\downarrow \\
\text{nat}(X) \\
\downarrow \\
\text{nat}(X) \\
\vdots
\end{array} \]
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16;
FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example

nat(X) ← nat(X)
has non-productive derivation:

\[
\begin{align*}
nat(X) \\
\downarrow \\
nat(X) \\
\downarrow \\
nat(X) \\
\vdots
\end{align*}
\]

Example

nat(s(X)) ← nat(X)
Definition (Productivity)
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16;
FP: Sijtsma 89; Endrullis et al. 08)
A productive non-terminating derivation does useful computations while
looping rather than just looping.

Example

\text{nat}(X) \leftarrow \text{nat}(X)

has non-productive derivation:

\text{nat}(X)
\downarrow
\text{nat}(X)
\downarrow
\text{nat}(X)
\ldots

Example

\text{nat}(s(X)) \leftarrow \text{nat}(X)

computes the first limit ordinal
\text{nat}(s(s(\ldots ))):
**Definition (Productivity)**
(LP: van Emden & Abdallah 86; Komendantskaya et al. 16; FP: Sijtsma 89; Endrullis et al. 08)

A productive non-terminating derivation does useful computations while looping rather than just looping.

Example

\[
\text{nat}(X) \leftarrow \text{nat}(X)
\]

has non-productive derivation:

\[
\begin{array}{c}
\text{nat}(X) \\
\downarrow \\
\text{nat}(X) \\
\downarrow \\
\text{nat}(X) \\
\vdots
\end{array}
\]

\[
\text{nat}(s(X)) \leftarrow \text{nat}(X)
\]

computes the first limit ordinal \(\text{nat}(s(s(\ldots)))\):

\[
\begin{array}{c}
\text{nat}(X) \\
\downarrow \\
x \mapsto s(X_2) \\
\text{nat}(X_2) \\
\downarrow \\
x_2 \mapsto s(X_3) \\
\text{nat}(X_3) \\
\vdots
\end{array}
\]
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is sufficient for knowing the whole derivation.

**Definition (Gupta et al. 07)**

CoLP = SLD resolution + loop detection rule.

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**

Successful coLP derivations only compute formulae in greatest fixed points.
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is suffice for knowing the whole derivation.

**Definition (Gupta et al. 07)**

CoLP = SLD resolution + loop detection rule.

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**

Successful coLP derivations only compute formulae in greatest fixed points.
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is sufficient for knowing the whole derivation.

**Definition (Gupta et al. 07)**

\[ \text{CoLP} = \text{SLD resolution} + \text{loop detection rule}. \]

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**

Successful coLP derivations only compute formulae in greatest fixed points.
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is sufficient for knowing the whole derivation.

**Definition (Gupta et al. 07)**

CoLP = SLD resolution + loop detection rule.

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is sufficient for knowing the whole derivation.

**Definition (Gupta et al. 07)**

\[ \text{CoLP} = \text{SLD resolution} + \text{loop detection rule.} \]

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**

Successful coLP derivations only compute formulae in greatest fixed points.
Now consider finite implementation of non-terminating SLD derivations. Since regular formulae have cyclic derivations, finding a cycle (loop) is suffice for knowing the whole derivation.

**Definition (Gupta et al. 07)**

\[
\text{CoLP} = \text{SLD resolution} + \text{loop detection rule}.
\]

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.

**Theorem (Coinductive soundness of coLP) (Gupta et al. 07)**

*Successful coLP derivations only compute formulae in greatest fixed points.*
Example (CoLP at work)

\[ \text{nat(s(X))} \leftarrow \text{nat(X) defines the first limit ordinal s(s(\ldots))} \]

We compare coLP and SLD derivation for goal `nat(X)`.

**SLD derivation (non-terminating)**

1. `nat(X)`
   - `G_0`
2. `nat(X)`
   - `G_1`
3. `\ldots`
   - `G_2`  \[ X \mapsto s(X_2) \]
   - `X_2` \[ \mapsto s(X_3) \]

**CoLP derivation (terminating)**

1. `nat(X)`
   - `G_0`
2. `nat(X_2)`
   - `G_1`
3. `\Box`
   - `G_2`
   - `X` \[ \mapsto s(X_2) \]
   - `X_2` \[ \mapsto X(G_1 \text{ unifies } G_0) \]

SLD derivation computes `s(s(\ldots))` by accumulating `X \mapsto s(X_2)`, `X_2 \mapsto s(X_3)`, \ldots

CoLP derivation computes `s(s(\ldots))` by circular binding `X \mapsto s(X)`. 

Y. Li (Heriot-Watt)  
Productive Corecursion in LP  
19 May 2017  
14 / 34
Example (CoLP at work)

\[ \text{nat}(s(X)) \leftarrow \text{nat}(X) \] defines the first limit ordinal \( s(s(\ldots)) \).
Example (CoLP at work)

nat(s(X)) ← nat(X) defines the first limit ordinal s(s(...)). We compare coLP and SLD derivation for goal nat(X).
Example (CoLP at work)

\[ \text{nat}(s(X)) \leftarrow \text{nat}(X) \] defines the first limit ordinal \( s(s(\ldots)) \). We compare coLP and SLD derivation for goal \( \text{nat}(X) \).

- **SLD derivation**
  - (non-terminating)

- **CoLP derivation**
  - (terminating)
Example (CoLP at work)

nat(s(X)) ← nat(X) defines the first limit ordinal s(s(...)). We compare coLP and SLD derivation for goal nat(X).

SLD derivation
(non-terminating)

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
\downarrow & \quad X \mapsto s(X_2) \\
G_1 & \quad \text{nat}(X_2) \\
\downarrow & \quad X_2 \mapsto s(X_3) \\
G_2 & \quad \text{nat}(X_3) \\
\cdots & \\
\end{align*}
\]

SLD derivation computes s(s(...)) by accumulating X \mapsto s(X_2), X_2 \mapsto s(X_3), ... .

CoLP derivation computes s(s(...)) by circular binding X \mapsto s(X).
Example (CoLP at work)

nat(s(X)) ← nat(X) defines the first limit ordinal s(s(…)). We compare coLP and SLD derivation for goal nat(X).

<table>
<thead>
<tr>
<th>SLD derivation</th>
<th>CoLP derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(non-terminating)</td>
<td>(terminating)</td>
</tr>
<tr>
<td>$G_0$ nat(X)</td>
<td></td>
</tr>
<tr>
<td>$\downarrow$ $X \mapsto s(X_2)$</td>
<td></td>
</tr>
<tr>
<td>$G_1$ nat($X_2$)</td>
<td></td>
</tr>
<tr>
<td>$\downarrow$ $X_2 \mapsto s(X_3)$</td>
<td></td>
</tr>
<tr>
<td>nat($X_3$)</td>
<td></td>
</tr>
<tr>
<td>$G_2$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
</tbody>
</table>
Example (CoLP at work)

\[ \text{nat}(s(X)) \leftarrow \text{nat}(X) \text{ defines the first limit ordinal } s(s(\ldots)). \]

We compare CoLP and SLD derivation for goal \( \text{nat}(X) \).

**SLD derivation (non-terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
& \quad \downarrow \ X \mapsto s(X_2) \\
G_1 & \quad \text{nat}(X_2) \\
& \quad \downarrow \ X_2 \mapsto s(X_3) \\
& \quad \text{nat}(X_3) \\
& \quad \vdots
\end{align*}
\]

**CoLP derivation (terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
& \quad \downarrow \ X \mapsto s(X_2) \\
G_1 & \quad \text{nat}(X_2) \\
& \quad \downarrow \ X_2 \mapsto X \ (G_1 \text{ unifies } G_0) \\
G_2 & \quad \Box
\end{align*}
\]
Example (CoLP at work)

`nat(s(X)) ← nat(X)` defines the first limit ordinal `s(s(…))`. We compare coLP and SLD derivation for goal `nat(X)`.

**SLD derivation** (non-terminating)

1. `G_0`: `nat(X)
   \[ X \mapsto s(X_2) \]
2. `G_1`: `nat(X_2)
   \[ X_2 \mapsto s(X_3) \]
3. `G_2`: `nat(X_3)
   \[ \vdots \]

**CoLP derivation** (terminating)

1. `G_0`: `nat(X)
   \[ X \mapsto s(X_2) \]
2. `G_1`: `nat(X_2)
   \[ X_2 \mapsto X (G_1 \text{ unifies } G_0) \]
3. `G_2`: `□`

SLD derivation computes `s(s(…))` by accumulating
Example (CoLP at work)

\[ \text{nat}(\text{s}(X)) \leftarrow \text{nat}(X) \] defines the first limit ordinal \( \text{s}(\text{s}(\ldots)) \). We compare coLP and SLD derivation for goal \( \text{nat}(X) \).

**SLD derivation** (non-terminating)

\[
\begin{align*}
G_0 & : \text{nat}(X) \\
G_1 & : \text{nat}(X_2) \\
G_2 & : \text{nat}(X_3) \\
\vdots \\
\end{align*}
\]

\( X \mapsto s(X_2) \)

\( X_2 \mapsto s(X_3) \)

**CoLP derivation** (terminating)

\[
\begin{align*}
G_0 & : \text{nat}(X) \\
G_1 & : \text{nat}(X_2) \\
G_2 & : \square \\
\end{align*}
\]

\( X \mapsto s(X_2) \)

\( X_2 \mapsto X \) (\( G_1 \) unifies \( G_0 \))

SLD derivation computes \( s(s(\ldots)) \) by accumulating \( X \mapsto s(X_2) \),
Example (CoLP at work)

nat(s(X)) ← nat(X) defines the first limit ordinal s(s(...)). We compare CoLP and SLD derivation for goal nat(X).

**SLD derivation (non-terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
\quad \downarrow & \quad X \mapsto s(X_2) \\
G_1 & \quad \text{nat}(X_2) \\
\quad \downarrow & \quad X_2 \mapsto s(X_3) \\
\quad \text{nat}(X_3) \\
\ldots & \\
\end{align*}
\]

**CoLP derivation (terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
\quad \downarrow & \quad X \mapsto s(X_2) \\
G_1 & \quad \text{nat}(X_2) \\
\quad \downarrow & \quad X_2 \mapsto X \quad \text{(}G_1\text{ unifies } G_0) \\
G_2 & \quad \Box \\
\end{align*}
\]

SLD derivation computes s(s(...)) by accumulating \( X \mapsto s(X_2), \) \( X_2 \mapsto s(X_3), \ldots \)
nat(s(X)) ← nat(X) defines the first limit ordinal s(s(\ldots)). We compare coLP and SLD derivation for goal nat(X).

**SLD derivation** (non-terminating)
- \( G_0 \) \( \text{nat}(X) \)
- \( X \mapsto s(X_2) \)
- \( G_1 \) \( \text{nat}(X_2) \)
- \( X_2 \mapsto s(X_3) \)
- \( \text{nat}(X_3) \)
- \( \ldots \)

**CoLP derivation** (terminating)
- \( G_0 \) \( \text{nat}(X) \)
- \( X \mapsto s(X_2) \)
- \( G_1 \) \( \text{nat}(X_2) \)
- \( X_2 \mapsto X \) (\( G_1 \) unifies \( G_0 \))
- \( G_2 \) \( \square \)

SLD derivation computes \( s(s(\ldots)) \) by accumulating \( X \mapsto s(X_2) \), \( X_2 \mapsto s(X_3) \), \ldots CoLP derivation computes \( s(s(\ldots)) \) by circular binding.
Example (CoLP at work)

\( \text{nat}(s(X)) \leftarrow \text{nat}(X) \) defines the first limit ordinal \( s(s(\ldots)) \). We compare coLP and SLD derivation for goal \( \text{nat}(X) \).

**SLD derivation (non-terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
\downarrow & \quad x \mapsto s(x_2) \\
G_1 & \quad \text{nat}(x_2) \\
\downarrow & \quad x_2 \mapsto s(x_3) \\
\vdots & \\
G_2 & \quad \text{nat}(x_3)
\end{align*}
\]

SLD derivation computes \( s(s(\ldots)) \) by accumulating \( X \mapsto s(X_2) \), \( X_2 \mapsto s(X_3) \), \ldots

**CoLP derivation (terminating)**

\[
\begin{align*}
G_0 & \quad \text{nat}(X) \\
\downarrow & \quad x \mapsto s(x_2) \\
G_1 & \quad \text{nat}(x_2) \\
\downarrow & \quad x_2 \mapsto x \quad (G_1 \text{ unifies } G_0) \\
G_2 & \quad \Box
\end{align*}
\]

CoLP derivation computes \( s(s(\ldots)) \) by circular binding \( X \mapsto s(X) \).
However, coLP does not take good care of productivity . . .
Assume some successful coLP derivation that computes an infinite formula.

Problem 1: It is not guaranteed that there exists a corresponding non-terminating SLD derivation.

e.g. For program $p(f(X), X) \leftarrow p(X, X)$ and goal $p(f(X), X)$, coLP computes $p(f(f(\ldots)), f(f(\ldots)))$ but here is no non-terminating SLD derivation.

**CoLP derivation**

$G_0 : \quad p(f(X), X)$

$\downarrow$

$G_1 : \quad p(X, X)$

$\downarrow \quad X \mapsto f(f(\ldots))$ by unifying $G_1$ with $G_0$

$\downarrow$

$G_2 : \quad \square$
However, coLP does not take good care of productivity . . . 

Assume some successful coLP derivation that computes an infinite formula.

Problem 1: *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*

e.g. For program \( p(f(X),X) \leftarrow p(X,X) \) and goal \( p(f(X),X) \), coLP computes \( p(f(f(...)),f(f(...))) \) but here is no non-terminating SLD derivation.

CoLP derivation

\[
\begin{align*}
G_0 : & \quad p(f(X),X) \\
\downarrow & \\
G_1 : & \quad p(X,X) \\
\downarrow & \quad X \mapsto f(f(...)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 : & \quad \square
\end{align*}
\]
However, coLP does not take good care of productivity . . . Assume some successful coLP derivation that computes an infinite formula. **Problem 1:** *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*  

*e.g.* For program \( p(f(X),X) \leftarrow p(X,X) \) and goal \( p(f(X),X) \), coLP computes \( p(f(f(\ldots)),f(f(\ldots))) \) but here is no non-terminating SLD derivation.

```
CoLP derivation

\[
G_0 : \quad p(f(X),X) \\
\downarrow \\
G_1 : \quad p(X,X) \\
\downarrow \quad X \mapsto f(f(\ldots)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 : \quad \Box
\]
```
However, coLP does not take good care of productivity . . .
Assume some successful coLP derivation that computes an infinite formula.

**Problem 1:** *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*

*e.g.* For program \( p(f(X),X) \leftarrow p(X,X) \) and goal \( p(f(X),X) \), coLP computes \( p(f(f(...)),f(f(...))) \) but here is no non-terminating SLD derivation.

**CoLP derivation**

\[
\begin{align*}
G_0 : & \quad p(f(X),X) \\
\downarrow \\
G_1 : & \quad p(X,X) \\
\downarrow & \quad X \mapsto f(f(...)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 : & \quad \square
\end{align*}
\]
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula.

**Problem 1:** *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*

e.g. For program $p(f(X),X) \leftarrow p(X,X)$ and goal $p(f(X),X)$, coLP computes $p(f(f(\ldots)),f(f(\ldots)))$ but here is no non-terminating SLD derivation.

CoLP derivation

$G_0 : p(f(X),X) \downarrow$

$G_1 : p(X,X) \downarrow X \mapsto f(f(\ldots))$ by unifying $G_1$ with $G_0$

$G_2 : \square$
However, coLP does not take good care of productivity . . .
Assume some successful coLP derivation that computes an infinite formula. **Problem 1:** *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*

e.g. For program \( p(f(X),X) \leftarrow p(X,X) \) and goal \( p(f(X),X) \), coLP computes \( p(f(f(\ldots)),f(f(\ldots))) \) but here is no non-terminating SLD derivation.

**SLD derivation**

\[
G_0 : \ p(f(X),X) \\
\downarrow \\
G_1 : \ p(X,X)
\]

\[† \ G_1 \text{ does not unify head of clause} \]
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula. **Problem 1:** *It is not guaranteed that there exists a corresponding non-terminating SLD derivation.*

*E.g.* For program \( p(f(X), X) \leftarrow p(X, X) \) and goal \( p(f(X), X) \), coLP computes \( p(f(f(\ldots)), f(f(\ldots))) \) but here is no non-terminating SLD derivation.

\[
\begin{align*}
G_0 & : \quad p(f(X), X) \\
G_1 & : \quad p(X, X)
\end{align*}
\]

\( \uparrow \)  \( G_1 \) does not unify head of clause
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula.

**Problem 2:** There exists a corresponding non-terminating SLD derivation but it computes a different formula.

e.g. For program \( q(f(X), Y) \leftarrow q(X, h(Y)) \) and goal \( q(f(X), Y) \), coLP computes \( q(f(f(\ldots)), h(h(\ldots))) \) but the corresponding non-terminating SLD derivation computes \( q(f(f(\ldots)), Y) \).

CoLP derivation

\[
\begin{align*}
G_0 &: q(f(X), Y) \\
\downarrow \\
G_1 &: q(X, h(Y)) \\
\downarrow & X \mapsto f(f(\ldots)), Y \mapsto h(h(\ldots)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 &: \square
\end{align*}
\]
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula. **Problem 2:** *There exists a corresponding non-terminating SLD derivation but it computes a different formula.*

e.g. For program \( q(f(X), Y) \leftarrow q(X, h(Y)) \) and goal \( q(f(X), Y) \), coLP computes \( q(f(f(\ldots)), h(h(\ldots))) \) but the corresponding non-terminating SLD derivation computes \( q(f(f(\ldots)), Y) \).

**CoLP derivation**

\[
\begin{align*}
G_0 & : q(f(X), Y) \\
G_1 & : q(X, h(Y)) \\
& \quad \downarrow X \mapsto f(f(\ldots)), Y \mapsto h(h(\ldots)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 & : \square
\end{align*}
\]
However, coLP does not take good care of productivity . . . Assume some successful coLP derivation that computes an infinite formula. 

**Problem 2:** *There exists a corresponding non-terminating SLD derivation but it computes a different formula.*

e.g. For program $q(f(X), Y) \leftarrow q(X, h(Y))$ and goal $q(f(X), Y)$, coLP computes $q(f(f(\ldots)), h(h(\ldots)))$ but the corresponding non-terminating SLD derivation computes $q(f(f(\ldots)), Y)$.

---

**CoLP derivation**

\[
\begin{align*}
G_0 : & \quad q(f(X), Y) \\
\downarrow & \\
G_1 : & \quad q(X, h(Y)) \\
\downarrow & \quad X \leftrightarrow f(f(\ldots)), \ Y \leftrightarrow h(h(\ldots)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 : & \quad \square
\end{align*}
\]
However, coLP does not take good care of productivity . . .
Assume some successful coLP derivation that computes an infinite formula.

**Problem 2:** There exists a corresponding non-terminating SLD derivation but it computes a different formula.

e.g. For program $q(f(X), Y) \leftarrow q(X, h(Y))$ and goal $q(f(X), Y)$, coLP computes $q(f(f(\ldots)), h(h(\ldots)))$ but the corresponding non-terminating SLD derivation computes $q(f(f(\ldots)), Y)$.

CoLP derivation

\[
\begin{align*}
G_0 & : q(f(X), Y) \\
\downarrow & \\
G_1 & : q(X, h(Y)) \\
\downarrow & X \mapsto f(f(\ldots)), Y \mapsto h(h(\ldots)) \text{ by unifying } G_1 \text{ with } G_0 \\
G_2 & : \square
\end{align*}
\]
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula.

**Problem 2:** *There exists a corresponding non-terminating SLD derivation but it computes a different formula.*

e.g. For program \( q(f(X), Y) \leftarrow q(X, h(Y)) \) and goal \( q(f(X), Y) \), coLP computes \( q(f(f(\ldots)), h(h(\ldots))) \) but the corresponding non-terminating SLD derivation computes \( q(f(f(\ldots)), Y) \).

\[
\begin{align*}
G_0 : & \quad q(f(X), Y) \\
& \downarrow \\
G_1 : & \quad q(X, h(Y)) \\
& \downarrow X \mapsto f(X_2) \\
G_2 : & \quad q(X_2, h(h(Y))) \\
& \downarrow X_2 \mapsto f(X_3) \\
G_3 : & \quad q(X_3, h(h(h(Y)))) \\
& \vdots
\end{align*}
\]
However, coLP does not take good care of productivity . . .

Assume some successful coLP derivation that computes an infinite formula.

**Problem 2:** There exists a corresponding non-terminating SLD derivation but it computes a different formula.

e.g. For program \( q(f(X), Y) \leftarrow q(X, h(Y)) \) and goal \( q(f(X), Y) \), coLP computes \( q(f(f(\ldots)), h(h(\ldots))) \) but the corresponding non-terminating SLD derivation computes \( q(f(f(\ldots)), Y) \).

\[
\begin{align*}
G_0 : & \quad q(f(X), Y) \\
\downarrow & \\
G_1 : & \quad q(X, h(Y)) \\
\quad \downarrow & \quad X \mapsto f(X_2) \\
G_2 : & \quad q(X_2, h(h(Y))) \\
\quad \downarrow & \quad X_2 \mapsto f(X_3) \\
G_3 : & \quad q(X_3, h(h(h(Y)))) \\
\vdots & \\
\end{align*}
\]
Motivation

Problem description

Our question

Can we have an implementation of infinite SLD derivation, that is not only sound regarding greatest fixed points, but also sound regarding productivity?

Our answer is affirmative.
Our question

Can we have an implementation of infinite SLD derivation,
Our question

*Can we have an implementation of infinite SLD derivation, that is not only sound regarding greatest fixed points,*
Our question

Can we have an implementation of infinite SLD derivation, that is not only sound regarding greatest fixed points, but also sound regarding productivity?
Our question

Can we have an implementation of infinite SLD derivation, that is not only sound regarding greatest fixed points, but also sound regarding productivity?

Our answer is affirmative.
1 Motivation
   - Overview of motivation
   - Background knowledge for understanding motivation
   - Problem description

2 Productive Corecursion
   - Loop detection rule review
   - Productivity guarantee

3 Conclusion

4 Future Work & Implementation
If two formulae unify \textit{without} occurs check and produce an infinite regular formula,
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart,
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check.
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

Case 1: They do not unify. e.g. \( p(f(X),X) \) and \( p(X,X) \);

Case 2: One is a variant or instance of the other. e.g. \( \text{nat}(X) \) and \( \text{nat}(s(X)) \);

Case 3: Otherwise. e.g. \( q(f(X),Y) \) and \( q(X,h(Y)) \);
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

\textbf{Case 1} They do not unify.
If two formulae unify* without occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version* with occurs check. The following are possible results.

**Case 1** They do not unify. e.g. \( p(f(X), X) \) and \( p(X, X) \);
If two formulae unify *without* occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version *with* occurs check. The following are possible results.

**Case 1** They do not unify. e.g. \( p(f(X),X) \) and \( p(X,X) \); \( p(f(X),X) \) and \( p(X_1,X_1) \).
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

\textbf{Case 1} They do not unify. e.g. \( p(f(X),X) \) and \( p(X,X) \); \( p(f(X),X) \) and \( p(X_1, X_1) \).

\textbf{Case 2} One is a variant or instance of the other.
If two formulae unify *without* occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version *with* occurs check. The following are possible results.

**Case 1**  They do not unify. e.g. $p(f(X),X)$ and $p(X,X)$; $p(f(X),X)$ and $p(X_1, X_1)$.

**Case 2**  One is a variant or instance of the other. e.g. $nat(X)$ and $nat(s(X))$;
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

\textbf{Case 1} They do not unify. e.g. \(p(f(X),X)\) and \(p(X,X)\); \(p(f(X),X)\) and \(p(X_1,X_1)\).

\textbf{Case 2} One is a variant or instance of the other. e.g. \(\text{nat}(X)\) and \(\text{nat}(s(X))\); \(\text{nat}(X)\) and \(\text{nat}(s(X_1))\).
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

**Case 1** They do not unify. e.g. \( p(f(X),X) \) and \( p(X,X) \); \( p(f(X),X) \) and \( p(X_1,X_1) \).

**Case 2** One is a variant or instance of the other. e.g. \( \text{nat}(X) \) and \( \text{nat}(s(X)) \); \( \text{nat}(X) \) and \( \text{nat}(s(X_1)) \).

**Case 3** Otherwise.
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

\textbf{Case 1} They do not unify. e.g. $p(f(X),X)$ and $p(X,X)$; $p(f(X),X)$ and $p(X_1,X_1)$.

\textbf{Case 2} One is a variant or instance of the other. e.g. $\text{nat}(X)$ and $\text{nat}(s(X))$; $\text{nat}(X)$ and $\text{nat}(s(X_1))$.

\textbf{Case 3} Otherwise. e.g. $q(f(X),Y)$ and $q(X,h(Y))$;
If two formulae unify \textit{without} occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version \textit{with} occurs check. The following are possible results.

\textbf{Case 1} They do not unify. e.g. $p(f(X),X)$ and $p(X,X)$; $p(f(X),X)$ and $p(X_1, X_1)$.

\textbf{Case 2} One is a variant or instance of the other. e.g. $\text{nat}(X)$ and $\text{nat}(s(X))$; $\text{nat}(X)$ and $\text{nat}(s(X_1))$.

\textbf{Case 3} Otherwise. e.g. $q(f(X),Y)$ and $q(X,h(Y))$; $q(f(X),Y)$ and $q(X_1,h(Y_1))$. 
If two formulae unify *without* occurs check and produce an infinite regular formula, we should standardize them apart, then try to unify the standardized version *with* occurs check. The following are possible results.

**Case 2** One is a variant or instance of the other. e.g. nat(X) and nat(s(X)); nat(X) and nat(s(X₁)).
To guarantee soundness regarding productivity, we propose using the following loop detection rule:
To guarantee soundness regarding productivity, we propose using the following loop detection rule

**Definition (Our loop detection rule)**

A goal succeeds if it’s a variant of its ancestor goal.
To guarantee soundness regarding productivity, we propose using the following loop detection rule

**Definition (Our loop detection rule)**

A goal succeeds if it’s a variant of its ancestor goal.

instead of

**Definition (Loop detection rule) (Gupta et al. 07)**

A goal succeeds if it unifies with its ancestor goal.
We also characterized a class of logic programs whose non-terminating SLD derivations, if any, are guaranteed to be productive.
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause's head.
Definition (Rewriting for LP) (Komendantskaya et al. 15)

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Example (Rewriting)

- From goal nat(X) and clause nat(X) ← nat(X), derive nat(X).
**Definition (Rewriting for LP) (Komendantskaya et al. 15)**

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

**Example (Rewriting)**

- From goal nat(s(X)) and clause nat(Y) ← nat(s(Y)), derive nat(s(s(Y))).
Definition (Rewriting for LP) (Komendantskaya et al. 15)
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)
A logic program that is

**Definition (Rewriting for LP) (Komendantskaya et al. 15)**

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

---

**Theorem (Productivity Guarantee)**

A logic program that is

1. terminating for rewriting, and
Definition (Rewriting for LP) (Komendantskaya et al. 15)

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)

A logic program that is

1. terminating for rewriting, and
2. free from existential variables,
Definition (Rewriting for LP) (Komendantskaya et al. 15)
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)
A logic program that is
1. terminating for rewriting, and
2. free from existential variables,
is guaranteed to be productive for its non-terminating SLD derivations, if any.
Definition (Rewriting for LP) (Komendantskaya et al. 15)

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)

A logic program that is

1. terminating for rewriting, and
2. free from existential variables,

is guaranteed to be productive for its non-terminating SLD derivations, if any.

Why termination for rewriting plays a role?
Definition (Rewriting for LP) (Komendantskaya et al. 15)
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)
A logic program that is
1. terminating for rewriting, and
2. free from existential variables,
is guaranteed to be productive for its non-terminating SLD derivations, if any.

Why termination for rewriting plays a role?

Assume SLD derivation is non-terminating,
Definition (Rewriting for LP) (Komendantskaya et al. 15)
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)
A logic program that is
1. terminating for rewriting, and
2. free from existential variables,
is guaranteed to be productive for its non-terminating SLD derivations, if any.

Why termination for rewriting plays a role?

Assume SLD derivation is non-terminating, where all rewriting terminates,
Definition (Rewriting for LP) (Komendantskaya et al. 15)

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)

A logic program that is

1. terminating for rewriting, and
2. free from existential variables,

is guaranteed to be productive for its non-terminating SLD derivations, if any.

Why termination for rewriting plays a role?

Assume SLD derivation is non-terminating,
where all rewriting terminates,
then it is guaranteed that
Definition (Rewriting for LP) (Komendantskaya et al. 15)

Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)

A logic program that is
1. terminating for rewriting, and
2. free from existential variables,

is guaranteed to be productive for its non-terminating SLD derivations, if any.

Why termination for rewriting plays a role?

Assume SLD derivation is non-terminating, where all rewriting terminates, then it is guaranteed that there are infinitely many productive SLD resolution steps.
Definition (Rewriting for LP) (Komendantskaya et al. 15)
Rewriting is a special case of SLD resolution, where the selected subgoal is an instance of the chosen program clause’s head.

Theorem (Productivity Guarantee)
A logic program that is
1. terminating for rewriting, and
2. free from existential variables,
is guaranteed to be productive for its non-terminating SLD derivations, if any.

Theorem (our main result: Productivity Semi-decision)
Productivity is semi-decidable for programs characterized above, by SLD resolution combined with our loop detection rule.
1 Motivation
   - Overview of motivation
   - Background knowledge for understanding motivation
   - Problem description

2 Productive Corecursion
   - Loop detection rule review
   - Productivity guarantee

3 Conclusion

4 Future Work & Implementation
non-terminating SLD derivation

CoLP

implementation

(greatest fixed point semantics)

Our Work

implementation

productivity

✓: sound

✗: not sound

(Jaffar & Stuckey 86)

(van Emden & Abdallah 85)

(Gupta et al. 07)

(Gupta et al. 07)

(Y. Li (Heriot-Watt))
We had to change the loop detection rule.
- We had to change the loop detection rule.
- Put conditions on clauses.
- We had to change the loop detection rule.
- Put conditions on clauses.
- These kinds of clauses characterize a rich class of productive corecursion in LP.
We had to change the loop detection rule.
Put conditions on clauses.
These kinds of clauses characterize a rich class of productive corecursion in LP.

Example (number streams) (Gupta et al. 07)
Streams of natural numbers, e.g. 3 1 4 1 5 9 2 . . . ,
We had to change the loop detection rule.
Put conditions on clauses.
These kinds of clauses characterize a rich class of productive corecursion in LP.

Example (number streams) (Gupta et al. 07)
Streams of natural numbers, e.g. 3 1 4 1 5 9 2 . . . , are defined by the corecursive clause nats([X|S]) ← nat(X),nats(S).
- We had to change the loop detection rule.
- Put conditions on clauses.
- These kinds of clauses characterize a rich class of productive corecursion in LP.

**Example (number streams) (Gupta et al. 07)**

Streams of natural numbers, e.g. 3 1 4 1 5 9 2 . . . , are defined by the corecursive clause \( \text{nats([X|S])} \leftarrow \text{nat(X)}, \text{nats(S)}. \)

**Example (increasing stream) (Simon et al. 06)**

Streams of consecutive numbers, e.g. 1 2 3 . . . or 99 100 101 . . . ,
We had to change the loop detection rule.
Put conditions on clauses.
These kinds of clauses characterize a rich class of productive corecursion in LP.

Example (number streams) (Gupta et al. 07)
Streams of natural numbers, e.g. 3 1 4 1 5 9 2 . . . , are defined by the corecursive clause nats([X|S]) ← nat(X),nats(S).

Example (increasing stream) (Simon et al. 06)
Streams of consecutive numbers, e.g. 1 2 3 . . . or 99 100 101 . . . , are defined by the corecursive clause from(X,[X|T]) ← from(s(X),T).
1 Motivation
   • Overview of motivation
   • Background knowledge for understanding motivation
   • Problem description

2 Productive Corecursion
   • Loop detection rule review
   • Productivity guarantee

3 Conclusion

4 Future Work & Implementation
Programs that contain existential variables?
Programs that contain existential variables?

Example (Fibonacci stream) (Komendantskaya et al. 15)

Streams of Fibonacci numbers, e.g. 1 1 2 3 5 8 ... or 10 4 14 18 32 ...,
• Programs that contain existential variables?

Example (Fibonacci stream) (Komendantskaya et al. 15)

Streams of Fibonacci numbers, e.g. 1 1 2 3 5 8 ... or 10 4 14 18 32 ..., are defined by a corecursive clause that has an existential variable.

\[
fibs(X,Y,[X\mid S]) \leftarrow \text{add}(X,Y,Z), \ fibs(Y,Z,S).
\]
Programs that contain existential variables?
Practical application in
Future Work & Implementation

- Programs that contain existential variables?
- Practical application in
  - type inference in programming languages, and

Example (Fibonacci stream) (Komendantskaya et al. 15)

Streams of Fibonacci numbers, e.g. 1 1 2 3 5 8 ... or 10 4 14 18 32 ..., are defined by a corecursive clause that has an existential variable:

\[ \text{fibs}(X, Y, [X | S]) \leftarrow \text{add}(X, Y, Z), \text{fibs}(Y, Z, S). \]
Future Work & Implementation

- Programs that contain existential variables?
- Practical application in
  - type inference in programming languages, and
  - internet programming

Example (Fibonacci stream) (Komendantskaya et al. 15)

Streams of Fibonacci numbers, e.g. 1 1 2 3 5 8 . . . or 10 4 14 18 32 . . . , are defined by a corecursive clause that has an existential variable.

fibs(X,Y,[X | S]) ← add(X,Y,Z), fibs(Y,Z,S).
Implementation is available at
GitHub / coalp / Productive-Corecursion
Thanks!