Coalgebraic Logic Programming: implicit versus explicit resource handling

Katya Komendantskaya, joint with J. Power and M. Schmidt

School of Computing, University of Dundee, UK

CoLP’12,
8 September 2012
Algebraic and coalgebraic semantics for LP

Least fixed point of $T_P$

Algebraic fibrational semantics

Greatest fixed point of $T_P$

Coalgebraic fibrational semantics

Finite SLD-derivations

Finite and Infinite SLD-derivations
Consider the logic program below.

\[
\begin{align*}
q(b,a) & \leftarrow s(a,b) \\
q(b,a) & \leftarrow \\
s(a,b) & \leftarrow \\
p(a) & \leftarrow q(b,a), s(a,b)
\end{align*}
\]
Examples of derivations

The action of $\bar{p} : \text{At} \rightarrow C(P_f P_f)(\text{At})$ on $p(a)$

\[\leftarrow p(a)\]

\[q(b, a) \quad s(a, b)\]

\[s(a, b) \quad \square \quad \square\]

\[\square\]
Examples of derivations

The action of \( \overline{p} : \text{At} \longrightarrow C(P_f P_f)(\text{At}) \) on \( p(a) \)

Match it? - The SLD derivation

\[ \leftarrow p(a) \]
\[ q(b, a) \]
\[ s(a, b) \]
\[ s(a, b) \]
\[ \square \]

\[ \leftarrow p(a) \]
\[ \leftarrow q(b, a), s(a, b) \]
\[ \leftarrow s(a, b), s(a, b) \]
\[ \leftarrow s(a, b) \]
\[ \square \]
Examples of a derivations

The action of $\overline{p} : \text{At} \rightarrow C(P_f P_f)(\text{At})$ on $p(a)$

Match it? - The proof tree
Examples of a derivations

The action of \( \bar{p} : \text{At} \rightarrow C(P_f P_f)(\text{At}) \) on \( p(a) \)

\[
\begin{align*}
\bar{p} : \text{At} & \rightarrow C(P_f P_f)(\text{At}) \\
p(a) \rightarrow q(b, a), s(a, b) \\
q(b, a) \rightarrow s(a, b) \\
s(a, b) & \rightarrow \square
\end{align*}
\]

Match it? - The SLD tree

\[
\begin{align*}
\bar{p} : \text{At} & \rightarrow C(P_f P_f)(\text{At}) \\
p(a) \rightarrow q(b, a), s(a, b) \\
q(b, a) \rightarrow s(a, b) \\
s(a, b) \rightarrow s(a, b), s(a, b) \\
s(a, b) & \rightarrow \square
\end{align*}
\]
Is there anything at all in practice of Logic Programming that corresponds to the action of $C(P_fP_f)$-comonad?

From the examples above, it’s clear that:

**Sequential SLD-derivation**
is the least suitable...

**Proof trees**
exhibit an *and-parallelism* in derivations...

**SLD-trees**
exhibit an *or-parallelism* in derivations...
It turns out that the answer lies in the combination of the two kinds of parallelism:

\[
\bar{p} : \text{At} \rightarrow C(P_f P_f)(\text{At}) \text{ on } p(a)
\]

The and-or parallel tree

Except for... and-or trees are unsound in the first-order case.
Why unsound?

\[
\text{list(cons(x, cons(y, x)))} \\
\text{nat(x)} \\
\text{list(cons(y, x))} \\
\text{nat(y)} \\
\text{list(x)} \\
\text{nat(x_1)} \\
\text{nat(x_1)} \\
\text{nat(z_1)} \text{list(z_2)} \\
\text{nat(z_1)} \text{list(z_2)} \\
\text{nat(z_1)} \text{list(z_2)} \\
\text{nat(z_1)} \text{list(z_2)}
\]
Why unsound?

This is how we realised we had to come up our own computational model for them.
Algebraic and coalgebraic semantics for LP

Least fixed point of $T_P$

Finite SLD-derivations

Algebraic fibrational semantics

Greatest fixed point of $T_P$

Finite and Infinite SLD-derivations

Coalgebraic fibrational semantics
Algebraic and coalgebraic semantics for LP

Greatest fixed point of $T_P$

Coalgebraic fibrational semantics

Finite and Infinite SLD-derivations

Coalgebraic Logic programming
First prototype (by M. Schmidt) is available on the Web.
Recursion and Corecursion in Logic Programming

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

\[
\begin{align*}
nat(0) & \leftarrow \\
nat(s(x)) & \leftarrow \text{nat}(x) \\
\text{list}(\text{nil}) & \leftarrow \\
\text{list}(\text{cons} \ x \ y) & \leftarrow \text{nat}(x), \\
\text{list}(y) & \\
\end{align*}
\]
SLD-resolution (+ unification) is behind LP derivations.

Example

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

\[\text{list}(y)\]

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.
Things go wrong

Example

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

Example

\begin{verbatim}
bit(0) ←
bit(1) ←
stream(scons x y) ←
    bit(x), stream(y)
\end{verbatim}

No answer, as derivation never terminates.
Things go wrong

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.
Semantics may go wrong as well.
Solution - 1 [Gupta, Simon et al., 2007 - 2008]

Use normal SLD-resolution but add a new rule:
If a formula repeatedly appears as a resolvent (modulo $\alpha$-conversion), then conclude the proof.

Example

bit(0) ←
bit(1) ←
stream(scons x y) ←

\[
\begin{align*}
\text{bit}(x), \text{stream}(y) & \leftarrow \text{stream}(\text{scons}(x, y)) \\
\text{bit}(x), \text{stream}(y) & \leftarrow \text{stream}(y) \\
\text{bit}(x_1), \text{stream}(y_1) & \leftarrow \text{stream}(y) \\
\text{stream}(y_1) & \leftarrow \text{stream}(y_1) \\
\end{align*}
\]

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

Use normal SLD-resolution but add a new rule:
If a formula repeatedly appears as a resolvent (modulo $\alpha$-conversion), then conclude the proof.

Example

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

The answer is: $x/0, y/\text{cons}(x_1, y_1)$. 
Explicitly-treated corecursion

To know whether to allow (co-LP) or disallow (standard LP) infinite loops, explicit annotation is needed.

Example

\[
\begin{align*}
\text{bit}^i(0) & \leftarrow \\
\text{bit}^i(1) & \leftarrow \\
\text{stream}^c(\text{scons}(x, y)) & \leftarrow \text{bit}^i(x), \text{stream}^c(y) \\
\text{list}^i(\text{nil}) & \leftarrow \\
\text{list}^i(\text{cons}(x, y)) & \leftarrow \text{bit}^i(x), \text{list}^i(y)
\end{align*}
\]
Drawbacks:

- some predicates may behave inductively or coinductively depending on the arguments provided, and such cases need to be resolved dynamically, and not statically; in which case mere predicate annotation fails.
- ... cannot mix induction and coinduction. — All clauses need to be marked as inductive or coinductive in advance.
- Can deal only with restricted sort of structures — the ones having finite regular pattern.

Example

0:: 1:: 0:: 1:: 0:: ... may be captured by such programs.
π represented as a stream may not.

- the derivation itself is not really a corecursive process.
... arose from considerations valid for coalgebraic semantics of logic programs
Solution - 2. Coinductive LP in [Komendantskaya, Power CSL’11]

... arose from considerations valid for coalgebraic semantics of logic programs
Technically:
- features parallel derivations;
- it is not a standard SLD-resolution any more, e.g. unification is restricted to term matching;
Coinductive trees

Definition

Let $P$ be a logic program and $G \leftarrow A$ be an atomic goal. The coinductive derivation tree for $A$ is a tree $T$ satisfying the following properties.

- $A$ is the root of $T$.
- Each node in $T$ is either an and-node or an or-node.
- Each or-node is given by $\bullet$.
- Each and-node is an atom.
- For every and-node $A'$ occurring in $T$, there exist exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in $P$ (a clause $C_i$ has the form $B_i \leftarrow B_{i1}^1, \ldots, B_{in_i}^i$, for some $n_i$), such that $A' = B_1 \theta_1 = \ldots = B_m \theta_m$, for some substitutions $\theta_1, \ldots, \theta_m$, then $A'$ has exactly $m$ children given by or-nodes, such that, for every $i \in m$, the $i$th or-node has $n_i$ children given by and-nodes $B_{i1}^i \theta_i, \ldots, B_{in_i}^i \theta_i$. 
An Example

\[ \theta_1 \rightarrow \]

\text{stream}(x)
An Example

\[ \theta_1 \rightarrow \]
\[ \text{stream}(x) \]

\[ \theta_2 \rightarrow \ldots \rightarrow \theta_3 \rightarrow \]
\[ \text{stream} \left( \text{scons}(z, y) \right) \]

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

Note that transitions \( \theta \) may be determined in a number of ways:
- using mgus;
- non-deterministically;
- randomly;
- in a distributed/parallel manner.
An Example

\[
\begin{aligned}
\theta_1 & \rightarrow \\
\text{stream}(x) & \quad \theta_2 \quad \ldots \quad \theta_3 \\
\text{stream(scons}(z, y)) & \\
\text{bit}(z) & \quad \text{stream}(y)
\end{aligned}
\]

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

- using mgus;
- non-deterministically;
- randomly;
- in a distributed/parallel manner.
An Example

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

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

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

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

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

Answers for x: \text{cons}(z, y) and \text{cons}(0, \text{cons}(y_1, z_1)). It’s a different (corecursive) approach to what a “terminating derivation” is.
Solution - 2. Coinductive LP in [Komendantskaya, Power CSL’11]

Advantages
- Works uniformly for both inductive and coinductive definitions, without having to classify the two into disjoint sets;
- in spirit of corecursion, derivations may feature an infinite number of finite structures.
- there does not have to be regularity or repeating patterns in derivations.
Guarding corecursion

(Co)-Recursion is always dangerous:
... and needs to be guarded against infinite loops. Both in FP and LP, such guards can be given semantically or syntactically ("guardness-by-construction").

Example
This program is not guarded-by-constructors:

1. `connected(x,x) ←`

2. `connected(x,y) ← edge(x,z), connected(z,y).`

... and it will produce infinite coinductive trees.
Infinite forests of infinite trees:

connected(0, z)

edge(0, y) connected(y, z))

edge(y, y_1) connected(y_1, z)

edge(s(y), y_1) connected(y_1, z_1)

edge(0, z)

edge(0, s(y)) connected(s(y), z))

edge(s(y), s(y_1)) connected(s(y_1), z_1)

edge(0, s(s(y))) connected(s(s(y)), z))

edge(s(s(y)), y_1) connected(y_1, z_1)
Guarding corecursion

(Co)-Recursion is always dangerous:

... and needs to be guarded against infinite loops. Both in FP and LP, such guards can be given semantically or syntactically (”guardeness-by-construction”).

Example

This program is not guarded-by-constructors:

1. connected(x,x) ←

2. connected(x,y) ← edge(x,z), connected(z,y).

... and it will produce infinite coinductive trees.

In reality, such programs will be disallowed by the termination checker, and will need to be reformulated.
Guarding corecursion, for example:

Example

\[
\begin{align*}
\text{connected}(X, \text{cons}(\text{Node}, \text{Path})) & \leftarrow \text{edge}(X, \text{Node}), \text{connected}(\text{Node}, \text{Path}) \\
\text{connected}(X, \text{nil}) & \leftarrow \\
\text{edge}(0, 0) & \leftarrow \\
\text{edge}(X, s(X)) & \leftarrow
\end{align*}
\]
Guarding corecursion, for example:

Example

\[
\begin{align*}
\text{connected}(X, \text{cons}(Node, Path)) & \leftarrow \text{edge}(X, Node), \text{connected}(Node, Path) \\
\text{connected}(X, \text{nil}) & \leftarrow \\
\text{edge}(0, 0) & \leftarrow \\
\text{edge}(X, s(X)) & \leftarrow \\
\end{align*}
\]

\[
\begin{align*}
\text{conn}(0, \text{cons}(y, z)) & \rightarrow \text{conn}(0, \text{cons}(s0, z)) & \rightarrow \text{conn}(0, \text{cons}(s0, \text{nil})) \\
\text{edge}(0, y) \ \text{conn}(y, z) & \quad \text{edge}(0, s0) \ \text{conn}(s0, z) & \quad \text{edge}(0, s0) \ \text{conn}(s0, \text{nil}) \\
\end{align*}
\]
More discipline?

Adapting this sort of programming discipline from lazy functional languages to LP may have its advantages. E.g., it will equally guard against programs that induce infinite SLD-derivations:

Example

1. \texttt{connected(x,y) ← connected(z,y), edge(x,z)}
2. \texttt{connected(x,x) ←}

While currently, it is up to a programmer to manually weed-out such cases.
Corecursion guarding parallelism:
Corecursion FREEING! parallelism:

Unification and SLD-resolution are P-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:

\[
\text{list} \left(\text{cons} \left(x, \text{cons} \left(y, x\right)\right)\right)\n\]

\[
\text{nat} \left(x\right)\n\]

\[
\square \text{nat} \left(x_1\right)\ldots\text{list} \left(\text{cons} \left(y, x\right)\right)\n\]

\[
\text{nat} \left(y\right)\n\]

\[
\square \text{nat} \left(x_1\right)\ldots\text{list} \left(z_2\right)\ldots\text{list} \left(z_2\right)\ldots\]
Corecursion FREEING! parallelism:

Unification and SLD-resolution are P-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:
Unification and SLD-resolution are $P$-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:

```
list(cons(x, cons(y, x)))
```

```
nat(x)          list(cons(y, x))
    /\               /
   /   \            /   \
  nat(x) nat(y)    list(x)
               /\   /
              /   \ /
             /     \ /
            /       \ /
           /         \ /
          nat(x_1) nat(x_1) nat(z_1) list(z_2)
```

Katya (Dundee) Coalgebraic Logic Programming: implicit versus explicit resource handling CoLP’12 31 / 34
Now, by the same lazy corecursive derivation:

```
list(c(x, c(y, x))) → list(c(0, c(y, 0))) → list(c(0, c(0, 0)))

nat(x) list(c(y, x)) → nat(0) list(c(y, 0)) → nat(0) list(c(0, 0))

nat(y) list(x) → □ nat(y) list(0) → □ nat(0) list(0) → □
```
Corecursion FREEING! parallelism:

Seq no more!

Where was unification, we bring term-matching!
Where was SLD-derivations, we bring corecursive derivations!
Both are parallelisable, and LP is free.

Variable Synchronization?
... is no longer in power...
Variables can live their own lazy corecursive lives.
Corecursion FREEING! parallelism:

Seq no more!

- Where was unification, we bring term-matching!
Corecursion FREEING! parallelism:

Seq no more!

- Where was unification, we bring term-matching!
- Where was SLD-derivations, we bring corecursive derivations!

Both are parallelisable, and LP is free.
Corecursion FREEING! parallelism:

Seq no more!
- Where was unification, we bring term-matching!
- Where was SLD-derivations, we bring corecursive derivations!

Both are parallelisable, and LP is free.

Variable Synchronization?
Corecursion FREEING! parallelism:

Seq no more!

- Where was unification, we bring term-matching!
- Where was SLD-derivations, we bring corecursive derivations!

Both are parallelisable, and LP is free.

Variable Synchronization? ... is no longer in power...
Corecursion FREEING! parallelism:

Seq no more!
- Where was unification, we bring term-matching!
- Where was SLD-derivations, we bring corecursive derivations!

Both are parallelisable, and LP is free.

Variable Synchronization? ... is no longer in ... in use.
Corecursion FREEING! parallelism:

Seq no more!

- Where was unification, we bring term-matching!
- Where was SLD-derivations, we bring corecursive derivations!

Both are parallelisable, and LP is free.

Variable Synchronization? ... is no longer in ... in use.

Variables can live their own lazy corecursive lives.
[Instead of] Conclusions...

So, what happened to the old Rule?
[Instead of] Conclusions...

So, what happened to the old Rule?

*Logic Programs = Logic + Control*

[Kowalski 1979]
So, what happened to the old Rule?

*Logic Programs = Logic + Control*

[Kowalski 1979]

We have new rules:

Corecursive Programs: **LOGIC is Control**
So, what happened to the old Rule?

*Logic Programs = Logic + Control*

[Kowalski 1979]

We have new rules:

**Corecursive Programs:** LOGIC is Control

... long live LOGIC!
So, what happened to the old Rule?

\textit{Logic Programs = Logic + Control}  
\[\text{[Kowalski 1979]}\]

We have new rules:

\textbf{Corecursive Programs: LOGIC is Control}

... long live LOGIC!

The End.