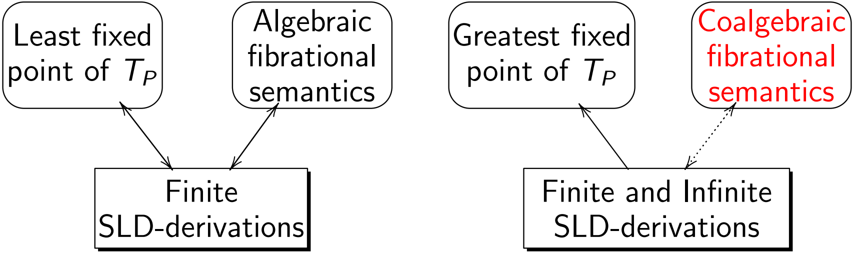
Coalgebraic Logic Programming: implicit versus explicit resource handling

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



Example

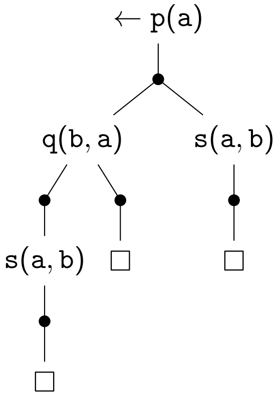
Example

Consider the logic program below .

$$
\begin{aligned}
\mathtt{q(b,a)} &\leftarrow \mathtt{s(a,b)} \\
\mathtt{q(b,a)} &\leftarrow \\
\mathtt{s(a,b)} &\leftarrow \\
\mathtt{p(a)} &\leftarrow \mathtt{q(b,a)}, \mathtt{s(a,b)}
\end{aligned}
$$

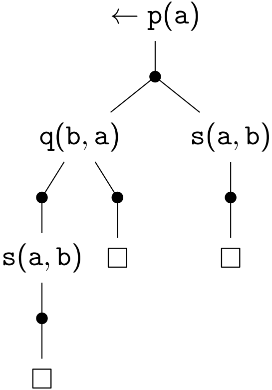
Examples of derivations

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



Examples of derivations

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

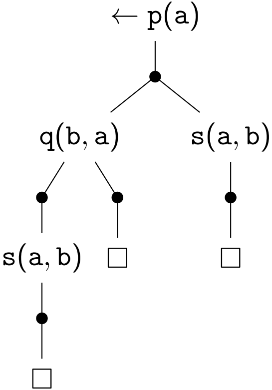


Match it? - The SLD derivation

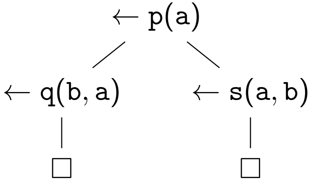
$$
\begin{array}{c}
\leftarrow \mathtt{p(a)} \\
| \\
\leftarrow \mathtt{q(b,a)}, \mathtt{s(a,b)} \\
| \\
\leftarrow \mathtt{s(a,b)}, \mathtt{s(a,b)} \\
| \\
\leftarrow \mathtt{s(a,b)} \\
| \\
\square
\end{array}
$$

Examples of a derivations

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

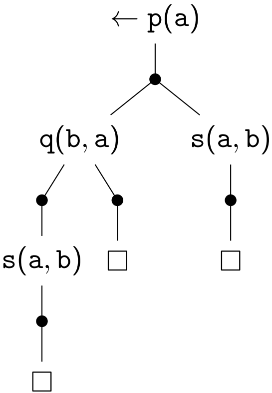


Match it? - The *proof tree*

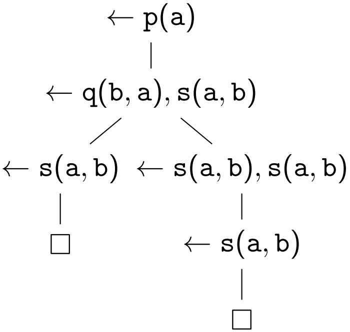


Examples of a derivations

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



Match it? - The SLD tree



Is there anything at all in practice of Logic Programming that corresponds to the action of $C(P_f P_f)$-comonad?

From the examples above, it's clear that:

Sequential SLD-derivation

is the least suitable...

Proof trees

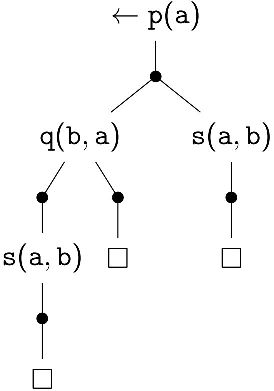
exhibit an and-parallelism in derivations...

SLD-trees

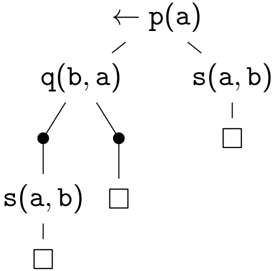
exhibit an or-parallelism in derivations...

It turns out that the answer lies in the combination of the two kinds of parallelism:

$\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$ on p(a)

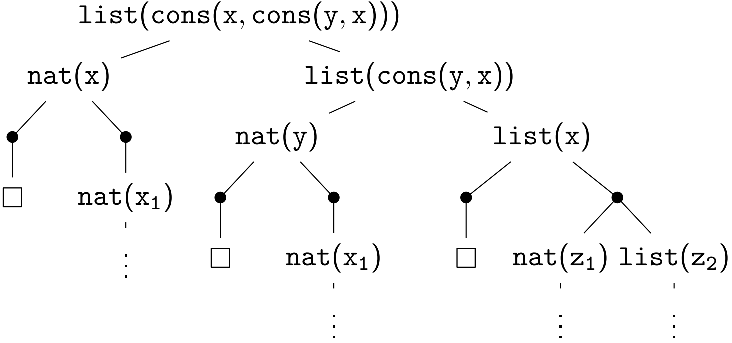


The and-or parallel tree

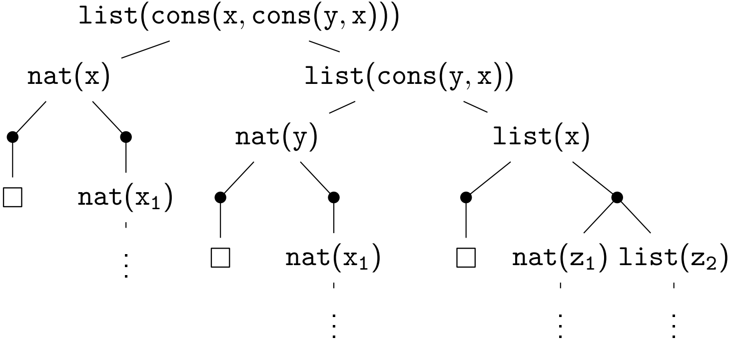


Except for... and-or trees are unsound in the first-order case.

Why unsound?

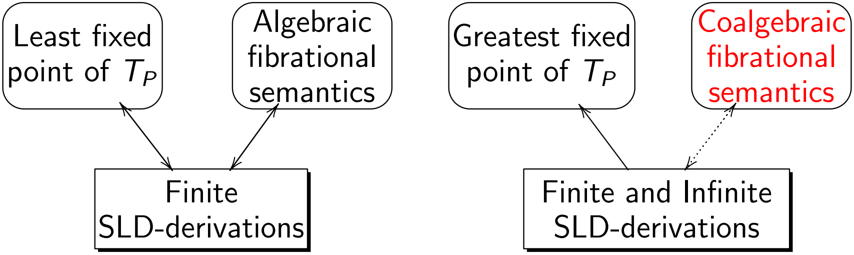

list(cons(x, cons(y, x)))
nat(x)
list(cons(y, x))
nat(y)
list(x)
□
nat(x1)
□
nat(x1)
□
nat(z1)
list(z2)

Why unsound?

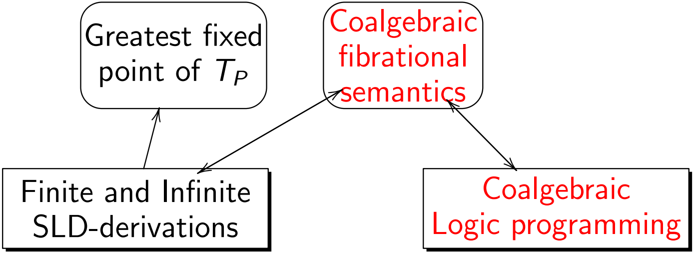


This is how we realised we had to come up our own computational model for them.

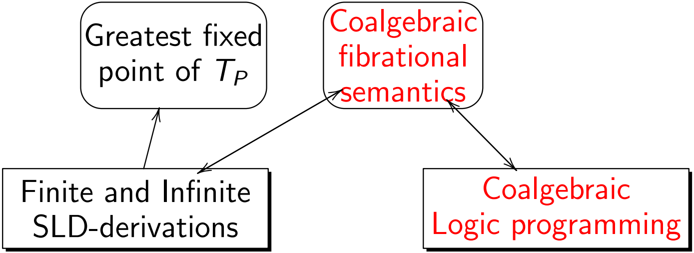
Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



First prototype (by M. Schmidt) is available on the Web.

Recursion and Corecursion in Logic Programming

Example

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

Example

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

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

Example

$\mathtt{nat(0)} \leftarrow$
$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$
$\mathtt{list(nil)} \leftarrow$
$\mathtt{list(cons\ x\ y)} \leftarrow \mathtt{nat(x)},$
$\mathtt{list(y)}$

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

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)
```

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

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

Example

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

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

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

Things go wrong

Example

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

Things go wrong

Example

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

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

No answer, as derivation never terminates.

Things go wrong

Example

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

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

No answer, as derivation never terminates.

Semantics may go wrong as well.

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

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 α-conversion), then conclude the proof.

Example

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

$$\begin{array}{c} \leftarrow \mathtt{stream}(\mathtt{scons}(x, y)) \\ | \\ \leftarrow \mathtt{bit}(x), \mathtt{stream}(y) \\ | \\ \leftarrow \mathtt{stream}(y) \\ | \\ \leftarrow \mathtt{bit}(x_1), \mathtt{stream}(y_1) \\ | \\ \leftarrow \mathtt{stream}(y_1) \\ | \\ \square \end{array}$$

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 α-conversion), then conclude the proof.

Example

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

The answer is: $x/0$, $y/cons(x_1, y_1)$.

$$
\begin{array}{c}
\leftarrow \mathtt{stream}(\mathtt{scons}(\mathtt{x}, \mathtt{y})) \\
| \\
\leftarrow \mathtt{bit}(\mathtt{x}), \mathtt{stream}(\mathtt{y}) \\
| \\
\leftarrow \mathtt{stream}(\mathtt{y}) \\
| \\
\leftarrow \mathtt{bit}(\mathtt{x}_1), \mathtt{stream}(\mathtt{y}_1) \\
| \\
\leftarrow \mathtt{stream}(\mathtt{y}_1) \\
| \\
\square
\end{array}
$$

Explicitly-treated corecursion

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

Example

$$
\begin{array}{rcl}
\mathtt{bit}^i(0) & \leftarrow & \\
\mathtt{bit}^i(1) & \leftarrow & \\
\mathtt{stream}^c(\mathtt{scons}(x, y)) & \leftarrow & \mathtt{bit}^i(x), \mathtt{stream}^c(y) \\
\mathtt{list}^i(\mathtt{nil}) & \leftarrow & \\
\mathtt{list}^i(\mathtt{cons}(x, y)) & \leftarrow & \mathtt{bit}^i(x), \mathtt{list}^i(y)
\end{array}
$$

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.

Solution - 2. Coinductive LP in [Komendantskaya, Power CSL'11]

- ... 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 $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i), such that $A' = B_1\theta_1 = ... = B_m\theta_m$, 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 ith or-node has n children given by and-nodes $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.

An Example

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

An Example

$\xrightarrow{\theta_1}$

stream(x)

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

stream(scons(z, y))

bit(z)   stream(y)

An Example

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

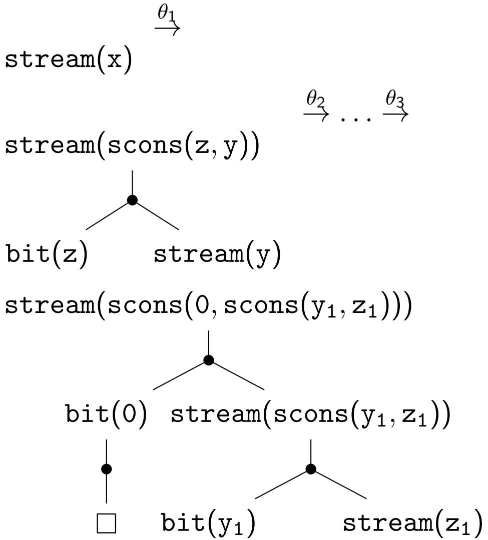
$\mathtt{stream(scons(z, y))} \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_3}$

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

Note that transitions θ may be determined in a number of ways:

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

An Example



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

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 ("guardeness-by-construction").

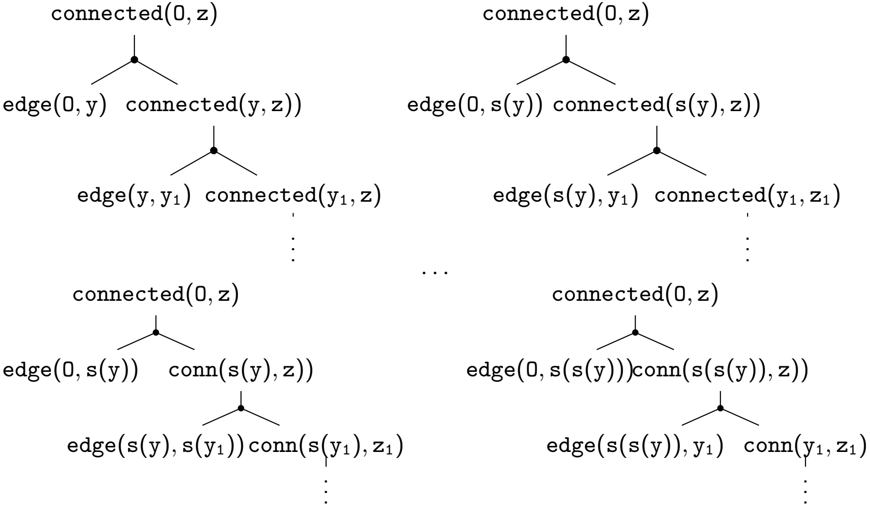
Example

This program is not guarded-by-constructors:

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

... and it will produce infinite coinductive trees.

Infinite forests of infinite trees:



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. $\texttt{connected(x,x)} \leftarrow$
2. $\texttt{connected(x,y)} \leftarrow \texttt{edge(x,z)}, \texttt{connected(z,y)}.$

... 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

$$
\begin{aligned}
\mathtt{connected}(X, cons(Node, Path)) &\leftarrow \mathtt{edge}(X, Node), \mathtt{connected}(Node, Path) \\
\mathtt{connected}(X, nil) &\leftarrow \\
\mathtt{edge}(0, 0) &\leftarrow \\
\mathtt{edge}(X, s(X)) &\leftarrow
\end{aligned}
$$

Guarding corecursion, for example:

Example

$$\begin{aligned}
\texttt{connected}(X, cons(Node, Path)) &\leftarrow \texttt{edge}(X, Node), \texttt{connected}(Node, Path) \\
\texttt{connected}(X, nil) &\leftarrow \\
\texttt{edge}(0, 0) &\leftarrow \\
\texttt{edge}(X, s(X)) &\leftarrow
\end{aligned}$$

$$\begin{array}{ccccc}
\begin{array}{c} \texttt{conn}(0, \texttt{cons}(\texttt{y}, \texttt{z})) \\ \texttt{edge}(0, \texttt{y}) \quad \texttt{conn}(\texttt{y}, \texttt{z})) \end{array}
& \rightarrow &
\begin{array}{c} \texttt{conn}(0, \texttt{cons}(\texttt{s0}, \texttt{z})) \\ \texttt{edge}(0, \texttt{s0}) \quad \texttt{conn}(\texttt{s0}, \texttt{z})) \\ \Box \end{array}
& \rightarrow &
\begin{array}{c} \texttt{conn}(0, \texttt{cons}(\texttt{s0}, \texttt{nil})) \\ \texttt{edge}(0, \texttt{s0}) \quad \texttt{conn}(\texttt{s0}, \texttt{nil})) \\ \Box \qquad\qquad \Box \end{array}
\end{array}$$

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. $\texttt{connected(x,y)} \leftarrow \texttt{connected(z,y),edge(x,z)}$
2. $\texttt{connected(x,x)} \leftarrow$

While currently, it is up to a programmer to manually weed-out such cases.

Corecursion guarding parallelism:

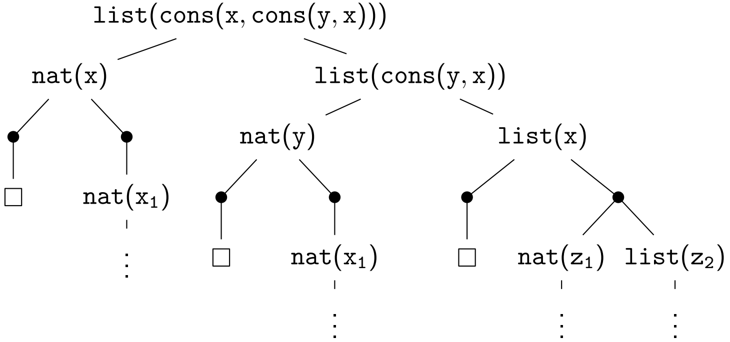
Corecursion FREEING! parallelism:

Corecursion FREEING! parallelism:

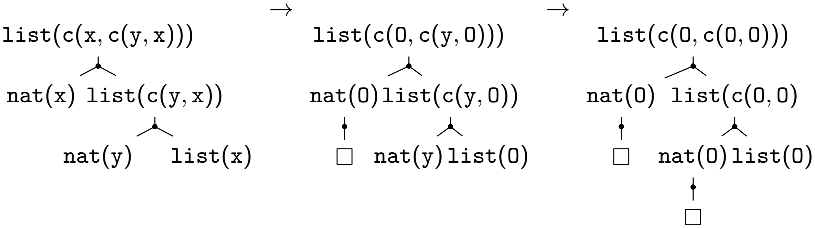
Unification and SLD-resolution are P-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:

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:



Now, by the same lazy corecursive derivation:



Corecursion FREEING! parallelism:

Seq no more!

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]

[Instead of] Conclusions...

So, what happened to the old Rule?

Logic Programs $=$ *Logic* $+$ *Control*

[Kowalski 1979]

We have new rules:

Corecursive Programs: **LOGIC is Control**

[Instead of] Conclusions...

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!

[Instead of] Conclusions...

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!

The End.