Coalgebraic Logic Programming for Type Inference

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

School of Computing, University of Dundee, UK

MSP-RAD'13,
6 June 2013

Outline

Outline

1. Motivation: LP in Type inference
2. Two old problems of LP:
   - Parallelism
   - Corecursion

Outline

1. Motivation: LP in Type inference
2. Two old problems of LP:
   - Parallelism
   - Corecursion
3. How Coalgebra Saved the Day

Outline

1. Motivation: LP in Type inference
2. Two old problems of LP:
   - Parallelism
   - Corecursion
3. How Coalgebra Saved the Day
4. What does this matter for type inference?

Milner, 1978 [Mil78]

"A theory of Type Polymorphism in Programming"

Milner, 1978 [Mil78]

"A theory of Type Polymorphism in Programming"
An elegant match between polymorphic λ-calculus and type inference by means of Robinson's unification/resolution algorithm.
One made another possible...
Principal type exists, and the Robinson's algorithm is [necessary and] sufficient to compute it.

Constraints and LP in Type inference

- Hindley-Milner Type inference [Milner78, Damas&Milner82] (used in ML, OCAML, Haskel, and some other languages) was based on first-order unification, and simultaneous generation and solving of constraints.

Constraints and LP in Type inference

- Hindley-Milner Type inference [Milner78, Damas&Milner82] (used in ML, OCAML, Haskel, and some other languages) was based on first-order unification, and simultaneous generation and solving of constraints.
- ... was generalised in Haskell by [Odersky, Sulzmann, Wehr 1999] to HM(X) – by means of generalising from Herbrand domains to arbitrary constraint domains (hence “X”).

Constraints and LP in Type inference

- Hindley-Milner Type inference [Milner78, Damas&Milner82] (used in ML, OCAML, Haskel, and some other languages) was based on first-order unification, and simultaneous generation and solving of constraints.
- ... was generalised in Haskell by [Odersky, Sulzmann, Wehr 1999] to HM(X) – by means of generalising from Herbrand domains to arbitrary constraint domains (hence "X").
- HM(X) type inference was shown to be equivalent to solving CLP(X) – constraint logic programming (with arbitrary constraint domains), in a very elegant paper [Sulzmann, Stuckey 2008]. [Constraint solving and constraint generation are separated.]

Constraints and LP in Type inference

- Hindley-Milner Type inference [Milner78, Damas&Milner82] (used in ML, OCAML, Haskel, and some other languages) was based on first-order unification, and simultaneous generation and solving of constraints.
- ... was generalised in Haskell by [Odersky, Sulzmann, Wehr 1999] to HM(X) – by means of generalising from Herbrand domains to arbitrary constraint domains (hence "X").
- HM(X) type inference was shown to be equivalent to solving CLP(X) – constraint logic programming (with arbitrary constraint domains), in a very elegant paper [Sulzmann, Stuckey 2008]. [Constraint solving and constraint generation are separated.]
- In fact, there have been publications on type inference in between, e.g. [Remy & Potier], but not in the direction of LP.

Trend in type inference:

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

- *Dependent Types* [BC02],
- *Type Classes* [WB89],
- *Generalised Algebraic Types* (GADTs) [JVWW06]
- *Dependent Type Classes* [SO08] and
- *Canonical Structures* [GZND11].

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

Trend in type inference:

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

- *Dependent Types* [BC02],
- *Type Classes* [WB89],
- *Generalised Algebraic Types* (GADTs) [JVWW06]
- *Dependent Type Classes* [SO08] and
- *Canonical Structures* [GZND11].

Milner-style decidable type inference does not always suffice (e.g. the *principal type* may no longer exist), and TI requires computation additional to compile-time. Implementations of new type inference algorithms include a variety of first-order decision procedures. notably Unification and Logic Programming (LP) [JVWW06], Constraint LP [OSW99, SS08, SJSV09, VJSS11], LP embedded into interactive tactics (Coq's *eauto*) [SO08], and LP supplemented by rewriting [GZND11].

Trend in type inference:

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

- *Dependent Types* [BC02],
- *Type Classes* [WB89],
- *Generalised Algebraic Types* (GADTs) [JVWW06]
- *Dependent Type Classes* [SO08] and
- *Canonical Structures* [GZND11].

Milner-style decidable type inference does not always suffice (e.g. the *principal type* may no longer exist), and TI requires computation additional to compile-time. Implementations of new type inference algorithms include a variety of first-order decision procedures. notably Unification and Logic Programming (LP) [JVWW06], Constraint LP [OSW99, SS08, SJSV09, VJSS11], LP embedded into interactive tactics (Coq's *eauto*) [SO08], and LP supplemented by rewriting [GZND11]. The latter claims that, for richer type systems, LP-style type inference is more efficient and natural than traditional tactic-driven proof development.

Recursion and Corecursion in Logic Programming

Example

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

Example

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

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

Example

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

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

$$|$$

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

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)
```

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

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.

Outline

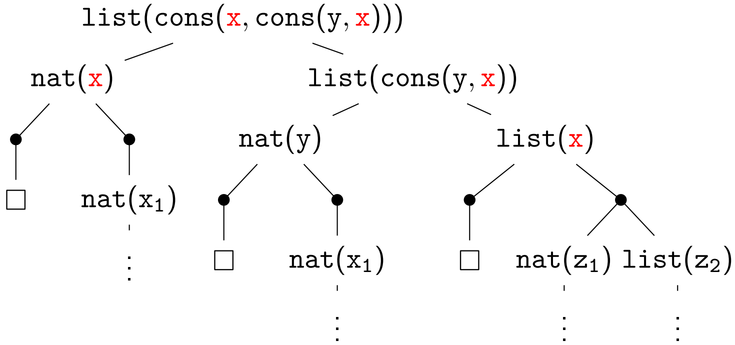
1. Motivation: LP in Type inference
2. Two old problems of LP:
    - Parallelism
    - Corecursion
3. How Coalgebra Saved the Day
4. What does this matter for type inference?

Let's parallelise it!

[A popular trend in the 90s...]

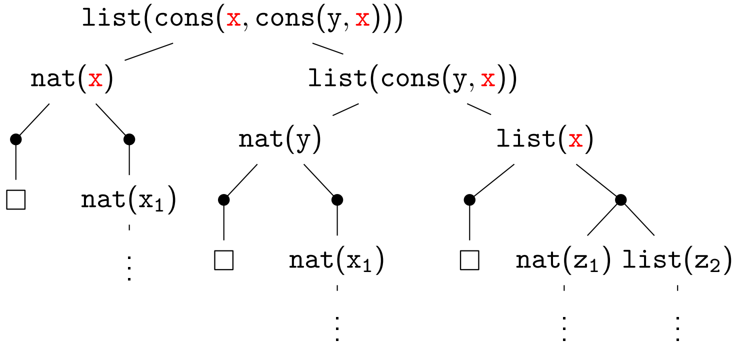
Let's parallelise it!

[A popular trend in the 90s...] Unsound and-or parallelism:



Let's parallelise it!

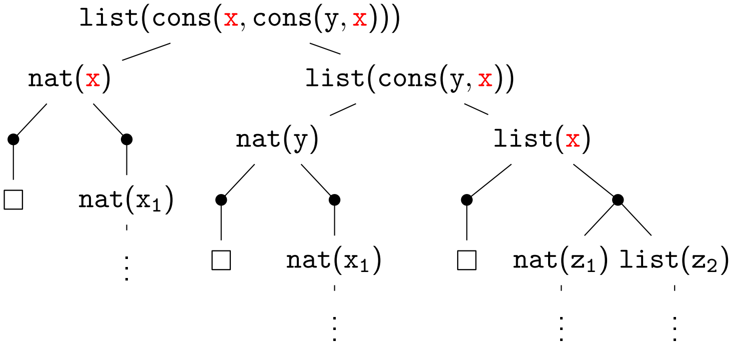
[A popular trend in the 90s...] Unsound and-or parallelism:



If unsound – lets synchronize variable substitution! – many engineering solutions...

Let's parallelise it!

[A popular trend in the 90s...] Unsound and-or parallelism:



If unsound – lets synchronize variable substitution! – many engineering solutions... but basically still a problem! [Big Survey [GPA+12]

Outline

Corecursion in LP?

Example

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

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

$\texttt{stream(scons x y)} \leftarrow$

$\qquad\qquad \texttt{bit(x), stream(y)}$

Corecursion in LP?

Example

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

No answer, as derivation never terminates.

Corecursion in LP?

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.

$$\begin{array}{c}
\leftarrow \mathtt{stream}(\mathtt{scons}(\mathtt{x},\mathtt{y})) \\
| \\
\leftarrow \mathtt{bit}(\mathtt{x}), \mathtt{stream}(\mathtt{y}) \\
| \\
\leftarrow \mathtt{stream}(\mathtt{y}) \\
| \\
\leftarrow \mathtt{bit}(\mathtt{x}_1), \mathtt{stream}(\mathtt{y}_1) \\
| \\
\leftarrow \mathtt{stream}(\mathtt{y}_1) \\
| \\
\leftarrow \mathtt{bit}(\mathtt{x}_2), \mathtt{stream}(\mathtt{y}_2) \\
| \\
\leftarrow \mathtt{stream}(\mathtt{y}_2) \\
| \\
\vdots
\end{array}$$

Solution - 1 [Gupta, Simon et al., [Ge07, Se07]

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

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

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

$\texttt{stream(scons x y)} \leftarrow$

$\qquad\qquad \texttt{bit(x), stream(y)}$

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

Solution - 1 [Gupta, Simon et al., [Ge07, Se07]

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

$$\leftarrow \mathtt{stream}(\mathtt{scons}(\mathtt{x},\mathtt{y}))$$

$$\leftarrow \mathtt{bit}(\mathtt{x}), \mathtt{stream}(\mathtt{y})$$

$$\leftarrow \mathtt{stream}(\mathtt{y})$$

$$\leftarrow \mathtt{bit}(\mathtt{x}_1), \mathtt{stream}(\mathtt{y}_1)$$

$$\leftarrow \mathtt{stream}(\mathtt{y}_1)$$

$$\square$$

Explicitly-treated corecursion

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

Example

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

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

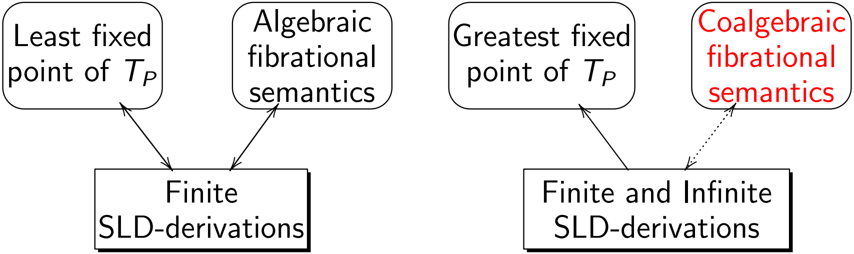
`0::  1::  0::  1::  0::  ...` may be captured by such programs.
π represented as a stream may not.

- the derivation itself is not really a corecursive process.

Outline

Algebraic and coalgebraic semantics for LP

... with John Power, 2008 - 2012. Initially, we were not aware of the two implementation trends above...



Coalgebraic Analysis of derivations in Logic Programs

Given a variable-free logic program P, let At be the set of all atoms appearing in P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(\mathrm{At}))$ sends an atom A to the set of bodies of those clauses in P with head A, each body being viewed as the set of atoms that appear in it.

Coalgebraic Analysis of derivations in Logic Programs

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

$$\ldots \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At} \times P_f P_f(\mathrm{At})) \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At}) \longrightarrow \mathrm{At}.$$

This chain has length ω.
We inductively define the objects $\mathrm{At}_0 = \mathrm{At}$ and $\mathrm{At}_{n+1} = \mathrm{At} \times P_f P_f \mathrm{At}_n$, and the cone

$$\begin{aligned} p_0 &= id : \mathrm{At} \longrightarrow \mathrm{At}(= \mathrm{At}_0) \\ p_{n+1} &= \langle id, P_f P_f(p_n) \circ p \rangle : \mathrm{At} \longrightarrow \mathrm{At} \times P_f P_f \mathrm{At}_n(= \mathrm{At}_{n+1}) \end{aligned}$$

and the limit determines the required coalgebra $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$.

Example

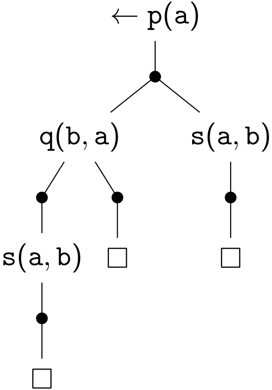
Example

Consider the logic program below .

$$
\begin{array}{rcl}
\texttt{q(b,a)} & \leftarrow & \texttt{s(a,b)} \\
\texttt{q(b,a)} & \leftarrow & \\
\texttt{s(a,b)} & \leftarrow & \\
\texttt{p(a)} & \leftarrow & \texttt{q(b,a)}, \texttt{s(a,b)}
\end{array}
$$

Examples of derivations

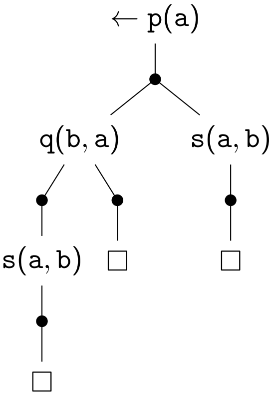
The action of
$\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$ on
p(a)



Examples of derivations

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

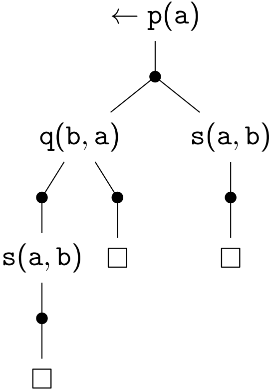
Match it? - The SLD derivation



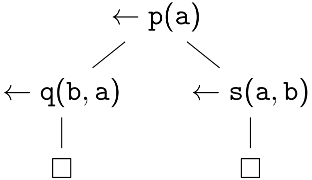
$$\leftarrow \mathtt{p(a)}$$
$$|$$
$$\leftarrow \mathtt{q(b,a), s(a,b)}$$
$$|$$
$$\leftarrow \mathtt{s(a,b), s(a,b)}$$
$$|$$
$$\leftarrow \mathtt{s(a,b)}$$
$$|$$
$$\square$$

Examples of a derivations

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



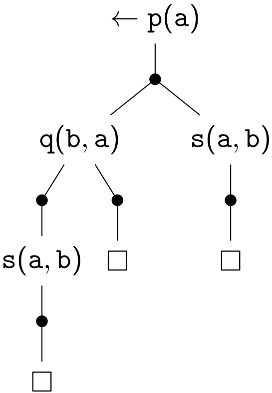
Match it? - The *proof tree*

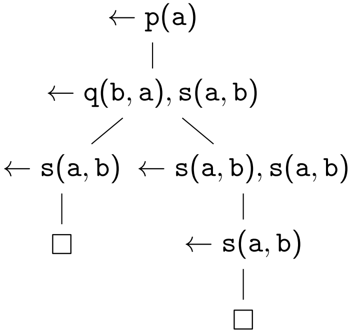


Examples of a derivations

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

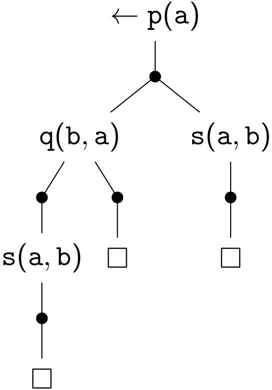
exhibit an and-parallelism in derivations...

SLD-trees

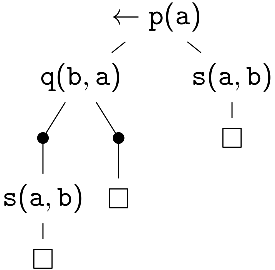
exhibit an or-parallelism in derivations...

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.

Lawvere theories and the first-order signature Σ

A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these are called *constants*.

Given a signature Σ, construct the Lawvere theory $\mathcal{L}_\Sigma$:

- Define the set $\mathrm{ob}(\mathcal{L}_\Sigma)$ to be the set of natural numbers.
- For each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
- Define $\mathrm{ob}(\mathcal{L}_\Sigma)(n, m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
- Define composition in $\mathcal{L}_\Sigma$ by substitution.

Example of Lawvere theory generated by a LP

Example

The constants 0 and `nil` are modelled by maps from 0 to 1 in $\mathcal{L}_\Sigma$, `s` is modelled by a map from 1 to 1, and `cons` is modelled by a map from 2 to 1. The term `s(0)` is therefore modelled by the map from 0 to 1 given by the composite of the maps modelling `s` and 0; similarly for the term `s(nil)`, although the latter does not make semantic sense.

We use Lawvere Theory $\mathcal{L}_\Sigma$ intead of set At

Some modifications are needed:

- we need to extend *Set* to *Poset*,
- natural transformations to *lax natural transformations*, and
- replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability).

We use Lawvere Theory $\mathcal{L}_\Sigma$ intead of set At

Some modifications are needed:

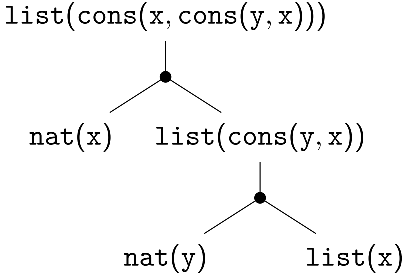
- we need to extend *Set* to *Poset*,
- natural transformations to *lax natural transformations*, and
- replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability).

Then $p : At \longrightarrow P_c P_f At$ gives a $Lax(\mathcal{L}_\Sigma^{op}, P_c P_f)$-coalgebra structure on At; and p determines a $Lax(\mathcal{L}_\Sigma^{op}, C(P_c P_f))$-coalgebra structure $\bar{p} : At \longrightarrow C(P_c P_f)(At)$.

Examples of first-order coinductive trees determined by the semantics:

$A(x, y) \in At(2)$

Then apply At to the map $(s, s) : 1 \to 2$ in $\mathcal{L}_\Sigma$



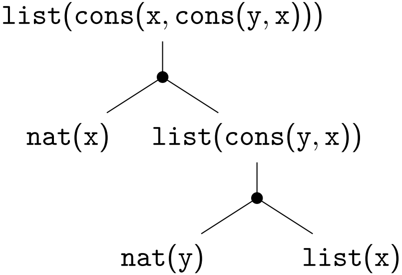
Examples of first-order coinductive trees determined by the semantics:

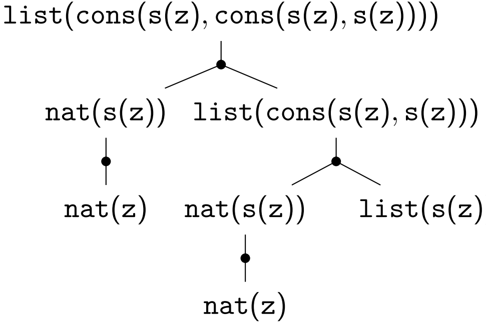
$A(x, y) \in At(2)$

Then apply At to the map $(s, s) : 1 \to 2$ in $\mathcal{L}_\Sigma$

$A(z) \in At(1)$

$At((s, s))(A(x, y))$ is an element of $P_c P_f At(1)$.

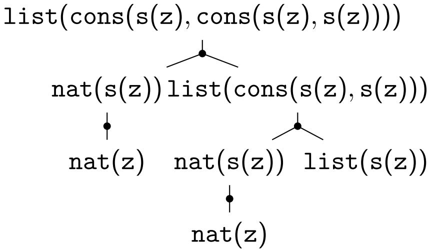




Examples of first-order coinductive trees determined by the semantics:

$A(z) \in At(1)$

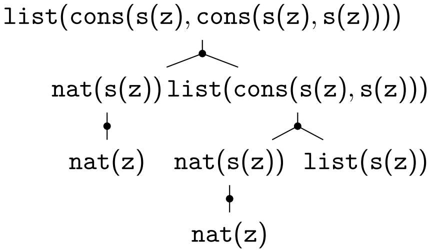
Then apply At to the map $O : 0 \to 1$ in $\mathcal{L}_\Sigma$.



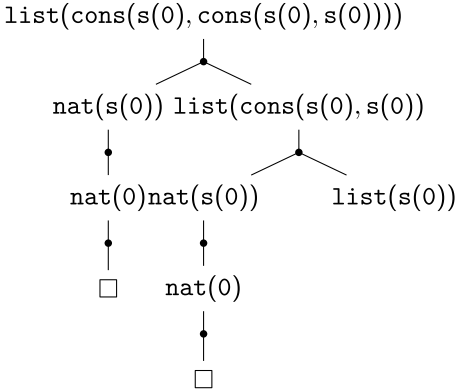
Examples of first-order coinductive trees determined by the semantics:

$A(z) \in At(1)$

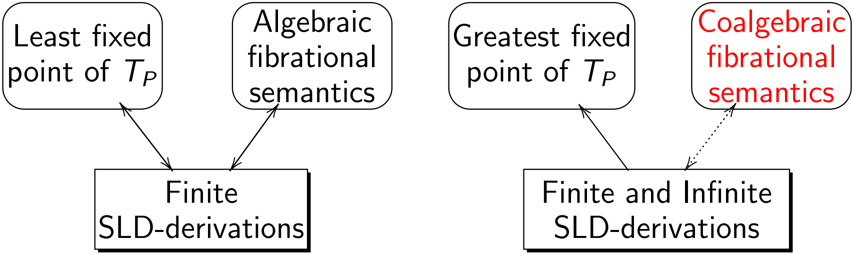
Then apply At to the map $O : 0 \to 1$ in $\mathcal{L}_\Sigma$.



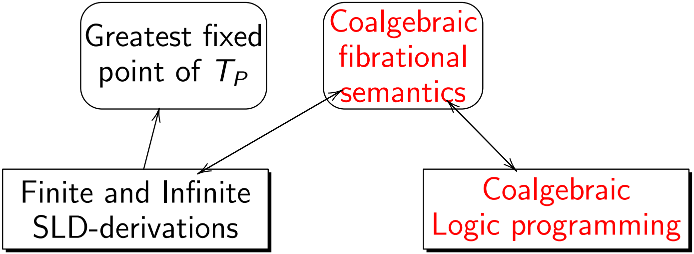
$pAt(0)At((s, s))(A(x, y))$



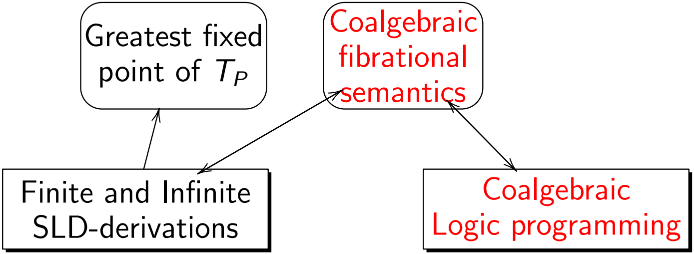
Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



First sequential (in PROLOG) and parallel (in GO) prototypes (by M. Schmidt) are available on the Web: www.computing.dundee.ac.uk/staff/katya/coalp.

Coalgebraic Logic Programming (CoALP)

- ... arose from considerations valid for coalgebraic semantics of logic programs

Coalgebraic Logic Programming (CoALP)

- … 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

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

An Example

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

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

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

An Example

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

stream(scons(z, y))

bit(z)　　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

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

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

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

$\mathtt{stream(scons(0, scons(y_1, z_1)))}$

$\mathtt{bit(0)} \quad \mathtt{stream(scons(y_1, z_1))}$

$\square \quad \mathtt{bit(y_1)} \qquad \mathtt{stream(z_1)}$

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.

CoALP's features

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").

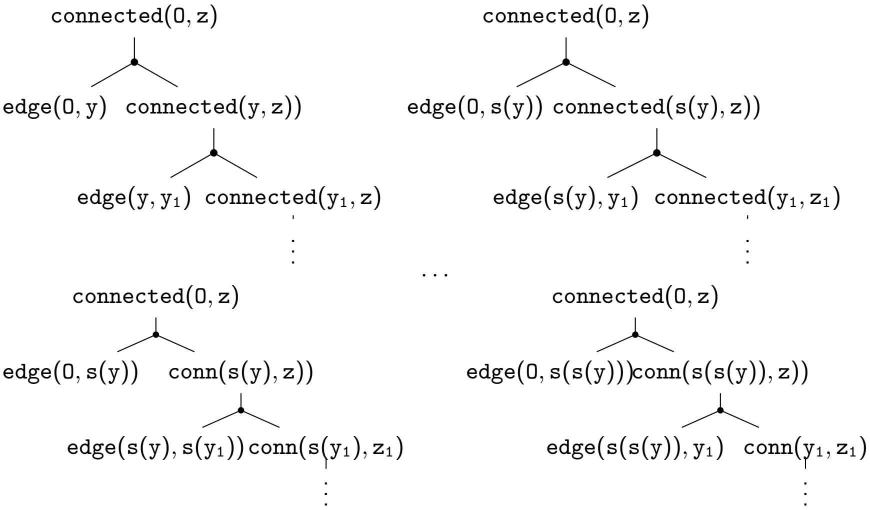
Example

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. `connected(x,x)` $\leftarrow$
2. `connected(x,y)` $\leftarrow$ `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

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

Guarding corecursion, for example:

Example

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

$$
\begin{array}{c}
\texttt{conn}(0, \texttt{cons}(\texttt{y}, \texttt{z})) \\
\texttt{edge}(0, \texttt{y}) \quad \texttt{conn}(\texttt{y}, \texttt{z}))
\end{array}
\rightarrow
\begin{array}{cc}
\multicolumn{2}{c}{\texttt{conn}(0, \texttt{cons}(\texttt{s0}, \texttt{z}))} \\
\texttt{edge}(0, \texttt{s0}) & \texttt{conn}(\texttt{s0}, \texttt{z})) \\
\Box &
\end{array}
\rightarrow
\begin{array}{cc}
\multicolumn{2}{c}{\texttt{conn}(0, \texttt{cons}(\texttt{s0}, \texttt{nil}))} \\
\texttt{edge}(0, \texttt{s0}) & \texttt{conn}(\texttt{s0}, \texttt{nil})) \\
\Box & \Box
\end{array}
$$

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

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

Corecursion guarding parallelism:

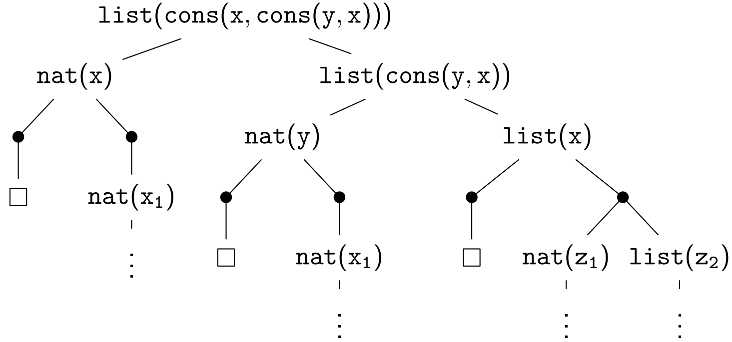
Corecursion FREEING! parallelism:

Corecursion FREEING! parallelism:

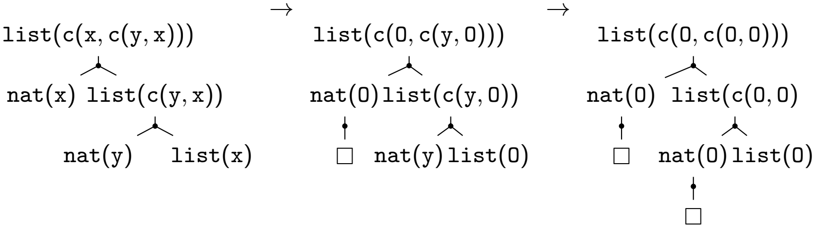
Unification and SLD-resolution are P-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:

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.

More generally...

So, what happened to the old Rule?

More generally...

So, what happened to the old Rule?

Logic Programs = Logic + Control

[Kowalski 1979]

More generally...

So, what happened to the old Rule?

Logic Programs = Logic + Control

[Kowalski 1979]

We have new rules:

Corecursive Programs: **LOGIC is Control**

.

Outline

1. Motivation: LP in Type inference
2. Two old problems of LP:
   - Parallelism
   - Corecursion
3. How Coalgebra Saved the Day
4. What does this matter for type inference?

What does this matter for type inference?

- Practical aspect: hopefully, CoALP's parallelism or corecursion (or some specific combination of the above) will be of some use for new type inference trends;
- Aesthetic: perhaps it is time to bring some harmony into the question of relationship between a type system and the underlying TI algorithm.
  Can we uniformly classify programming languages in terms of extensions of the Hindley-Milner inference algorithm? What impact does it have on operational semantics?

Other questions one may ask:

- (SLD-)Resolution methods are involved in TI in two novel extensions of Coq: in type classes [SO08] and canonical structures [GZND11]. In both cases, enriched type systems give rise to type inference search that exploits many typing options at once. This seems an ideal application for CoALP. Will it be, in practice? Could it be a basis for unifying the two Coq extensions?
- Can constraint LP algorithms implemented in Haskell be efficiently and elegantly combined with CoALP (cf. the combination of sequential Co-LP with Constraints [SG12])? If so, can this yield further improvements in type inference such as in speed, parallelisation or expressiveness?
- Co-LP [Ge07, Ge11] was implemented for type inference in FJ [ALZ09, AL11]. CoALP allows us to program a wider class of corecursive programs than Co-LP does, and it allows us to mix recursion and corecursion, which was impossible in Co-LP. Can these properties of CoALP help to improve type inference in FJ or in other functional languages?

The End.

Bibliography I

D. Ancona and G. Lagorio.
Idealized coinductive type systems for imperative object-oriented programs.
RAIRO - Th. Inf. and Applications, 45(1):3–33, 2011.

Davide Ancona, Giovanni Lagorio, and Elena Zucca.
Type inference by coinductive logic programming.
In *TYPES*, volume 5497 of *LNCS*, pages 1–18, 2009.

G. Barthe and T. Coquand.
An introduction to dependent type theory.
In *Applied Semantics*, volume 2395 of *LNCS*, pages 1–41, 2002.

G. Gupta and et al.
Coinductive logic programming and its applications.
In *ICLP 2007*, volume 4670 of *LNCS*, pages 27–44, 2007.

Bibliography II

Gopal Gupta and et al.
Infinite computation, co-induction and computational logic.
In *CALCO*, volume 6859 of *LNCS*, pages 40–54, 2011.

G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo.
Parallel execution of prolog programs: a survey.
ACM Trans. Computational Logic, pages 1–126, 2012.

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer.
How to make ad hoc proof automation less ad hoc.
In *ICFP*. ACM, 2011.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn.
Simple unification-based type inference for GADTs.
In *ICFP*, pages 50–61. ACM, 2006.

Bibliography III

Robin Milner.
A theory of type polymorphism in programming.
J. Comput. Syst. Sci., 17(3):348–375, 1978.

Martin Odersky, Martin Sulzmann, and Martin Wehr.
Type inference with constrained types.
TAPOS, 5(1):35–55, 1999.

L. Simon and et al.
Co-logic programming: Extending logic programming with coinduction.
In *ICALP*, volume 4596 of *LNCS*, pages 472–483. Springer, 2007.

Neda Saeedloei and Gopal Gupta.
Coinductive constraint logic programming.
In *FLOPS*, volume 7294 of *LNCS*, pages 243–259, 2012.

Bibliography IV

Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis.
Complete and decidable type inference for gadts.
In *ICFP*, pages 341–352. ACM, 2009.

Matthieu Sozeau and Nicolas Oury.
First-class type classes.
In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.

Martin Sulzmann and Peter J. Stuckey.
Hm(x) type inference is clp(x) solving.
J. Funct. Program., 18(2):251–283, 2008.

Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann.
Outsidein(x) modular type inference with local assumptions.
J. Funct. Program., 21(4-5):333–412, 2011.

Bibliography V

Philip Wadler and Stephen Blott.
How to make ad-hoc polymorphism less ad-hoc.
In *POPL*, pages 60–76. ACM Press, 1989.