Designing a small programming language, coalgebraically

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

School of Computing, University of Dundee, UK

Shonan'13, Coinduction for Computation Structures and Programming Languages
9 October 2013

Outline

1. Motivation: LP key facts

Outline

1. Motivation: LP key facts
2. Coalgebraic Semantics and its Implementation
   - Variable-free case
   - General Case

Outline

1. Motivation: LP key facts
2. Coalgebraic Semantics and its Implementation
   - Variable-free case
   - General Case
3. Comparing with the actual state-of-the art...

Outline

1. Motivation: LP key facts

2. Coalgebraic Semantics and its Implementation
   - Variable-free case
   - General Case

3. Comparing with the actual state-of-the art...

4. Future: CoALP for Type Inference?

LP Key Facts

- An un-typed declarative language, with eager evaluation.
- ... Based on Predicate logic [*vanEmden and Kowalski, "The Semantics of Predicate Logic as a Programming Language", 1976*]
- ... Set-theoretic semantics given by lfp or gfp of the semantic operator T_P.
- ... Operational semantics given by SLD-resolution. [*Robinson "A Machine-Oriented Logic Based on the Resolution Principle", 1965*]
- Many dialects exist: Prolog, Datalog, etc.
- Applications: Data bases, Proof theory (Automated First Order Theorem Provers), AI, Hindley-Milner style Type inference in Functional languages.

Recursion in Logic Programming

Example

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

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

Example

$\mathtt{nat(0)} \leftarrow$

$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$

$\mathtt{list(nil)} \leftarrow$

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

$\mathtt{list(y)}$

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

$$|$$

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

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

Example

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

$\texttt{nat(s(x))} \leftarrow \texttt{nat(x)}$

$\texttt{list(nil)} \leftarrow$

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

$\texttt{list(y)}$

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

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)
```

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

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.

Relation to Coalgebra? – Poor

- Coinduction fails (because of eager evaluation)
- Concurrency fails (because of unification and variable dependencies)

Problems with Corecursion in LP?

Example

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

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

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

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

Problems with Corecursion in LP?

Example

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

No answer, as derivation never terminates.

Problems with 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.

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

Relation to Coalgebra? – Poor

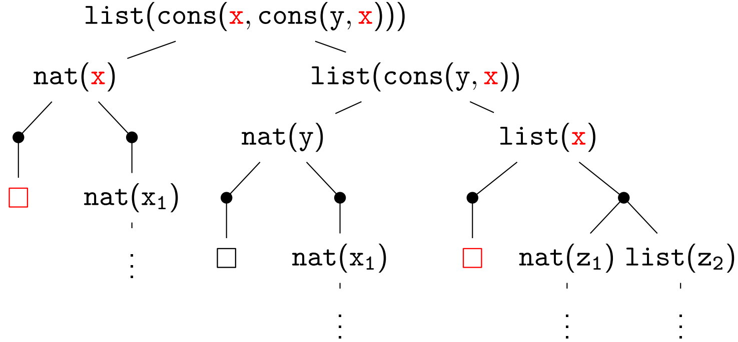
- Coinduction fails (because of eager evaluation)
- Concurrency fails (because unification and variable dependencies)

Problems with concurrency/corecursion in LP

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

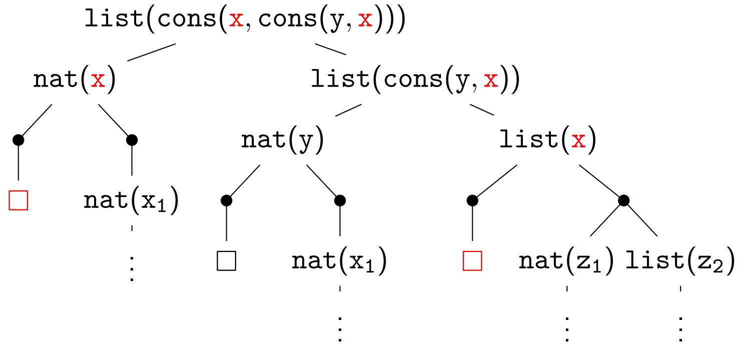
Problems with concurrency/corecursion in LP

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



Problems with concurrency/corecursion in LP

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



If unsound – lets synchronize variable substitution! – many engineering solutions... Synchronisation breaks parallelisation, in the general case.

Relation to Coalgebra? – Poor

- Coinduction fails (because of eager evaluation)
- Concurrency fails (because unification and variable dependencies)

Lets time-travel to early 70s and fix it...

... create coalgebraically oriented alternative from scratch...

Outline

CoALP semantics bibliography

- E.Komendantskaya and J.Power. Coalgebraic derivations in logic programming. International conference Computer Science Logic, CSL'11.
- E.Komendantskaya and J.Power. Coalgebraic semantics for derivations in logic programming. International conference on Algebra and Coalgebra CALCO'11.
- E. Komendantskaya, G. McCusker and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. Proceedings of AMAST'2010.

Current work – implementation and applications...

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

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.

Example

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

$p(q(b,a)) = \{\{\}, \{s(a,b)\}\}$

Coalgebraic Analysis of ground 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

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

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

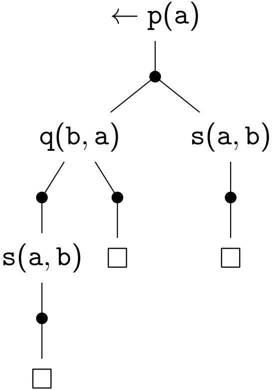
$$\begin{aligned} p_0 &= id : \mathrm{At} \longrightarrow \mathrm{At}(= \mathrm{At}_0) \\ p_{n+1} &= \langle id, P_f P_f(p_n) \circ p \rangle : \mathrm{At} \longrightarrow \mathrm{At} \times P_f P_f \mathrm{At}_n(= \mathrm{At}_{n+1}) \end{aligned}$$

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

Semantics, graphically represented

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

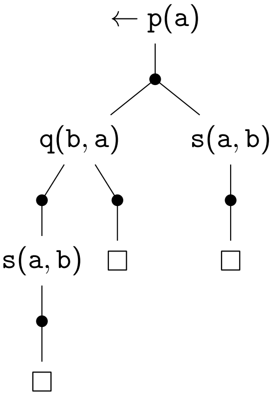
for logic program:




```
q(b,a)  ←  s(a,b)
q(b,a)  ←
s(a,b)  ←
  p(a)  ←  q(b,a),s(a,b)
```

Language design

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



We transform this construction verbatim to logic algorithm

- ... corresponds to and-or parallel trees introduced for LP in the 90s
- if we are in the 70s, we “win” 20 years.
- Ready-to-use algorithm for Datalog programs or equivalent finite-model LP fragments.
- Non-determinism – if or-nodes are considered as points of non-determinism

Implementation

[2012, with M.Schmidt] - First prototype in Prolog

[2012-2013, with M.Schmidt, J.Heras] - Parallel implementation in Go (language for concurrency inspired by Hoare logic).
Very nice results in terms of the speed-up.

Implementation

[2012, with M.Schmidt] - First prototype in Prolog

[2012-2013, with M.Schmidt, J.Heras] - Parallel implementation in Go (language for concurrency inspired by Hoare logic).
Very nice results in terms of the speed-up.

[2013 - , with J.Heras, V.Komendantsky] – Parallel Haskell implementation. We do not have thorough evaluation yet...

Outline

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.

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

Fibrations modeling the set At

Intuition is to replace At with the functor $At : \mathcal{L}_{\Sigma}^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ, set of predicates of the given program and n distinct variables.
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).

Fibrations modeling the set *At*

Intuition is to replace At with the functor $At : \mathcal{L}_{\Sigma}^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ, set of predicates of the given program and n distinct variables.
Some modifications are needed:

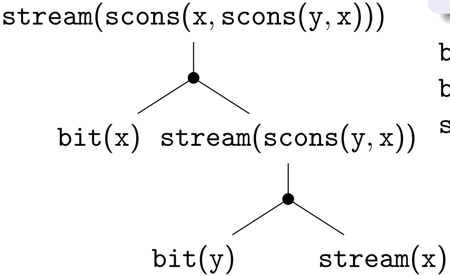
- we need to extend Set to $Poset$,
- natural transformations to *lax natural transformations*, and
- replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability).

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

But cf. Bonchi&Zanasi CALCO'13 paper on getting rid of laxness.

The semantics, graphically:

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

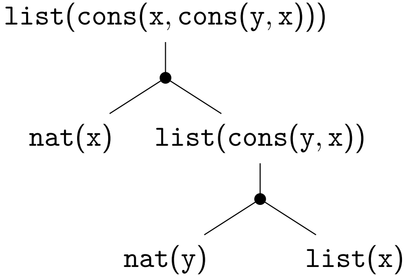


for a program:

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

The semantics, graphically:

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



for a program

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

Language design

- Again, we take the construction of the trees "almost" verbatim in the language design;
- We call the trees arising from the logic algorithm – *coinductive trees*
- The effect of fibrations modelled by term-matching (rather than unification) used in derivations.
- Note the finite tree for `Stream`!!! – looks like lazy evaluation!!!
- As before, we give a parallel implementation for computations of every node.
- Note also because of the "laziness" , a single coinductive tree may not give entire derivation.
- We had a "bad" case of "typing", but the coinductive trees had no unsound substitutions.
- Guardedness ...

Guarding corecursion

(Co)-Recursion

... needs to be guarded against non-termination. Both in FP and LP, such guards can be given semantically or syntactically (later is often known as "guardeness-by-constructors").

`Stream` is guarded by constructors and has finite coinductive trees:

Example

bit(0) $\leftarrow$

bit(1) $\leftarrow$

stream(scons x y) $\leftarrow$, bit(x), stream(y)

Guarding corecursion

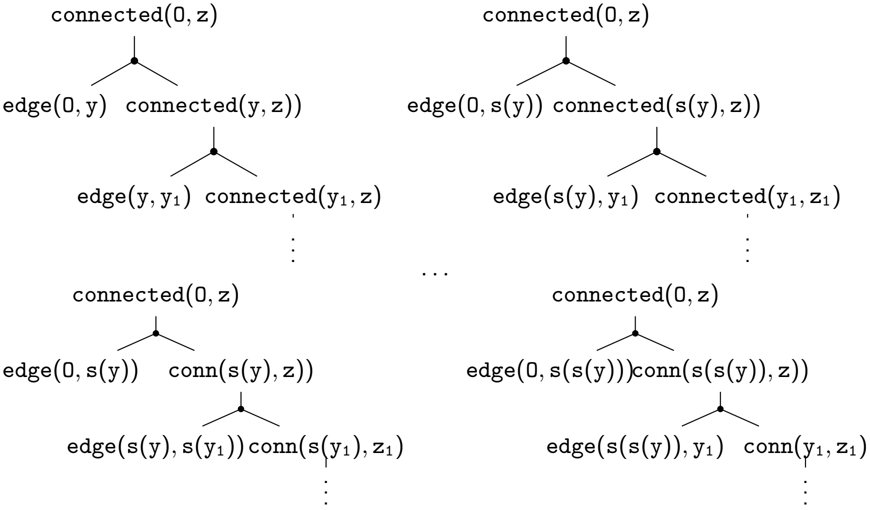
Example

This program (graph connectivity) 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 (infinite-breadth and infinite-depth trees):



Guarding corecursion, by constructors:

Example

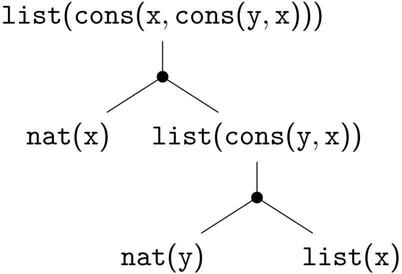
$$\begin{aligned}
\texttt{connected}(X, cons(Node, Path)) &\leftarrow \texttt{edge}(X, Node), \texttt{connected}(Node, Path)\\
\texttt{connected}(X, nil) &\leftarrow\\
\texttt{edge}(0, 0) &\leftarrow\\
\texttt{edge}(X, s(X)) &\leftarrow
\end{aligned}$$

- Cf. Coq/Agda guarding (co)-recursion by constructors;
- There is a bit more to it than that (more conditions needed to keep variable substitution under control in absence of types.)

Maps between fibers, graphically:

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

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



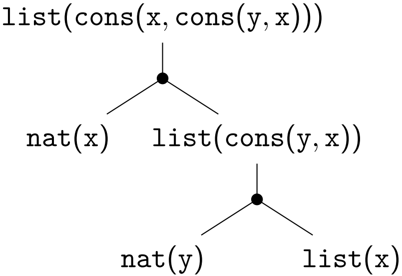
Maps between fibers, graphically:

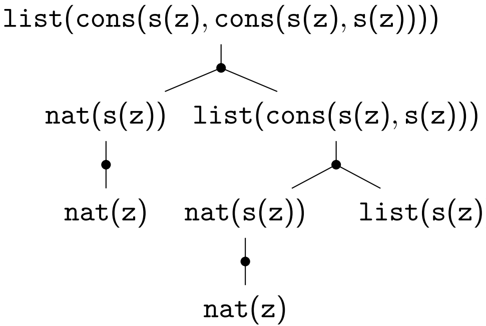
$A(x, y) \in At(2)$

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 $P_c P_f At(1)$.

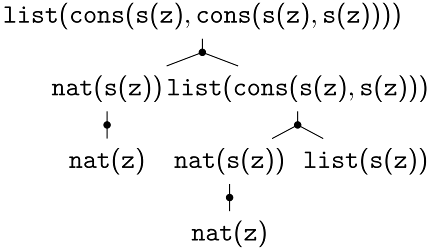




Maps between fibers, graphically:

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

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

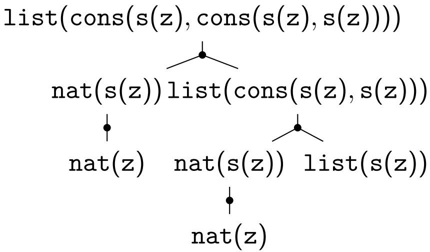


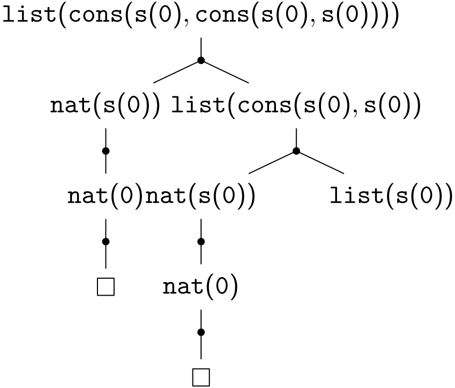
Maps between fibers, graphically:

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

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

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





Language design

- We take the above fiber transitions "almost" verbatim;
- Compute only some maps, not all maps to do derivations;
- Note the gracious way variable dependencies have been handled in "bad typing" case;
- Use the old MGU algorithm (on tree leaves and clauses) to compute substitutions;
- These can be computed in parallel or non-deterministic way...
- Lazy corecursion in full power: potentially infinite transitions between finitely computable coindutive trees;
- Coinductive trees = finite observations;
- Note also the variable length of the size of the finite observations: (i-productivity, but with dynamic i?)

An Example

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

An Example

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

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

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

An Example

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

$\texttt{bit(z)}$ $\texttt{stream(y)}$

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

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

An Example

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

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

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

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

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

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

Answers for x: $cons(z, y)$ and $cons(0, cons(y_1, z_1))$.

Guarding corecursive derivations, lazily:

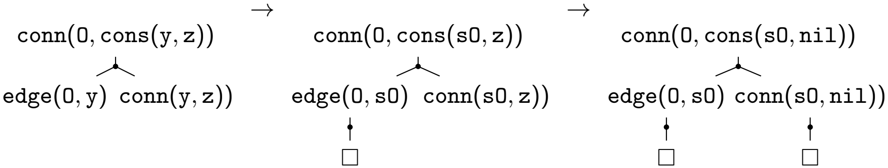
Example

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

Guarding corecursive derivations, lazily:

Example

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



Guardedness: 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. `connected(x,y)` $\leftarrow$ `connected(z,y), edge(x,z)`
2. `connected(x,x)` $\leftarrow$

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

Outline

Back to 2013...

... we find out that:

- Parallel LP has been flourishing around 90s, with general-case parallelism still being a problem (took 20 years);
- Coinductive LP was suggested in 2007, with limited data structures (took 30 years)
- LP was adapted in Hindley-Milner Type inference algorithm (70s), but the rest of algorithmic development of type-inference was chaotic, on a "hack-by-need" basis... 2010s showing a need for coinductive and parallel inference algorithms...

Co-LP [Gupta, Simon et al., 2007

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)
```

$$\leftarrow \mathtt{stream(scons(x,y))}$$
$$|$$
$$\leftarrow \mathtt{bit(x), stream(y)}$$
$$|$$
$$\leftarrow \color{red}{\mathtt{stream(y)}}$$
$$|$$
$$\leftarrow \mathtt{bit(x_1), stream(y_1)}$$
$$|$$
$$\leftarrow \color{red}{\mathtt{stream(y_1)}}$$
$$|$$
$$\square$$

Co-LP [Gupta, Simon et al., 2007

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

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

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

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

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

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

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

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.

CoALP vs Co-LP

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.

Corecursion guarding parallelism:

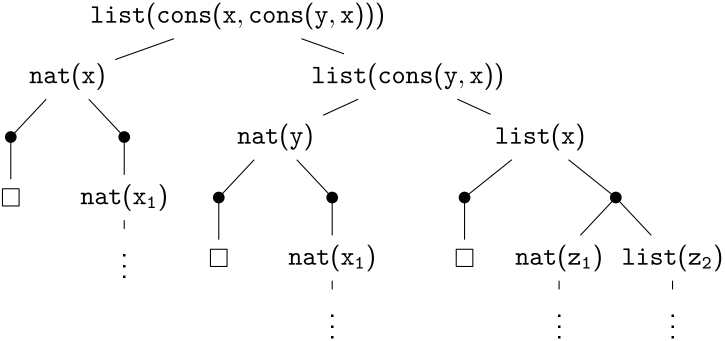
Corecursion FREEING! parallelism:

Corecursion FREEING! parallelism:

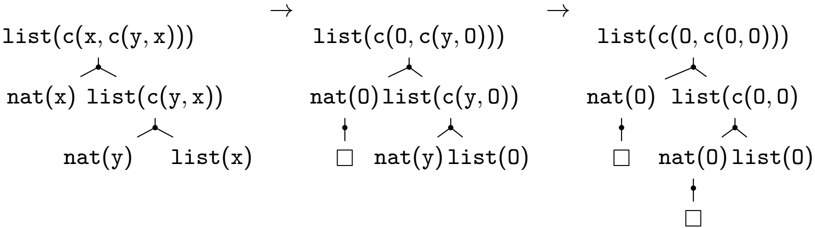
Unification and SLD-resolution are P-complete algorithms. Parallel LP community has to be very inventive in the ways to trick it. In particular, variable synchronization is a huge sequential barrier:

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:



So, the same guarded corecursive algorithm does the work for free.

Outline

Milner, 1978

"A theory of Type Polymorphism in Programming"

Milner, 1978

"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*,
- *Type Classes* [Wadler&Blott89],
- *Generalised Algebraic Types* (GADTs) [Jones&al,06]
- *Dependent Type Classes* [Sozeau&Oury,08] and
- *Canonical Structures* [Gonthier&al,11].

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

Trend in type inference:

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

- *Dependent Types*,
- *Type Classes* [Wadler&Blott89],
- *Generalised Algebraic Types* (GADTs) [Jones&al,06]
- *Dependent Type Classes* [Sozeau&Oury,08] and
- *Canonical Structures* [Gonthier&al,11].

Milner-style decidable type inference does not always suffice (e.g. the *principal type* may no longer exist), and TI requires 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) [Jones&al,06], Constraint LP [Odersky,Sulzmann,Schrijvers,Vytiniotis, 1999-2011], LP embedded into interactive tactics (Coq's *eauto*) [Sozeau&Oury,08], and LP supplemented by rewriting [Gonthier&al,11].

What can CoALP do 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?

We have minds/hands and EPSRC money to pursue that, so if you see this useful in *your* language, we will be happy to try that.