Coinductive Logic Programming for Type Inference

Katya Komendantskaya

School of Computing, University of Dundee, UK

TYPES'11,
10 September 2011

Motivation

There was a talk at TYPES'08 [Ancona, Lagorio, et al.] on coinductive logic programming for type inference (in featherweight JAVA)

Things I wanted to understand ever since were:

- ... what is the difference between "normal" LP and coinductive LP?

Motivation

There was a talk at TYPES'08 [Ancona, Lagorio, et al.] on coinductive logic programming for type inference (in featherweight JAVA)

Things I wanted to understand ever since were:

- ... what is the difference between "normal" LP and coinductive LP?
- ... the role of coinductive LP for type inference.

Motivation

There was a talk at TYPES'08 [Ancona, Lagorio, et al.] on coinductive logic programming for type inference (in featherweight JAVA)

Things I wanted to understand ever since were:

- ... what is the difference between "normal" LP and coinductive LP?
- ... the role of coinductive LP for type inference.
- ... is this role specific to FW-JAVA, or can be extended to functional languages?

Motivation

There was a talk at TYPES'08 [Ancona, Lagorio, et al.] on coinductive logic programming for type inference (in featherweight JAVA)

Things I wanted to understand ever since were:

- ... what is the difference between "normal" LP and coinductive LP?
- ... the role of coinductive LP for type inference.
- ... is this role specific to FW-JAVA, or can be extended to functional languages?
- ... we have proposed an alternative coinductive LP algorithm (together with J.Power, [CSL'11]). Would this new version be any better for type inference?

Motivation

There was a talk at TYPES'08 [Ancona, Lagorio, et al.] on coinductive logic programming for type inference (in featherweight JAVA)

Things I wanted to understand ever since were:

- ... what is the difference between "normal" LP and coinductive LP?
- ... the role of coinductive LP for type inference.
- ... is this role specific to FW-JAVA, or can be extended to functional languages?
- ... we have proposed an alternative coinductive LP algorithm (together with J.Power, [CSL'11]). Would this new version be any better for type inference?

The last item is largely future work, so please step forward if you would like to join!

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 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 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 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.

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 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.

If CLP(X) is strong enough to substitute Hindley-Milner type inference, where does Co-LP come in?

Recursion and Corecursion in Logic Programming

Example

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

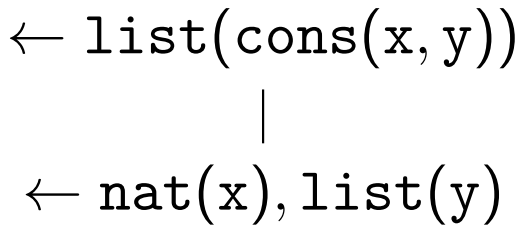
Example

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

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

Example

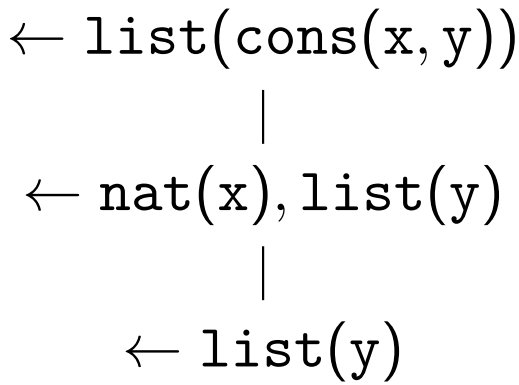
```
nat(0) ←
nat(s(x)) ← nat(x)
list(nil) ←
list(cons x y) ← nat(x),
                 list(y)
```



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

Example

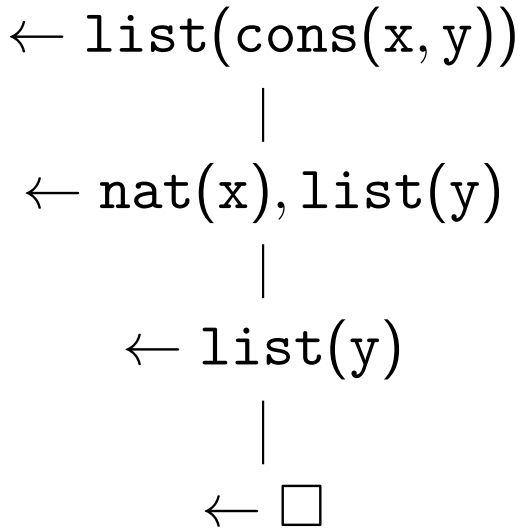
```
nat(0) ←
nat(s(x)) ← nat(x)
list(nil) ←
list(cons x y) ← nat(x),
                  list(y)
```



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)
```



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.

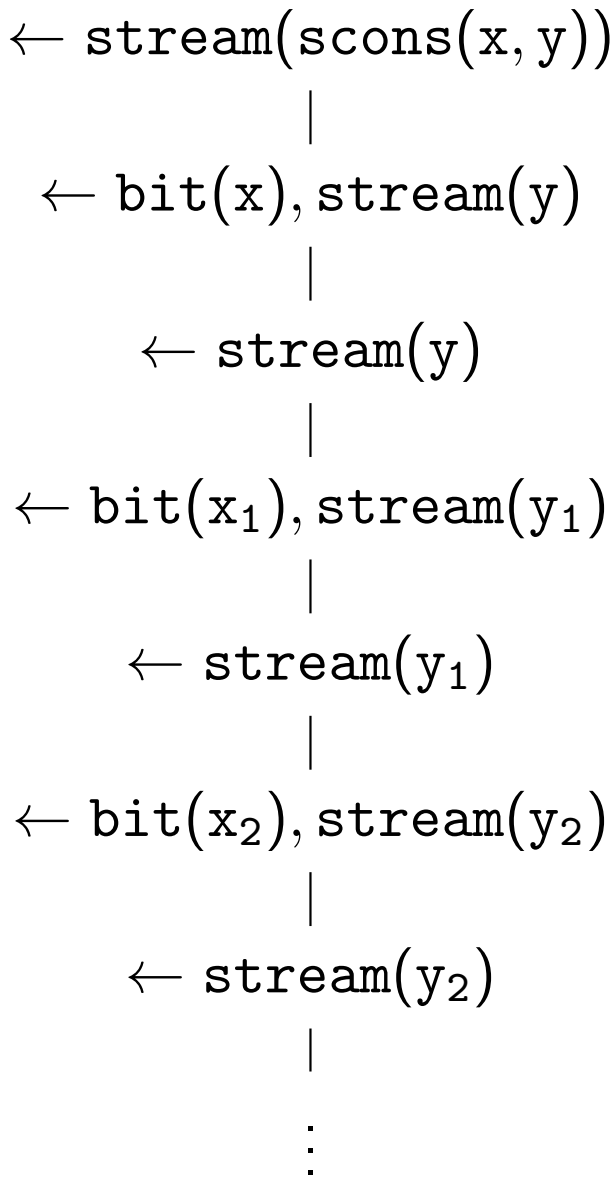
Things go wrong

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.



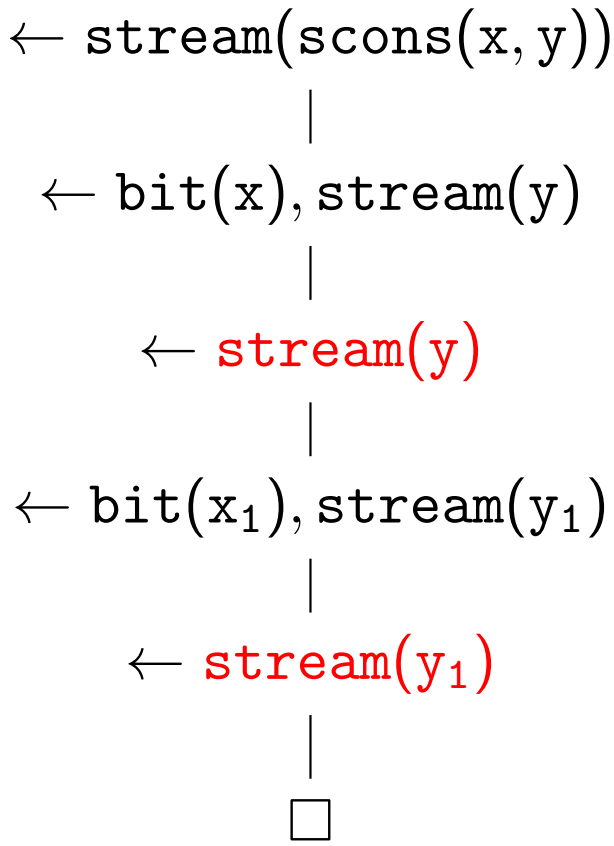
Solution - 1 [Gupta, Simon et al., 2007 - 2008]

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)
```



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

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

Drawbacks:

- ... 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.

`1::  2::  3::  4::  5::  ...` may not

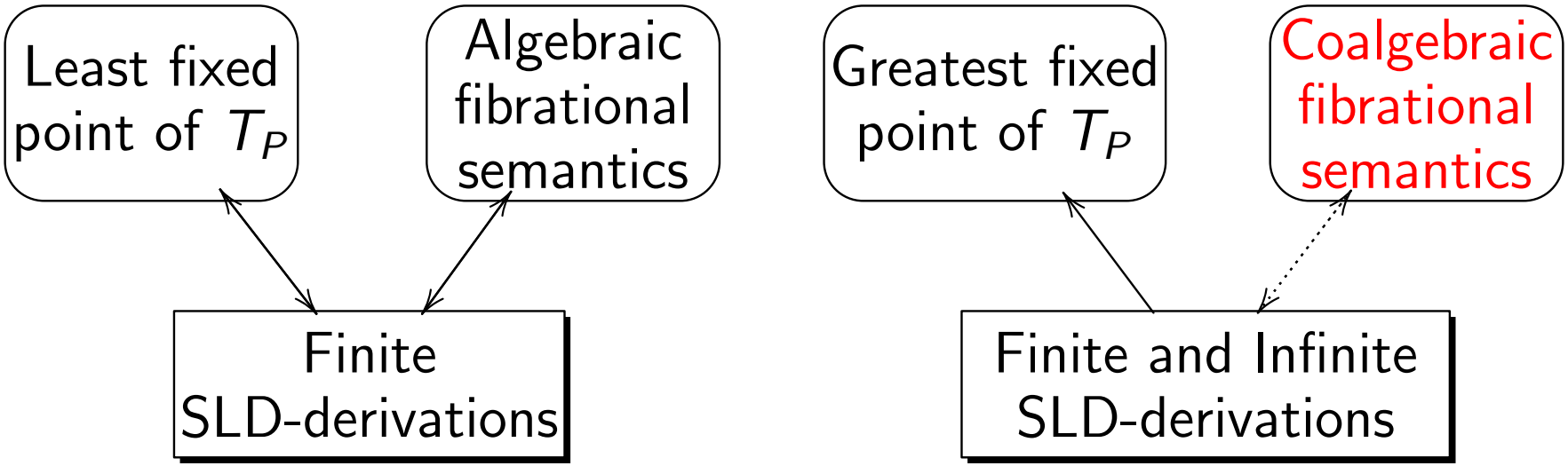
π represented as a stream may not.

- the derivation itself is not really a corecursive process.

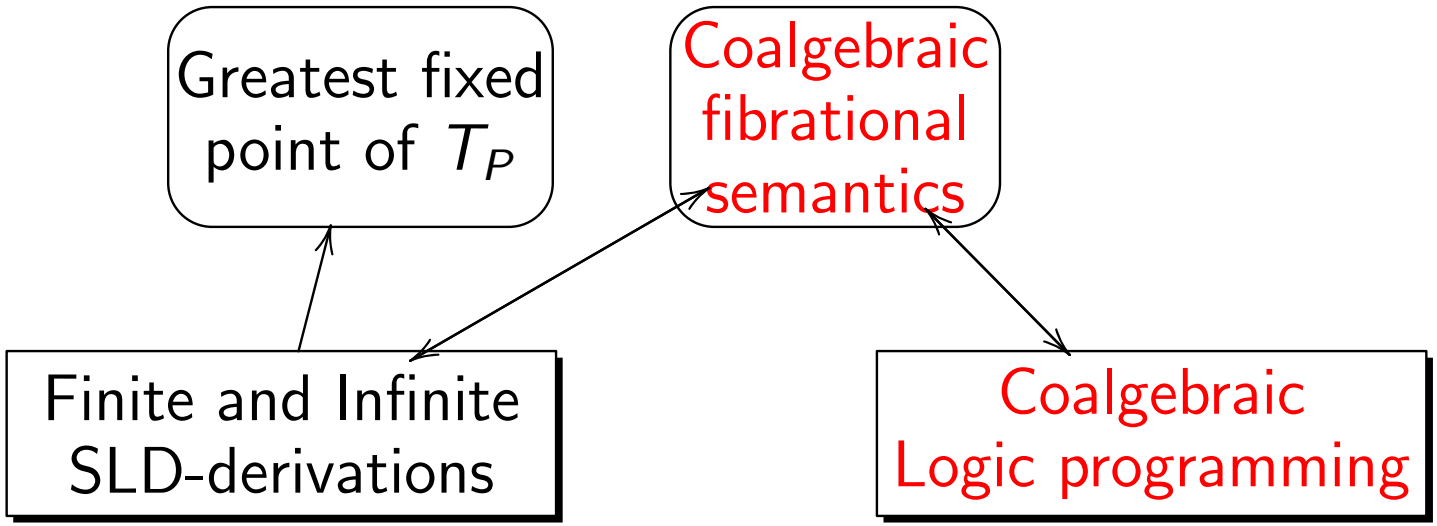
Solution - 2. Coinductive LP in [Komendantskaya, Power CSL'11]

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

Algebraic and coalgebraic semantics for LP



Algebraic and coalgebraic semantics for LP



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

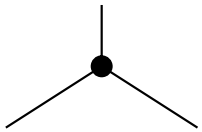
An Example

$$\xrightarrow{\theta_1}$$

`stream(x)`

An Example

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



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

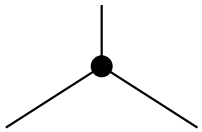
An Example

$\xrightarrow{\theta_1}$

$\texttt{stream(x)}$

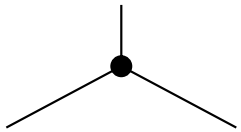
$\xrightarrow{\theta_2} \ldots \xrightarrow{\theta_3}$

$\texttt{stream(scons(z, y))}$

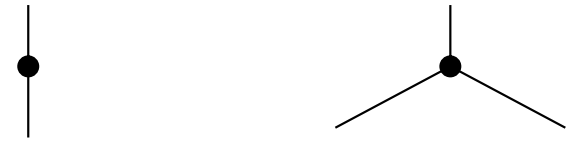


$\texttt{bit(z)} \qquad \texttt{stream(y)}$

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



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



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

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.

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

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

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.

Questions answered

- ... what is the difference between "normal" LP and coinductive LP?
- ... what is the role of coinductive LP for type inference?

- ... is this role of Co-LP specific to FW-JAVA and object-oriented languages, or can be extended to functional languages?
- ... can alternative coinductive LP algorithm [CSL'11] be any better for type inference?

Questions answered

- ... what is the role of coinductive LP for type inference?
- ... is this role of Co-LP specific to FW-JAVA and object-oriented languages, or can be extended to functional languages?
- ... can alternative coinductive LP algorithm [CSL'11] be any better for type inference?

Questions answered

- ... what is the role of coinductive LP for type inference?
- Why there is no place for Co-LP in type inference by CLP(X)?
- ... is this role of Co-LP specific to FW-JAVA and object-oriented languages, or can be extended to functional languages?
- ... can alternative coinductive LP algorithm [CSL'11] be any better for type inference?

Why is there no place for Co-LP in type inference by CLP(X)?

The answer is:

CLP(X) uses a restricted form of logic programs - the one that does not allow recursion.

LP, [Sulzmann, Stuckey 2008]

```
Head H ::= p(a1, . . . , an)
Atom L ::= p(t1, . . . , tn)
Goal G ::= L | C | G ∧ G
Rule R ::= H ←
```

Type inference by CLP(X) comes with a proof of termination. (Programs are not recursive.)

The trick is due to separating derivations in LP stage and constraint solving stage.

Example of a LP used for type inference:

Example

g y = let f x = x in (f True, f y)

$$f(t) \leftarrow t = t_x \rightarrow t$$

$$g(t) \leftarrow t = t_y \rightarrow (t_1, t_2) \wedge f(t_{f_1}) \wedge t_{f_1} = \mathit{Bool} \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

Example of a LP used for type inference:

Example

g y = let f x = x in (f True, f y)

$$f(t) \leftarrow t = t_x \rightarrow t$$

$$g(t) \leftarrow t = t_y \rightarrow (t_1, t_2) \wedge f(t_{f_1}) \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

$$\rightarrow_f t = t_y \rightarrow (t_1, t_2) \wedge t_{f_1} = t_x \rightarrow t_x \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

Example of a LP used for type inference:

Example

g y = let f x = x in (f True, f y)

$$f(t) \leftarrow t = t_x \rightarrow t$$

$$g(t) \leftarrow t = t_y \rightarrow (t_1, t_2) \wedge f(t_{f_1}) \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

$$\rightarrow_f t = t_y \rightarrow (t_1, t_2) \wedge t_{f_1} = t_x \rightarrow t_x \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

$$\rightarrow_f t = t_y \rightarrow (t_1, t_2) \wedge t_{f_1} = t_x \rightarrow t_x \wedge t_{f_1} = Bool \rightarrow t_1 \wedge t_{f_2} = t'_x \rightarrow t'_x \wedge t_{f_2} = t_y \rightarrow f_2$$

Example of a LP used for type inference:

Example

g y = let f x = x in (f True, f y)

$$f(t) \leftarrow t = t_x \rightarrow t$$

$$g(t) \leftarrow t = t_y \rightarrow (t_1, t_2) \wedge f(t_{f_1}) \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

$$\rightarrow_f t = t_y \rightarrow (t_1, t_2) \wedge t_{f_1} = t_x \rightarrow t_x \wedge t_{f_1} = Bool \rightarrow t_1 \wedge f(t_{f_2}) \wedge t_{f_2} = t_y \rightarrow t_2$$

$$\rightarrow_f t = t_y \rightarrow (t_1, t_2) \wedge t_{f_1} = t_x \rightarrow t_x \wedge t_{f_1} = Bool \rightarrow t_1 \wedge t_{f_2} = t'_x \rightarrow t'_x \wedge t_{f_2} = t_y \rightarrow f_2$$

After solving the constraints,

g's type is $\forall t_y . t_y \rightarrow (Bool, t_y)$

Features of CLP(X) method:

- Constraints describe types of expressions;
- each rule describes type of a function (hence only 1 rule per function admitted)
- let-bound function names are renamed to guarantee that the rule heads have distinct predicates;
- no explicit type schemes for let-defined functions, only rules;
- polymorphism is acheived by replicating the constraints for let-definitions.
- recursion is handled by equating the type of the recursive call with the type of the function.

Example

`f x = (let g y = rec g in λ y.  g x in g x)`

$g(t, l) \leftarrow \; t = t_y \rightarrow t_1 \wedge l = [t_x] \wedge t_g = t_x \rightarrow t_1 \wedge t_g = t.$

- separation of constraint generation and inference;
- flexible and accurate type error diagnosis.

Role of Co-LP in FW-JAVA [Ancona et al. 2008]

[Since TYPES'08, eight (!) more papers by the authors on the subject. Also, incorporating some ideas from Sulzman's CLP(X).]

A general observation

Object-oriented languages make heavy use of inheritance, interfaces, and method overriding in a word, subtyping. Naively attempting to expand from Damas-Milners unification to solving a set of subtyping inequality constraints results in an instance of the semi-unification problem, which is generally undecidable.

Role of Co-LP in FW-JAVA [Ancona et al. 2008]

[Since TYPES'08, eight (!) more papers by the authors on the subject. Also, incorporating some ideas from Sulzman's CLP(X).]

A general observation

Object-oriented languages make heavy use of inheritance, interfaces, and method overriding in a word, subtyping. Naively attempting to expand from Damas-Milners unification to solving a set of subtyping inequality constraints results in an instance of the semi-unification problem, which is generally undecidable.

Problem: a general analysis of the method is lacking, e.g. in statements of adequacy, soundness and completeness.
Method of generating logic programs from typed programs is different from Sulzmann, which makes direct comparison hard. In paricular, I suspect [at least some of] (co)recursive LPs shown in [Ancona et al.] are due to the method difference.

Example

```
invoke ( obj(C,R),M,A1 ,RT_ET) ← val_types (A1 ,A2),
exc_types (A1 ,ET), has_meth (C,M ,[ obj(C,R) | A2 ],RT ).
invoke ( obj(C,R),M,A,ET) ← no_val_types (A), exc_types
(A,ET ).
invoke (T1_T2 ,M,A,RT1_RT2) ← invoke (T1 ,M,A, RT1), invoke
(T2 ,M,A, RT2 ).
invoke (ex(C),M,A,ex(C )) ←.
```

Most of such clauses would not be allowed in previous CLP(X) approach, and would be reformulated:

Example

```
invoke ( obj(C,R),M,A1 ,RT_ET) ← val_types (A1 ,A2),
exc_types (A1 ,ET), has_meth (C,M ,[ obj(C,R) | A2 ],RT ).
invoke ( obj(C,R),M,A,ET) ← no_val_types (A), exc_types
(A,ET ).
invoke (T1_T2 ,M,A,RT1_RT2) ← invoke (T1 ,M,A, RT1), invoke
(T2 ,M,A, RT2 ).
invoke (ex(C),M,A,ex(C )) ←.
```

Most of such clauses would not be allowed in previous CLP(X) approach, and would be reformulated:

E.g. the highlighted clause would be $\texttt{invoke}(t, l) \leftarrow t = t_1 \vee t_2, \ldots$.

Doubts summarised

Two major accounts [Ancona & Sulzmann] of LP for type inference

present general approaches to formulating a LP + constraints from function definitions. They could be cross applied, subject to careful description of suitable fragments of type systems.

- I am not entirely convinced that type inference in FW Java cannot be done using Sulzmann's method (terminating LPs); at any rate, this issue is not analysed in the literature;
- If indeed coinductive LPs cannot be avoided for CLP(X) type inference for certain type systems, I am not convinced co-LPs [Simon, Gupta] are expressive enough to handle this – especially, they do not allow to mix inductive and coinductive predicates in one clause and can work only with finite regular patterns (which has to be determined in advance of inference);
- chance for our new coindutive LPs?

Conclusions. Coinductive LP [KP'11] could be useful:

- when delayed/partial type inference may be welcome (on place of infinitary term rewriting? for some mixture of static and dynamic inference?)
  [akin reconciliation of static and dynamic inference mentioned by G.Gonthier yesterday?]
  [Another yesterday's talk *Type Classes: instance resolution cannot handle cyclic instances*]
- OR where concurrency and parallelism are important;
- in case combining inductive and coinductive propositions in clauses is important;
- in case a regular pattern in an infinite structure described by a LP either does not exist or is too expensive to observe in advance.

Conclusions. Coinductive LP [KP'11] could be useful:

- when delayed/partial type inference may be welcome (on place of infinitary term rewriting? for some mixture of static and dynamic inference?)
  [akin reconciliation of static and dynamic inference mentioned by G.Gonthier yesterday?]
  [Another yesterday's talk *Type Classes: instance resolution cannot handle cyclic instances*]
- OR where concurrency and parallelism are important;
- in case combining inductive and coinductive propositions in clauses is important;
- in case a regular pattern in an infinite structure described by a LP either does not exist or is too expensive to observe in advance.

Research question

Where this could be applied, and how can it be implemented?

Conclusions. Coinductive LP [KP'11] could be useful:

- when delayed/partial type inference may be welcome (on place of infinitary term rewriting? for some mixture of static and dynamic inference?)
  [akin reconciliation of static and dynamic inference mentioned by G.Gonthier yesterday?]
  [Another yesterday's talk *Type Classes: instance resolution cannot handle cyclic instances*]
- OR where concurrency and parallelism are important;
- in case combining inductive and coinductive propositions in clauses is important;
- in case a regular pattern in an infinite structure described by a LP either does not exist or is too expensive to observe in advance.

Research question

Where this could be applied, and how can it be implemented?
Personal question: Is the RQ worth investigating?

Any Questions?

or, better, Any Answers?