Coalgebraic Logic Programming

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

School of Computing, University of Dundee, UK

3 July 2014

Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types

Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types
2. Coalgebraic Logic Programming

Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types
2. Coalgebraic Logic Programming
3. Parallelism

Outline

1. Recursion and Corecursion
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Recursion and Corecursion without types
2. Coalgebraic Logic Programming
3. Parallelism
4. Future directions: Applications to type inference

Outline

Today's talk...

...continuation of Thanos'es talk of yesterday:

- about logic programming (LP);
- about first order (= in Thanos'es terms infinite) language for LP;
- about how much we can merge methods of FP and LP...
- may be you will see some references to a possible game semantics.

Inductive Types and Recursive Functions

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

Recursive functions have arguments of inductive types.

```
Fixpoint length (A:Type) (l: list A) : nat :=
match l with
 | nil => 0
 | _ :: l' => S (length l')
end.
```

Coinductive Types and Corecursive Functions

```
CoInductive stream (A:Set) :  Set :=
SCons:  A -> stream A -> stream A.
```

Corecursive functions have outputs of coinductive types. (Type of input arguments is not important.)

```
CoFixpoint map (s:Stream A) : Stream B :=
SCons (f (hd s)) (map (tl s)).
```

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow\leftarrow$ types; proofs $\rightarrow\leftarrow$ programs): non-terminating proofs can lead to inconsistency.

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow\leftarrow$ types; proofs $\rightarrow\leftarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.

Productive Values

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

Productive Values

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.
The element of the stream at position n can be found by:

Definition

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

A given stream s is productive if we can be sure that the computation of the list `nth n s` is guaranteed to terminate, whatever the value of `n` is.

We call a function *productive at the input value i*, if it outputs a productive value at *i*.

Deciding Termination: Structural Recursion

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Deciding Termination: Structural Recursion

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Example

```
Fixpoint length (A:Type) (l:  list A) : nat :=
match l with
 | nil => O
 | _ ::  l' => S (length l')
end.
```

Deciding Productivity: Guardedness

The guardedness condition insures that

* each corecursive call is made under at least one constructor;

** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Deciding Productivity: Guardedness

The guardedness condition insures that

* each corecursive call is made under at least one constructor;

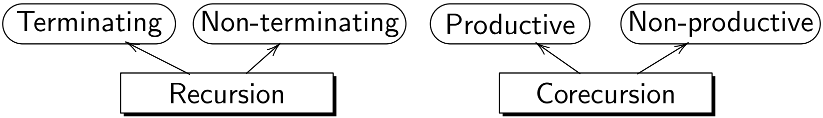
** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Example

```
CoFixpoint map (s:Stream A) : Stream B :=
SCons (f (hd s)) (map (tl s)).
```

To notice:



- The role of types in definition of (co)recursive functions;
- The role of constructors and (co)-pattern matching;

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

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

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

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

Example

nat(0) $\leftarrow$

nat(s(x)) $\leftarrow$ nat(x)

list(nil) $\leftarrow$

list(cons x y) $\leftarrow$ nat(x), list(y)

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

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)
```

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

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.

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)

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

$$|$$

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

$$|$$

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

$$|$$

$$\leftarrow \Box$$

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.

Nice, clean semantics: least Herbrand model exists, sound&complete, etc.: see Thanos'es Viva of yesterday.

Corecursion in LP?

Example

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

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

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

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

Corecursion in LP?

Example

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

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

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

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

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: least Herbrand models will contain an infinite term corresponding to stream: so completeness fails.

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

It can be worse....

Example

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

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

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

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

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

It can be worse....

Example

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

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

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

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

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

No answer, as derivation never terminates.

It can be worse....

Example

```
bit(0) ←
bit(1) ←
list(cons(x, y)) ←
            bit(x), list(y)
list(nil) ←
```

No answer, as derivation never terminates.
Semantics goes wrong: this time, soundness!

$$\leftarrow \texttt{list}(\texttt{cons}(x, y))$$
$$|$$
$$\leftarrow \texttt{bit}(x), \texttt{list}(y)$$
$$|$$
$$\leftarrow \texttt{list}(y)$$
$$|$$
$$\leftarrow \texttt{bit}(x_1), \texttt{list}(y_1)$$
$$|$$
$$\leftarrow \texttt{list}(y_1)$$
$$|$$
$$\leftarrow \texttt{bit}(x_2), \texttt{list}(y_2)$$
$$|$$
$$\leftarrow \texttt{list}(y_2)$$
$$|$$
$$\vdots$$

To notice:

- Distinction between (co)inductive type, (co)recursive function over (co)inductive type and a proof by (co)induction is erased.
- Without types guarding (co)recursion, things get messy:
  - ...not "just" termination, but also semantics
- We do not have a formalism to speak about termination and productivity, or generally, recursion/corecursion.

Note aside: LP has instances of dependent types, mixed induction/coinduction, recursion/corecursion....

Outline

CoALP: what is it about?

- syntactically – first-order logic programming;
- operationally – lazy (co)recursion;
- inspired by coalgebraic fibrational semantics;
- explores the tree-structure of partial proofs – "coinductive trees";
- uses lazy guarded corecursion using measures of corecursive steps given by coinductive trees (cf. "clocked corecursion");
- parallel...

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

1. Let At be the set of all atoms appearing in a program P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(At))$ sends an atom A to the set of bodies of those clauses in P with head A.

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

1. Let At be the set of all atoms appearing in a program P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(At))$ sends an atom A to the set of bodies of those clauses in P with head A.
2. Taking $p : At \longrightarrow P_f P_f(At)$, 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: $C(P_f P_f)(At)$ is given by a limit of the form

$$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

This gives a "tree-like" structure: we call them $\&V$-trees.

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Operational View of Logic Programming

1. Let At be the set of all atoms appearing in a program P. Then P can be identified with a $P_f P_f$-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(At))$ sends an atom A to the set of bodies of those clauses in P with head A.
2. Taking $p : At \longrightarrow P_f P_f(At)$, 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: $C(P_f P_f)(At)$ is given by a limit of the form

   $$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

   This gives a "tree-like" structure: we call them $\&V$-trees.
3. For first order extension: Take *Lawvere Theory* $\mathcal{L}_\Sigma$ to model the signature Σ (objects are natural numbers, arrows – term arities, composition = substitution), and take $\mathcal{L}_\Sigma \rightarrow Set$ – to model (predicates in) At.

Examples

Program `Stream`: fibers are term arities. Take the fiber of 1. $\&V$-trees:

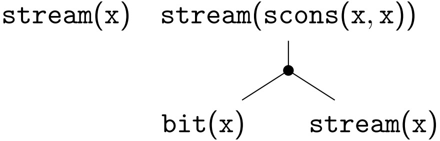
Examples

Program `Stream`: fibers are term arities. Take the fiber of 1. &V-trees:

`stream(x)`

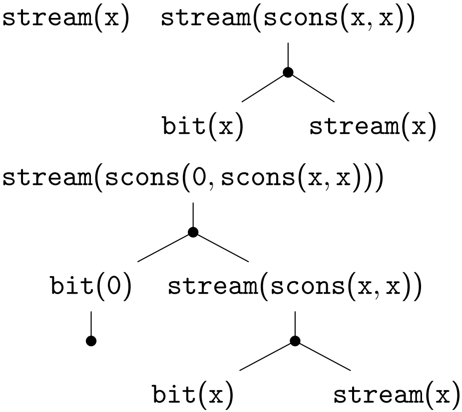
Examples

Program `Stream`: fibers are term arities. Take the fiber of 1. &*V*-trees:



Examples

Program `Stream`: fibers are term arities. Take the fiber of 1. &V-trees:



Computationally essential:

1. for coinductive `Stream` program, the $\&V$-trees are finite!!! – both in depth and in breadth;
2. each tree gives only a partial computation – it is not like eager SLD-trees we have seen earlier;
3. the effect of fibers is best modelled by restricting unification to term-matching (note resemblance to the pattern-matching in Functional setting).

Computationally essential:

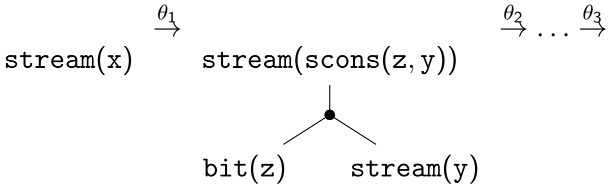
1. for coinductive `Stream` program, the &V-trees are finite!!! – both in depth and in breadth;
2. each tree gives only a partial computation – it is not like eager SLD-trees we have seen earlier;
3. the effect of fibers is best modelled by restricting unification to term-matching (note resemblance to the pattern-matching in Functional setting).

1. ⇒ gives hope for a formalism to describe termination and productivity
2. ⇒ hints there may be laziness involved...

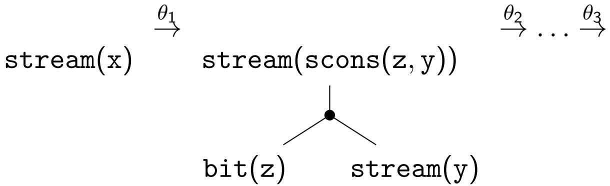
Lazy Corecursion in CoALP: Coinductive trees

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

Lazy Corecursion in CoALP: Coinductive trees



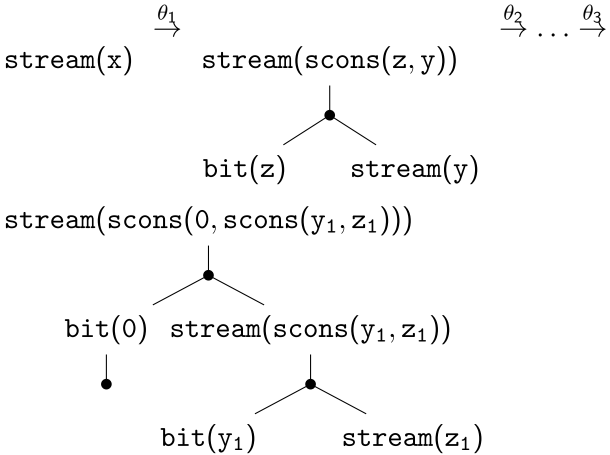
Lazy Corecursion in CoALP: Coinductive trees



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

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

Lazy Corecursion in CoALP



The above would correspond to one-branch of SLD-derivations we have seen! The main driving force: separation of layers of computations into different dimensions.

Properties:

Komendantskaya, Power, Schmidt: Coalgebraic Logic Programming: from Semantics to Implementation, Journal of Logic and Computation, 2014.

- Sound and complete with respect to the coalgebraic semantcs;
- Finite computations are sound and complete with respect to the least Herbrand model semantics (so, we can do as much as standard Prolog for sure).
- Adequacy result for observational semantics.

Properties:

Komendantskaya, Power, Schmidt: Coalgebraic Logic Programming: from Semantics to Implementation, Journal of Logic and Computation, 2014.

- Sound and complete with respect to the coalgebraic semantcs;
- Finite computations are sound and complete with respect to the least Herbrand model semantics (so, we can do as much as standard Prolog for sure).
- Adequacy result for observational semantics.

What does it tell us beyond LP?

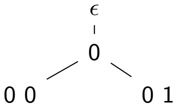
CoALP: the three-dimensional calculus of trees

1. Dimension 1: term-trees;
2. Dimension 2: coinductive trees;
3. Dimension 3: derivation trees.

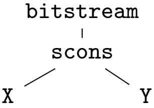
Dimension-1: Term-trees

Take a tree-language $\mathbb{N}^*$.
Given an $L \in \mathbb{N}^*$, a term tree is a map $L \to \Sigma$, satisfying term arity restrictions.
Example:

0.



1.



Notation:

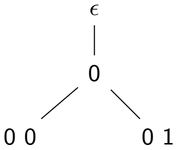
$\mathbf{Term}(\Sigma)$	*finite* term trees over Σ
$\mathbf{Term}^{\infty}(\Sigma)$	*infinite* term trees over Σ
$\mathbf{Term}^{\omega}(\Sigma)$	*finite and infinite* term trees over Σ

Dimension-2: Coinductive trees

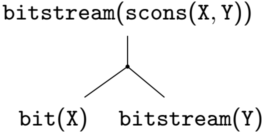
Given an $L \in \mathbb{N}^*$, a coinductive tree is a map $L \to \mathbf{Term}(\Sigma_P)$, satisfying coinductive tree construction for P.

Example:

0.



2.

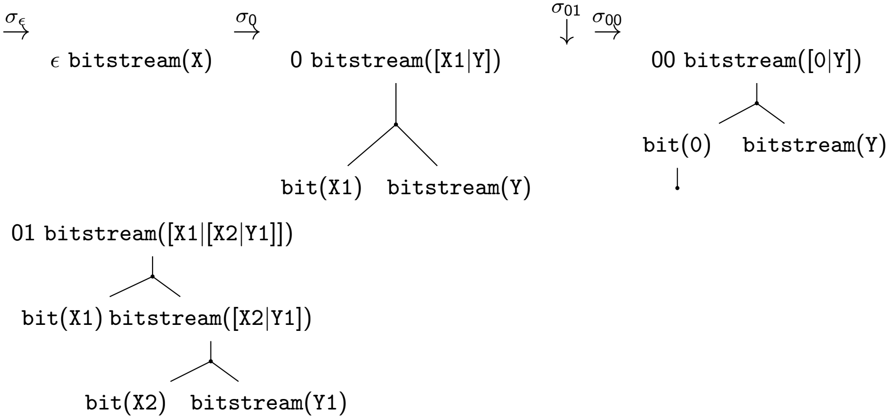


Notation:

$\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P)$	all *finite* coinductive trees over $\mathbf{Term}(\Sigma_P)$
$\mathbf{CTree}^{\infty}(\mathbf{Term}(\Sigma_P), P)$	all *infinite* coinductive trees over $\mathbf{Term}(\Sigma_P)$
$\mathbf{CTree}^{\omega}(\mathbf{Term}(\Sigma_P), P)$	all *finite and infinite* coinductive trees over $\mathbf{Term}(\Sigma_P)$

Dimension-3: Derivation trees

Given an $L \in \mathbb{N}^*$, a coinductive derivation is a map $L \to \mathbf{CTree}(\mathbf{Term}(\Sigma_P), P)$, satisfying the mgu requirement.



Dimension-3 notation

$\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$	all *finite* coinductive derivations over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$
$\mathbf{CDer}^{\infty}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$	all *infinite* coinductive trees over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$
$\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$	all *finite and infinite* coinductive trees over $(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree CT with the root t belongs to $CTree(\mathbf{Term}(\Sigma_P), P)$.

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree CT with the root t belongs to $CTree(\mathbf{Term}(\Sigma_P), P)$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$, E.g. **bit**.

Theory of Productivity for LP

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree CT with the root t belongs to $CTree(\mathbf{Term}(\Sigma_P), P)$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$, E.g. **bit**.
- inductive LPs all derivations for which are in $\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$; E.g. **ListNat**.

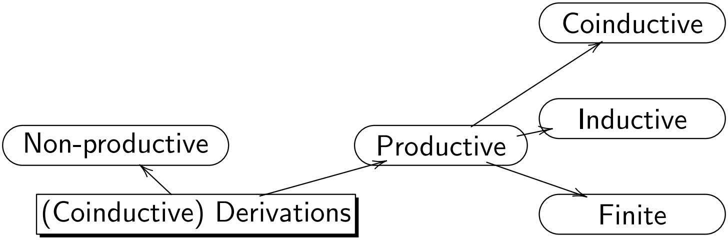
Theory of Productivity for LP

A first-order logic program P is *productive* if

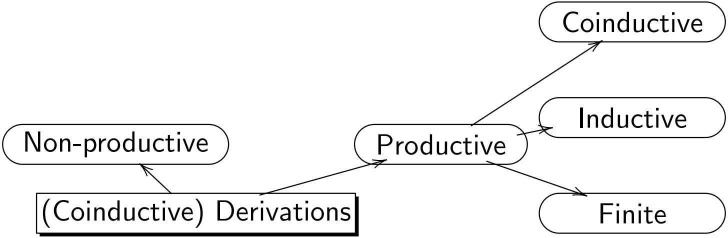
for any term $t \in \mathbf{Term}(\Sigma_P)$, the coinductive tree CT with the root t belongs to $CTree(\mathbf{Term}(\Sigma_P), P)$.

- In the class of Productive LPs, we can further distinguish finite LP that give rise to derivations in $\mathbf{CDer}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$, E.g. **bit**.
- inductive LPs all derivations for which are in $\mathbf{CDer}^{\omega}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P))$; E.g. **ListNat**.
- coinductive LPs all derivations for which are in $\mathbf{CDer}^{\infty}(\mathbf{CTree}(\mathbf{Term}(\Sigma_P), P)))$ E.g. **Stream**.

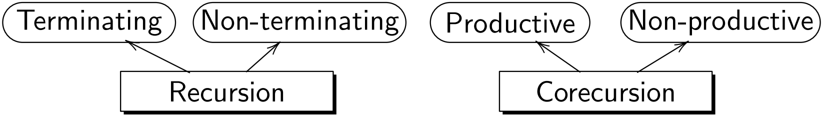
Theory of Productivity in LP



Theory of Productivity in LP

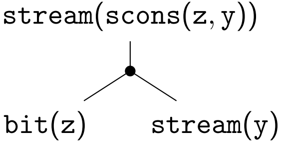


Compare with:



Deciding Productivity: Guardedness

- Dimension 1. Measures of reduction on term trees: `stream(y)` is a reduction of `stream(scons(x,y))`
- Dimension 2. Reduction on coinductive tree loops:



- Dimension 3. Discovery of derivation loops.

Example of guardedness issues

```
p(s(X1),X2,Y1,Y2) ← q(X2,X2,Y1,Y2)
q(X1,X2,s(Y1),Y2) ← p(X1,X2,Y2,Y2)
```

Example of guardedness issues

```
p(s(X1),X2,Y1,Y2) ← q(X2,X2,Y1,Y2)
q(X1,X2,s(Y1),Y2) ← p(X1,X2,Y2,Y2)
```

```
p(s(X1),X2,Y1,Y2)
        |
  q(X2,X2,Y1,Y2)
```

Example of guardedness issues

```
p(s(X1),X2,Y1,Y2) ← q(X2,X2,Y1,Y2)
q(X1,X2,s(Y1),Y2) ← p(X1,X2,Y2,Y2)
```

```
p(s(X1),X2,Y1,Y2)
        •
 q(X2,X2,Y1,Y2)
```

$$p(s(x1), s(x2), s(y1), s(y2))$$
$$|$$
$$q(s(x2), s(x2), s(y1), s(y2))$$
$$|$$
$$p(s(x2), s(x2), s(y2), s(y2))$$
$$|$$
$$q(s(x2), s(x2), s(y2), s(y2))$$
$$|$$
$$\vdots$$

Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

- Imposing guardedness conditions, to ensure every coinductive tree is finite.
  To ban programs that are not guarded by constructors:
  `stream(scons x,y)` $\leftarrow$ `stream(scons x,y)`

Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

- Imposing guardedness conditions, to ensure every coinductive tree is finite.

  To ban programs that are not guarded by constructors:

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

  Unlike termination checks in Coq/Agda cannot be done fully statically (no types to help!), and needs some proof search in Dimension 3.

Soundness of Corecursion in LP

- CoALP is sound and complete for inductive programs;
- Soundness of coinductive programs is our next step.

Two directions:

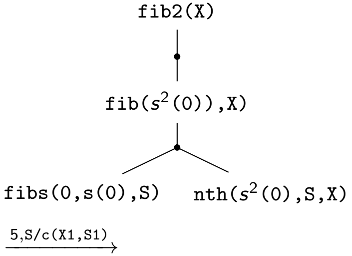
- Imposing guardedness conditions, to ensure every coinductive tree is finite.
  To ban programs that are not guarded by constructors:
  `stream(scons x,y)` $\leftarrow$ `stream(scons x,y)`
  Unlike termination checks in Coq/Agda cannot be done fully statically (no types to help!), and needs some proof search in Dimension 3.
- Determining when it is safe to make a coinductive conclusion (and finding a right coinductive hypothesis).
  (Again, the troubles come form un-typed setting.)

Stream of Fibonacci numbers:

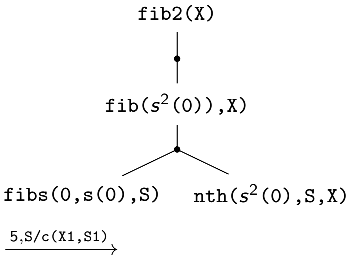
Falls into infinite loops in Prolog and Prolog-like version of CoLP [Gupta et al. 2007] [Both are eager...] Those powerful SAT/SMT solvers would not do it either.

```
add(0,Y,Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
fibs(X,Y,cons(X,S)) :- add(X,Y,Z), fibs(Y,Z,S).
nth(0,cons(X,S),X).
nth(s(N),cons(X,S),Y) :- nth(N,S,Y).
fib(N,X) :- fibs(0,s(0),S), nth(N,S,X).
fib2(X) :- fib(s(s(0)),X).
```

Examples of derivations with Fib: lazy step 1

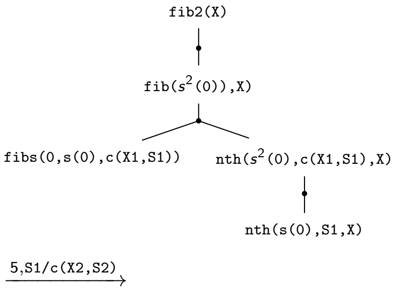


Examples of derivations with Fib: lazy step 1

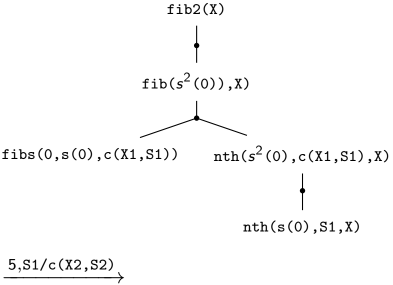



```
1.  add(0,Y,Y).
2.  add(s(X),Y,s(Z)) :-
add(X,Y,Z).
3.  fibs(X,Y,cons(X,S)) :-
add(X,Y,Z), fibs(Y,Z,S).
4.  nth(0,cons(X,S),X).
5.  nth(s(N),cons(X,S),Y) :-
nth(N,S,Y).
6.  fib(N,X) :- fibs(0,s(0),S),
nth(N,S,X).
7.  fib2(X) :- fib(s(s(0)),X).
```

Examples of derivations with Fib: lazy step 2

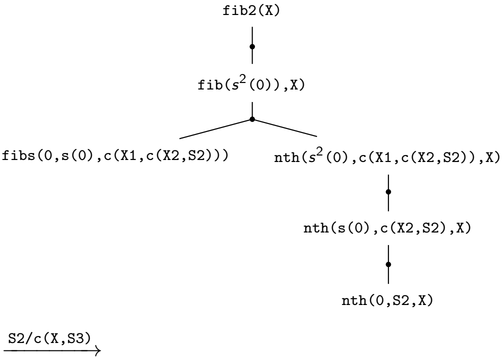


Examples of derivations with Fib: lazy step 2

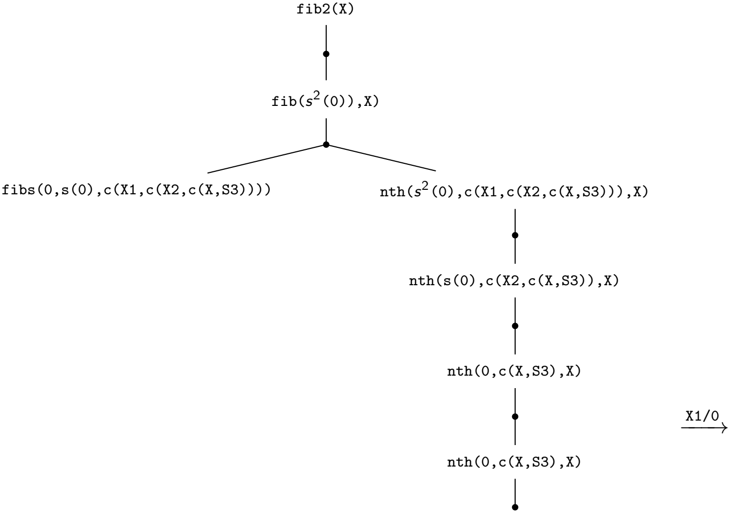



```
1.  add(0,Y,Y).
2.  add(s(X),Y,s(Z)) :-
add(X,Y,Z).
3.  fibs(X,Y,cons(X,S)) :-
add(X,Y,Z), fibs(Y,Z,S).
4.  nth(0,cons(X,S),X).
5.  nth(s(N),cons(X,S),Y) :-
nth(N,S,Y).
6.  fib(N,X) :- fibs(0,s(0),S),
nth(N,S,X).
7.  fib2(X) :- fib(s(s(0)),X).
```

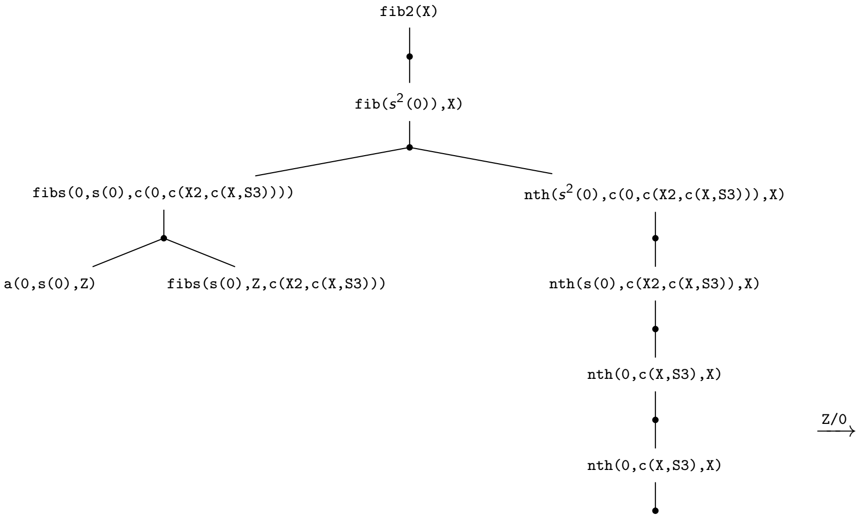
Examples of derivations with Fib: lazy step 3



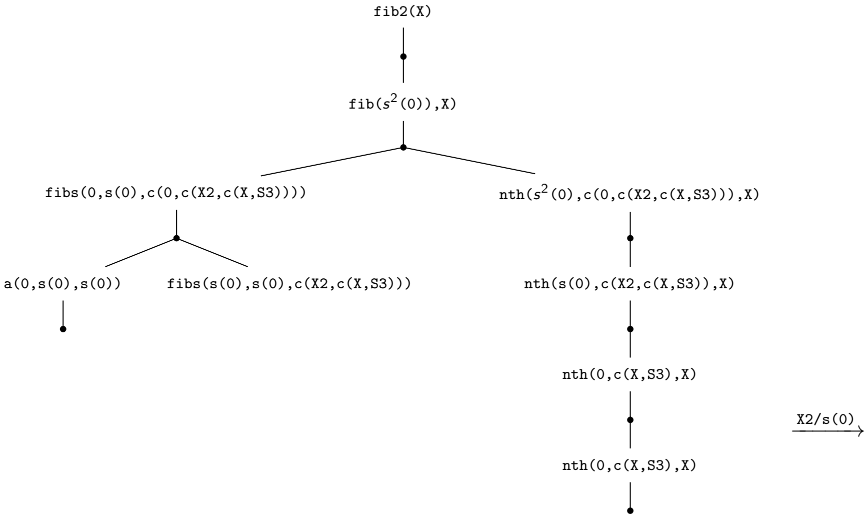
Examples of derivations with Fib: lazy step 4



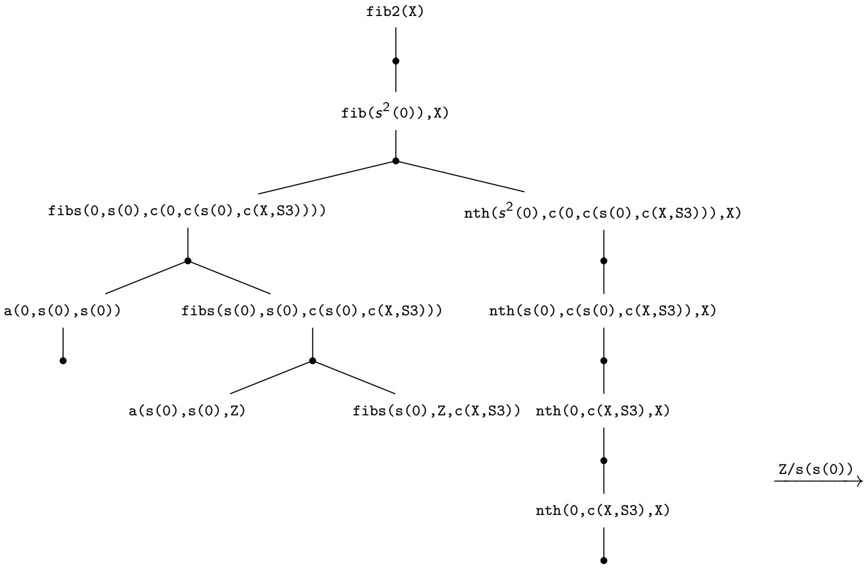
Examples of derivations with Fib: lazy step 5



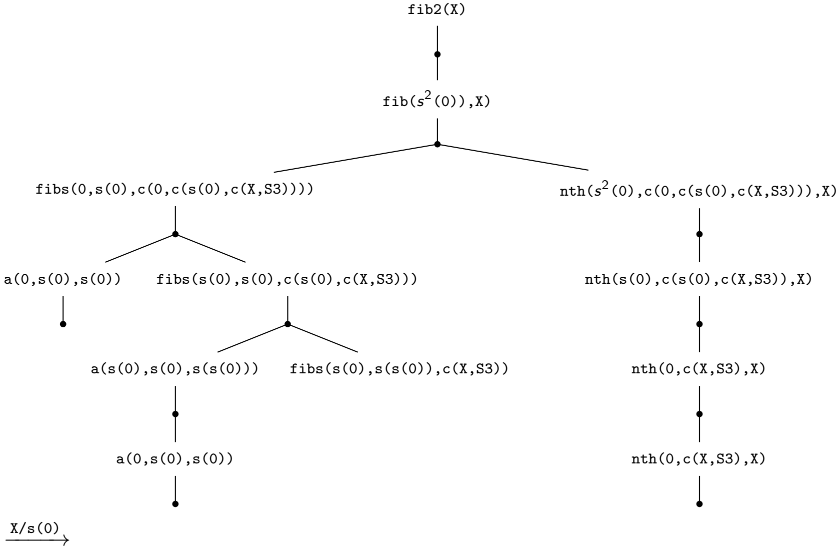
Examples of derivations with Fib: lazy step 6



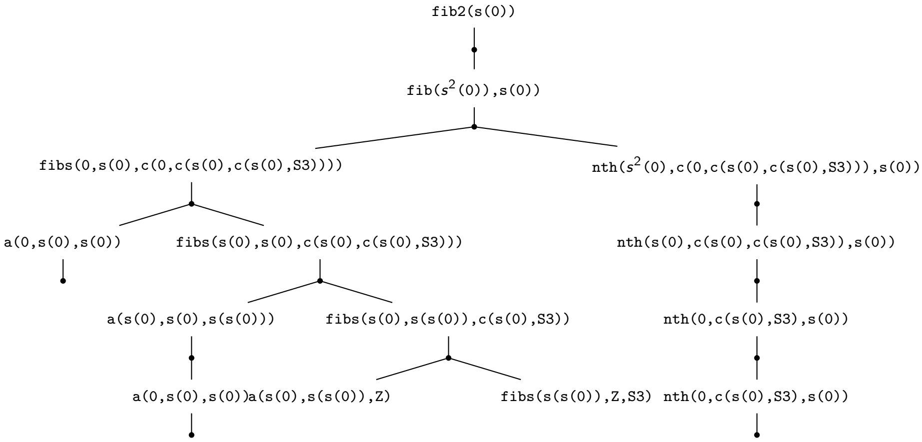
Examples of derivations with Fib: lazy step 7



Examples of derivations with Fib: lazy step 8



Examples of derivations with Fib: lazy step 9



Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Fib example	No	No	No	Yes
Execution	Eager	Eager	Eager	Lazy
Corecursion				
Mode of execution				
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Fib example	No	No	No	Yes
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Guardedness by constructors
Mode of execution				
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Fib example	No	No	No	Yes
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Guardedness by constructors
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics				
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Fib example	No	No	No	Yes
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Guardedness by constructors
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics	lfp	lfp	gfp (restricted)	coalgebraic
Operational semantics				

Logic Programming dialects, compared

	Prolog	Parallel Prolog	Co-LP	CoALP
Fib example	No	No	No	Yes
Execution	Eager	Eager	Eager	Lazy
Corecursion	No	No	by Regular Loop detection	Guardedness by constructors
Mode of execution	Sequential	Parallel	Sequential	Parallel
Declarative semantics	lfp	lfp	gfp (restricted)	coalgebraic
Operational semantics	transitions; states: lists of formulae	transitions; states: lists of formulae	transitions; states: lists of formulae	transitions; states: coinductive trees

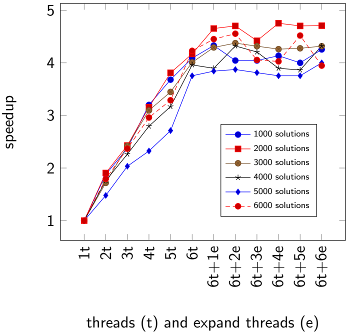
Outline

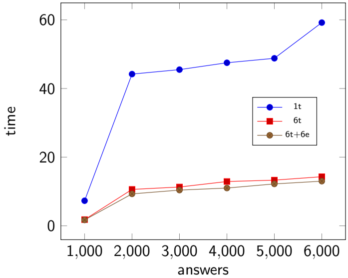
Parallelising CoALP

Komendantskaya, Schmidt, Heras: *Exploiting Parallelism in Coalgebraic Logic Programming*, ENTCS, 2014

```
1.  bit(0).
2.  bit(1).
3.  btree(empty).
4.  btree(tree(L,X,R)) ← btree(L), bit(X), btree(R).
```

Parallel CoALP





Directions we are exploring

Haskell implementation is nearly finished.

Current task: to find a "right" language to try CoALP-based type inference

- Using CoALP in Hume: for analysis of stream-based networks and/or for type inference;
- Type-inference in Haskell;
- SSReflect: overloading in canonical structures currently requires the use of back-tracking in LP-like algorithm. It could be parallel CoALP execution instead;
- CoALP for global type analysis in object-oriented languages: CoLP is already used for that.
- Formal Verification of CoALP-based type inference

The end

- Komendantskaya, Power, Schmidt: Coalgebraic Logic Programming: from Semantics to Implementation, Journal of Logic and Computation, 2014.
- Komendantskaya, Schmidt, Heras: *Exploiting Parallelism in Coalgebraic logic Programming*, ENTCS, 2014.
- A paper on implementing lazy guarded corecursion in CoALP using Haskell is in preparation...
- CoALP webpage has various prototype implementations to play with... http://staff.computing.dundee.ac.uk/katya/CoALP/

We will be happy to apply CoALP for TI (or other purposes) in *YOUR* language!

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.

Trend in type inference:

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

- *Dependent Types*,
- *Type Classes* [Wadler&Blott 89],
- *Generalised Algebraic Types* (GADTs) [Peyton Jones & al, 2006]
- *Dependent Type Classes* [Sozeau & al 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 additional inference algorithms.

Trend in type inference:

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

- *Dependent Types*,
- *Type Classes* [Wadler&Blott 89],
- *Generalised Algebraic Types* (GADTs) [Peyton Jones & al, 2006]
- *Dependent Type Classes* [Sozeau & al 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 additional inference algorithms.

Implementations of new type inference algorithms include a variety of first-order decision procedures, notably Unification and Logic Programming (LP) [Peyton Jones & al, 2006], Constraint LP [Odersky Sulzmann, Vytiniotis & many more 1999-], LP embedded into interactive tactics (Coq's *eauto*) Sozeau & al. 08], and LP supplemented by rewriting [Gonthier & al, 11].

Motivation: type inference with Polymorphic types

List Length in Haskell

```
length ::  [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

Logic program for type inference

$$
\begin{aligned}
&\texttt{cons}(\texttt{X}) && \leftarrow \texttt{X} = \texttt{Y} \rightarrow \texttt{list}(\texttt{Y}) \rightarrow \texttt{list}(\texttt{Y}).\\
&\texttt{plus}(\texttt{X}) && \leftarrow \texttt{X} = \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}.\\
&\texttt{nil}(\texttt{X}) && \leftarrow \texttt{X} = \texttt{list}(\texttt{Y}).\\
&\texttt{length}(\texttt{X}) && \leftarrow (\texttt{X} = \texttt{Y} \rightarrow \texttt{Z}) \ \& \ \texttt{nil}(\texttt{Y}) \ \& \ \texttt{Z} = \texttt{int} \ \& \ \texttt{cons}(\texttt{W}) \ \&\\
& && \quad (\texttt{W} = \texttt{W1} \rightarrow \texttt{W2} \rightarrow \texttt{Y}) \ \& \ \texttt{plus}(\texttt{U}) \ \&\\
& && \quad (\texttt{U} = \texttt{int} \rightarrow \texttt{Z} \rightarrow \texttt{Z}) \ \& \ \texttt{W2} = \texttt{Y}.
\end{aligned}
$$

Query: length(X)?

Answer (any existing PROLOG version): $X = list(\_) \rightarrow int$.

Trend to do more by type-inference:

... session types,

... writing contracts by means of types:

Example

Vytiniotis et al. "HALO: Haskell to Logic Through Denotational Semantics" [POPL'13]

```
f xs = head (reverse (True :  xs))
g xs = head (reverse xs)
```

Both f and g are well typed and "'can't go wrong"' in Milner's sense, but g will crash for empty list, and `f` will never crash.

Contract:

$$\texttt{reverse} \in (\texttt{xs} : \texttt{CF}) \rightarrow \{\texttt{ys} \mid \texttt{null xs} <=> \texttt{null ys}\}$$

Requires strong first-order type inference engines: Z3, Vampire, E...

Could it get any better?

- Clear trend on Type theory side: increase in type expressivenes (dependent types, GADTs, type classes, session types, etc etc)

Could it get any better?

- Clear trend on Type theory side: increase in type expressivenes (dependent types, GADTs, type classes, session types, etc etc)
- Chaotic use of type-inference engines, also known in the literature as "using off-the-shelf" first order TPs.

Could it get any better?

- Clear trend on Type theory side: increase in type expressivenes (dependent types, GADTs, type classes, session types, etc etc)
- Chaotic use of type-inference engines, also known in the literature as "using off-the-shelf" first order TPs.
- Would it pay-off to get more conceptually elegant on type inference side? – especially bearing in mind the big emphasis on type inference in more expressive type systems.

Could it get any better?

- Clear trend on Type theory side: increase in type expressivenes (dependent types, GADTs, type classes, session types, etc etc)
- Chaotic use of type-inference engines, also known in the literature as "using off-the-shelf" first order TPs.
- Would it pay-off to get more conceptually elegant on type inference side? – especially bearing in mind the big emphasis on type inference in more expressive type systems.
- Would our "Coalgebraic Logic programming" grow to become a type-inference specific theorem prover (with stronger theoretical background and motivation than state-of-the-art SAT/SMT-solvers)?