Structural Resolution meets Curry-Howard

Katya Komendantskaya (Joint work with Patricia Johann and Peng Fu)

School of Computing, University of Dundee, UK

Birmingham, 06 November 2015

About myself

- 1998 – 2003 BSc+MSc degree in Logic, Moscow State University.
- 2004 – 2007 PhD in the University College Cork, Ireland.
- 2007 – 2008 First postdoc project with Yves Bertot in INRIA, France - on guardedness of corecursive functions in Coq. (I study coinduction/corecursion in different domains since then.)
- 2008 – 2011 EPSRC TCS research fellowship first in St Andrews, later transferred to the School of Computing in Dundee.
- 2010 – 2016 – Lecturer, Senior Lecturer, Reader in Computing, University of Dundee.
- 2016 – ?? – Associate Professor at Heriot-Watt University, Edinburgh.

Big Picture: Coinduction in Theorem Proving

- Pre-history is given be the studies of recursion and fixed points: Knaster, Tarski, Kleene, first half of XX century;
- 1994 Coquand and 1996 Gimenez – bring inductive/coinductive methods to Type theory and Coq;
- Nowadays, most mainstream Interactive Theorem Provers (ITP) have support for induction and coinduction:
  - Inductive and Coinductive types
  - Recursive and Corecursive functions
  - Inductive and Coinductive proof principles.

Big Picture: Coinduction in Theorem Proving

- Pre-history is given be the studies of recursion and fixed points: Knaster, Tarski, Kleene, first half of XX century;
- 1994 Coquand and 1996 Gimenez – bring inductive/coinductive methods to Type theory and Coq;
- Nowadays, most mainstream Interactive Theorem Provers (ITP) have support for induction and coinduction:
  - Inductive and Coinductive types
  - Recursive and Corecursive functions
  - Inductive and Coinductive proof principles.

It happens that type-theoretic setting gives just the right background for development of coinductive methods: pattern and co-pattern matching, laziness, productivity and guardedness, constructive view of proofs.

Coinduction in Automated Theorem Proving (ATP)

- Coinductive data types are as natural and common (in verification, CS) as inductive;
- There is a need to reason about infinite/cyclic computation
- Would enrich ATPs and allow more elegant programs.

Coinduction in Automated Theorem Proving (ATP)

- Coinductive data types are as natural and common (in verification, CS) as inductive;
- There is a need to reason about infinite/cyclic computation
- Would enrich ATPs and allow more elegant programs.

. CoInductive LP [Gupta et al, 2007]

. [Leino 2013]: Coinduction in Dafny. (A big part of motivation is to mimic ITPs)

. [Reynolds and Blanchette 2015]: Coinduction for SMT Solvers, CADE 2015.

. Corecursion arising in Type inference in Functional Programming [Lamel, Peyton Jones], and OO Programming [Ancona]

Coinduction in Automated Theorem Proving (ATP)

- Coinductive data types are as natural and common (in verification, CS) as inductive;
- There is a need to reason about infinite/cyclic computation
- Would enrich ATPs and allow more elegant programs.

. CoInductive LP [Gupta et al, 2007]

. [Leino 2013]: Coinduction in Dafny. (A big part of motivation is to mimic ITPs)

. [Reynolds and Blanchette 2015]: Coinduction for SMT Solvers, CADE 2015.

. Corecursion arising in Type inference in Functional Programming [Lamel, Peyton Jones], and OO Programming [Ancona]

Note: introduced MUCH later and in a limited form.

Bigger Goal and Motivation

1. Understand reasons why coinduction does not come to ATP as easily as to ITP;
2. Find new – more structural and more constructive – foundations for proof-search in ATP;
3. ... thereby introducing better (more natural and expressive) coinductive methods to ATP;
4. Bridge the gap between ITP and ATP, in which best sides of the two worlds are presented.

Bigger Goal and Motivation

1. Understand reasons why coinduction does not come to ATP as easily as to ITP;
2. Find new – more structural and more constructive – foundations for proof-search in ATP;
3. ... thereby introducing better (more natural and expressive) coinductive methods to ATP;
4. Bridge the gap between ITP and ATP, in which best sides of the two worlds are presented.

My goal for today:

- illustrate and explain points 1-3 by means of (first-order) Horn Clause Logic and Resolution;
- invite discussion (and collaborators!)

Outline

Syntax of Horn-clause Logic

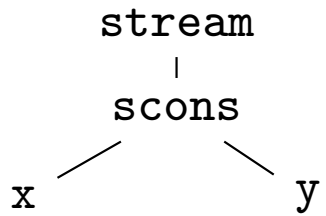
First-order signature Σ and terms, term-trees

- function symbols with arity;
- variables.

Example

- `stream` – arity 1
- `scons` – arity 2

Term-trees are trees over $\Sigma \cup V$, subject to branching $\approx$ arity:



Term Notation:

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

$\mathbf{GTerm}(\Sigma)$, $\mathbf{GTerm}^{\infty}(\Sigma)$, $\mathbf{GTerm}^{\omega}(\Sigma)$ will denote sets of ground (variable free) terms.

Syntax of Horn-clause Logic

Horn Clauses

Given $A, B_1, \ldots, B_n \in \mathbf{Term}(\Sigma)$,

- a definite clause $A \leftarrow B_1, \ldots, B_k$
- a goal clause $\leftarrow B_1, \ldots, B_k$

Universal quantification is assumed.

A (definite) logic program is a finite set of definite clauses

... Gives us a Turing-complete programming language.

Example: lists of natural numbers

Example

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

Logic Programming...

SLD resolution = Unification + Search

SLD-resolution

Program **NatList**:

Example

1.`nat(0)` $\leftarrow$
2.`nat(s(x))` $\leftarrow$ `nat(x)`
3.`list(nil)` $\leftarrow$
4.`list(cons(x,y))` $\leftarrow$

`nat(x), list(y)`

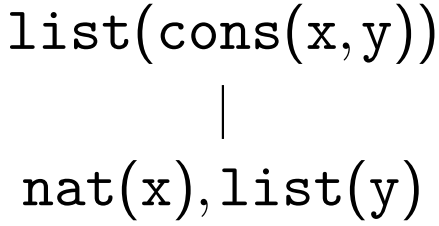
`list(cons(x,y))`

SLD-resolution

Example

1. `nat(0)` $\leftarrow$
2. `nat(s(x))` $\leftarrow$ `nat(x)`
3. `list(nil)` $\leftarrow$
4. `list(cons(x,y))` $\leftarrow$

`nat(x), list(y)`

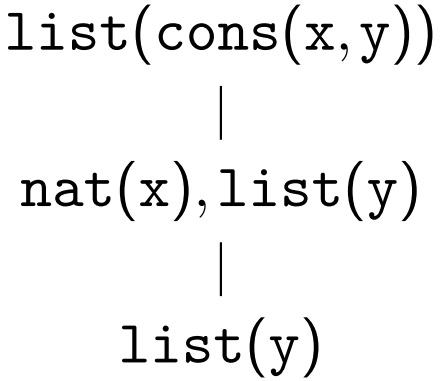


SLD-resolution

Example

1. `nat(0)` $\leftarrow$
2. `nat(s(x))` $\leftarrow$ `nat(x)`
3. `list(nil)` $\leftarrow$
4. `list(cons(x,y))` $\leftarrow$

`nat(x), list(y)`

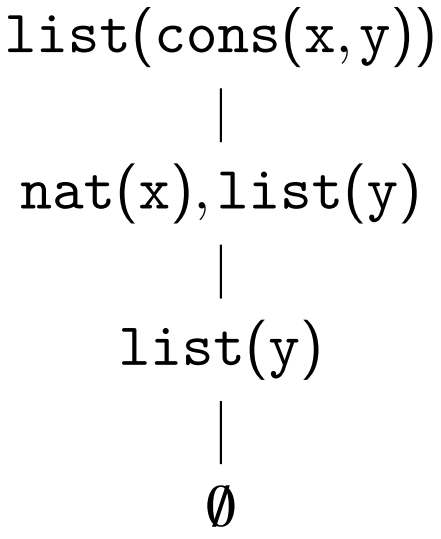


SLD-resolution

Example

1.`nat(0)` $\leftarrow$
2.`nat(s(x))` $\leftarrow$ `nat(x)`
3.`list(nil)` $\leftarrow$
4.`list(cons(x,y))` $\leftarrow$

`nat(x), list(y)`



The answer is "Yes", `NatList` $\vdash$ *list*(*cons*(x, y)) if $x/0$, y/nil, but we can get more substitutions by backtracking.
SLD-refutation = finite successful SLD-derivation.

Horn Clauses in Type Class Inference (Haskell)

Equality type class

```
class Eq a where
(==) :: a -> a-> Bool.
```

Equality class instance declaration for datatype List

```
instance Eq x => Eq (List x) where...
```

Horn-Clause view of equality class instances for List and Char:

$$\begin{aligned} \kappa_1 : &\quad \mathrm{Eq}(x) \Rightarrow \mathrm{Eq}(\mathrm{List}(x)) \\ \kappa_2 : &\quad \Rightarrow \mathrm{Eq}(\mathrm{Char}) \end{aligned}$$

Outline

Big-Step Semantics of LP

Definition (Big step rule)

$$\frac{P \models \sigma(B_1), \ldots, P \models \sigma(B_n)}{P \models \sigma(A)},$$

for some grounding substitution σ, and $A \leftarrow B_1, \ldots B_n \in P$.

Example

Logic program *Nat*

1.`nat(0)` $\leftarrow$

2.`nat(s(x))` $\leftarrow$ `nat(x)`

Inductive Semantics of LP

Definition (Big step rule)

$$\frac{P \models \sigma(B_1), \ldots, P \models \sigma(B_n)}{P \models \sigma(A)},$$

for some grounding substitution σ, and $A \leftarrow B_1, \ldots B_n \in P$.

Definition

The *least Herbrand model* for P is the smallest set $M_P \subseteq \mathbf{GTerm}(\Sigma)$ *closed forward* under the rules.

Example

Taking the logic program *Nat*, we obtain the set $M_{Nat} = \{\texttt{nat(0)}, \texttt{nat(s(0))}, \texttt{nat(s(s(0)))}, \ldots\}$.

CoInductive Semantics of LP

Definition (Big step rule)

$$\frac{P \models \sigma(B_1), \ldots, P \models \sigma(B_n)}{P \models \sigma(A)},$$

for some grounding substitution σ, and $A \leftarrow B_1, \ldots B_n \in P$.

Definition

The *greatest complete Herbrand model* for P is the largest set $M_P^\omega \subseteq \mathbf{GTerm}^\omega(\Sigma)$ *closed backward* under the rules.

Example

M_{Nat}^ω will now be given by the set:
$\{\mathtt{nat(0)}, \mathtt{nat(s(0))}, \mathtt{nat(s(s(0)))}, \ldots\} \bigcup \{\mathtt{nat(s(s(...)))}\}$.

Coinductive programs

Some programs have only one natural interpretation:

Example

1.`bit(0)` $\leftarrow$
2.`bit(1)` $\leftarrow$
3.`stream(scons(x,y))` $\leftarrow$ `bit(x), stream(y)`

$M_{Stream} = \{$`bit(0)`$,$`bit(1)`$\}$
$M^{\omega}_{Stream} = \{$`bit(0)`$,$`bit(1)`$,$`stream(scons(0,scons(0, ...))`$,$
`stream(scons(1,scons(0, ...))`$,\ldots\}$

Outline

SLD-Resolution as a reduction system

Given a logic program P, and terms $t_1, \ldots, t_i, \ldots, t_n$ we define

- SLD-reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightsquigarrow [\sigma(t_1), \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, \sigma(t_n)]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $t_i \sim_\sigma A$.

unifiers

$t \sim_\sigma t'$ denotes a unifier of t and t', i.e. $\sigma(t) = \sigma(t')$

SLD-Resolution as a reduction system

Given a logic program P, and terms $t_1, \ldots, t_i, \ldots, t_n$ we define

- SLD-reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightsquigarrow [\sigma(t_1), \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, \sigma(t_n)]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $t_i \sim_\sigma A$.

unifiers

$t \sim_\sigma t'$ denotes a unifier of t and t', i.e. $\sigma(t) = \sigma(t')$

SLD-resolution is sound and complete, inductively [70s Apt, Van Emden, Kowalski]

- If $P \vdash t \rightsquigarrow^n \emptyset$, for some $t \in \mathbf{Term}(\Sigma)$, then there exists a substitution θ such that $\theta(t) \in M_P$.
- If $\theta(t) \in M_P$ for some grounding substitution θ, then there exists a reduction $P \vdash t \rightsquigarrow^n \emptyset$, with an answer σ such that there exists θ' making $\theta = \theta'\sigma$.

Coinductive soundness of SLD-resolution: "Computations at infinity" [80s, van Emden&Abdallah, Lloyd]

Definition

An infinite term t is *SLD-computable at infinity* with respect to a program P if there exist a finite term t' and an infinite fair SLD-derivation $G_0 = (? \leftarrow t'), G_1, G_2, \ldots G_k \ldots$ with mgus $\theta_1, \theta_2, \ldots \theta_k \ldots$ such that $d(t, \theta_k \ldots \theta_1(t')) \to 0$ as $k \to \infty$.

An SLD-derivation is *fair* if either it is finite, or it is infinite and, for every atom B appearing in some goal in the derivation, (a further instantiated version of) B is chosen within a finite number of steps.

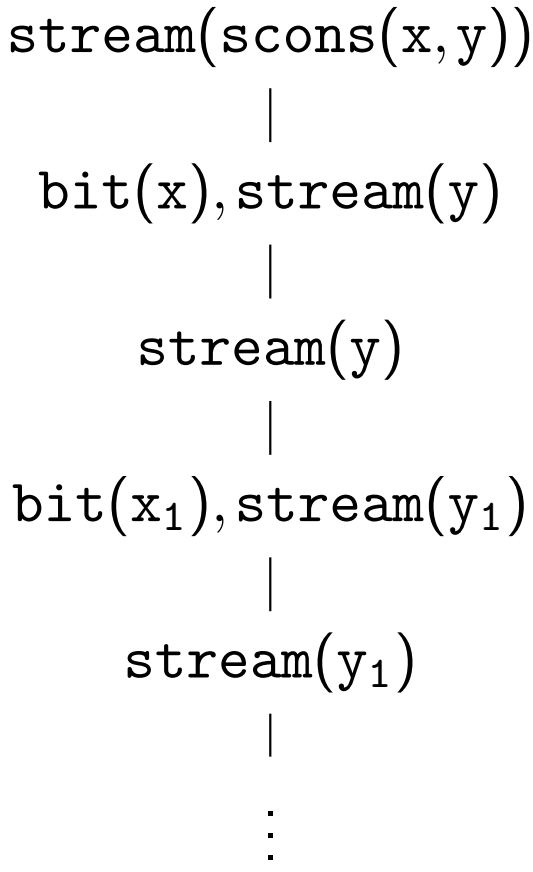
Example

Program **Stream**:

Example

1.bit(0) $\leftarrow$
2.bit(1) $\leftarrow$
3.stream(scons(x,y)) $\leftarrow$

bit(x), stream(y)



At infinity, a term scons(0, scons(0, ...)) is computed.

Computations at infinity are sound

Defining $C_P = \{t \in \mathbf{GTerm}^{\infty}(\Sigma) \mid$ there exists a term t' such that t is SLD-computable at infinity with respect to P by $t'\}$.

Theorem (Van Emden&Abdallah, Lloyd, 80s)

Given a $P \in \mathbf{LP}(\Sigma)$, $C_P \subseteq M_P^{\omega}$.

Computations at infinity are sound

Defining $C_P = \{t \in \mathbf{GTerm}^{\infty}(\Sigma) \mid$ there exists a term t' such that t is SLD-computable at infinity with respect to P by $t'\}$.

Theorem (Van Emden&Abdallah, Lloyd, 80s)

Given a $P \in \mathbf{LP}(\Sigma)$, $C_P \subseteq M_P^{\omega}$.

A great insight:

This work was an early example of productive coinduction: requiring an infinite computation must produce an infinite term.

Two problems:

- Computations at infinity are not complete;
- They do not give rise to an implementable algorithm.

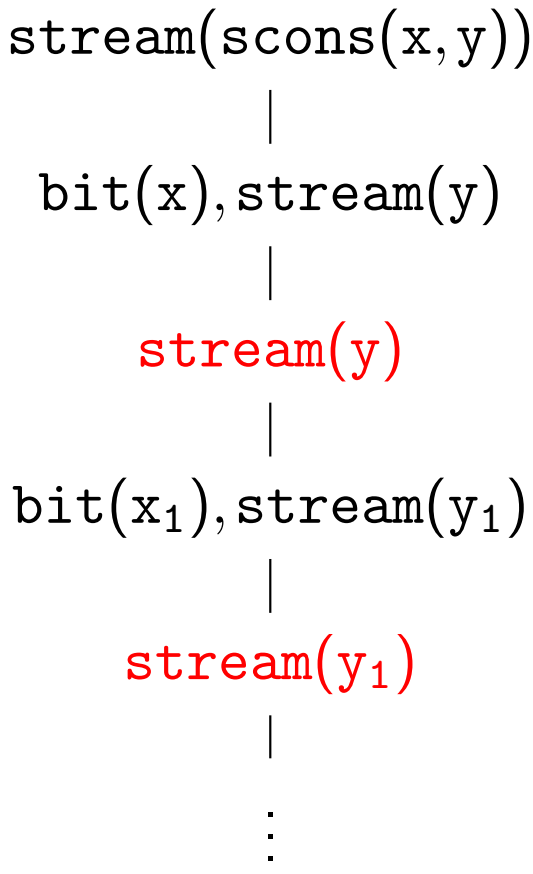
First Solution: CoLP [Gupta et al, 2007]

Program **Stream**:

Example

1.bit(0) ←
2.bit(1) ←
3.stream(scons(x,y)) ←

bit(x), stream(y)



Look for a loop in resolvents: if it is found, close by circular unification: taking $\mathtt{stream}(\mathtt{y})$ and $\mathtt{stream}(\mathtt{scons}(0,\mathtt{y}_1))$ by circular unification gives $\mathtt{y} \mapsto \mathtt{scons}(0,\mathtt{y})$. Thus, we get a coinductive answer $\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \mathtt{scons}(0,\mathtt{y})$.

Properties of CoLP

1. Sound relative to greatest complete Herbrand models;
2. Incomplete relative to greatest complete Herbrand models;
3. Neither sound nor complete relative to computations at infinity:

Example (Unsound)

Program:
$\mathtt{bad(x)} \leftarrow \mathtt{bad(x)}$
Will give rise to a reduction:
$\mathtt{bad(x)} \rightsquigarrow \mathtt{bad(x)} \rightsquigarrow \mathtt{bad(x)} \rightsquigarrow \ldots$
CoLP will conclude $bad(x)$ is entailed by the program, using loop detection. However, no infinite term is computed at infinity for this program. Its Herbrand model will not contain infinite terms, either.

Conclusion: CoLP's loop detection does not guarantee, on its own, productivity of computations.

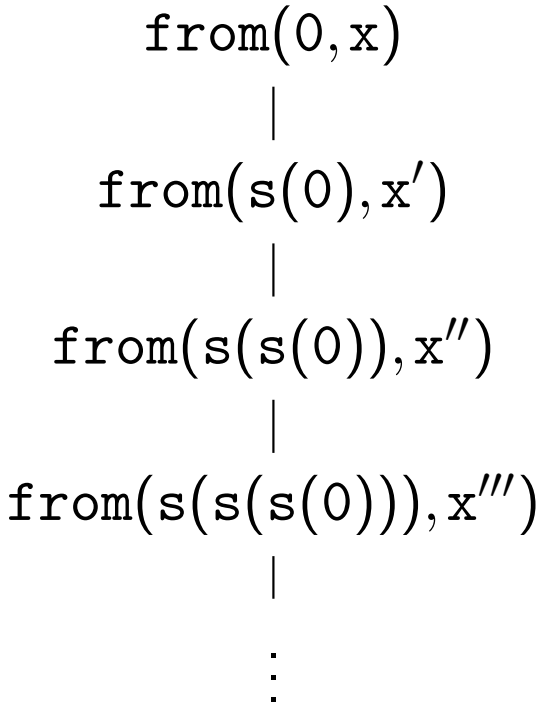
Incompleteness of CoLP for Irregular term trees

Program **From**:

Example

$1.\texttt{from(x,scons(x,y))} \leftarrow$

$\texttt{from(s(x),y)}$



No loop found, and CoLP does not terminate. But the term $\texttt{from(0,scons(0,scons(s(0),scons(s(s(0)),...))))}$ is computable at infinity.

A Fundamental Problem

Productivity theory for LP is absent:

- Operational Semantics of the 80s gives a notion of global productivity, but it is not implementable;
- Loop detection of CoLP is implementable but does not ensure global productivity;
- Notion of Observational Productivity is missing in both cases

Global and Observational Productivity elsewhere in Theorem Proving

In interactive theorem provers (Coq, Agda)
Values in co-inductive types are observationally productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

Observational Productivity

We call a function (observationally) productive, if, for any given input, it outputs a productive value.

Global and Observational Productivity elsewhere in Theorem Proving

Observational productivity is guaranteed by means of *guardedness checks* imposed on all corecursive functions:

```
CoInductive Stream A : Type :=
    SCons : A → Stream A → Stream A.
```

Observationally productive function:

```
CoFixpoint fib (n m : nat) :=SCons n (fib m (n+m)).
```

Observationally non-productive function:

```
CoFixpoint bad (f : A → A) (x : A) : Stream A :=
bad f (f x).
```

Observational productivity guarantees global productivity.

In search of a missing theory of productivity for ATP

- Direct borrowing of methods from ITP is hard:
  1. We have no types and type constructors to use as "guards";
  2. We have no pattern-matching.

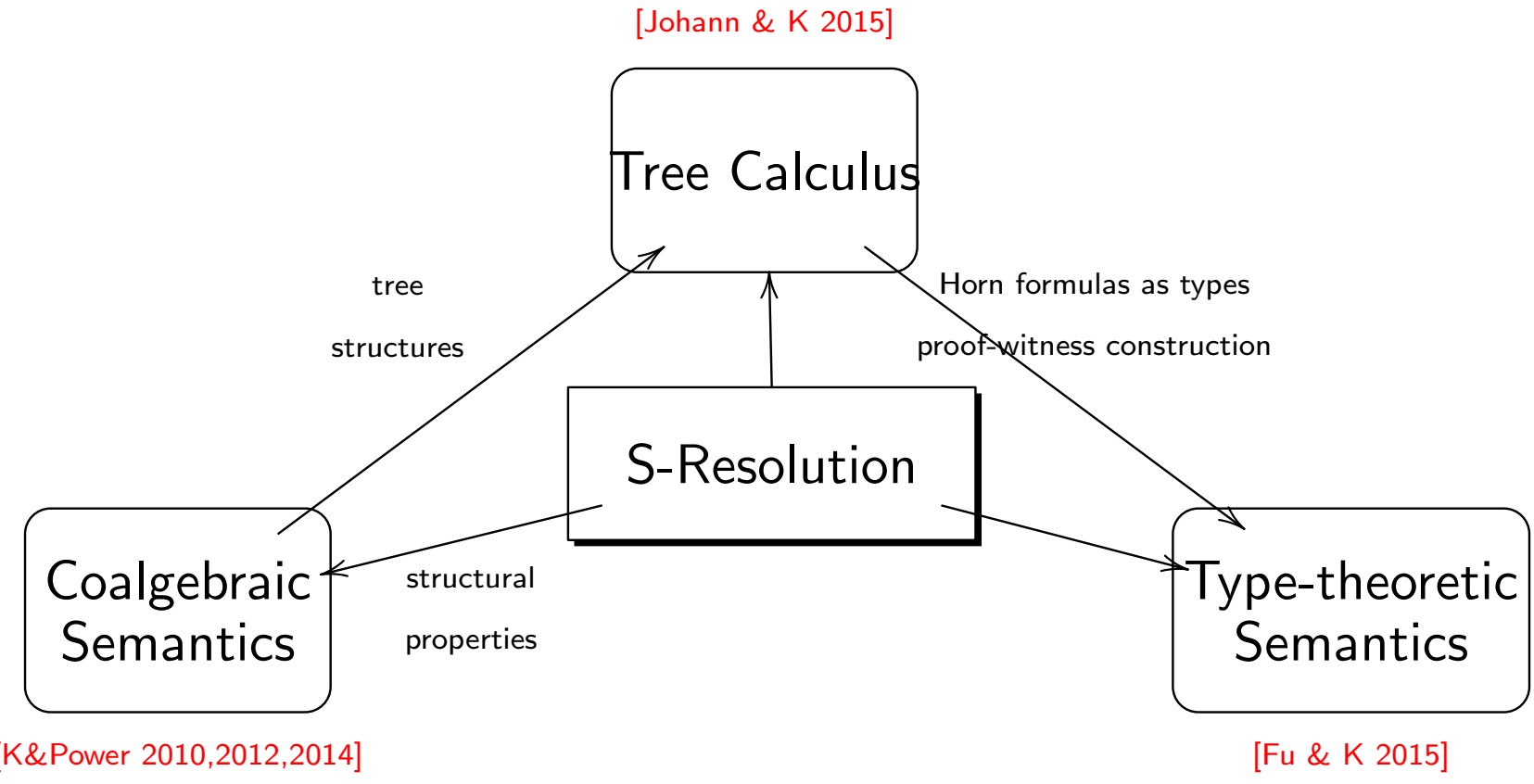
Is there a mysterious Missing productivity theory for LP and wider for ATP?

– Structural Resolution (also S-Resolution)

Outline

Structural Resolution:

a new operational semantics for productive coinductive proof search.



S-resolution reductions

matchers

$t \prec_\sigma t'$ denotes a matcher of t and t', i.e. $\sigma(t) = t'$

- SLD-reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightsquigarrow [\sigma(t_1), \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, \sigma(t_n)]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $t_i \sim_\sigma A$.
- Term-Matching reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightarrow [t_1, \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, t_n]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $A \prec_\sigma t_i$.

S-resolution reductions

matchers

$t \prec_\sigma t'$ denotes a matcher of t and t', i.e. $\sigma(t) = t'$

- SLD-reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightsquigarrow [\sigma(t_1), \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, \sigma(t_n)]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $t_i \sim_\sigma A$.
- Term-Matching reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightarrow [t_1, \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, t_n]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $A \prec_\sigma t_i$.
- Substitutional reduction: $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \hookrightarrow [\sigma(t_1), \ldots, \sigma(t_i), \ldots, \sigma(t_n)]$ if $A \leftarrow B_1, \ldots, B_m \in P$, and $t_i \sim_\sigma A$.
- S-resolution reduction: $P \vdash [\overline{t}] \rightarrow^\mu \circ \hookrightarrow^1 [\overline{t'}]$.

Then, $(P, \rightsquigarrow)$ is a reduction system that models SLD-reductions for P, and $(P, \rightarrow^\mu \circ \hookrightarrow^1)$ is a reduction system that models S-resolution reductions for P.

Example

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

1. SLD-resolution reduction:
   $[stream(x)] \rightsquigarrow [bit(x'), stream(y)] \rightsquigarrow [stream(y)] \rightsquigarrow$
   $[bit(x''), stream(y')] \rightsquigarrow \ldots$
2. Term-matching reduction: $[stream(x)] \nrightarrow$
3. S-resolution reduction:
   $[stream(x)] \hookrightarrow^1 [stream(scons(x', y))] \rightarrow^\mu$
   $[bit(x'), stream(y)] \hookrightarrow^1 [bit(0), stream(y)] \rightarrow^\mu [stream(y)] \hookrightarrow^1$
   $[stream(scons(x'', y'))] \rightarrow^\mu [bit(x''), stream(y')] \ldots$

Note how term-matching ($\approx$ pattern-matching) behaves for this coinductive program!

Productivity for free

Definition (Observational Productivity)

A logic program P is observationally productive if the reduction system $(P, \rightarrow)$ is strongly normalising, i.e. if every term-matching reduction is finite for P.

Example

All programs we have seen are productive with the exception of `Bad`. Program `From`, troublesome for CoLP's loop detection, is also productive, just as it would be in e.g. Coq.

Productivity for free

Definition (Observational Productivity)

A logic program P is observationally productive if the reduction system $(P, \rightarrow)$ is strongly normalising, i.e. if every term-matching reduction is finite for P.

Example

All programs we have seen are productive with the exception of `Bad`. Program `From`, troublesome for CoLP's loop detection, is also productive, just as it would be in e.g. Coq.

Benefits:

1. first-ever notion of observational productivity for LP;
2. simple and natural
3. more abstract than (and independent of) loop detection
4. Guarantees global productivity?

Lost completeness

By decomposing $\rightsquigarrow$ into $\rightarrow^{\mu} \circ \hookrightarrow^{1}$ we lost completeness of search (compared to SLD-resolution).

Example

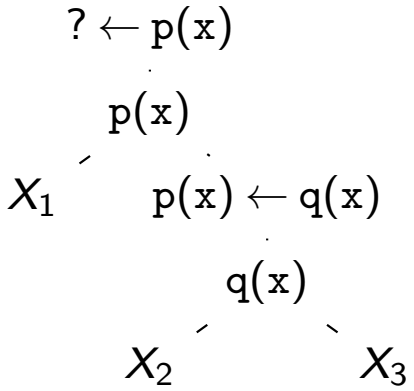
Consider the following program:
0. $\mathtt{p(c)} \leftarrow$
1. $\mathtt{p(X)} \leftarrow \mathtt{q(X)}$

We can show that $P \models \mathtt{p(c)}$, but $\mathtt{p(X)}$ will not result in a successful S-resolution reduction: $P \vdash \mathtt{p(X)} \rightarrow \mathtt{q(X)} \not\rightarrow$

Before we talk about coinductive properties of S-resolution, we must re-gain completeness of search.

Rewriting trees

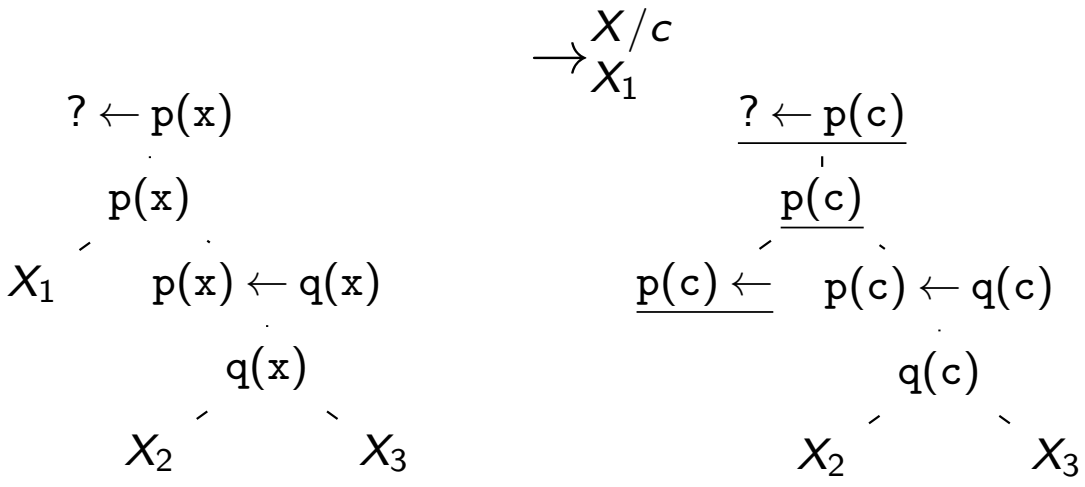
A rewriting tree is a tree over $\mathbf{Term}(\Sigma) \cup \mathbf{Clause}(\Sigma) \cup Var_R$, subject to arity conditions: arity of an and-node is a number of clauses in the program, arity of an or-node is a number of terms in the clause body.



our example

0. $\mathtt{p(c)} \leftarrow$
1. $\mathtt{p(x)} \leftarrow \mathtt{q(x)}$

Rewriting tree substitutions and transitions

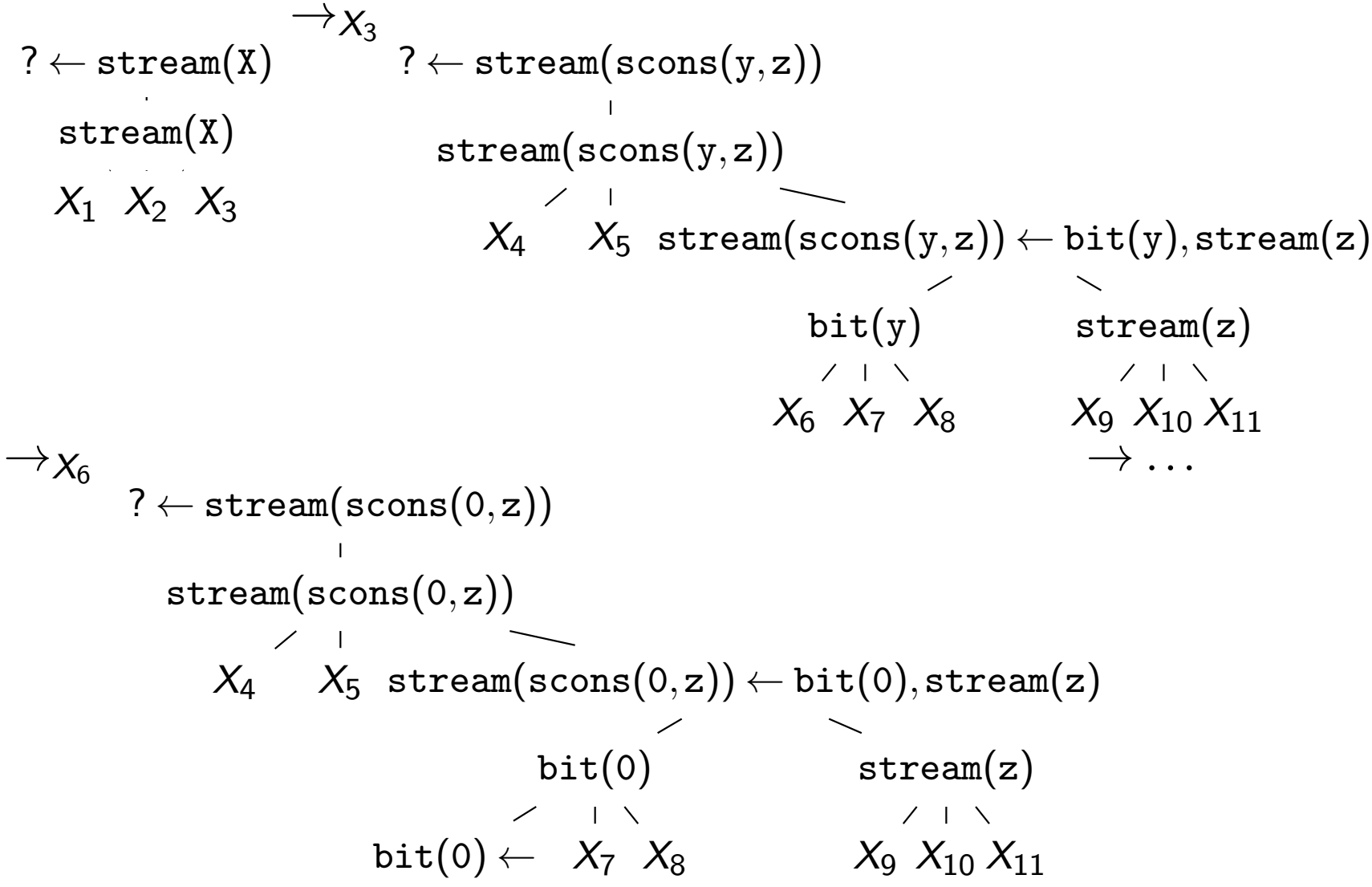


We call such transitions S-derivations

our example

0. p(c) ←
1. p(x) ← q(x)

A coinductive productive S-derivation



We separate proof construction (done "vertically" by term matching) and proof search (done "horizontally" by unification)

Logic and Control

[Kowalski, 74]:

Logic Programming = Logic + Control

In fact, it was:

SLD-derivations = SLD-resolution + Searching strategies

Logic and Control

[Kowalski, 74]:

Logic Programming = Logic + Control

In fact, it was:

SLD-derivations = SLD-resolution + Searching strategies

CoLP = SLD-resolution + Searching strategies + Loop detection

Logic and Control

[Kowalski, 74]:

Logic Programming = Logic + Control

In fact, it was:

SLD-derivations = SLD-resolution + Searching strategies

CoLP = SLD-resolution + Searching strategies + Loop detection

We now have:

S-derivations = S-resolution + Tree Calculus

- S-resolution separates proof-construction from proof-search;
- Tree calculus makes it inductively sound and complete.
- Observational Productivity is a part of Logic (rather than Control)

Logic and Control

[Kowalski, 74]:

Logic Programming = Logic + Control

In fact, it was:

SLD-derivations = SLD-resolution + Searching strategies

CoLP = SLD-resolution + Searching strategies + Loop detection

We now have:

S-derivations = S-resolution + Tree Calculus

- S-resolution separates proof-construction from proof-search;
- Tree calculus makes it inductively sound and complete.
- Observational Productivity is a part of Logic (rather than Control)

We now resolve the coinductive properties

Global Productivity of S-Derivations

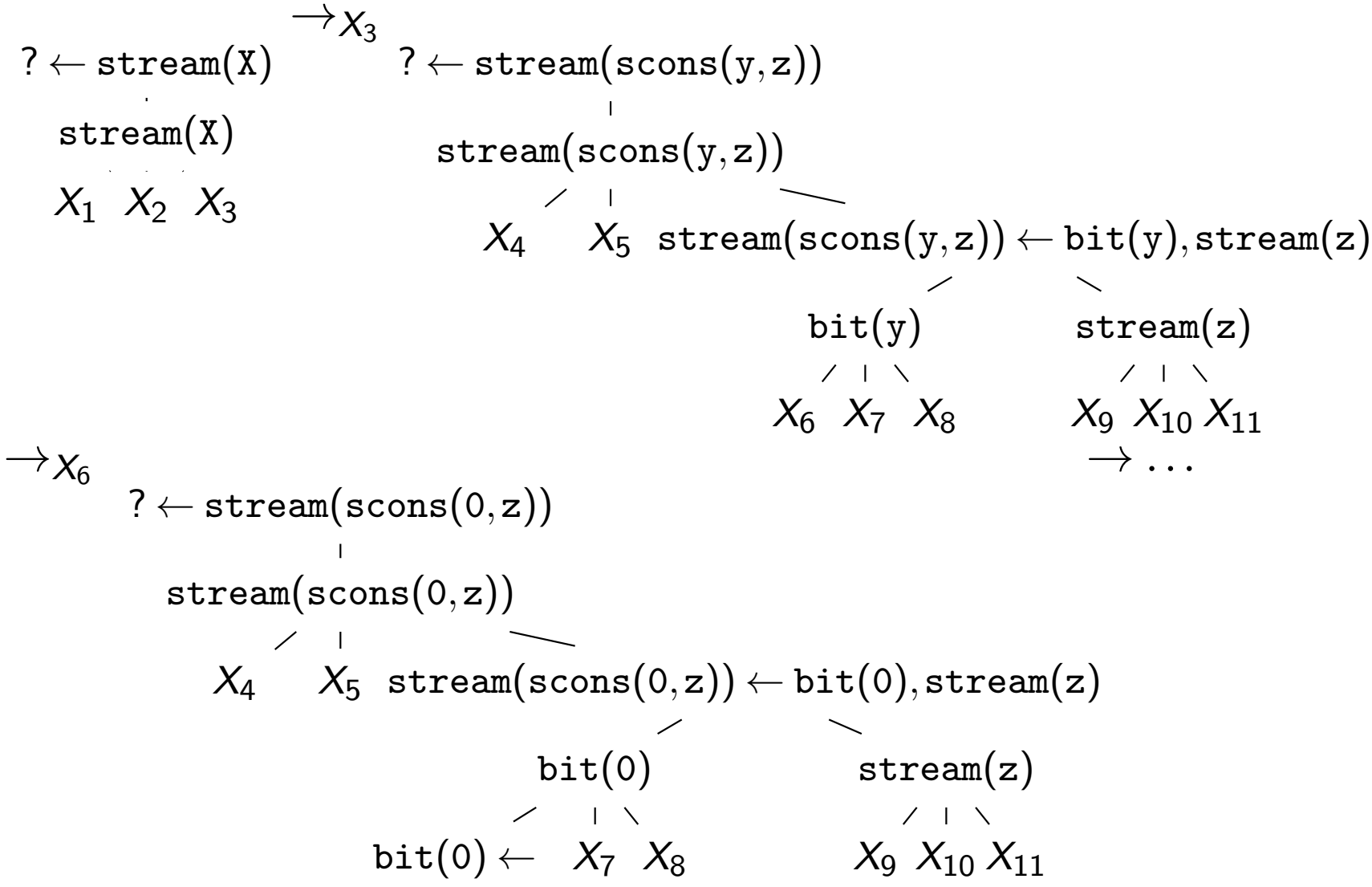
... assuming light inductive/coinductive typing Ty on predicates.

Definition

Let P be observationally productive, and let $t \in \mathbf{GTerm}^{\infty}(\Sigma)$. We say that $t' \in \mathbf{Term}(\Sigma)$ *finitely approximates* t if the following hold:

1. There is an infinite S-derivation $T_0 \to T_1 \to \ldots T_k \to \ldots$ with T_0 being a rewriting tree for t', and associated resolvents $\theta_1, \theta_2, \ldots \theta_k \ldots$ such that $d(t, \theta_k \ldots \theta_1(t')) \to 0$ as $k \to \infty$, and
2. (inductively-typed nodes are successfully closed infinitely often in the S-derivation).

Term t is *S-computable at infinity with respect to P and Ty* if there is a $t' \in \mathbf{Term}(\Sigma)$ such that t' finitely approximates t.



The term stream(scons(0, scons(0, ...))) is S-computable at infinity.

S-derivations and Global productivity of the 80s:

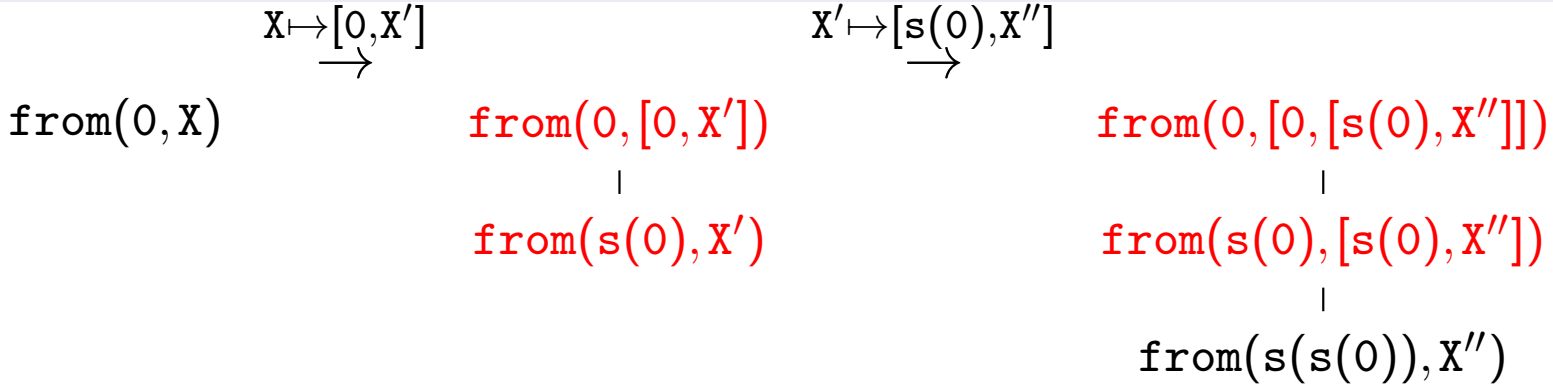
S-computations at infinity generalise computations at infinity

If a term t is SLD-computable at infinity, then it is S-computable at infinity.

How about finite proof-search procedure?

Coinductive Proof Principle

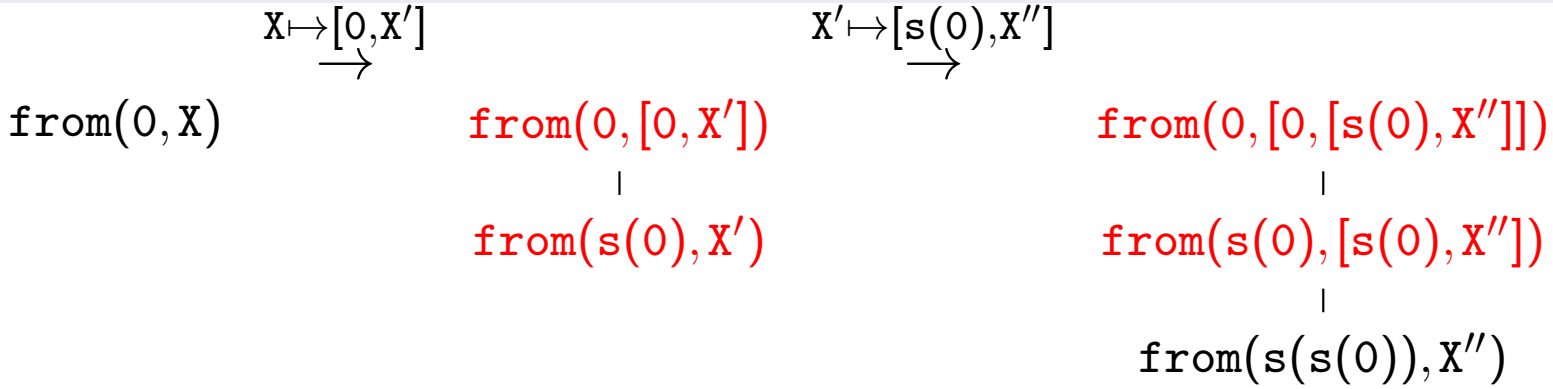
$$\mathtt{from}(\mathtt{x},\mathtt{scons}(\mathtt{x},\mathtt{y})) \leftarrow \mathtt{from}(\mathtt{s}(\mathtt{x}),\mathtt{y})$$



1. Form a coinductive hypothesis observing the second tree: $\mathtt{from}(\mathtt{s}(0),\mathtt{X}') \leftarrow \mathtt{from}(0,\mathtt{scons}(0,\mathtt{X}'))$
2. Apply it at the third tree (guarding the application by transition)
3. Close the coinductive proof by coinductive hypothesis application at the third tree.

Coinductive Proof Principle

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



1. Form a coinductive hypothesis observing the second tree: $\mathtt{from}(\mathtt{s}(0),\mathtt{X}') \leftarrow \mathtt{from}(0,\mathtt{scons}(0,\mathtt{X}'))$
2. Apply it at the third tree (guarding the application by transition)
3. Close the coinductive proof by coinductive hypothesis application at the third tree.

General Coinductive Proof principle:

$$\frac{T \rightarrow \ldots \rightarrow T' \qquad T' \vdash_{CH} \theta(C)}{T \vdash_{CH} C}$$

Coinductive proof principle for S-resolution is...

...Sound and complete relative to S-computations at infinity

Given P is observationally productive, and $t \in \mathbf{Term}(\Sigma)$,
there exists a $t^* \in \mathbf{GTerm}^{\infty}(\Sigma)$ that is S-computable at infinity with respect to P and t
iff
there exists a coinductively observed proof for t.

...Sound relative to the greatest complete Herbrand models

Let P be observationally productive, and $t \in \mathbf{Term}(\Sigma)$. If there exists a coinductively observed proof for t, then there exists an infinite term $t^* \in M_P^{\omega}$ that is S-computable at infinity with respect to P and t.

Summary

- We have just given a coinductive proof-search method;
- that generalises computations at infinity from the 80s;
- but at the same time gives a coinductive proof principle and thus is implementable.
- It incorporates observational and global productivity on par with ITP;
- and generalises CoLP [Gupta, 2007]

Outline

Formalization of a Type System

Horn Formulas as Types

Proof evidence as Terms

Term $t \ ::= \ x \mid f(t_1, ..., t_n)$
Atomic Formula $A, B, C, D \ ::= \ P(t_1, ..., t_n)$
(Horn) Formula $F \ ::= \ A_1, ..., A_n \Rightarrow A$
Proof Term $p, e \ ::= \ \kappa \mid a \mid \lambda a.e \mid e \ e'$

$$\frac{}{\kappa : \forall \underline{x}.F} \ axiom \qquad \frac{e : F}{e : \forall \underline{x}.F} \ gen$$

$$\frac{e : \forall \underline{x}.F}{e : [\underline{t}/\underline{x}]F} \ inst \qquad \frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2 \ \underline{b}) \ (e_1 \ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C} \ cut$$

Soundness of SLD and term-matching reductions

- If $P \vdash \{A\} \rightsquigarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$.
- If $P \vdash \{A\} \rightarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$.

Example

κ_1 `nat(0)` $\leftarrow$
κ_2 `nat(s(x))` $\leftarrow$ `nat(x)`
κ_3 `list(nil)` $\leftarrow$
κ_4 `list(cons(x,y))` $\leftarrow$ `nat(x)`, `list(y)`

$\{\mathrm{list}(\mathrm{cons}(x, y))\} \rightsquigarrow \{\mathrm{nat}(\mathrm{x}), \mathrm{list}(y)\} \rightsquigarrow \{\mathrm{list}(y)\} \rightsquigarrow \emptyset$
yields a proof $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3 : \mathrm{list}(\mathrm{cons}(0, nil))$
(β-reducible to $(\kappa_4 \kappa_3)\kappa_1 : \mathrm{list}(\mathrm{cons}(0, nil))$).

Soundness of SLD and term-matching reductions

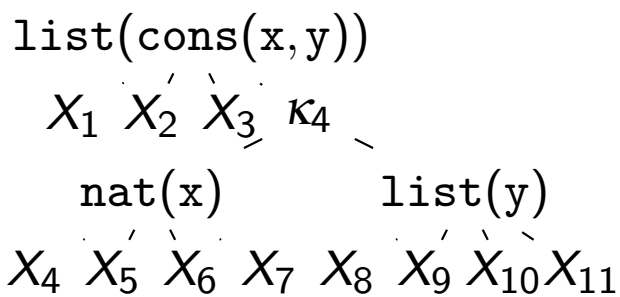
- If $P \vdash \{A\} \rightsquigarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$.
- If $P \vdash \{A\} \rightarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$.

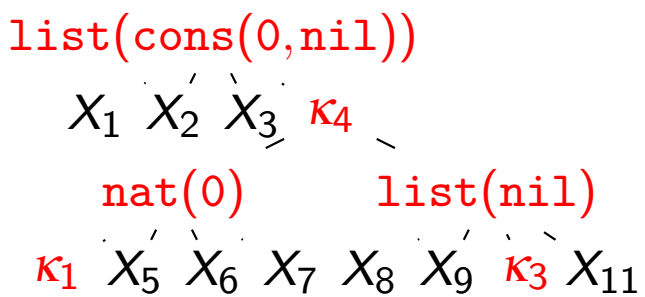
Example

κ_1 `nat(0)` $\leftarrow$
κ_2 `nat(s(x))` $\leftarrow$ `nat(x)`
κ_3 `list(nil)` $\leftarrow$
κ_4 `list(cons(x,y))` $\leftarrow$ `nat(x)`, `list(y)`

$\{\text{list}(\text{cons}(x, y))\} \rightsquigarrow \{\text{nat(x)}, \text{list}(y)\} \rightsquigarrow \{\text{list}(y)\} \rightsquigarrow \emptyset$
yields a proof $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3 : \text{list}(\text{cons}(0, nil))$
(β-reducible to $(\kappa_4 \kappa_3)\kappa_1 : \text{list}(\text{cons}(0, nil))$).

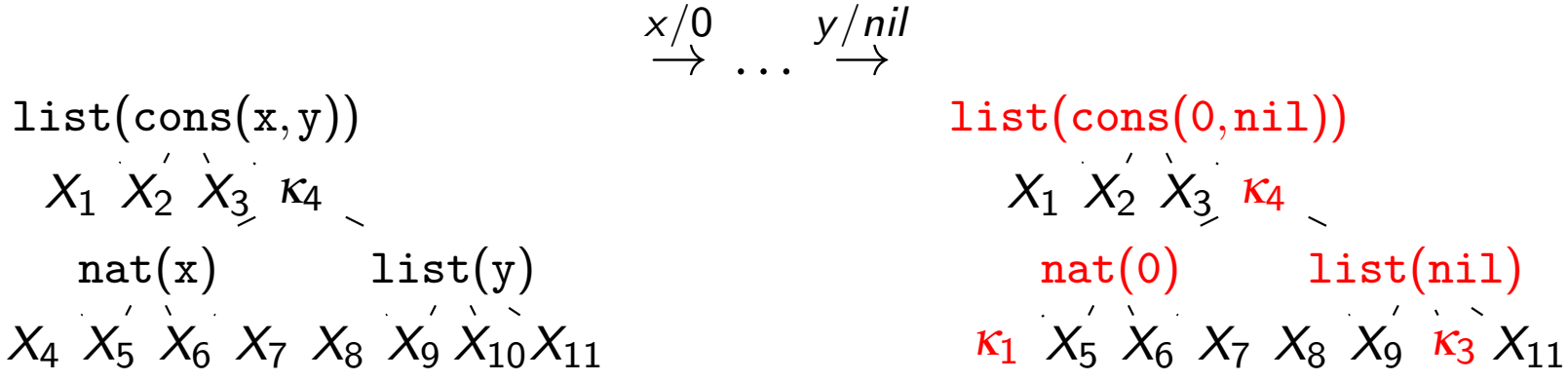
$\xrightarrow{x/0} \ \ldots \ \xrightarrow{y/nil}$





Structural resolution meets Curry-Howard

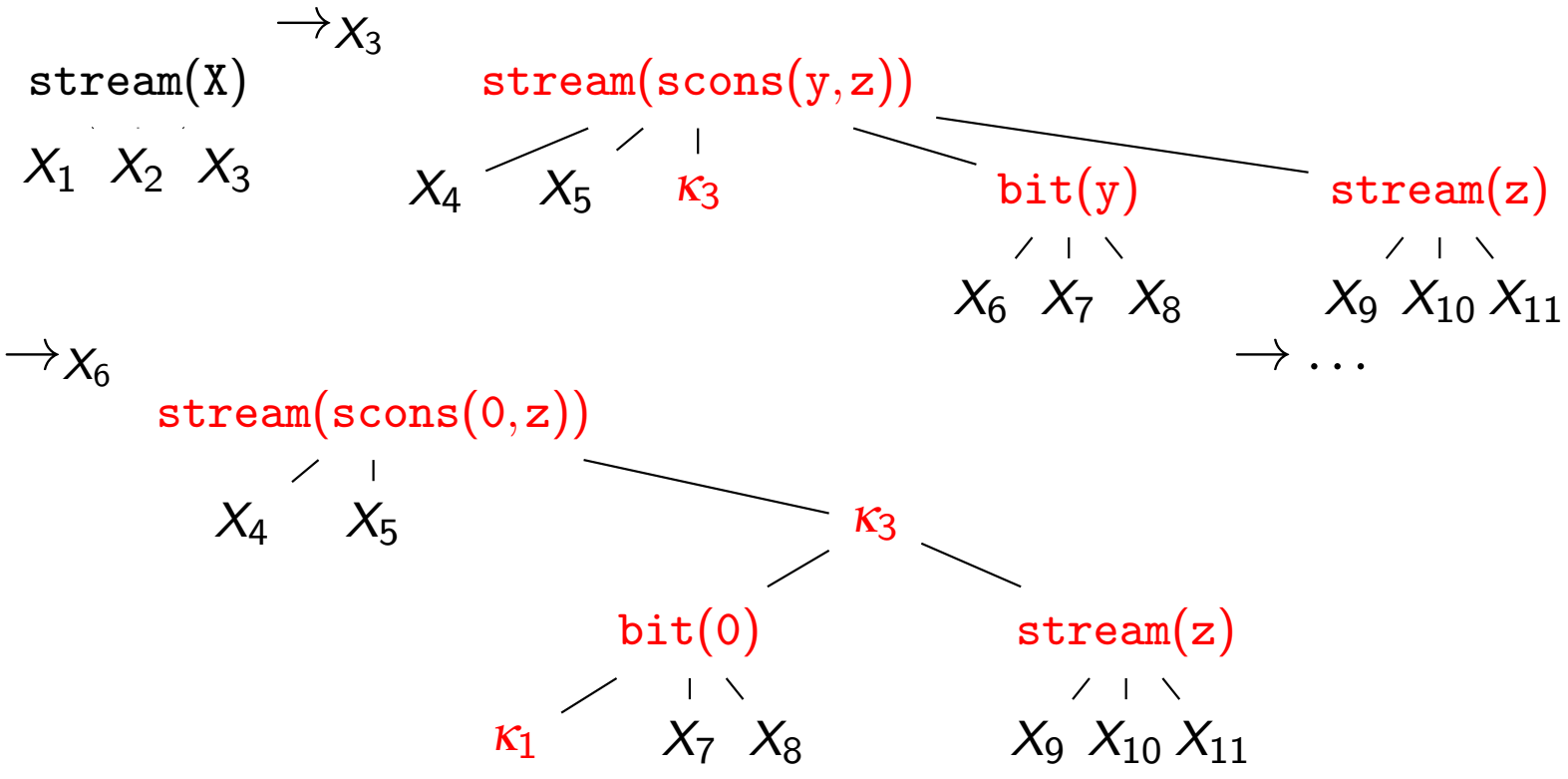
- Success rewriting trees always give a proof $e : A$, in which A is atomic root of the tree and e is a witness for clause applications.



$$(\kappa_4\,\kappa_3)\,\kappa_1 :: \mathrm{list}(\mathrm{cons}(0, \mathit{nil}))$$

Structural resolution meets Curry-Howard

- All open rewriting trees will correspond to a proof $e : B_1, \ldots, B_n \Rightarrow A$, where e is λ-term, A the root of that tree, and $B_1, \ldots, B_n$ – all open proof obligations.



$\lambda\alpha.(\kappa_3\alpha)\kappa_1 : \mathtt{stream(z)} \Rightarrow \mathtt{stream(scons(0,z))}$

Next step: fully formalise the coinductive proof principle.

Conclusions-1: Q and A

Q Why is it harder to implement coinduction in ATP than in ITP?

A ...The notions of proof and proof-search are badly separated.

A ... In absence of types and pattern-matching, we have fewer tools for structural analysis of programs and computations

Conclusions-1: Q and A

Q Why is it harder to implement coinduction in ATP than in ITP?

A ...The notions of proof and proof-search are badly separated.

A ... In absence of types and pattern-matching, we have fewer tools for structural analysis of programs and computations

Q Is it possible, in principle, to bridge the gap?

A Yes, by using structural resolution to separate proof and proof search components (term-matching and unification, rewriting trees and their transitions)

A Yes, by taking a Curry-Howard view on proof component of proof-search.

Current work

Applications of the above to Type Inference

To construct equality class instance for datatype list and int:

$$\begin{aligned} \kappa_1 : &\quad \mathrm{Eq}(x) \Rightarrow \mathrm{Eq}(\mathrm{list}(x)) \\ \kappa_2 : &\quad \Rightarrow \mathrm{Eq}(\mathrm{int}) \end{aligned}$$

When we call a query $\mathrm{Eq}(\mathrm{list}(\mathrm{int}))$, we can use LP-TM to construct a proof for $\mathrm{Eq}(\mathrm{list}(\mathrm{int}))$, which is $\kappa_2\ \kappa_1$, and then the evaluation of $\kappa_2\ \kappa_1$ will correspond to the process of evidence construction, thus yielding computational meaning of the proof.

Thank you!

S-resolution and CoALP webpage:
http://staff.computing.dundee.ac.uk/katya/CoALP/

S-resolution/CoALP authors, contributors, implementors:

- John Power
- Martin Schmidt
- Jonathan Heras
- Vladimir Komendantskiy
- Patty Johann
- Andrew Pond
- Peng Fu
- Frantisek Farka

We always welcome collaborators!
(Also currently have funding (and looking for) 1 PhD student to start working on it at Heriot-Watt.)