Structural Resolution

Katya Komendantskaya

School of Computing, University of Dundee, UK

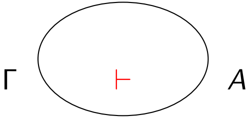
12 May 2015

Outline

Proof methods: structural, unstructured, and?

Abstracting from the details, all proof-search and proof-inference methods can be classified as

more or less Structural...



Proof inference methods: structural

Constructive Type theory

is more Structural...

$$\Gamma \quad \vdash p : \quad A$$

To prove $\Gamma \vdash A$, we need to show that type A has inhabitant p; namely, we have to conSTRUCT it.

Proof inference methods

Resolution-based first-order automated theorem provers (ATPs)

are less Structural...

$$\Gamma \qquad \vdash \qquad A$$

To prove $\Gamma \vdash A$, we need to assume A is false, and derive a contradiction from $\Gamma \cup \neg A$.

It only matters if resolution <u>finitely succeeds</u>; the proof structure is irrelevant.

Logic Programming...

SLD resolution $=$ Unification $+$ Search

SLD-resolution + unification in LP derivations.

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)`

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

SLD-resolution + unification in LP derivations.

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)`

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

SLD-resolution (+ unification) in LP derivations.

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)`

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

SLD-resolution (+ unification) in LP derivations.

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)`

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

The answer is "Yes", $\texttt{NatList} \vdash \mathit{list}(\mathit{cons}(x, y))$ if $x/0$, y/nil, but we can get more substitutions by backtracking.

SLD-refutation = finite successful SLD-derivation. SLD-refutations are sound and complete.

Problem

LP has never received a coherent, uniform theory of *Universal Termination*.

the program P is terminating, if, given any term A, a derivation for $P \vdash A$ returns an answer in a finite number of derivation steps.

- The survey [deSchreye, 1994] lists some 119 approaches to termination in LP, neither using universal termination.
- The consensus has not been reached to this day.

Problem

LP has never received a coherent, uniform theory of *Universal Termination*.

the program P is terminating, if, given any term A, a derivation for $P \vdash A$ returns an answer in a finite number of derivation steps.

- The survey [deSchreye, 1994] lists some 119 approaches to termination in LP, neither using universal termination.
- The consensus has not been reached to this day.

Reasons? – The lack of structural theory, namely:

Reason-1. *Non-determinism of proof-search in LP: –* termination depends on the searching strategy and order of clauses.

NatList2:

Example

1. $\mathtt{nat(0)} \leftarrow$
2. $\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$
3. $\mathtt{list(cons(x,y))} \leftarrow$

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

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

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

Reason-1. *Non-determinism of proof-search in LP: –* termination depends on the searching strategy and order of clauses.

NatList2:

Example

1.$\mathtt{nat(0)} \leftarrow$
2.$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$
3.$\mathtt{list(cons(x,y))} \leftarrow$
$\qquad \mathtt{nat(x),\ list(y)}$
4.$\mathtt{list(nil)} \leftarrow$

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

We have no means to analyse the structure of computations but run a search... which may be deceiving.

Reason 2. *Termination and (deciding) entailment are closely connected in LP.*

This creates an obstacle on the way to reasoning about coinductive programs, that do not assume finite success in derivations.

Reason 2. *Termination and (deciding) entailment are closely connected in LP.*

This creates an obstacle on the way to reasoning about coinductive programs, that do not assume finite success in derivations.
Program **Stream**:

Example

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

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

Reason 2. *Termination and (deciding) entailment are closely connected in LP.*

This creates an obstacle on the way to reasoning about coinductive programs, that do not assume finite success in derivations.

Program **Stream**:

Example

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

2.`bit(1)` $\leftarrow$

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

`bit(x), stream(y)`

No answer, as derivation never terminates. Neverthless, the program could be given a coindutive meaning...

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

Reason 2. *Termination and (deciding) entailment are closely connected in LP.*

This creates an obstacle on the way to reasoning about coinductive programs, that do not assume finite success in derivations.

Program **Stream**:

Example

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

`bit(x), stream(y)`

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

No answer, as derivation never terminates. Neverthless, the program could be given a coindutive meaning...

No distinction between type, function definition, and proof that could help to separate the issues...

Reason 3. *"Lack of directionality" in LP:*

Structurally recursive addition:

$$
\begin{array}{lll}
1.\mathtt{add}(\mathtt{0}, \mathtt{Y}, \mathtt{Y}) & \leftarrow & \\
2.\mathtt{add}(\mathtt{s}(\mathtt{X}), \mathtt{Y}, \mathtt{s}(\mathtt{Z})) & \leftarrow & \mathtt{add}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})
\end{array}
$$

If the third argument in add is "thought of" as the "output", and the other arguments – as "inputs", then, giving queries with variable-free "inputs" will guarantee termination by structural recursion on the first argument. But otherwise, there will be non-terminating derivations for queries to add.
As a consequence, in LP, it is common to talk about existential termination (only for some derivations, for queries of certain kinds, or satisfying certain conditions/measures), not programs in general.

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **NatList2**...

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **NatList2**...
- Or may be it is a recursive program with coinductive interpretation? (again, **NatList2**)

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **NatList2**...
- Or may be it is a recursive program with coinductive interpretation? (again, **NatList2**)
- Or may be it is just some bad loop without particular computational meaning:

$$badstream(scons(x, y)) \leftarrow badstream(scons(x, y))$$

Problems...

This unstructured approach to $\vdash$ gives us too little formal support to analyse termination

What does it mean if your program does not terminate?

- May be it is a corecursive program, like **Stream**...
- May be it is a recursive program, but badly ordered, like **NatList2**...
- Or may be it is a recursive program with coinductive interpretation? (again, **NatList2**)
- Or may be it is just some bad loop without particular computational meaning:

$$badstream(scons(x, y)) \leftarrow badstream(scons(x, y))$$

We are missing a theory, a language, to talk about such things...

Problems with LP termination and static program analysis

From its conception in 1960's, LP/ATP has not formulated a theory of universal termination!

All below programs do not terminate, and fail to produce any answer in PROLOG.

★**1.** P_1. Peano numbers.	★**2.** P_2. Infinite streams.	★**3.** P_3. Bad recursion.
$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$ $\mathtt{nat(0)} \leftarrow$	$\mathtt{stream(scons(x,y))} \leftarrow \mathtt{nat(x), stream(y)}$	$\mathtt{bad(x)} \leftarrow \mathtt{bad(x)}$

Problems with LP termination and static program analysis

From its conception in 1960's, LP/ATP has not formulated a theory of universal termination!

All below programs do not terminate, and fail to produce any answer in PROLOG.

★**1.** P_1. Peano numbers.	★**2.** P_2. Infinite streams.	★**3.** P_3. Bad recursion.
$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$ $\mathtt{nat(0)} \leftarrow$ inductive definition	$\mathtt{stream(scons(x,y))} \leftarrow \mathtt{nat(x), stream(y)}$ coinductive definition	$\mathtt{bad(x)} \leftarrow \mathtt{bad(x)}$ non-well-founded

Problems with LP termination and static program analysis

From its conception in 1960's, LP/ATP has not formulated a theory of universal termination!

All below programs do not terminate, and fail to produce any answer in PROLOG.

★**1.** P_1. Peano numbers.	★**2.** P_2. Infinite streams.	★**3.** P_3. Bad recursion.
$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$ $\mathtt{nat(0)} \leftarrow$ inductive definition	$\mathtt{stream(scons(x,y))} \leftarrow \mathtt{nat(x), stream(y)}$ coinductive definition	$\mathtt{bad(x)} \leftarrow \mathtt{bad(x)}$ non-well-founded

No termination – no program analysis

New methods. In search of a missing link

New methods. In search of a missing link

Is there a mysterious Missing link theory?

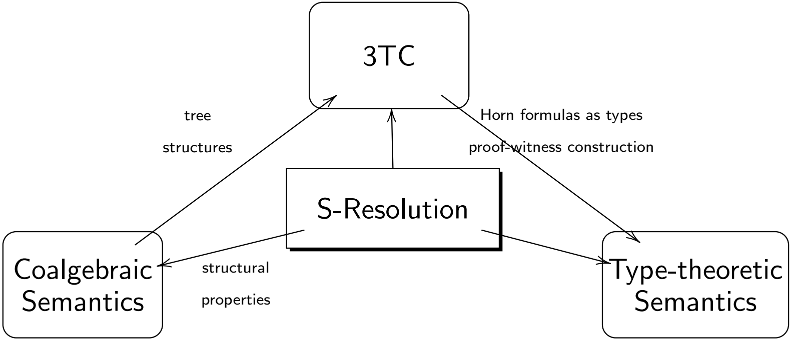
– Structural Resolution (also S-Resolution)

Is there place for a DISCOVERY here, which could expose A BETTER STRUCTURED resolution?

What IS

S-Resolution?

Structural Resolution:



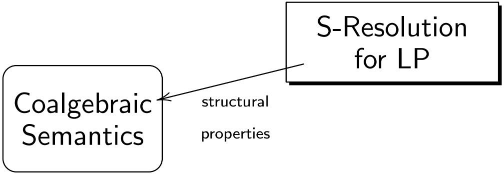
Outline

Structural Resolution:

[2010-2014, K, Power]

Discovery A:

(A) Structural Properties of Programs Uniquely determine Structural Properties of Computations



Outline

Motivation

Coalgebraic Semantics for Structural Resolution

The Three Tier Tree calculus for Structural Resolution

Type-Theoretic view of Structural Resolution

Conclusions and Future work

Our running example

Example

1. $\mathtt{nat}(\mathtt{s}(\mathtt{x})) \leftarrow \mathtt{nat}(\mathtt{x})$
2. $\mathtt{nat}(\mathtt{0}) \leftarrow$
3. $\mathtt{stream}(\mathtt{scons}(\mathtt{x}, \mathtt{y})) \leftarrow \mathtt{nat}(\mathtt{x}), \mathtt{stream}(\mathtt{y})$

Note: double-hopeless for SLD-resolution-based ATP!

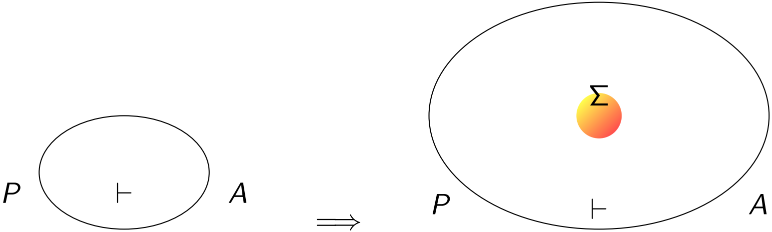
Defining structural resolution from first principles...

Main credo: we do not impose types or extra annotations, but look deep for “sub-atomic” structures innate in first-order proofs.

Defining structural resolution from first principles...

Main credo: we do not impose types or extra annotations, but look deep for "sub-atomic" structures innate in first-order proofs.

Given a logic program P there is a first-order signature Σ in P...



Example

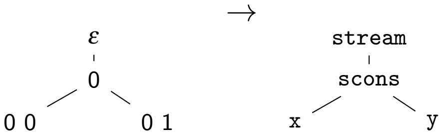
For our example, $\Sigma = \{0, \mathtt{s}, \mathtt{scons}, \mathtt{nat}, \mathtt{stream}\}$ + Variables.

Tier-1: Term-trees, given Σ:

Let $\mathbb{N}^*$ denote the set of all finite words over $\mathbb{N}$.

A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language*, satisfying prefix closedness conditions.

A term tree is a map $L \rightarrow \Sigma \cup Var$, satisfying term **arity** restrictions.

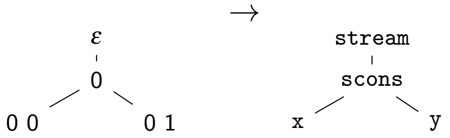


Tier-1: Term-trees, given Σ:

Let $\mathbb{N}^*$ denote the set of all finite words over $\mathbb{N}$.

A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language*, satisfying prefix closedness conditions.

A term tree is a map $L \to \Sigma \cup Var$, satisfying term **arity** restrictions.



Calculus: first-order unification

Given two terms t_1, t_2, and a substitution θ, θ is a unifier if $\theta(t_1) = \theta(t_2)$, and matcher if $t_1 = \theta(t_2)$.

Tier-1: Term-trees, given Σ:

Let $\mathbb{N}^*$ denote the set of all finite words over $\mathbb{N}$.

A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language*, satisfying prefix closedness conditions.

A term tree is a map $L \to \Sigma \cup Var$, satisfying term **arity** restrictions.

$$\begin{array}{ccc} & \varepsilon & \\ & | & \\ & 0 & \\ / & & \backslash \\ 0\,0 & & 0\,1 \end{array} \quad \longrightarrow \quad \begin{array}{ccc} & \texttt{stream} & \\ & | & \\ & \texttt{scons} & \\ / & & \backslash \\ \texttt{x} & & \texttt{y} \end{array}$$

Calculus: first-order unification

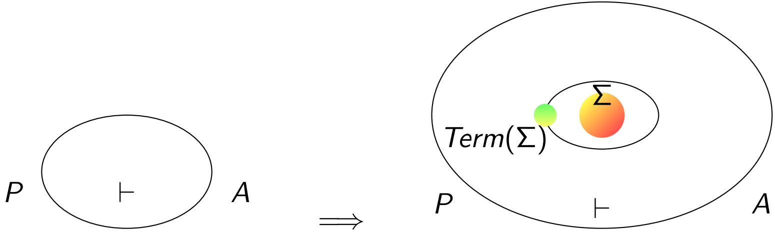
Given two terms t_1, t_2, and a substitution θ, θ is a unifier if $\theta(t_1) = \theta(t_2)$, and matcher if $t_1 = \theta(t_2)$.

Notation:

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

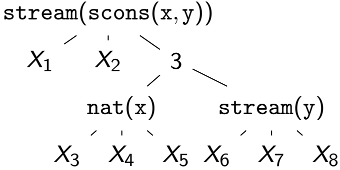
Constructing the structural resolution from first principles...

- Given a logic program P there is a first-order signature Σ...
- First tier of Terms builds on it...



Tier-2: rewriting trees

A rewriting tree is a map $L \to \textbf{Term}(\Sigma) \cup \textbf{Clause}(\Sigma) \cup Var_R$, subject to arity conditions.



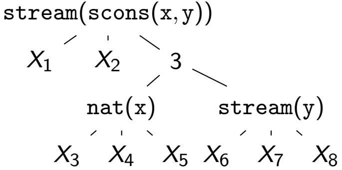
our running example

1. $\mathtt{nat(s(x))} \leftarrow$
2. $\mathtt{nat(0)} \leftarrow$
3. $\mathtt{stream(scons(x,y))} \leftarrow$
$\mathtt{nat(x), stream(y)}$

Interesting: Variables of Tier 2 and finiteness of rewriting trees for our “difficult” example!

Tier-2: rewriting trees

A rewriting tree is a map $L \to \textcolor{red}{\mathbf{Term}(\Sigma) \cup \mathbf{Clause}(\Sigma) \cup \mathit{Var}_R}$, subject to arity conditions.



our running example

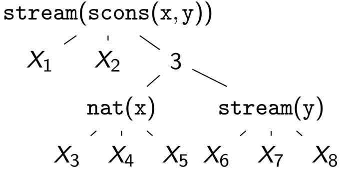
1. $\mathtt{nat}(\mathtt{s}(\mathtt{x})) \leftarrow$
2. $\mathtt{nat}(0) \leftarrow$
3. $\mathtt{stream}(\mathtt{scons}(\mathtt{x},\mathtt{y})) \leftarrow \mathtt{nat}(\mathtt{x}), \mathtt{stream}(\mathtt{y})$

Interesting: Variables of Tier 2 and finiteness of rewriting trees for our “difficult” example!

Calculus: tree transition by Tier-2 variable substitution

Tier-2: rewriting trees

A rewriting tree is a map $L \to \textbf{Term}(\Sigma) \cup \textbf{Clause}(\Sigma) \cup Var_R$, subject to arity conditions.



our running example

1. $\mathtt{nat}(\mathtt{s}(\mathtt{x})) \leftarrow$
2. $\mathtt{nat}(0) \leftarrow$
3. $\mathtt{stream}(\mathtt{scons}(\mathtt{x},\mathtt{y})) \leftarrow$
$\mathtt{nat}(\mathtt{x}), \mathtt{stream}(\mathtt{y})$

Interesting: Variables of Tier 2 and finiteness of rewriting trees for our “difficult” example!

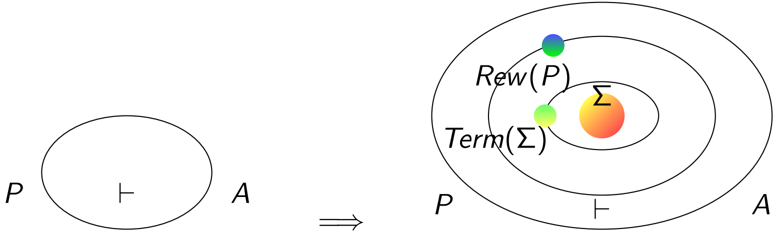
Calculus: tree transition by Tier-2 variable substitution

Notation:

$\textbf{Rew}(P)$	all *finite* rewriting trees over P and $\textbf{Term}(\Sigma)$
$\textbf{Rew}^{\infty}(P)$	all *infinite* rewriting trees over P and $\textbf{Term}(\Sigma)$
$\textbf{Rew}^{\omega}(P)$	all *finite and infinite* rewriting trees over P and $\textbf{Term}(\Sigma)$

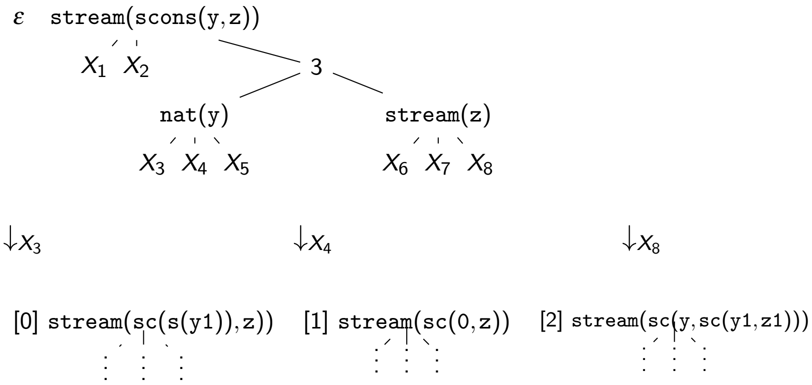
Constructing the structural resolution from first principles...

- Given a logic program P there is a first-order signature Σ...
- First tier of Terms builds on it...
- Term-trees give rise to a new tier of rewriting trees...



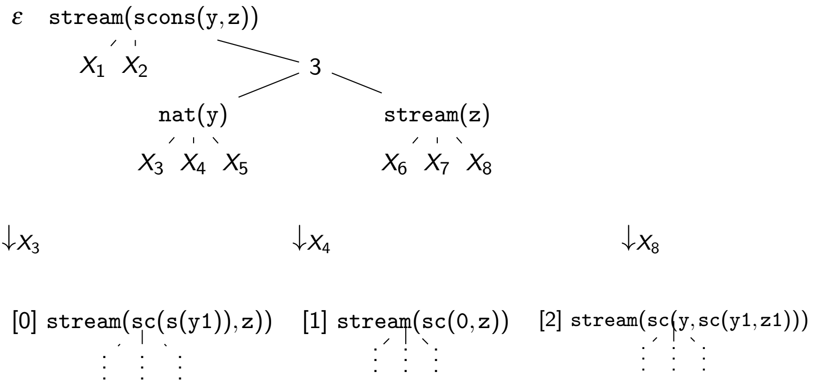
Tier-3: Derivation trees

A derivation tree is a map $L \rightarrow \mathbf{Rew}(P)$, subject to Arity condition (given by the number of clauses in P).



Tier-3: Derivation trees

A derivation tree is a map $L \to$ **Rew**(P), subject to Arity condition (given by the number of clauses in P).



Note: this derivation tree is infinite.

Tier-3 laws and notation

Notation:

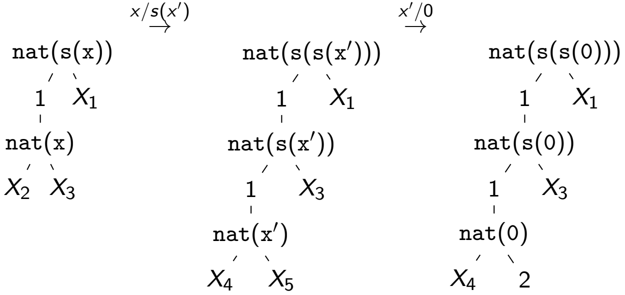
$\mathbf{Der}(P)$	all *finite* derivation trees over $\mathbf{Rew}(P)$
$\mathbf{Der}^{\infty}(P)$	all *infinite* derivation trees over $\mathbf{Rew}(P)$
$\mathbf{Der}^{\omega}(P)$	all *finite and infinite* derivation trees over $\mathbf{Rew}(P)$

Tier-3 laws and notation

Notation:

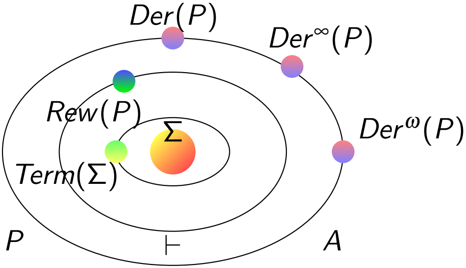
Der(P)	all *finite* derivation trees over **Rew**(P)
Der$^\infty(P)$	all *infinite* derivation trees over **Rew**(P)
Der$^\omega(P)$	all *finite and infinite* derivation trees over **Rew**(P)

An SLD-derivation for a program P and goal A corresponds to a branch in a derivation tree for P and A.



Constructing the structural resolution from first principles...

- Given a logic program P there is a first-order signature Σ...
- First tier of Terms builds on it...
- Term-trees give rise to a new tier of rewriting trees.
- And then, derivations by Structural resolution emerge!

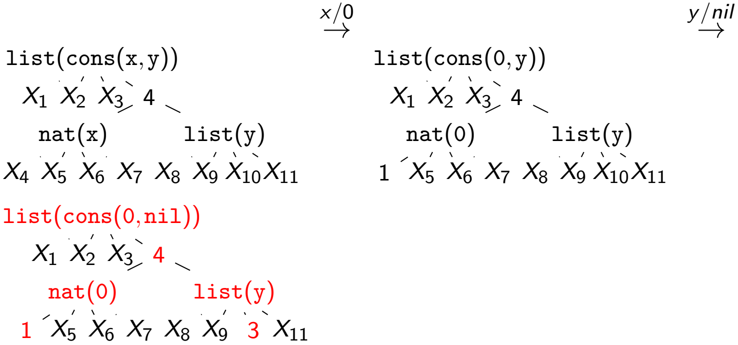


Gains:

- We found a missing theory of constructive resolution!
- Now to prove $P \vdash A$, we need to construct a rewriting tree $rew \in Rew(P)$ that proves A:

$$P \vdash rew : A$$

To prove $ListNat \vdash list(cons(x, y))$, we need to construct a rewriting tree that proves it:



New theory of universal productivity for resolution

A program P is productive, if it gives rise to rewriting trees only in **Rew**(P).

New theory of universal productivity for resolution

A program P is productive, if it gives rise to rewriting trees only in $\mathbf{Rew}(P)$.

In the class of Productive LPs, we can further distinguish:

- finite LP that give rise to derivations in $\mathbf{Der}(P)$,
- inductive LPs all derivations for which are in $\mathbf{Der}^{\omega}(P)$;
- coinductive LPs all derivations for which are in $\mathbf{Der}^{\infty}(P)$

New theory of universal productivity for resolution

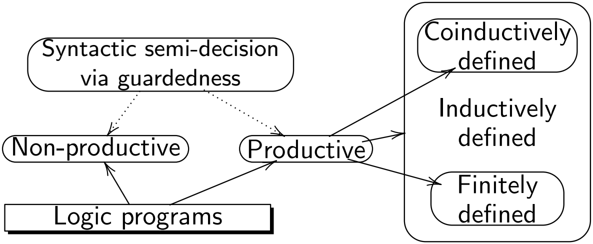
A program P is productive, if it gives rise to rewriting trees only in **Rew**(P).

In the class of Productive LPs, we can further distinguish:

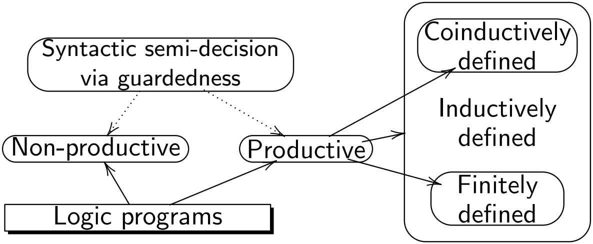
- finite LP that give rise to derivations in **Der**(P),
- inductive LPs all derivations for which are in $\mathbf{Der}^{\omega}(P)$;
- coinductive LPs all derivations for which are in $\mathbf{Der}^{\infty}(P)$

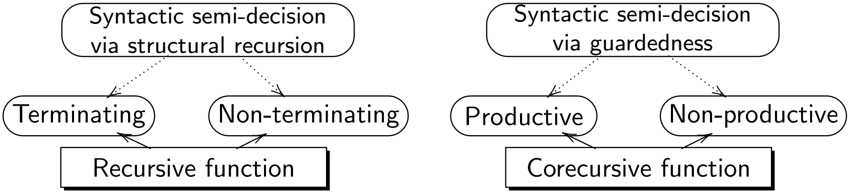
★**1.** P_1. Peano numbers.	★**2.** P_2. Infinite streams.	★**3.** P_3. Bad recursion.
`nat(s(x))` $\leftarrow$ `nat(x)` `nat(0)` $\leftarrow$ inductive definition	`stream(scons(x,y))` $\leftarrow$ `nat(x),stream(y)` coinductive definition	`bad(x)` $\leftarrow$ `bad(x)` non-well-founded
Productive inductive program	Productive coinductive program	Non-productive program
rewriting trees in $Rew(P)$, derivation trees $Der^{\omega}(P)$	rewriting trees in $Rew(P)$, derivation trees in $Der^{\infty}(P)$	rewriting trees do not belong to $Rew(P)$

Theory of universal Productivity in LP!



Theory of universal Productivity in LP!

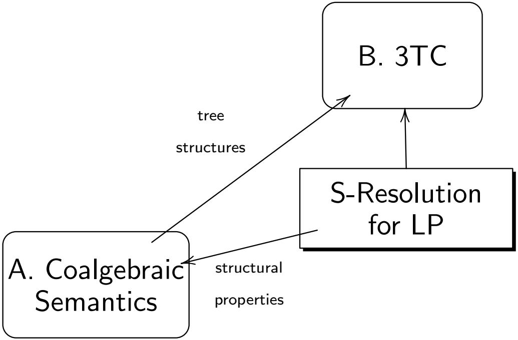




Structural Resolution:

Discovery B:

(B) Structures suggested by (A) can give a sound calculus, and solve problems known to be hard for LP: universal productivity and coinductive proof inference.



More questions still:

- What is the proof-theoretic meaning of S-Resolution?
- What is the constructive content of proofs by resolution?
- How do the rewriting trees relate to term rewriting systems?
- Does the informal analogy of 3TC

  $$P \vdash \mathit{rew} : A$$

  really have any relation to type theory?
- How exactly does the intuition that rewriting trees may serve as proof-witnesses in S-derivations relate to the type theory setting?

Outline

Horn formula view of LP

$$\kappa_1 : \Rightarrow \mathrm{Nat}(0)$$
$$\kappa_2 : \ \mathrm{Nat}(x) \Rightarrow \mathrm{Nat}(s(x))$$
$$\kappa_3 : \Rightarrow \mathrm{List}(\mathrm{nil})$$
$$\kappa_4 : \mathrm{Nat}(x), \mathrm{List}(y) \Rightarrow \mathrm{List}(\mathrm{cons}(x, y))$$

Formalism: LP-Unif, LP-TM and LP-Struct

- **Term-matching reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightarrow_{\kappa,\sigma} \{A_1, ..., \sigma B_1, ..., \sigma B_m, ..., A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \mapsto_\sigma A_i$.

Formalism: LP-Unif, LP-TM and LP-Struct

- **Term-matching reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \to_{\kappa,\sigma} \{A_1, ..., \sigma B_1, ..., \sigma B_m, ..., A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \mapsto_\sigma A_i$.
- **Unification reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \leadsto_{\kappa,\gamma \cdot \gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.

Formalism: LP-Unif, LP-TM and LP-Struct

- **Term-matching reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \to_{\kappa,\sigma} \{A_1, ..., \sigma B_1, ..., \sigma B_m, ..., A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \mapsto_\sigma A_i$.
- **Unification reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \leadsto_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.
- **Substitutional reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.

Formalism: LP-Unif, LP-TM and LP-Struct

- **Term-matching reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \to_{\kappa,\sigma} \{A_1, ..., \sigma B_1, ..., \sigma B_m, ..., A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \mapsto_\sigma A_i$.
- **Unification reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.
- **Substitutional reduction:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\}$, if there exists $\kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.
- **LP-TM:** $(\Phi, \to)$
  LP-Unif: $(\Phi, \rightsquigarrow)$
  LP-Struct: $(\Phi, \to^\mu \cdot \hookrightarrow^1)$

Execution behavior of LP-TM

- Consider query $\text{List}(\text{cons}(x,y))$:
  $\{\text{List}(\text{cons}(x,y))\} \rightarrow_{\kappa_4,[x/x_1,y/y_1]} \{\text{Nat}(x), \text{List}(y)\}$
  Note Partial nature

Execution behavior of LP-TM

- Consider query $\text{List}(\text{cons}(x, y))$:
  $\{\text{List}(\text{cons}(x, y))\} \rightarrow_{\kappa_4, [x/x_1, y/y_1]} \{\text{Nat}(x), \text{List}(y)\}$
  Note Partial nature
- Consider following Stream predicate:
  $\kappa : \text{Stream}(y) \Rightarrow \text{Stream}(\text{cons}(x, y))$
- In LP-TM:
  $\{\text{Stream}(\text{cons}(x, y))\} \rightarrow_{\kappa, [x/x_1, y/y_1]} \{\text{Stream}(y)\}$

Execution behavior of LP-TM

- Consider query $\mathrm{List}(\mathrm{cons}(x,y))$:
  $\{\mathrm{List}(\mathrm{cons}(x,y))\} \rightarrow_{\kappa_4,[x/x_1,y/y_1]} \{\mathrm{Nat}(x),\mathrm{List}(y)\}$
  Note Partial nature
- Consider following Stream predicate:
  $\kappa : \mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x,y))$
- In LP-TM:
  $\{\mathrm{Stream}(\mathrm{cons}(x,y))\} \rightarrow_{\kappa,[x/x_1,y/y_1]} \{\mathrm{Stream}(y)\}$

Note finiteness

LP-Struct: BList

For query $\mathrm{List}(\mathrm{cons}(x, y))$, in LP-Struct:

- $\{\mathrm{List}(\mathrm{cons}(x, y))\} \to \{\mathrm{Nat}(\mathrm{x}), \mathrm{List}(y)\}$

LP-Struct: BList

For query $\mathrm{List}(\mathrm{cons}(x, y))$, in LP-Struct:

- $\{\mathrm{List}(\mathrm{cons}(x, y))\} \to \{\mathrm{Nat}(\mathrm{x}), \mathrm{List}(y)\}$
- $\hookrightarrow_{[0/x]} \{\mathrm{Nat}(0), \mathrm{List}(y)\} \to \{\mathrm{List}(y)\}$

LP-Struct: BList

For query $\mathrm{List}(\mathrm{cons}(x,y))$, in LP-Struct:

- $\{\mathrm{List}(\mathrm{cons}(x,y))\} \to \{\mathrm{Nat}(\mathrm{x}), \mathrm{List}(y)\}$
- $\hookrightarrow_{[0/x]} \{\mathrm{Nat}(0), \mathrm{List}(y)\} \to \{\mathrm{List}(y)\}$
- $\hookrightarrow_{[0/x, \mathrm{nil}/y]} \{\mathrm{List}(\mathrm{nil})\} \to \emptyset$

LP-Struct: Stream

$\kappa : \mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x, y))$
For query $\mathrm{Stream}(\mathrm{cons}(x, y))$, in LP-Struct:

- $\{\mathrm{Stream}(\mathrm{cons}(x, y))\} \to \{\mathrm{Stream}(y)\}$

LP-Struct: Stream

$\kappa : \text{Stream}(y) \Rightarrow \text{Stream}(\text{cons}(x, y))$
For query $\text{Stream}(\text{cons}(x, y))$, in LP-Struct:

- $\{\text{Stream}(\text{cons}(x, y))\} \rightarrow \{\text{Stream}(y)\}$
- $\hookrightarrow_{[\text{cons}(x_1, y_1)/y]} \{\text{Stream}(\text{cons}(x_1, y_1))\} \rightarrow \{\text{Stream}(y_1)\}$

LP-Struct: Stream

$\kappa : \mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x, y))$
For query $\mathrm{Stream}(\mathrm{cons}(x, y))$, in LP-Struct:

- $\{\mathrm{Stream}(\mathrm{cons}(x, y))\} \to \{\mathrm{Stream}(y)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_1, y_1)/y]} \{\mathrm{Stream}(\mathrm{cons}(x_1, y_1))\} \to \{\mathrm{Stream}(y_1)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_2, y_2)/y_1, \mathrm{cons}(x_1, \mathrm{cons}(x_2, y_2))/y]} \{\mathrm{Stream}(\mathrm{cons}(x_2, y_2))\} \to \{\mathrm{Stream}(y_2)\}$

LP-Struct: Stream

$\kappa : \mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x, y))$
For query Stream(cons(x, y)), in LP-Struct:

- $\{\mathrm{Stream}(\mathrm{cons}(x, y))\} \to \{\mathrm{Stream}(y)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_1,y_1)/y]} \{\mathrm{Stream}(\mathrm{cons}(x_1, y_1))\} \to \{\mathrm{Stream}(y_1)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_2,y_2)/y_1,\mathrm{cons}(x_1,\mathrm{cons}(x_2,y_2))/y]} \{\mathrm{Stream}(\mathrm{cons}(x_2, y_2))\} \to \{\mathrm{Stream}(y_2)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_3,y_3)/y_2,\mathrm{cons}(x_2,\mathrm{cons}(x_3,y_3))/y_1,\mathrm{cons}(x_1,\mathrm{cons}(x_2,\mathrm{cons}(x_3,y_3)))/y]}$ $\{\mathrm{Stream}(\mathrm{cons}(x_3, y_3))\} \to \{\mathrm{Stream}(y_3)\}$

LP-Struct: Stream

$\kappa : \mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x, y))$
For query $\mathrm{Stream}(\mathrm{cons}(x, y))$, in LP-Struct:

- $\{\mathrm{Stream}(\mathrm{cons}(x, y))\} \to \{\mathrm{Stream}(y)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_1,y_1)/y]} \{\mathrm{Stream}(\mathrm{cons}(x_1, y_1))\} \to \{\mathrm{Stream}(y_1)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_2,y_2)/y_1,\mathrm{cons}(x_1,\mathrm{cons}(x_2,y_2))/y]} \{\mathrm{Stream}(\mathrm{cons}(x_2, y_2))\} \to \{\mathrm{Stream}(y_2)\}$
- $\hookrightarrow_{[\mathrm{cons}(x_3,y_3)/y_2,\mathrm{cons}(x_2,\mathrm{cons}(x_3,y_3))/y_1,\mathrm{cons}(x_1,\mathrm{cons}(x_2,\mathrm{cons}(x_3,y_3)))/y]}$ $\{\mathrm{Stream}(\mathrm{cons}(x_3, y_3))\} \to \{\mathrm{Stream}(y_3)\}$
- ...
- Partial answer: $\mathrm{cons}(x_1, \mathrm{cons}(x_2, \mathrm{cons}(x_3, y_3)))/y$

Formalization of a Type System

- 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 $p, e ::= \kappa \mid a \mid \lambda a.e \mid e\ e'$

Formalization of a Type System

- 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 $p, e ::= \kappa \mid a \mid \lambda a.e \mid e\ e'$
- Girard's observation on intuitionistic sequent calculus with atomic formulas

$$\frac{}{\underline{B} \vdash A}\ axiom \qquad \frac{\underline{B} \vdash C}{\sigma \underline{B} \vdash \sigma C}\ subst \qquad \frac{\underline{A} \vdash D \quad \underline{B}, D \vdash C}{\underline{A}, \underline{B} \vdash C}\ cut$$

Formalization of a Type System

- 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 $p, e ::= \kappa \mid a \mid \lambda a.e \mid e\ e'$
- Girard's observation on intuitionistic sequent calculus with atomic formulas

$$\frac{}{\underline{B} \vdash A}\ \mathit{axiom} \qquad \frac{\underline{B} \vdash C}{\sigma\underline{B} \vdash \sigma C}\ \mathit{subst} \qquad \frac{\underline{A} \vdash D \quad \underline{B}, D \vdash C}{\underline{A}, \underline{B} \vdash C}\ \mathit{cut}$$

- Is $\vdash Q$ provable?

Formalization of a Type System

- Term $t ::= x \mid f(t_1, \dots, t_n)$
  Atomic Formula $A, B, C, D ::= P(t_1, \dots, t_n)$
  (Horn) Formula $F ::= A_1, \dots, A_n \Rightarrow A$
  Proof Term $p, e ::= \kappa \mid a \mid \lambda a.e \mid e\ e'$
- Girard's observation on intuitionistic sequent calculus with atomic formulas

$$\frac{}{\underline{B} \vdash A}\ \mathit{axiom} \qquad \frac{\underline{B} \vdash C}{\sigma \underline{B} \vdash \sigma C}\ \mathit{subst} \qquad \frac{\underline{A} \vdash D \quad \underline{B}, D \vdash C}{\underline{A}, \underline{B} \vdash C}\ \mathit{cut}$$

- Is $\vdash Q$ provable?
- We internalized "$\vdash$" as "$\Rightarrow$" and add proof term annotations

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

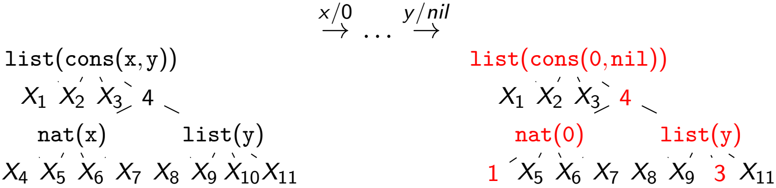
$$\frac{e : \forall \underline{x}.F}{e : [\underline{t}/\underline{x}]F}\ \mathit{inst} \qquad \frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow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

Soundness of LP-TM and LP-Unif

- *Soundness of LP-Unif*
  If $\Phi \vdash \{A\} \rightsquigarrow^*_\gamma \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$ given axioms Φ.
- *Soundness of LP-TM*
  If $\Phi \vdash \{A\} \rightarrow^* \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$ given axioms Φ.
- For example:
  $\{\mathrm{BList}(\mathrm{cons}(x, y))\} \rightsquigarrow \{\mathrm{Bit}(\mathrm{x}), \mathrm{BList}(y)\} \rightsquigarrow_{[0/x]} \{\mathrm{BList}(y)\}$
  $\rightsquigarrow_{[0/x, \mathrm{nil}/y]} \rightsquigarrow \emptyset$
- yields a proof $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3 : \mathrm{BList}(\mathrm{cons}(0, nil))$
  (β-reducible to $(\kappa_4 \kappa_3)\kappa_1$).

Soundness of LP-TM and LP-Unif

- *Soundness of LP-Unif*
  If $\Phi \vdash \{A\} \leadsto^*_\gamma \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$ given axioms Φ.
- *Soundness of LP-TM*
  If $\Phi \vdash \{A\} \to^* \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$ given axioms Φ.
- For example:
  $\{\mathrm{BList}(\mathrm{cons}(x, y))\} \leadsto \{\mathrm{Bit}(\mathrm{x}), \mathrm{BList}(y)\} \leadsto_{[0/x]} \{\mathrm{BList}(y)\}$
  $\leadsto_{[0/x,\mathrm{nil}/y]} \leadsto \emptyset$
- yields a proof $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3 : \mathrm{BList}(\mathrm{cons}(0, nil))$ (β-reducible to $(\kappa_4 \kappa_3)\kappa_1$).
- Compare with the 3TC proof-witness:



LP-Struct is equivalent to LP-Unif

... for logic programs subject to realisability transformation

$\kappa_1 : \Rightarrow \mathrm{Nat}(0, c_{\kappa_1})$
$\kappa_2 : \ \mathrm{Nat}(x, u) \Rightarrow \mathrm{Nat}(s(x), f_{\kappa_2}(u))$
$\kappa_3 : \Rightarrow \mathrm{BList}(\mathrm{nil}, c_{\kappa_3})$
$\kappa_4 : \mathrm{Bit}(x, u_1), \mathrm{BList}(y, u_2) \Rightarrow \mathrm{BList}(\mathrm{cons}(x, y, f_{\kappa_4}(u_1, u_2)))$

- $\{\mathrm{BList}(\mathrm{cons}(x, y, u))\} \hookrightarrow_{[f_{\kappa_4}(u_1,u_2)/u]}$
  $\{\mathrm{BList}(\mathrm{cons}(x, y, f_{\kappa_4}(u_1, u_2)))\} \rightarrow \{\mathrm{Bit}(\mathrm{x}, \mathrm{u}_1), \mathrm{BList}(y, u_2)\}$

LP-Struct is equivalent to LP-Unif

... for logic programs subject to realisability transformation

$\kappa_1 : \Rightarrow \mathrm{Nat}(0, c_{\kappa_1})$
$\kappa_2 : \ \mathrm{Nat}(x, u) \Rightarrow \mathrm{Nat}(s(x), f_{\kappa_2}(u))$
$\kappa_3 : \Rightarrow \mathrm{BList}(\mathrm{nil}, c_{\kappa_3})$
$\kappa_4 : \mathrm{Bit}(x, u_1), \mathrm{BList}(y, u_2) \Rightarrow \mathrm{BList}(\mathrm{cons}(x, y, f_{\kappa_4}(u_1, u_2)))$

- $\{\mathrm{BList}(\mathrm{cons}(x, y, u))\} \hookrightarrow_{[f_{\kappa_4}(u_1,u_2)/u]}$ $\{\mathrm{BList}(\mathrm{cons}(x, y, f_{\kappa_4}(u_1, u_2)))\} \to \{\mathrm{Bit}(\mathrm{x}, \mathrm{u}_1), \mathrm{BList}(y, u_2)\}$
- $\hookrightarrow_{[0/x, c_{\kappa_1}/u_1]} \{\mathrm{Bit}(0, c_{\kappa_1}), \mathrm{BList}(y, u_2)\} \to \{\mathrm{BList}(y, u_2)\}$
- $\hookrightarrow_{[0/x, \mathrm{nil}/y, c_{\kappa_3}/u_2]} \{\mathrm{BList}(\mathrm{nil}, c_{\kappa_3})\} \to \emptyset$

Note the substitution for $u/f_{\kappa_4}(c_{\kappa_1}, c_{\kappa_3})$ matches the earlier computed proof term $(\kappa_4 \kappa_3)\kappa_1$.

Results about Realizability Transformation

- *Guarantees productivity = Termination of term-matching reduction*
  Directly inherited from 3TC
- *Preserves Provability*
- *Records Proof*
  in the extra argument substitutions
- *Preserves Computational behaviour of LP-Unif*
- *Helps to prove Operational Equivalence of LP-Unif and LP-Struct*
- *Helps to prove soundness of LP-Struct*

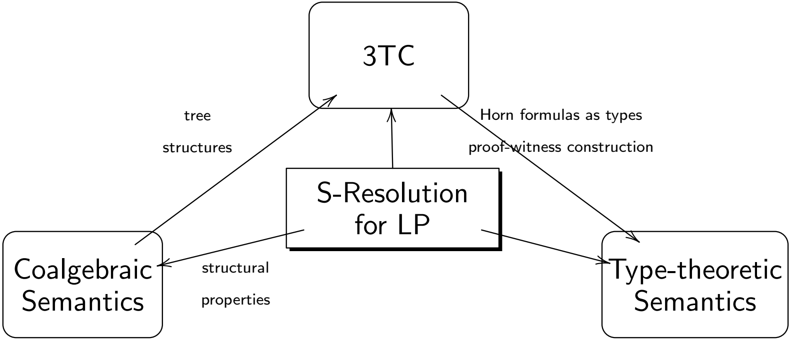
Gains from type-theoretic semantics for S-Resolution:

1. We established a direct relation to term-rewriting via LP-Struct;
2. We established a natural typed λ-calculus characterisation;
3. LP-Struct is sound wrt the type system;
4. Proof-witness is now formally defined as type inhabitant;
   directly inherited from 3TC
5. S-resolution is not equivalent to SLD-resolution, in general;
6. We exactly described the class of LPs that have structural properties (for which S-resolution and SLD-resolution are equivalent);
   directly inherited from 3TC
7. and gave an automated and static way to transform LPs to their constructive variants (via realisability transformation).

Structural Resolution:

Discovery C:

(C) The 3 Tier Tree calculus gives genuine insight into constructive nature of first-order automated proof: Horn-formulas as types and proof-witnesses as type inhabitants.



Outline

Structural Resolution ABC

S-resolution is Automated proof-search by resolution

in which:

(A) Structural Properties of Programs Uniquely determine Structural Properties of Computations

(B) These structures define a sound calculus, and solve problems known to be hard for LP: universal productivity and coinductive proof inference.

(C) The 3 Tier Tree calculus gives genuine insight into constructive nature of first-order automated proof

Current work

Applications of the above to Type Inference

... see the next talk by Fu Peng.

To construct equality class instance for datatype list and int:

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

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.

Current work

Applications of the above to Type Inference

... see the next talk by Fu Peng.

To construct equality class instance for datatype list and int:

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

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.

Dreams for the Future

Structural resolution as a new —
better structured and more constructive —
foundation for Automated Proof Search, starting from LP and reaching as far as Resolution-based SAT and SMT solvers.

Thank you!

CoALP webpage:
http://staff.computing.dundee.ac.uk/katya/CoALP/

CoALP authors and contributors:

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