Structural Resolution

Katya Komendantskaya

School of Computing, University of Dundee, UK

07 May 2015
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
### Why do we call Computing Computer Science?

Because it has areas/methods/foundations that have been discovered, rather than engineered...

### Example

Programming languages are engineered; Their semantics – e.g. \( \lambda \)-calculus have been discovered...

Programming language semantics discovers foundations of programming languages.
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*...

\[ \Gamma \vdash p : A \]

To prove \( \Gamma \vdash A \), we need to show that type \( A \) has inhabitant \( p \); namely, we have to *conSTRUCT* it.
Resolution-based first-order automated theorem provers (ATPs) are less Structural...

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 **finitely succeeds**; the proof structure is irrelevant.
Logic Programming...

SLD resolution = Unification + Search

Note: it is an engineered language, in the sense of the first slide...
SLD-resolution + unification in LP derivations.

Program **NatList**:  

<table>
<thead>
<tr>
<th>Example</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. nat(0) ←</td>
<td></td>
</tr>
<tr>
<td>2. nat(s(x)) ← nat(x)</td>
<td></td>
</tr>
<tr>
<td>3. list(nil) ←</td>
<td></td>
</tr>
<tr>
<td>4. list(cons(x,y)) ←</td>
<td></td>
</tr>
<tr>
<td>nat(x), list(y)</td>
<td>← list(cons(x,y))</td>
</tr>
</tbody>
</table>
SLD-resolution + unification in LP derivations.

Example

1. \texttt{nat(0) ←}
2. \texttt{nat(s(x)) ← nat(x)}
3. \texttt{list(nil) ←}
4. \texttt{list(cons(x,y)) ←}

\begin{align*}
nat(x), & \text{ list(y)}
\end{align*}

\begin{align*}
\leftarrow & \text{ list(cons(x,y))} \\
\leftarrow & \text{ nat(x), list(y)}
\end{align*}
SLD-resolution (+ unification) in LP derivations.

Example

1. nat(0) ←
2. nat(s(x)) ← nat(x)
3. list(nil) ←
4. list(cons(x,y)) ←
   nat(x), list(y)

← list(cons(x,y))
   |  
   ← nat(x), list(y)
   |  
   ← list(y)
SLD-resolution (+ unification) in LP derivations.

Example

1. \texttt{nat(0)} ←
2. \texttt{nat(s(x))} ← \texttt{nat(x)}
3. \texttt{list(nil)} ←
4. \texttt{list(cons(x,y))} ←
   \texttt{nat(x)}, \texttt{list(y)}

The answer is “Yes”, \texttt{NatList} ⊢ \texttt{list(cons(x,y))} if \texttt{x}/0, \texttt{y}/\texttt{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.
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.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. nat(0) ←</td>
</tr>
<tr>
<td>2. nat(s(x)) ← nat(x)</td>
</tr>
<tr>
<td>3. list(cons(x,y)) ←</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4. list(nil) ←</td>
</tr>
</tbody>
</table>

We have no means to analyse the structure of computations but run a search... which may be deceiving.
Reason-1. Non-determinism of proof-search in LP: – termination depends on the searching strategy and order of clauses.

NatList2:

Example

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

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

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

No answer, as derivation never terminates. Nonetheless, the program could be given a coinductive meaning...
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) ←
2. bit(1) ←
3. stream(scons(x,y)) ←
   
   bit(x), stream(y)

No answer, as derivation never terminates. Nevertheless, the program could be given a coinductive meaning...

```
← stream(scons(x,y))
   | 
← bit(x), stream(y)
   | 
← stream(y)
   | 
← bit(x_1), stream(y_1)
   | 
← stream(y_1)
   | 
...```

No distinction between type, function definition, and proof that could help to separate the issues...
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) ←
2. bit(1) ←
3. stream(scons(x,y)) ←
   bit(x), stream(y)
```

No answer, as derivation never terminates. Nevertheless, the program could be given a coinductive meaning...

No distinction between type, function definition, and proof that could help to separate the issues...
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 \texttt{Stream}...
Problems...

This unstructured approach to $\uparrow$ 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...
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:

\[
\text{badstream}(\text{scons}(x, y)) \leftarrow \text{badstream}(\text{scons}(x, y))
\]
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.

<table>
<thead>
<tr>
<th>★1.</th>
<th>P₁. Peano numbers.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nat(s(x)) ← nat(x)</td>
</tr>
<tr>
<td></td>
<td>nat(0) ←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>★2.</th>
<th>P₂. Infinite streams.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stream(scons(x,y)) ←</td>
</tr>
<tr>
<td></td>
<td>nat(x), stream(y)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bad(x) ← bad(x)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{nat}(s(x)) \leftarrow \text{nat}(x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{nat}(0) \leftarrow$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inductive definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{stream}(\text{scons}(x,y)) \leftarrow \text{nat}(x), \text{stream}(y)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coinductive definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{bad}(x) \leftarrow \text{bad}(x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-well-founded</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nat($s(x)$) ← nat($x$)</td>
<td>stream($scons(x,y)$) ←</td>
<td>bad($x$) ← bad($x$)</td>
</tr>
<tr>
<td>nat(0) ←</td>
<td>nat($x$), stream($y$)</td>
<td>non-well-founded</td>
</tr>
<tr>
<td>inductive definition</td>
<td>coinductive definition</td>
<td></td>
</tr>
</tbody>
</table>

No termination – no program analysis
New methods. In search of a missing link
New methods. In search of a missing link

<table>
<thead>
<tr>
<th>Is there a mysterious Missing link theory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Structural Resolution (also S-Resolution)</td>
</tr>
</tbody>
</table>

Is there place for a DISCOVERY here, which could expose A BETTER STRUCTURED resolution?
What IS S-Resolution?
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
Fibrational Coalgebraic Semantics of LP in 3 ideas

Idea 1: Logic programs as coalgebras

Definition

For a functor $F$, a coalgebra is a pair $(U, c)$ consisting of a set $U$ and a function $c : U \to F(U)$.

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 $p : At \to P_f(P_f(At))$ sends an atom $A$ to the set of bodies of those clauses in $P$ with head $A$.

Example

$T \leftarrow Q, R$
$T \leftarrow S$
$p(T) = \{\{Q, R\}, \{S\}\}$
Idea 2: Derivations modelled by coalgebra for the comonad on $P_f P_f$

In general, if $U : H-coalg \longrightarrow C$ has a right adjoint $G$, the composite functor $UG : C \longrightarrow C$ possesses the canonical structure of a comonad $C(H)$, called the cofree comonad on $H$. One can form a coalgebra for a comonad $C(H)$.

- 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.
This models and-or parallel trees known in LP [AMAST 2010]
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

Definition

A Lawvere theory consists of a small category $L$ with strictly associative finite products, and a strict finite-product preserving functor $I : \mathbb{N}^{op} \to L$.

Take Lawvere Theory $\mathcal{L}_\Sigma$ to model the terms over $\Sigma$

* $\text{ob}(\mathcal{L}_\Sigma)$ is $\mathbb{N}$.
** For each $n \in \text{Nat}$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
*** $\text{ob}(\mathcal{L}_\Sigma)(n, m)$ is the set of $m$-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in $\Sigma$ and variables $x_1, \ldots, x_n$.
**** composition in $\mathcal{L}_\Sigma$ is first-order substitution.
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

Definition

A Lawvere theory consists of a small category $L$ with strictly associative finite products, and a strict finite-product preserving functor $I : \mathbb{N}^{op} \rightarrow L$.

Take Lawvere Theory $\mathcal{L}_\Sigma$ to model the terms over $\Sigma$

* $\text{ob}(\mathcal{L}_\Sigma)$ is $\mathbb{N}$.
** For each $n \in \text{Nat}$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
*** $\text{ob}(\mathcal{L}_\Sigma)(n, m)$ is the set of $m$-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in $\Sigma$ and variables $x_1, \ldots, x_n$.
**** composition in $\mathcal{L}_\Sigma$ is first-order substitution.

Take the functor $At : \mathcal{L}_\Sigma^{op} \rightarrow \text{Set}$ that sends a natural number $n$ to the set of all atomic formulae generated by $\Sigma$ and $n$ variables.
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 3: Add Lawvere Theories to model first-order signature

**Definition**

A *Lawvere theory* consists of a small category $L$ with strictly associative finite products, and a strict finite-product preserving functor $I : \mathbb{N}^{op} \to L$.

Take the *Lawvere Theory* $L_\Sigma$ to model the terms over $\Sigma$

* $\text{ob}(L_\Sigma)$ is $\mathbb{N}$.

** For each $n \in \text{Nat}$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.

*** $\text{ob}(L_\Sigma)(n, m)$ is the set of $m$-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in $\Sigma$ and variables $x_1, \ldots, x_n$.

**** composition in $L_\Sigma$ is first-order substitution.

Take the functor $At : L_\Sigma^{op} \to \text{Set}$ that sends a natural number $n$ to the set of all atomic formulae generated by $\Sigma$ and $n$ variables.

Model a program $P$ by the $[L_\Sigma^{op}, P_f P_f]$-coalgebra.
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1 to model all terms with 1 free variable. Then $\mathcal{V}$-trees:
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1 to model all terms with 1 free variable. Then \&V-trees:

\[
\text{stream}(x)
\]
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1 to model all terms with 1 free variable. Then \&\textit{V}-trees:

\[
\text{stream}(x) \quad \text{stream}(\text{scons}(x, x))
\]

\[
\text{bit}(x) \quad \text{stream}(x)
\]
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1 to model all terms with 1 free variable. Then $\&V$-trees:

\[
\text{stream}(x) \quad \text{stream}(\text{scons}(x,x)) \\
\quad \downarrow \\
\text{bit}(x) \quad \text{stream}(x)
\]

---

Note the finite size

\[
\text{stream}(\text{scons}(0,\text{scons}(x,x))) \\
\quad \downarrow \\
\text{bit}(0) \quad \text{stream}(\text{scons}(x,x)) \\
\quad \downarrow \\
\text{bit}(x) \quad \text{stream}(x)
\]
Examples

Program **ListNat**: “fibers” given by term arities. Take the fiber of 2 to model all terms with 2 free variables. Then &V-trees:
Examples

Program **ListNat**: “fibers” given by term arities. Take the fiber of 2 to model all terms with 2 free variables. Then &V-trees:

\[
\text{list}(X) \quad \text{list}(\text{nil})
\]

\[
\text{list}(\text{cons}(0, \text{nil})) \quad \text{list}(\text{cons}(X, Y))
\]

\[
\text{nat}(0) \quad \text{list}(\text{nil}) \quad \text{nat}(X) \quad \text{list}(Y)
\]

Note the partial nature...
Discovery A:

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

Structures suggested by the CoAlgebraic semantics do not really fit into LP tradition

- each $\& \lor$-tree gives only partial computation compared to SLD-resolution;
- seems to suggest laziness?
- introduces the (alien to LP) restriction on substitutions, due to fibers;
- the restriction works almost like term-matching…
- seems to suggest connection to term-rewriting systems?
- accounts for many choices in rewriting…
- seems to suggest and-or parallelism?
A Problem:

Structures suggested by the CoAlgebraic semantics do not really fit into LP tradition

- each \&\lor-tree gives only partial computation compared to SLD-resolution;
- seems to suggest laziness?
- introduces the (alien to LP) restriction on substitutions, due to fibers;
- the restriction works almost like term-matching...
- seems to suggest connection to term-rewriting systems?
- accounts for many choices in rewriting...
- seems to suggest and-or parallelism?

In short,

it introduced more questions than answers...
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

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $\text{nat}(s(x)) \leftarrow \text{nat}(x)$</td>
</tr>
<tr>
<td>2. $\text{nat}(0) \leftarrow$</td>
</tr>
<tr>
<td>3. $\text{stream}(\text{scons}(x, y)) \leftarrow \text{nat}(x), \text{stream}(y)$</td>
</tr>
</tbody>
</table>

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 $\Sigma$ in $P$...

Example

For our example, $\Sigma = \{0, s, scons, \text{nat}, \text{stream}\} + \text{Variables}$. 
Tier-1: Term-trees, given $\Sigma$:

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 $L \rightarrow \Sigma \cup \text{Var}$, satisfying term arity restrictions.

\[
\begin{array}{c}
00 \quad 01 \\
\downarrow \quad \downarrow \\
0 \quad 1 \\
\downarrow \\
\varepsilon \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\text{stream} \\
\downarrow \\
\text{scons} \\
\quad \downarrow \\
\quad x \quad y
\end{array}
\]
Tier-1: Term-trees, given $\Sigma$:

Let $N^*$ denote the set of all finite words over $N$. A set $L \subseteq N^*$ is a (finitely branching) tree language, satisfying prefix closedness conditions. A term tree is a map $L \rightarrow \Sigma \cup \text{Var}$, satisfying term arity restrictions.

Given two terms $t_1$, $t_2$, and a substitution $\theta$, $\theta$ is a unifier if $\theta(t_1) = \theta(t_2)$, and matcher if $t_1 = \theta(t_2)$. 
Tier-1: Term-trees, given $\Sigma$:

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 $L \rightarrow \Sigma \cup \text{Var}$, satisfying term arity restrictions.

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

**Notation:**

<table>
<thead>
<tr>
<th>$\text{Term}(\Sigma)$</th>
<th>Set of <em>finite</em> term trees over $\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Term}^\infty(\Sigma)$</td>
<td>Set of <em>infinite</em> term trees over $\Sigma$</td>
</tr>
<tr>
<td>$\text{Term}^{\omega}(\Sigma)$</td>
<td>Set of <em>finite and infinite</em> term trees over $\Sigma$</td>
</tr>
</tbody>
</table>
Constructing the structural resolution from first principles...

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

\[
\vdash P \quad \text{and} \quad A \quad \Rightarrow \quad \Sigma \vdash P \quad \text{Term}(\Sigma)
\]
Tier-2: rewriting trees

A rewriting tree is a map \( L \to \text{Term}(\Sigma) \cup \text{Clause}(\Sigma) \cup \text{Var}_R \), subject to conditions \((\text{Term-matching})\).

```
stream(scons(x,y))
  X_1  X_2  \_  \_  3
    \_\_\_\_\_\_\_\_\_\_
    nat(x)  stream(y)
  X_3  X_4  X_5  X_6  X_7  X_8
```

```
 our running example

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

Interesting: all rewriting trees are finite for our “difficult” example!
Tier-2: rewriting trees

A rewriting tree is a map \( L \rightarrow \text{Term}(\Sigma) \cup \text{Clause}(\Sigma) \cup \text{Var}_R \), subject to conditions (Term-matching).

stream(scons(x,y))

\[
\begin{array}{c}
 X_1 \quad X_2 \quad 3 \\
 \quad \quad \quad \quad \text{nat}(x) \quad \text{stream}(y) \\
 \quad \quad \quad X_3 \quad X_4 \quad X_5 \quad X_6 \quad X_7 \quad X_8 \\
\end{array}
\]

Interesting: all rewriting trees are finite for our “difficult” example!

Notation:

| \( \text{Rew}(P) \) | all finite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |
| \( \text{Rew}^\infty(P) \) | all infinite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |
| \( \text{Rew}^{\omega}(P) \) | all finite and infinite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |

our running example

1. \( \text{nat}(s(x)) \leftarrow \)
2. \( \text{nat}(0) \leftarrow \)
3. \( \text{stream}(scons(x,y)) \leftarrow \)
   \( \text{nat}(x), \text{stream}(y) \)
Tier-2: rewriting trees

A rewriting tree is a map \( L \rightarrow \text{Term}(\Sigma) \cup \text{Clause}(\Sigma) \cup \text{Var}_R \), subject to conditions (Term-matching).

\[
\text{stream}(\text{scons}(x, y))
\]

\[
X_1 \quad X_2 \quad 3
\]

\[
\text{nat}(x) \quad \text{stream}(y)
\]

\[
X_3 \quad X_4 \quad X_5 \quad X_6 \quad X_7 \quad X_8
\]

Interesting: all rewriting trees are finite for our “difficult” example!

Notation:

| \( \text{Rew}(P) \) | all finite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |
| \( \text{Rew}^\infty(P) \) | all infinite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |
| \( \text{Rew}^\omega(P) \) | all finite and infinite rewriting trees over \( P \) and \( \text{Term}(\Sigma) \) |
Constructing the structural resolution from first principles...

- Given a logic program $P$ there is a first-order signature $\Sigma$...
- 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 $L \rightarrow \text{Rew}(P)$.

\[
\begin{align*}
\varepsilon & \quad \text{stream}(\text{scons}(y,z)) \\
X_1 & \quad X_2 & \quad 3 & \quad \text{nat}(y) & \quad \text{stream}(z) \\
X_3 & \quad X_4 & \quad X_5 & \quad X_6 & \quad X_7 & \quad X_8 \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
& [0 \ \text{stream}(\text{sc}(s(y1)),z)) & [1 \ \text{stream}(\text{sc}(0,z)) & [2 \ \text{stream}(\text{sc}(y,\text{sc}(y1,z1)))) & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{align*}
\]

Note: this derivation tree is infinite.
A derivation tree is a map $L \rightarrow \text{Rew}(P)$.

Note: this derivation tree is infinite.
## Tier-3 laws and notation

**Notation:**

<table>
<thead>
<tr>
<th>$\text{Der}(P)$</th>
<th>all <em>finite</em> derivation trees over $\text{Rew}(P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Der}^\infty(P)$</td>
<td>all <em>infinite</em> derivation trees over $\text{Rew}(P)$</td>
</tr>
<tr>
<td>$\text{Der}^\omega(P)$</td>
<td>all <em>finite and infinite</em> derivation trees over $\text{Rew}(P)$</td>
</tr>
</tbody>
</table>
Tier-3 laws and notation

Notation:

<table>
<thead>
<tr>
<th>Der((P))</th>
<th>all finite derivation trees over Rew((P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Der(^\infty)((P))</td>
<td>all infinite derivation trees over Rew((P))</td>
</tr>
<tr>
<td>Der(^\omega)((P))</td>
<td>all finite and infinite derivation trees over Rew((P))</td>
</tr>
</tbody>
</table>

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

\[
x / s(x') \rightarrow\]

\[
\text{nat}(s(x)) \quad \text{nat}(s(s(x'))) \quad \text{nat}(s(s(0)))
\]

\[
1 \quad \quad 1 \quad \quad 1
\]

\[
X_1 \
X_2 \quad X_3
\]

\[
x' / 0 \rightarrow\]

\[
\text{nat}(x') \quad \text{nat}(s(0))
\]

\[
1 \quad \quad 1
\]

\[
X_3 \
X_4 \quad X_5
\]

\[
x' \quad 2
\]
Given a logic program $P$ there is a first-order signature $\Sigma$...
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:

\[
\begin{array}{c}
\text{list(cons(x,y))} \\
X_1 \ X_2 \ X_3 \ X_4 \\
\text{nat(x)} \ \text{list(y)} \\
X_5 \ X_6 \ X_7 \ X_8 \ X_9 \ X_{10} \ X_{11}
\end{array}
\quad
\begin{array}{c}
\text{list(cons(0,y))} \\
X_1 \ X_2 \ X_3 \ X_4 \\
\text{nat(0)} \ \text{list(y)} \\
1 \ X_5 \ X_6 \ X_7 \ X_8 \ X_9 \ X_{10} \ X_{11}
\end{array}
\]

\[
\begin{array}{c}
\text{list(cons(0,nil))} \\
X_1 \ X_2 \ X_3 \ X_4 \\
\text{nat(0)} \ \text{list(y)} \\
1 \ X_5 \ X_6 \ X_7 \ X_8 \ X_9 \ 3 \ X_{11}
\end{array}
\]
The structural approach allowed to:

- Formulate the theory of Universal Productivity
- Show Finite derivations sound and complete wrt Herbrand models;
- Show Infinite derivations sound wrt Complete Herbrand models;
- Formulate finite coinductive proofs matching infinite derivations.
New theory of universal productivity for resolution

A program $P$ is **productive**, if it gives rise to rewriting trees only in $\text{Rew}(P)$. 
New theory of universal productivity for resolution

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

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

- finite LP that give rise to derivations in $\text{Der}(P)$,
- inductive LPs all derivations for which are in $\text{Der}^\omega(P)$;
- coinductive LPs all derivations for which are in $\text{Der}^\infty(P)$
A program $P$ is **productive**, if it gives rise to rewriting trees only in $\text{Rew}(P)$.

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{nat}(s(x)) \leftarrow \text{nat}(x)$</td>
<td>$\text{stream}(\text{scons}(x, y)) \leftarrow \text{nat}(x), \text{stream}(y)$</td>
<td>$\text{bad}(x) \leftarrow \text{bad}(x)$</td>
</tr>
<tr>
<td>$\text{nat}(0) \leftarrow \text{inductive definition}$</td>
<td>$\text{coinductive definition}$</td>
<td>non-well-founded</td>
</tr>
<tr>
<td><strong>Productive inductive program</strong></td>
<td><strong>Productive coinductive program</strong></td>
<td><strong>Non-productive program</strong></td>
</tr>
<tr>
<td>rewriting trees in $\text{Rew}(P)$, derivation trees $\text{Der}^{\omega}(P)$</td>
<td>rewriting trees in $\text{Der}^{\infty}(P)$</td>
<td>rewriting trees do not belong to $\text{Rew}(P)$</td>
</tr>
</tbody>
</table>
Theory of universal Productivity in LP!

- Non-productive
  - Logic programs
  - Syntactic semi-decision via guardedness
- Productive
  - Coinductively defined
  - Inductively defined
  - Finitely defined
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

\[
P \vdash \text{rew} : A
\]

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

Motivation

Coalgebraic Semantics for Structural Resolution

The Three Tier Tree calculus for Structural Resolution

Type-Theoretic view of Structural Resolution

Conclusions and Future work
Horn formula view of LP

\[ \kappa_1 : \Rightarrow \text{Nat}(0) \]
\[ \kappa_2 : \text{Nat}(x) \Rightarrow \text{Nat}(s(x)) \]
\[ \kappa_3 : \Rightarrow \text{List}(\text{nil}) \]
\[ \kappa_4 : \text{Nat}(x), \text{List}(y) \Rightarrow \text{List}(\text{cons}(x, y)) \]
Formalism: LP-Unif, LP-TM and LP-Struct

- **Term-matching reduction:**
  \[ \Phi \vdash \{ A_1, \ldots, A_i, \ldots, A_n \} \rightarrow_{\kappa, \sigma} \{ A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n \}, \text{ if there exists } \kappa : \forall \chi. B_1, \ldots, B_n \Rightarrow C \in \Phi \text{ such that } C \rightarrow_{\sigma} A_i. \]
Formalism: LP-Unif, LP-TM and LP-Struct

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

- **Unification reduction:**
  \[ \Phi \vdash \{ A_1, \ldots, A_i, \ldots, A_n \} \leadsto_{\kappa, \gamma, \gamma} \{ \gamma A_1, \ldots, \gamma B_1, \ldots, \gamma B_m, \ldots, \gamma A_n \}, \text{ if there exists } \kappa : \forall \underline{x}. B_1, \ldots, B_n \Rightarrow C \in \Phi \text{ such that } C \sim_{\gamma} A_i. \]
Formalism: LP-Unif, LP-TM and LP-Struct

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

- **Unification reduction:**
  \( \Phi \vdash \{A_1, \ldots, A_i, \ldots, A_n\} \leadsto_{\kappa, \gamma, \gamma'} \{\gamma A_1, \ldots, \gamma B_1, \ldots, \gamma B_m, \ldots, \gamma A_n\} \), if there exists \( \kappa : \forall x. B_1, \ldots, B_n \Rightarrow C \in \Phi \) such that \( C \sim_{\gamma} A_i \).

- **Substitutional reduction:**
  \( \Phi \vdash \{A_1, \ldots, A_i, \ldots, A_n\} \hookrightarrow_{\kappa, \gamma, \gamma'} \{\gamma A_1, \ldots, \gamma A_i, \ldots, \gamma A_n\} \), if there exists \( \kappa : \forall x. B_1, \ldots, B_n \Rightarrow C \in \Phi \) such that \( C \sim_{\gamma} A_i \).
Formalism: LP-Unif, LP-TM and LP-Struct

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

- **Unification reduction:**
  \[
  \Phi \vdash \{A_1, ..., A_i, ..., A_n\} \sim \rightarrow_{\kappa,\gamma,\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}, \text{ if there exists } \kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi \text{ such that } C \sim \gamma A_i.
  \]

- **Substitutional reduction:**
  \[
  \Phi \vdash \{A_1, ..., A_i, ..., A_n\} \leftrightarrow_{\kappa,\gamma,\gamma'} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\}, \text{ if there exists } \kappa : \forall \underline{x}. B_1, ..., B_n \Rightarrow C \in \Phi \text{ such that } C \sim \gamma A_i.
  \]

- **LP-TM:** \((\Phi, \rightarrow)\)
- **LP-Unif:** \((\Phi, \sim \rightarrow)\)
- **LP-Struct:** \((\Phi, \rightarrow^{\mu} \cdot \leftrightarrow^{1})\)
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
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:
\[
\{\text{Stream}(\text{cons}(x, y))\} \rightarrow_{\kappa,[x/x_1,y/y_1]} \{\text{Stream}(y)\}
\]
Consider query List(cons(x, y)):
\{List(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)\}

Note finiteness
LP-Struct: BList

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

\[
\begin{align*}
\{\text{List}(\text{cons}(x,y))\} & \rightarrow \{\text{Nat}(x), \text{List}(y)\}
\end{align*}
\]
LP-Struct: BList

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

- $\{\text{List}(\text{cons}(x, y))\} \rightarrow \{\text{Nat}(x), \text{List}(y)\}$
- $\leftarrow [0/x] \{\text{Nat}(0), \text{List}(y)\} \rightarrow \{\text{List}(y)\}$
For query $\text{List}(\text{cons}(x, y))$, in LP-Struct:

- $\{\text{List}(\text{cons}(x, y))\} \rightarrow \{\text{Nat}(x), \text{List}(y)\}$
- $\leftarrow_[0/x] \{\text{Nat}(0), \text{List}(y)\} \rightarrow \{\text{List}(y)\}$
- $\leftarrow_[0/x,\text{nil}/y] \{\text{List}(\text{nil})\} \rightarrow \emptyset$
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:

\( \Rightarrow \{ \text{Stream}(\text{cons}(x, y)) \} \rightarrow \{ \text{Stream}(y) \} \)
\( \kappa : \text{Stream}(y) \Rightarrow \text{Stream}(\text{cons}(x, y)) \)

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

1. \( \rightarrow \{ \text{Stream} \left( \text{cons} (x, y) \right) \} \rightarrow \{ \text{Stream} (y) \} \)
2. \( \leftarrow \left[ \text{cons}(x_1, y_1) \right] \{ \text{Stream} \left( \text{cons} (x_1, y_1) \right) \} \rightarrow \{ \text{Stream} (y_1) \} \)
LP-Struct: Stream

κ : Stream(y) ⇒ Stream(cons(x, y))
For query Stream(cons(x, y)), in LP-Struct:

▶ \{Stream(cons(x, y))\} → \{Stream(y)\}
▶ ↦_{[\text{cons}(x_1, y_1)/y]} \{Stream(\text{cons}(x_1, y_1))\} → \{Stream(y_1)\}
▶ ↦_{[\text{cons}(x_2, y_2)/y_1, \text{cons}(x_1, \text{cons}(x_2, y_2))/y]} \{Stream(\text{cons}(x_2, y_2))\} → \{Stream(y_2)\}
LP-Struct: Stream

κ : Stream(y) ⇒ Stream(cons(x, y))

For query Stream(cons(x, y)), in LP-Struct:

- ▶ {Stream(cons(x, y))} → {Stream(y)}
- ↪ [cons(x_1, y_1)/y] {Stream(cons(x_1, y_1))} → {Stream(y_1)}
- ↪ [cons(x_2, y_2)/y_1, cons(x_1, cons(x_2, y_2))/y_1] {Stream(cons(x_2, y_2))} → {Stream(y_2)}
- ↪ [cons(x_3, y_3)/y_2, cons(x_2, cons(x_3, y_3))/y_1, cons(x_1, cons(x_2, cons(x_3, y_3)))/y_1] {Stream(cons(x_3, y_3))} → {Stream(y_3)}
\( \kappa : \text{Stream}(y) \Rightarrow \text{Stream}(\text{cons}(x, y)) \)

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

- \( \{ \text{Stream}(\text{cons}(x, y)) \} \rightarrow \{ \text{Stream}(y) \} \)
- \( \leftarrow [\text{cons}(x_1, y_1)/y] \{ \text{Stream}(\text{cons}(x_1, y_1)) \} \rightarrow \{ \text{Stream}(y_1) \} \)
- \( \leftarrow [\text{cons}(x_2, y_2)/y_1, \text{cons}(x_1, \text{cons}(x_2, y_2))/y] \{ \text{Stream}(\text{cons}(x_2, y_2)) \} \rightarrow \{ \text{Stream}(y_2) \} \)
- \( \leftarrow [\text{cons}(x_3, y_3)/y_2, \text{cons}(x_2, \text{cons}(x_3, y_3))/y_1, \text{cons}(x_1, \text{cons}(x_2, \text{cons}(x_3, y_3)))/y] \{ \text{Stream}(\text{cons}(x_3, y_3)) \} \rightarrow \{ \text{Stream}(y_3) \} \)
- \( \ldots \)
- Partial answer: \( \text{cons}(x_1, \text{cons}(x_2, \text{cons}(x_3, y_3)))/y \)
Formalization of a Type System

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

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

Girard’s observation on intuitionistic sequent calculus with atomic formulas

\[
\begin{align*}
B \vdash A & \quad \text{axiom} \\
B \vdash C & \quad \text{subst} \\
\sigma B \vdash \sigma C & \quad \text{cut}
\end{align*}
\]
Formalization of a Type System

- Term $t ::= x \mid f(t_1, \ldots, t_n)$
  - Atomic Formula $A, B, C, D ::= P(t_1, \ldots, t_n)$
  - (Horn) Formula $F ::= A_1, \ldots, A_n \Rightarrow A$

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

- Girard’s observation on intuitionistic sequent calculus with atomic formulas

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

- Is $\vdash Q$ provable?
Formalization of a Type System

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

- Girard’s observation on intuitionistic sequent calculus with atomic formulas

\[
\begin{align*}
\frac{}{B \vdash A} & \quad \text{axiom} & \frac{B \vdash C}{\sigma B \vdash \sigma C} & \quad \text{subst} & \frac{A \vdash D, B, D \vdash C}{A, B \vdash C} & \quad \text{cut}
\end{align*}
\]

- Is $\vdash Q$ provable?

- We internalized “$\vdash$” as “$\Rightarrow$” and add proof term annotations

\[
\begin{align*}
\frac{}{\kappa : \forall x. F} & \quad \text{axiom} & \frac{e : F}{e : \forall x. F} & \quad \text{gen}
\end{align*}
\]

\[
\begin{align*}
\frac{e : \forall x. F}{e : [t/x]F} & \quad \text{inst} & \frac{e_1 : A \Rightarrow D, e_2 : B, D \Rightarrow C}{\lambda a. \lambda b. (e_2 b) (e_1 a) : A, B \Rightarrow C} & \quad \text{cut}
\end{align*}
\]
Soundness of LP-TM and LP-Unif

- **Soundness of LP-Unif**
  If $\Phi \vdash \{A\} \rightsquigarrow^*_\gamma \emptyset$, then there exists a proof $e : \forall x. \Rightarrow \gamma A$ given axioms $\Phi$.

- **Soundness of LP-TM**
  If $\Phi \vdash \{A\} \rightarrow^* \emptyset$, then there exists a proof $e : \forall x. \Rightarrow A$ given axioms $\Phi$.

- For example:
  \[
  \{\text{BList(cons}(x, y))\} \rightsquigarrow \{\text{Bit}(x), \text{BList}(y)\} \rightsquigarrow [0/x] \{\text{BList}(y)\} \\
  \rightsquigarrow [0/x, \text{nil}/y] \rightsquigarrow \emptyset
  \]

- yields a proof $(\lambda a.(\kappa_4 a) \kappa_1) \kappa_3$, $\beta$-reducible to $(\kappa_4 \kappa_3) \kappa_1$. 
Soundness of LP-TM and LP-Unif

- **Soundness of LP-Unif**
  If $\Phi \vdash \{A\} \rightsquigarrow^* \gamma \emptyset$, then there exists a proof $e : \forall x. \Rightarrow \gamma A$ given axioms $\Phi$.

- **Soundness of LP-TM**
  If $\Phi \vdash \{A\} \rightarrow^* \emptyset$, then there exists a proof $e : \forall x. \Rightarrow A$ given axioms $\Phi$.

- For example:
  \[
  \{\text{BList(cons}(x, y))\} \rightsquigarrow \{\text{Bit}(x), \text{BList}(y)\} \rightsquigarrow [0/x] \{\text{BList}(y)\} \\
  \rightsquigarrow [0/x, \text{nil}/y] \rightsquigarrow \emptyset
  \]
  yields a proof $(\lambda a. (\kappa_4 a) \kappa_1) \kappa_3$, $\beta$-reducible to $(\kappa_4 \kappa_3) \kappa_1$.

- Compare with the 3TC proof-witness:

  $$
  \begin{array}{cccccc}
  \text{list(cons}(x, y)) & \rightarrow & \ldots & \rightarrow & \text{list(cons}(0, \text{nil})) \\
  X_1 & X_2 & X_3 & 4 \backslash \ \ \\
  \text{nat}(x) & \text{list}(y) & \ \\
  X_4 & X_5 & X_6 & X_7 & X_8 & X_9 & X_{10} & X_{11}
  \end{array}
  $$
LP-Struct is equivalent to LP-Unif

... for logic programs subject to realisability transformation

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

\[
\{ \text{BList}(\text{cons}(x, y, u)) \} \xrightarrow{[f_{\kappa_4}(u_1, u_2)/u]} \{ \text{Bit}(x, u_1), \text{BList}(y, u_2) \}
\]
LP-Struct is equivalent to LP-Unif

... for logic programs subject to realisability transformation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\kappa_1: \Rightarrow \text{Nat}(0, c_{\kappa_1}))</td>
<td></td>
</tr>
<tr>
<td>(\kappa_2: \text{Nat}(x, u) \Rightarrow \text{Nat}(s(x), f_{\kappa_2}(u)))</td>
<td></td>
</tr>
<tr>
<td>(\kappa_3: \Rightarrow \text{BList}(\text{nil}, c_{\kappa_3}))</td>
<td></td>
</tr>
<tr>
<td>(\kappa_4: \text{Bit}(x, u_1), \text{BList}(y, u_2) \Rightarrow \text{BList}(\text{cons}(x, y, f_{\kappa_4}(u_1, u_2))))</td>
<td></td>
</tr>
</tbody>
</table>

- \(\{\text{BList}(\text{cons}(x, y, u))\} \xrightarrow{[f_{\kappa_4}(u_1, u_2)/u]} \{\text{BList}(\text{cons}(x, y, f_{\kappa_4}(u_1, u_2)))\} \rightarrow \{\text{Bit}(x, u_1), \text{BList}(y, u_2)\}\)
- \(\xleftarrow{[0/x, c_{\kappa_1}/u_1]} \{\text{Bit}(0, c_{\kappa_1}), \text{BList}(y, u_2)\} \rightarrow \{\text{BList}(y, u_2)\}\)
- \(\xleftarrow{[0/x, \text{nil}/y, c_{\kappa_3}/u_2]} \{\text{BList}(\text{nil}, c_{\kappa_3})\} \rightarrow \emptyset\)

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 $\lambda$-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

Motivation

Coalgebraic Semantics for Structural Resolution

The Three Tier Tree calculus for Structural Resolution

Type-Theoretic view of Structural Resolution

Conclusions and Future work
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

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://staffcomputing.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