Structural Resolution

Katya Komendantskaya

School of Computing, University of Dundee, UK

07 May 2015

Outline

Programming Language Semantics

Why do we call Computing Computer Science?

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

Example

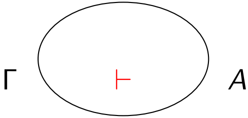
Programming languages are engineered; Their semantics – e.g. λ-calculus have been discovered...

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...

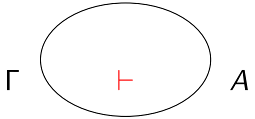
$$\Gamma \vdash p : A$$

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

Proof inference methods

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**:

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)

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

$$|$$

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

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)

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

$$|$$

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

$$|$$

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

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)`

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

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

`nat(x), list(y)`

4.`list(nil)` $\leftarrow$

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

$|$

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

$|$

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

$|$

$\dots$

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

NatList2:

Example

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

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

|

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

|

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

|

$\ldots$

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)` $\leftarrow$

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

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

`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...

$$\leftarrow \texttt{stream}(\texttt{scons}(\texttt{x},\texttt{y}))$$
$$|$$
$$\leftarrow \texttt{bit}(\texttt{x}), \texttt{stream}(\texttt{y})$$
$$|$$
$$\leftarrow \texttt{stream}(\texttt{y})$$
$$|$$
$$\leftarrow \texttt{bit}(\texttt{x}_1), \texttt{stream}(\texttt{y}_1)$$
$$|$$
$$\leftarrow \texttt{stream}(\texttt{y}_1)$$
$$|$$
$$\vdots$$

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...

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

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 **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.
$\texttt{nat(s(x))} \leftarrow \texttt{nat(x)}$ $\texttt{nat(0)} \leftarrow$ inductive definition	$\texttt{stream(scons(x,y))} \leftarrow \texttt{nat(x), stream(y)}$ coinductive definition	$\texttt{bad(x)} \leftarrow \texttt{bad(x)}$ 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?

Outline

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

Example

$T \leftarrow Q, R$
$T \leftarrow S$
$p(T) = \{\{Q, R\}, \{S\}\}$

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 2: Derivations modelled by coalgebra for the comonad on $P_f P_f$

In general, if $U : H\text{-}coalg \longrightarrow C$ 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

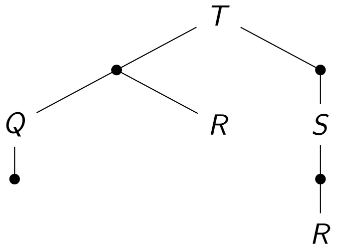
  $$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

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

Example

Example

$T \leftarrow Q, R$
$T \leftarrow S$
$Q \leftarrow$
$S \leftarrow R$



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* $\mathscr{L}_\Sigma$ to model the terms over Σ
$*$ $\mathtt{ob}(\mathscr{L}_\Sigma)$ is $\mathbb{N}$.
$**$ For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
$***$ $\mathtt{ob}(\mathscr{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
$****$ composition in $\mathscr{L}_\Sigma$ 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 $I : \mathbb{N}^{op} \to L$.

Take *Lawvere Theory* $\mathscr{L}_\Sigma$ to model the terms over Σ
$*$ $\mathtt{ob}(\mathscr{L}_\Sigma)$ is $\mathbb{N}$.
$**$ For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
$***$ $\mathtt{ob}(\mathscr{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
$****$ composition in $\mathscr{L}_\Sigma$ is first-order substitution.
Take the functor $At : \mathscr{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ 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 *Lawvere Theory* $\mathscr{L}_\Sigma$ to model the terms over Σ
$*$ $\mathtt{ob}(\mathscr{L}_\Sigma)$ is $\mathbb{N}$.
$**$ For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
$***$ $\mathtt{ob}(\mathscr{L}_\Sigma)(n, m)$ is the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$.
$****$ composition in $\mathscr{L}_\Sigma$ is first-order substitution.
Take the functor $At : \mathscr{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ and n variables.
Model a program P by the $[\mathscr{L}_\Sigma^{op}, P_f P_f]$-coalgebra.

Examples

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

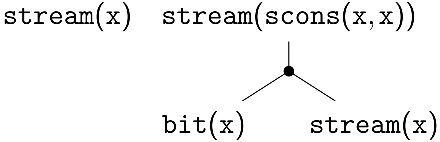
Examples

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

```
stream(x)
```

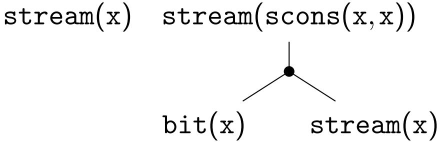
Examples

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

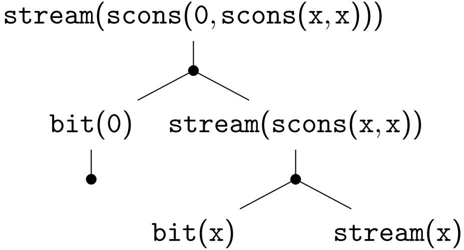


Examples

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



Note the finite size

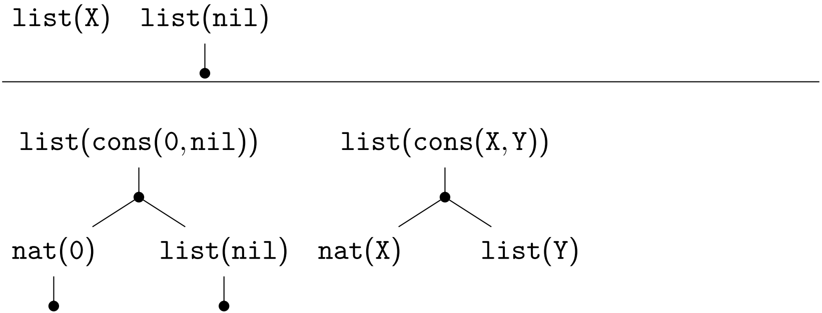


Examples

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:

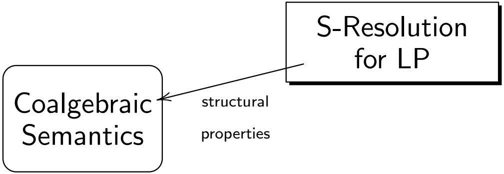


Note the partial nature...

Structural Resolution:

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 &$\vee$-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 &∨-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

Our running example

Example

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

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

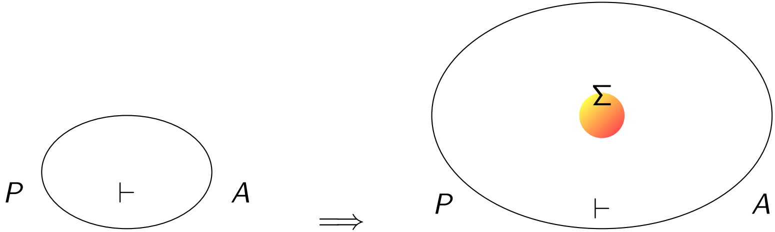
Defining structural resolution from first principles...

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...

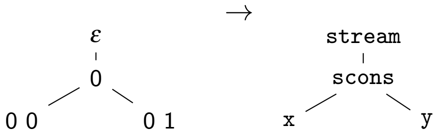


Example

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

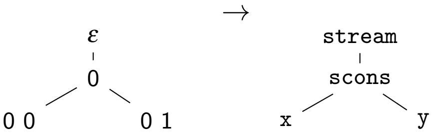
Tier-1: Term-trees, given Σ:

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:

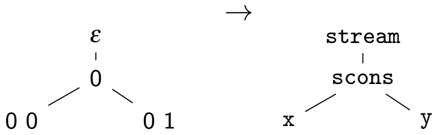
Let $\mathbb{N}^*$ denote the set of all finite words over $\mathbb{N}$.
A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language*, satisfying prefix closedness conditions.
A term tree is a map $L \rightarrow \Sigma \cup Var$, satisfying term arity restrictions.



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.



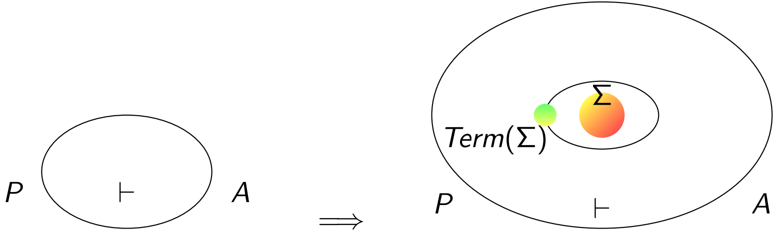
Given two terms t_1, t_2, and a substitution θ, θ is a unifier if $\theta(t_1) = \theta(t_2)$, and matcher if $t_1 = \theta(t_2)$.

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

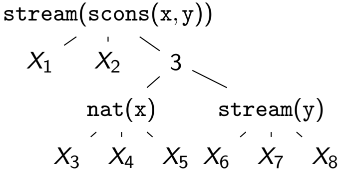
Constructing the structural resolution from first principles...

- 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 $L \rightarrow$ **Term**$(\Sigma) \cup$ **Clause**$(\Sigma) \cup Var_R$, subject to conditions (Term-matching).



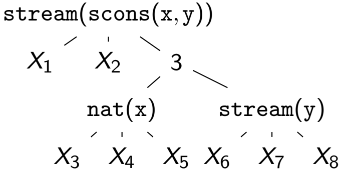
our running example

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

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

Tier-2: rewriting trees

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



our running example

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

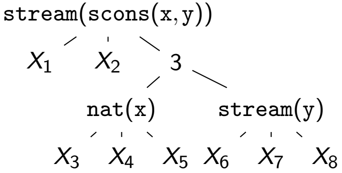
Interesting: all rewriting trees are finite for our "difficult" example!

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 $\textbf{Term}(\Sigma)$

Tier-2: rewriting trees

A rewriting tree is a map $L \to$ **Term**$(\Sigma) \cup$ **Clause**$(\Sigma) \cup Var_R$, subject to conditions (Term-matching).



our running example

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

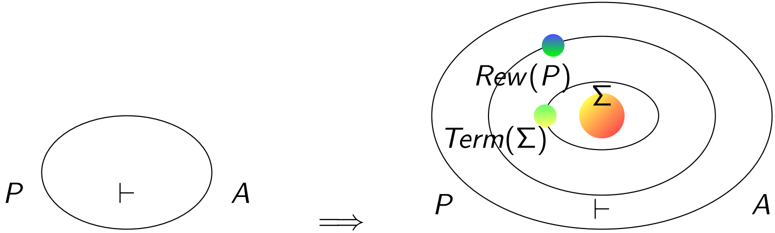
Interesting: all rewriting trees are finite for our "difficult" example!

Notation:

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

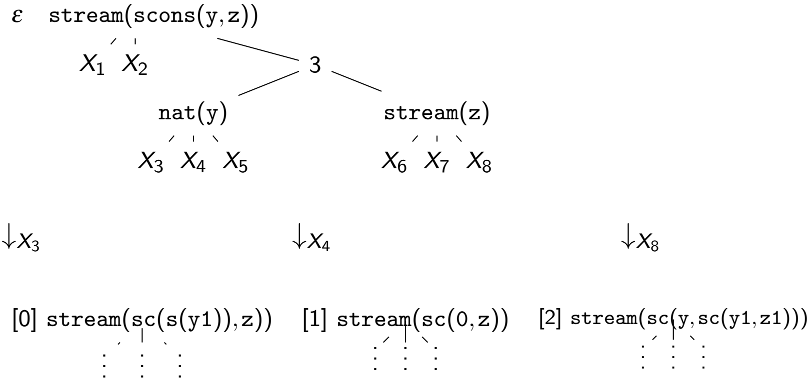
Constructing the structural resolution from first principles...

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



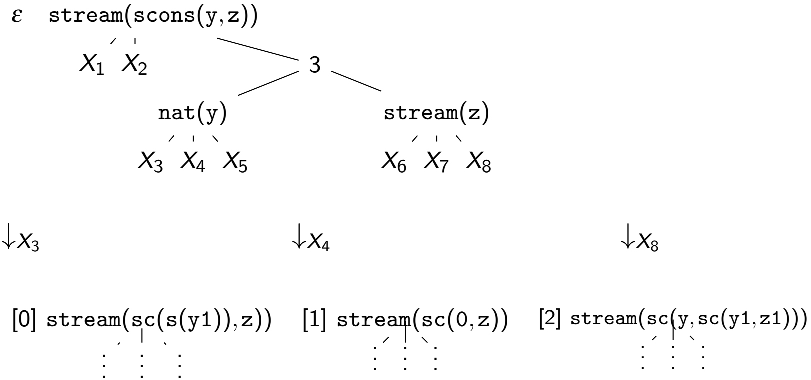
Tier-3: Derivation trees

A derivation tree is a map $L \to$ **Rew**(P).



Tier-3: Derivation trees

A derivation tree is a map $L \to \textbf{Rew}(P)$.



Note: this derivation tree is infinite.

Tier-3 laws and notation

Notation:

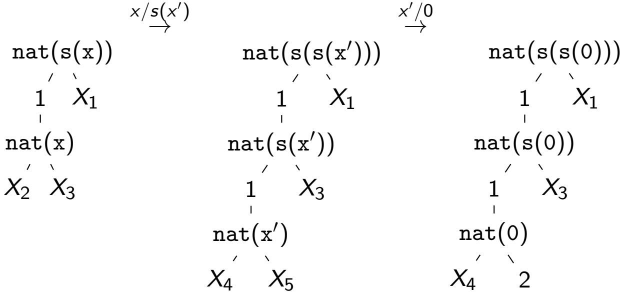
$\mathbf{Der}(P)$	all *finite* derivation trees over $\mathbf{Rew}(P)$
$\mathbf{Der}^{\infty}(P)$	all *infinite* derivation trees over $\mathbf{Rew}(P)$
$\mathbf{Der}^{\omega}(P)$	all *finite and infinite* derivation trees over $\mathbf{Rew}(P)$

Tier-3 laws and notation

Notation:

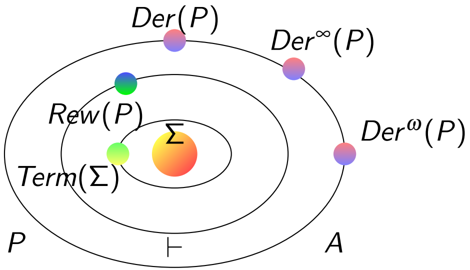
Der(P)	all *finite* derivation trees over **Rew**(P)
Der$^\infty$(P)	all *infinite* derivation trees over **Rew**(P)
Der$^\omega$(P)	all *finite and infinite* derivation trees over **Rew**(P)

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!

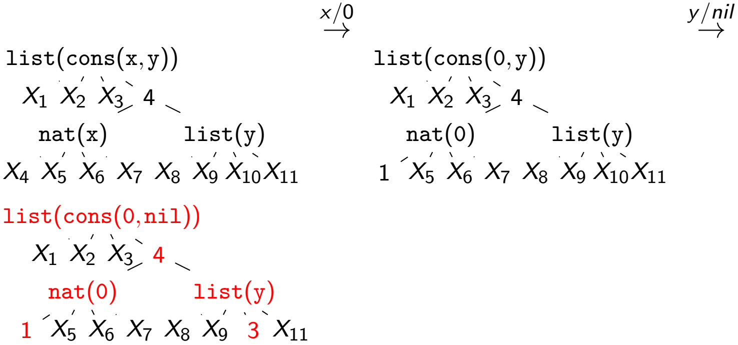


Gains:

- 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:



Gains

The structural approach allowed to:

- Formulate the theory of Universal Productivity
- Show Finite derivations sound and complete wrt Herbrnad 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 **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

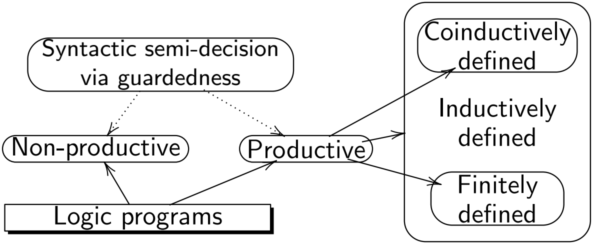
A program P is productive, if it gives rise to rewriting trees only in **Rew**(P).

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 **Der**$^{\omega}(P)$;
- coinductive LPs all derivations for which are in **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 $Rew(P)$

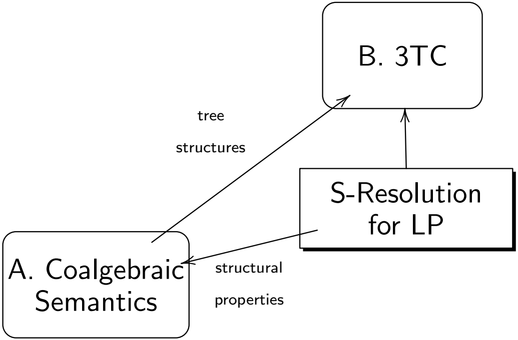
Theory of universal Productivity in LP!



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

  $$P \vdash rew : A$$

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

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:**
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$, 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 $C \sim_\gamma A_i$.

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 $C \sim_\gamma A_i$.
- **LP-TM:** $(\Phi, \to)$
  LP-Unif: $(\Phi, \leadsto)$
  LP-Struct: $(\Phi, \to^\mu \cdot \hookrightarrow^1)$

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

Execution behavior of LP-TM

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

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:
  $\{\text{Stream}(\text{cons}(x, y))\} \rightarrow_{\kappa, [x/x_1, y/y_1]} \{\text{Stream}(y)\}$

Note finiteness

LP-Struct: BList

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

- $\{\text{List}(\text{cons}(x, y))\} \rightarrow \{\text{Nat}(\text{x}), \text{List}(y)\}$

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

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

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

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

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

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:

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

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:

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

Formalization of a Type System

- Term $t ::= x \mid f(t_1, ..., t_n)$
  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

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

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 $t ::= x \mid f(t_1, ..., t_n)$
  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?
- We internalized "$\vdash$" as "$\Rightarrow$" and add proof term annotations

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

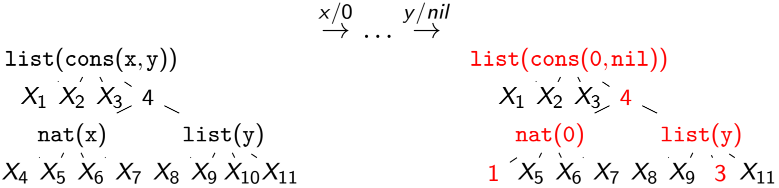
$$\frac{e : \forall \underline{x}.F}{e : [\underline{t}/\underline{x}]F}\ \mathit{inst} \qquad \frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow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

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 $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3$, 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 $(\lambda a.(\kappa_4\ a)\ \kappa_1)\ \kappa_3$, 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

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

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

LP-Struct is equivalent to LP-Unif

... for logic programs subject to realisability transformation

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

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

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*

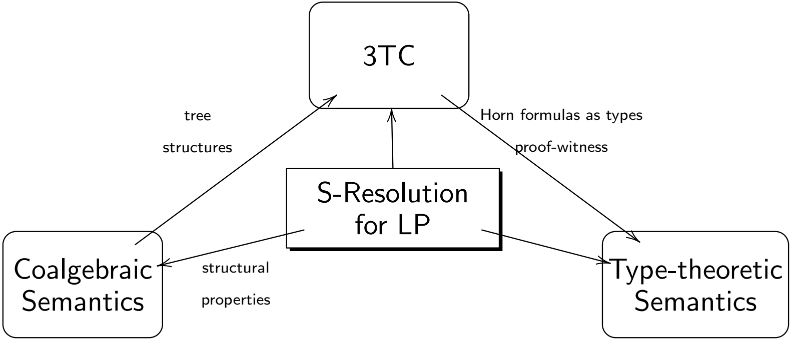
Gains from type-theoretic semantics for S-Resolution:

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

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