Horn clause logic: the knowns and the unknowns

Ekaterina Komendantskaya

School of Mathematical and Computer Sciences, Heriot-Watt University

ALCOP’17, 11 April 2017
Outline

Motivation
Outline

Motivation

The “knowns”:
   Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
   Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
   Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP
Outline

Motivation

The “knowns”:
  Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
  Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
  Structural resolution
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value
Outline

Motivation

The “knowns”:
   Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
   Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
   Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
The Semantic triangle [Lambek and Scott]
Horn clause logic is a fragment of predicate logic, in which all formulae are written in clausal form. Turing complete if taken as a programming language. Logician A. Horn first pointed out its significance in 1951.
Horn clause logic is a fragment of predicate logic, in which all formulae are written in clausal form.
Horn clause logic is a fragment of predicate logic, in which all formulae are written in clausal form.

- Turing complete if taken as a programming language.
This talk:

- Horn clause logic is a fragment of predicate logic, in which all formulae are written in clausal form.
- Turing complete if taken as a programming language.
- Logician A. Horn first pointed out its significance in 1951.
Syntax of Horn-clause Logic

First-order signature $\Sigma$ and terms, term-trees

- function symbols with arity;
- variables.

Example

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

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

```
  stream
  |   
  scons
  x   y
```
### Sets of terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Term}(\Sigma)$</td>
<td>Set of <em>finite</em> term trees over $\Sigma$</td>
</tr>
<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>
Sets of terms:

<table>
<thead>
<tr>
<th>Term((\Sigma))</th>
<th>Set of <em>finite</em> term trees over (\Sigma)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term^∞((\Sigma))</td>
<td>Set of <em>infinite</em> term trees over (\Sigma)</td>
</tr>
<tr>
<td>Term^ω((\Sigma))</td>
<td>Set of <em>finite and infinite</em> term trees over (\Sigma)</td>
</tr>
</tbody>
</table>

GTerm(\(\Sigma\)), GTerm^∞(\(\Sigma\)), GTerm^ω(\(\Sigma\)) will denote sets of ground (variable free) terms.
Syntax of Horn-clause Logic

Horn Clauses

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

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

Universal quantification is assumed.

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

... Gives us a Turing-complete programming language.
Example: lists of natural numbers

Example

\[
\begin{align*}
nat(0) & \leftarrow \\
nat(s(x)) & \leftarrow \ nat(x) \\
list(nil) & \leftarrow \\
list(cons(x,y)) & \leftarrow \ nat(x), \ list(y)
\end{align*}
\]
Why Horn clause Logic?

Well-defined model-theoretic properties:

- clean denotational (least and greatest) fixed point semantics. (Fixpoint construction for a monotone functor á la Knaster-Tarski.)
Why Horn clause Logic?

Well-defined model-theoretic properties:

- clean denotational (least and greatest) fixed point semantics. (Fixpoint construction for a monotone functor à la Knaster-Tarski.)

Sweet spot between expressivity and automation:

- It is long known to yield efficient proofs by resolution
  - logic of choice for first implementations of Prolog in 70s and 80s;
    - and many resolution-based provers in the 90s.
- SLD-resolution is not just sound but also complete relative to the least fixed point semantics.
Given a logic program $P$, and terms $t_1, \ldots, t_i, \ldots, t_n$ we define

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

Program **NatList**:  

**Example**

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

\( \text{nat}(x), \text{list}(y) \)
SLD-resolution

Example

1. \text{nat}(0) \leftarrow \\
2. \text{nat}(\text{s}(x)) \leftarrow \text{nat}(x) \\
3. \text{list}(\text{nil}) \leftarrow \\
4. \text{list}(\text{cons}(x,y)) \leftarrow \\
   \begin{array}{l}
   \text{nat}(x), \text{list}(y)
   \end{array}
Example

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

   \[ \text{nat}(x), \text{list}(y) \]

\[
\text{list}(\text{cons}(x,y)) \\
\mid \text{nat}(x), \text{list}(y) \\
\mid \text{list}(y)
\]
Example

1. $\text{nat}(0) \leftarrow$
2. $\text{nat}(s(x)) \leftarrow \text{nat}(x)$
3. $\text{list}($nil$) \leftarrow$
4. $\text{list}($cons$(x,y)) \leftarrow$
   $\text{nat}(x), \text{list}(y)$
   
The answer is “Yes”, NatList $\vdash \text{list}($cons$(x,y))$ if $x/0$, $y/\text{nil}$, but we can get more substitutions by backtracking.

SLD-refutation $= \text{finite successful SLD-derivation}$. 
Why Horn clause Logic?

In 2000s, emerged as a unifying language of ATP:

- allows elegant extensions to constraint LP and other enriched variants;
- a neat connection to Hoare Logic was discovered in 1987;
- in 2000s, Horn constraints have been shown to relate to Craig interpolation, which is one of the main techniques used to construct and refine abstractions in verification, and to synthesise inductive loop invariants;
- from 2010 onwards, increasingly used in SMT-solvers, model checkers, abstract interpretation (Bjorner, Rybalchenko);
- higher-order Horn clauses are used in model checkers of functional languages (Ong, Kobayashi)
Why Horn clause logic?

In 1989, Girard suggested to use the cut rule to model resolution for Horn formulas.

In the 1990s, Miller et. al. use cut-free sequent calculus to represent proofs in Horn clause logic.

Interactive theorem prover Twelf (by Pfenning et al.) pioneered implementation of proof search for Horn clause logic on top of a dependently typed system called LF (Harper & Licata).

In 2016, Fu & Komendantskaya gave a Horn clause-as-types (proofs as terms in STLC) interpretation to a fragment of Horn clause logic.
Why Horn clause Logic?

Applications in Programming languages, via type inference

- Type classes in Haskell (Jones, 90s, ext. 2000s)
- GADTs in Haskell (Stuckey, Schrijvers, et al. late 90s onwards)
- Type Classes in Coq and SSReflect (Ziliani, Gonthier et al. 2014)
- Class inference in Java (Ancona et al, 2000s)
class Eq x where
    eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
    eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
    eq x y = primitiveIntEq x y
class Eq x where
    eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
    eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
    eq x y = primitiveIntEq x y

This translates into the following logic program:

\[ Eq (x), Eq (y) \Rightarrow Eq (x, y) \]
\[ \Rightarrow Eq (Int) \]
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primitiveIntEq x y

This translates into the following logic program:

\[ \text{Eq}(x), \text{Eq}(y) \Rightarrow \text{Eq}(x, y) \]
\[ \Rightarrow \text{Eq}(\text{Int}) \]

Resolve the query \( \text{Eq}(\text{Int}, \text{Int}) \).

▷ We have the following reduction by SLD-resolution:

\[ \Phi \vdash \text{Eq}(\text{Int}, \text{Int}) \rightarrow \text{Eq}(\text{Int}), \text{Eq}(\text{Int}) \rightarrow \text{Eq}(\text{Int}) \rightarrow \emptyset \]
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Inductive semantics of LP, 70s

Least Herbrand models

Horn Clause Logic

SLD-resolution

Big Step semantics
Inductive Semantics of LP

Definition (Big step rule)

\[
P \models \sigma(B_1), \ldots, P \models \sigma(B_n) \\
P \models \sigma(A)
\]

for some grounding substitution \(\sigma\), and \(A \leftarrow B_1, \ldots B_n \in P\).

Definition

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

Example

Taking the logic program \(Nat\), we obtain the set \(M_{Nat} = \{\text{nat}(0), \text{nat}(s(0)), \text{nat}(s(s(0))), \ldots\}\).
Least Herbrand models as a least fixed point construction

**Definition**

The function $T_P : \mathcal{P}(\text{GTerm}(\Sigma)) \rightarrow \mathcal{P}(\text{GTerm}(\Sigma))$ is defined by $T_P(A) = A \cup \{ \theta(t) \in \text{GTerm}(\Sigma) \mid t \leftarrow t_1, \ldots, t_n \in P \text{ and each } \theta(t_i) \in A \}$.

**Definition**

The *least Herbrand model* for $P \in \text{LP}(\Sigma)$ is the smallest set $M_P \in \mathcal{P}(\text{GTerm}(\Sigma))$ such that

- if a clause $A \leftarrow \in P$ then $\theta(A) \in M_P$ for every substitution $\theta$ such that $\theta(A) \in \text{GTerm}(\Sigma)$, and
- $T_P(M_P) = M_P$. 
Soundness and Completeness

[70s: Apt, van Emden, Kowalski]

**Theorem (Soundness and Completeness of Derivations)**

**Soundness.** Given a logic program $P$, and an atom $A$, if there is an SLD-refutation for $P$ and $\leftarrow A$, then there is a grounding substitution $\theta$, such that $\theta(A) \in M_P$.

**Completeness.** Given a logic program $P$, and an atom $A \in M_P$, there is an SLD-refutation for $A$. 
Colinductive semantics

Greatest Herbrand models

Horn Clause Logic

Big Step semantics
**ColInductive Semantics of LP**

**Definition (Big step rule)**

\[
P \models \sigma(B_1), \ldots, P \models \sigma(B_n)
\]

\[
P \models \sigma(A)
\]

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

**Definition**

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

**Example**

$M_{Nat}^\omega$ will now be given by the set:

\[
\{\text{nat}(0), \text{nat}(s(0)), \text{nat}(s(s(0))), \ldots\} \cup \{\text{nat}(s(s(\ldots)))\}.
\]
Some programs have only one natural interpretation:

Example

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

\[ M_{\text{Stream}} = \{ \text{bit}(0), \text{bit}(1) \} \]
\[ M_{\omega \text{Stream}}^{\omega} = \{ \text{bit}(0), \text{bit}(1), \text{stream}(\text{scons}(0, \text{scons}(0, \ldots)), \text{stream}(\text{scons}(1, \text{scons}(0, \ldots))), \ldots) \} \]
Coinductive programs

Some programs have only one natural interpretation:

**Example**

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

\[ M_{Stream} = \{ \text{bit}(0), \text{bit}(1) \} \]
\[ M^{\omega}_{Stream} = \{ \text{bit}(0), \text{bit}(1), \text{stream}(\text{scons}(0, \text{scons}(0, \ldots))), \text{stream}(\text{scons}(1, \text{scons}(0, \ldots))), \ldots \} \]

... now the coinductive semantics does not match with the actual SLD-refutations...
Colinductive semantics, the 80s

- Greatest Herbrand models
- Big Step semantics
- SLD-computations at infinity
- Horn Clause Logic

?
“SLD Computations at infinity” [80s, van Emden & Abdallah, Lloyd]

Definition

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

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

Program **Stream**:  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>bit(0) ←</td>
</tr>
<tr>
<td>2.</td>
<td>bit(1) ←</td>
</tr>
<tr>
<td>3.</td>
<td>stream(scons(x,y)) ←</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So far, we have computed that \( x \mapsto 0, \ y \mapsto scons(x_1,y_1), \) and \( x_1 \mapsto 0. \) At infinity, the term \( scons(0,scons(0,...)) \) is computed.
Computations at infinity as globally productive computations

<table>
<thead>
<tr>
<th>Program definition</th>
<th>For query $\ ? \leftarrow p(x)$, computes the answer:</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(x) \leftarrow p(x)$</td>
<td>$id$</td>
<td>${p(a), p(f(a), p(f(f(a))), \ldots, p(f^\omega))}$</td>
</tr>
<tr>
<td>$p(x) \leftarrow p(f(x))$</td>
<td>$id$</td>
<td>${p(a), p(f(a), p(f(f(a))), \ldots, p(f^\omega))}$</td>
</tr>
<tr>
<td>$p(f(x)) \leftarrow p(x)$</td>
<td>${x \mapsto f(f\ldots)}$</td>
<td>${p(f^\omega)}$</td>
</tr>
</tbody>
</table>
Computations at infinity are sound

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

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

Given a $P \in \text{LP}(\Sigma)$, $C_P \subseteq M_P^\infty$. 
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Coalgebraic operational semantics, K&Power, 2010-14

SLD-resolution

Horn Clause Logic

Coalgebraic Semantics

Small Step semantics

?
Fibrational Coalgebraic Semantics of CoALP 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 $c : U \rightarrow F(U)$.

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 $p : At \rightarrow P_f(P_f(At))$ 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 as Comonads

In general, if $U : H\text{-coalg} \to C$ has a right adjoint $G$, the composite functor $UG : C \to C$ possesses the canonical structure of a comonad $C(H)$, called the cofree comonad on $H$. One can form a coalgebra for a comonad $C(H)$. 
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 2: Derivations as Comonads

In general, if \( U : H\text{-}coalg \rightarrow C \) has a right adjoint \( G \), the composite functor \( UG : C \rightarrow C \) 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 \( p : At \rightarrow P_fP_f(At) \), the corresponding \( C(P_fP_f)\)-coalgebra where \( C(P_fP_f) \) is the cofree comonad on \( P_fP_f \) is given as follows: \( C(P_fP_f)(At) \) is given by a limit of the form

\[
\ldots \rightarrow At \times P_fP_f(At \times P_fP_f(At)) \rightarrow At \times P_fP_f(At) \rightarrow At.
\]

This gives a “tree-like” structure: \&\&V-trees.
Example

This models and-or parallel trees known in LP [AMAST 2010]
A Lawvere theory consists of a small category $L$ with strictly associative finite products, and a strict finite-product preserving identity-on-objects functor $I : \mathbb{N}^{op} \to L$.

- Take 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.
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

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

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

- Take 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 $n$ vars.
Fibrational Coalgebraic Semantics of CoALP in 3 ideas

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

A Lawvere theory consists of a small category $L$ with strictly associative finite products, and a strict finite-product preserving identity-on-objects 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 $\text{At} : \mathcal{L}_\Sigma^{op} \rightarrow \text{Set}$ that sends a natural number $n$ to the set of all atomic formulae generated by $\Sigma$ and $n$ vars.
- model a program $P$ by the $[\mathcal{L}_\Sigma^{op}, P_f P_f]$-coalgebra $p : \text{At} \rightarrow P_f P_f \text{At}$ on the category $[\mathcal{L}_\Sigma^{op}, \text{Set}]$. 
Final remarks

Actually, some modifications are needed:

- we need to extend $\text{Set}$ to $\text{Poset}$,
- natural transformations to $\text{lax natural transformations}$, and
- replace the outer instance of $P_f$ by $P_c$ - the countable powerset functor (as recursion generates countability).
Final remarks

Actually, some modifications are needed:

- we need to extend \( \text{Set} \) to \( \text{Poset} \),
- natural transformations to \textit{lax natural transformations}, and
- replace the outer instance of \( P_f \) by \( P_c \) - the countable powerset functor (as recursion generates countability).

Then \( p : At \rightarrow P_cP_fAt \) gives a \( \text{Lax}(\mathcal{L}_\Sigma^{op}, P_cP_f) \)-coalgebra structure on \( At \).
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1. & V-trees:
Examples

Program **Stream**: “fibers” given by term arities. Take the fiber of 1. $\& V$-trees:

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

Program **Stream**: “fibers” given by term arities. Take the fiber of 1. \&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. \& \quad V
trees:

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

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

\[
\text{stream}(\text{scons}(0, \text{scons}(\text{x}, \text{x})))
\]

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

\[
\text{bit}(\text{x}) \quad \text{stream}(\text{x})
\]
Program $p(x) ← p(f(x))$: “fibers” given by term arities. Take the fiber of 1. & $V$-trees:
Examples

Program \( p(x) \leftarrow p(f(x)) \): “fibers” given by term arities. Take the fiber of 1. \& \( V \)-trees:

\[
\begin{align*}
p(x) & \quad p(f(x)) \\
p(f(x)) & \quad p(f(f(x))) \\
p(f(f(x))) & \quad p(f(f(f(x)))) \\
& \quad \ldots
\end{align*}
\]
Examples

Program \( p(f(x)) \leftarrow p(x) \): “fibers” given by term arities. Take the fiber of 1. & \( V \)-trees:
Examples

Program $p(f(x)) \leftarrow p(x)$: “fibers” given by term arities. Take the fiber of 1. & $V$-trees:

```
  p(x)  p(f(x))  p(f(f(x))) ...
     |     |      |
     v     v      v
  p(x)  p(f(x))
     |     |
     v     v
  p(x)
     |    
     v   
  p(x)
```

Whatever it is, it is no longer SLD-resolution!
Outline

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
S-resolution reductions

Matchers

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

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

- Term-Matching (Rewriting) reduction: \( P \vdash [t_1, \ldots, t_i, \ldots, t_n] \rightarrow [t_1, \ldots, \sigma(B_0), \ldots, \sigma(B_m), \ldots, t_n] \) if \( A \leftarrow B_1, \ldots, B_m \in P \), and \( A \prec_\sigma t_i \).
**S-resolution reductions**

**Matchers**

$t ≼_σ t'$ denotes a matcher of $t$ and $t'$, i.e. $σ(t) = t'$

- **SLD-reduction:** $P \vdash [t_1, \ldots, t_i, \ldots, t_n] \leadsto [σ(t_1), \ldots, σ(B_0), \ldots, σ(B_m), \ldots, σ(t_n)]$ if $A ← B_1, \ldots, B_m ∈ P$, and $t_i ≈_σ A$.

- **Term-Matching (Rewriting) reduction:**
  $P \vdash [t_1, \ldots, t_i, \ldots, t_n] → [t_1, \ldots, σ(B_0), \ldots, σ(B_m), \ldots, t_n]$ if $A ← B_1, \ldots, B_m ∈ P$, and $A ≼_σ t_i$.

- **Substitutional reduction:**
  $P \vdash [t_1, \ldots, t_i, \ldots, t_n] ⇔ [σ(t_1), \ldots, σ(t_i), \ldots, σ(t_n)]$ if $A ← B_1, \ldots, B_m ∈ P$, and $t_i ≈_σ A$.

- **S-resolution reduction:** $P \vdash \left[\overline{t}\right] →^μ ∈_1 \left[\overline{t'}\right]$.

Then, $(P, \leadsto)$ is a reduction system that models SLD-reductions for $P$, and $(P, →^μ ∈_1)$ is a reduction system that models S-resolution reductions for $P$. 
Example

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

1. SLD-resolution reduction:
   
   \[
   \text{stream}(x) \rightsquigarrow \text{bit}(x'), \text{stream}(y) \rightsquigarrow \text{stream}(y) \rightsquigarrow \text{bit}(x''), \text{stream}(y') \rightsquigarrow \ldots
   \]

2. Term-matching reduction: \( \text{stream}(x) \not\leftrightarrow \)

3. S-resolution reduction:
   
   \[
   \text{stream}(x) \leftarrow^1 \text{stream(scons}(x', y)) \rightarrow^\mu
   \text{bit}(x'), \text{stream}(y) \leftarrow^1 \text{bit}(0), \text{stream}(y) \rightarrow^\mu \text{stream}(y) \leftarrow^1
   \text{stream(scons}(x'', y')) \rightarrow^\mu \text{bit}(x''), \text{stream}(y') \ldots
   \]

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

Motivation

The “knowns”:
Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Is $S$-resolution really interesting?

- Inductively sound and complete (relative to least Herbrand models)
- Coinductively sound and complete (relative to SLD-computations at infinity)
- Coinductively sound (relative to greatest Herbrand models)

Relating $S$-resolution to the Coalgebraic semantics is subtle

- first theorems appeared in JLC’16 paper by K&Power;
- coalgebraic semantics is further refined in CMCS’16 paper to allow for closer correspondence;
What did it teach us?

- First ever notion of observational productivity for LP
- ...
Productivity for LP

Observational productivity of a program $P$ is a conjunction of two properties of $P$:

1. *universal observability*: termination of all rewriting derivations, and

2. *existential liveness*: existence of at least one non-terminating S-resolution or SLD-resolution derivation.

Program definition For query $? \leftarrow p(x)$, computes the answer:

```
    p(x) ← p(x)  \quad \text{id}
    p(x) ← p(f(x))  \quad \text{id}
    p(f(x)) ← p(x)  \quad \{x \mapsto f(f(\ldots))\}
```

Implementation of the productivity checker is available at https://github.com/coalp
What did it teach us?

- First Ever notion of productivity for LP
- Connection of LP to term-rewriting
- ...

Given a Horn clause \( A \leftarrow B_1, \ldots, B_n \)
get a rewriting rule: \( A \longrightarrow \kappa(B_1, \ldots, B_n) \).
Get one-to-one correspondence between rewriting reductions in LP and rewriting in TRS.
What did it teach us?

- First Ever notion of productivity for LP
- Connection of LP to term-rewriting via rewriting reductions
- Arising natural type-theoretic semantics
Type Theoretic semantics, Fu and K, 2015-2016

- Structural Resolution
  - Horn Clause Logic
    - Horn clauses as Types
    - terms as proofs by rewriting reductions
  - Coalgebraic Semantics
  - Small Step semantics

Horn Formulas as Types

Proof evidence as Terms

Term \( t \) ::= \( x \) | \( K(t_1, ..., t_n) \)

Atomic Formula \( A, B, C, D \) ::= \( P(t_1, ..., t_n) \)

Formula \( F \) ::= \( A \) | \( F \Rightarrow F' \) | \( \forall x. F \)

Horn Formula/Horn Clause \( H \) ::= \( \forall x. A_1, ..., A_n \Rightarrow B \)

Proof Evidence \( p, e \) ::= \( \kappa \) | \( a \) | \( e \ e' \) | \( \lambda a. e \)

Axioms/Logic Programs \( \Phi \) ::= \( \cdot \) | \( \kappa : H, \Phi \) | \( a : F, \Phi \)

Note: finite formulas only
Type System for S-resolution

Horn Formulas as Types

Proof evidence as Terms

Term $t ::= x \mid K(t_1,\ldots,t_n)$

Atomic Formula $A, B, C, D ::= P(t_1,\ldots,t_n)$

Formula $F ::= A \mid F \Rightarrow F' \mid \forall x. F$

Horn Formula/Horn Clause $H ::= \forall x. A_1,\ldots,A_n \Rightarrow B$

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

Axioms/Logic Programs $\Phi ::= \cdot \mid \kappa : H, \Phi \mid a : F, \Phi$

Note: finite formulas only
Simply Typed $\lambda$ calculus

\[
\frac{(\kappa : H) \in \Phi}{\Phi \vdash \kappa : H} \quad \text{Axiom} \quad \frac{(a : F) \in \Phi}{\Phi \vdash a : F} \quad \text{VAR} \quad \frac{\Phi \vdash e_1 : F_1 \Rightarrow F_2}{\Phi \vdash e_1 \ e_2 : F_2} \quad \text{APP}
\]

\[
\frac{\Phi \vdash e : \forall x. F}{\Phi \vdash \lambda \ a. (e_2 \ b) \ (e_1 \ a) : A, B \Rightarrow C} \quad \text{ABS}
\]

The cut rule that models Horn clause resolution is admissible (We can use rules ABS and APP to emulate CUT rule):

\[
\frac{\Phi \vdash e_1 : A \Rightarrow D \quad \Phi \vdash e_2 : B, D \Rightarrow C}{\Phi \vdash \lambda \ a. \lambda b. (e_2 \ b) \ (e_1 \ a) : A, B \Rightarrow C} \quad \text{CUT}
\]
Soundness of SLD and term-matching reductions

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

**Example**

\[
\begin{align*}
k_1 : & \text{nat}(0) \leftarrow \\
k_2 : & \text{nat}(s(x)) \leftarrow \text{nat}(x) \\
k_3 : & \text{list}(\text{nil}) \leftarrow \\
k_4 : & \text{list}(\text{cons}(x,y)) \leftarrow \text{nat}(x), \text{list}(y)
\end{align*}
\]

\[
\{\text{list}(\text{cons}(x,y))\} \rightsquigarrow \{\text{nat}(x), \text{list}(y)\} \rightsquigarrow \{\text{list}(y)\} \rightsquigarrow \emptyset
\]
yields a proof $(\lambda a. (k_4 \ a) \ k_1) \ k_3 : \text{list}(\text{cons}(0, \text{nil}))$

($\beta$-reducible to $k_4 k_3 k_1 : \text{list}(\text{cons}(0, \text{nil}))$).
Type Theoretic semantics, Fu and K, 2015-2016

Structural Resolution (rewriting fragment)

Horn Clause Logic

Coalgebraic Semantics

Horn clauses as Types
terms as proofs by rewriting reductions

Small Step semantics
What did it teach us?

- First Ever notion of productivity for LP
- Connection of LP to term-rewriting via rewriting reductions
- Arising natural type-theoretic semantics
- Coinduction in LP as an instance of the fixpoint rule
If we add a fixpoint rule

\[
\Phi, (\alpha : A \Rightarrow B) \vdash e : A \Rightarrow B \quad \text{HNF}(e)
\]

\[
\Phi \vdash \nu\alpha. e : A \Rightarrow B
\]  

(\text{NU})
If we add a fixpoint rule

\[
\Phi, (\alpha : A \Rightarrow B) \vdash e : A \Rightarrow B \quad \text{HNF}(e) \\
\Phi \vdash \nu \alpha.e : A \Rightarrow B
\]

\((\text{Nu})\)

We can (coinductively) prove \(p(a)\) from a program:

\[
\kappa : p(x) \Rightarrow p(x)
\]

\[
\vdash P; \alpha : p(a) \vdash \alpha : p(a) \\
\vdash P; \alpha : p(a) \vdash \kappa \alpha : p(a) \\
\vdash P \vdash \nu \alpha . \kappa \alpha : p(a)
\]

\((\text{CUT})\) \quad \text{\((\text{Nu})\)}
If we add a fixpoint rule

\[
\Phi, (\alpha : A \Rightarrow B) \vdash e : A \Rightarrow B \quad \text{HNF}(e)
\]
\[
\Phi \vdash \nu \alpha. e : A \Rightarrow B
\]

(\text{Nu})

We can (coinductively) prove \( p(a) \) from a program:

\[
\kappa : p(x) \Rightarrow p(x)
\]

:**

\[
\begin{array}{c}
P; \alpha : p(a) \vdash \alpha : p(a) \\
\hline
P; \alpha : p(a) \vdash \kappa \alpha : p(a) \\
\hline
P \vdash \nu \alpha. \kappa \alpha : p(a)
\end{array}
\]

CUT

\text{NU}

Note: still finite formulas only
Coinductive Type Theoretic semantics, Fu and K, 2015-2016

Nonterminating Rewriting reductions

Coalgebraic Semantics

Horn Clause Logic

Horn clauses as Types

terms as proofs by fixpoint rule

Small Step semantics
Applications in Type inference

- Rewriting reductions for Horn clauses are used for type class inference in Haskell (Fu&K&Schrijvers);
- We were able to extend the coinductive proof principle to extend state-of-the-art in Haskell type class inference with non-terminating resolution.
- Work on “Proof relevant resolution” or “Certified Resolution” (Fu & K, 2016)
Coinductive Type Theoretic semantics, Farka, K, Hammond, 2016

Nonterminating Rewriting reductions

Greater Herbrand Models

Horn Clause Logic

Horn clauses as Types

terms as proofs by fixpoint rule

Big Step semantics
Outline

Motivation

The “knowns”:
   Inductive and Coinductive Big Step Semantics for LP

The “known unknown”:
   Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
   Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work
Small-step categorical semantics for looping derivations?

- Nonterminating Rewriting reductions
- Horn Clause Logic
- Small Step semantics
- Categorical semantics?
- Horn clauses as Types
- terms as proofs by fixpoint rule

Rewriting reductions give only universal fragment of Horn clause logic
Infinite computations result in computation of finite terms (non-productive computations)
Small-step categorical semantics for looping derivations?

- Rewriting reductions give only universal fragment of Horn clause logic
- Infinite computations result in computation of finite terms (non-productive computations)
Type theoretic semantics for full fragment of S-resolution?

Nonterminating S-resolution reductions?

Horn Clause Logic

Note: existential fragment and infinite terms (productive computations)
Type theoretic semantics for full fragment of S-resolution?

Nonterminating S-resolution reductions

Horn Clause Logic

Horn clauses as Types?

Note: existential fragment and infinite terms (productive computations)
Small-step categorical semantics for looping S-resolution?

- Categorical semantics?
- Nonterminating S-resolution reductions
- Horn Clause Logic
- Horn clauses as Types?
- terms as proofs by fixpoint rule

Small Step semantics
Patterns and copatterns?

Horn Clause Logic

Pattern-Copattern resolution

Horn clauses as Types?

terms as proofs by fixpoint rule

Inductive-Coinductive semantics?

Big Step semantics

Horn Clause

→

↑

↖

↙

↑

↗

←

←

→
Your own game?

Horn Clause Logic

- some new proof system?
- some categorical semantics?
- new system for Horn clauses as Types

Horn clause logic is a researcher’s treasure chest.

Coinduction in Horn clause logic is still not well understood.

Impact of this research is high, due to continuing development of Horn clause logic in Automated Theorem Proving and Programming Language implementations.
Horn clause logic is a researcher’s treasure chest
Coinduction in Horn clause logic is still not well understood (!!!)
Impact of this research is high, due to continuing development of Horn clause logic in Automated Theorem Proving and Programming Language implementations
Thank you!