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

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

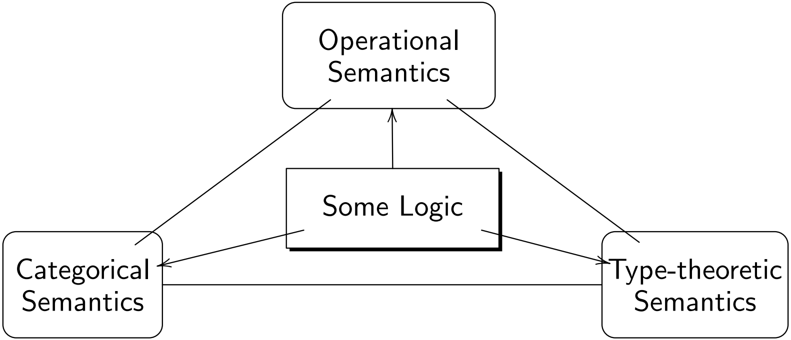
The "known unknown":
Small Step (Co)algebraic Semantics for LP

The "unknown unknown":
Structural resolution

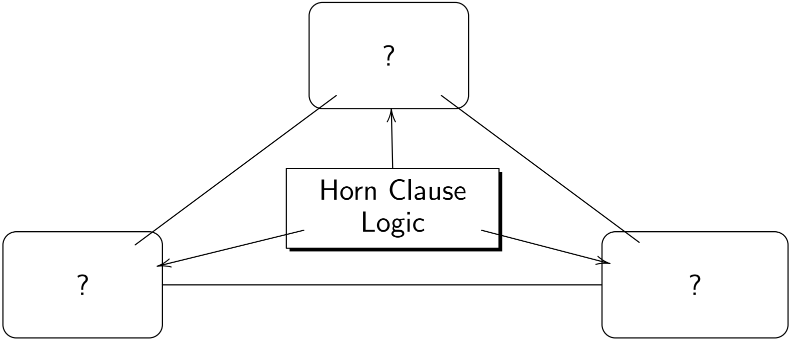
Structural resolution: impact and scientific value

New "unknowns": the future work

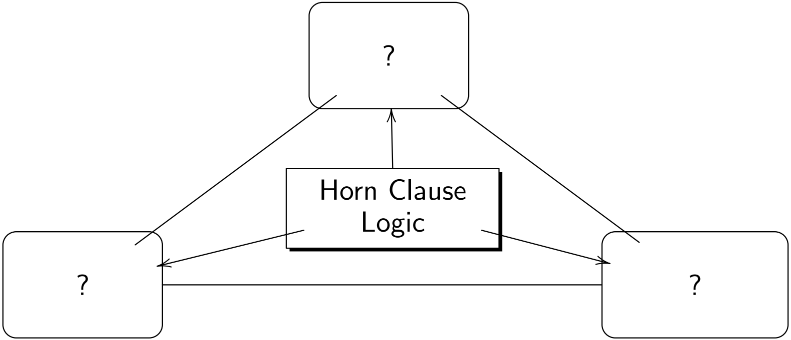
The Semantic triangle [Lambek and Scott]



This talk:

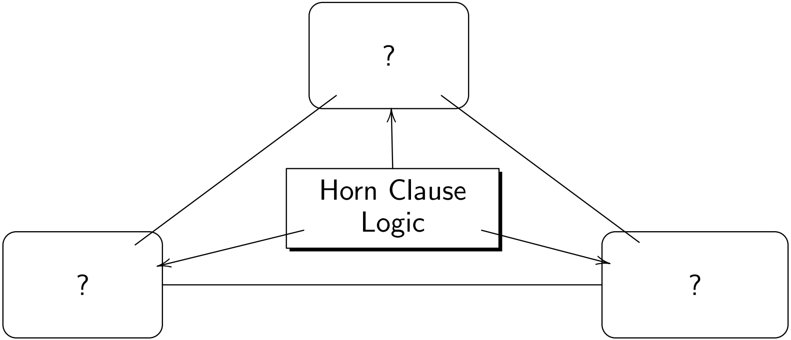


This talk:



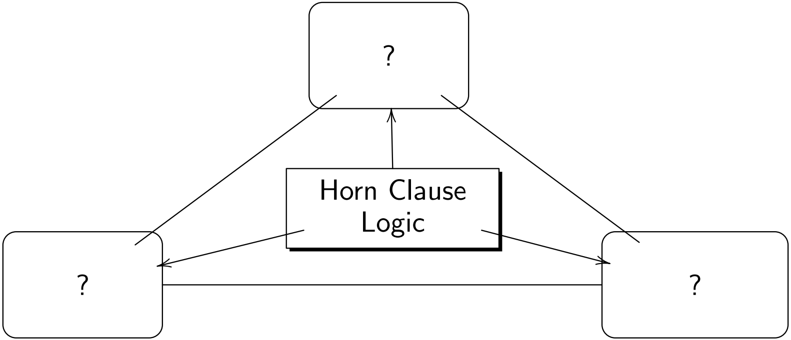
- Horn clause logic is a fragment of predicate logic, in which all formulae are written in clausal form.

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.

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

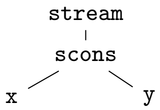
First-order signature Σ and terms, term-trees

- function symbols with arity;
- variables.

Example

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

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



Sets of terms:

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

Sets of terms:

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

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

Syntax of Horn-clause Logic

Horn Clauses

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

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

Universal quantification is assumed.

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

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

Example: lists of natural numbers

Example

$$
\begin{aligned}
\texttt{nat(0)} &\leftarrow \\
\texttt{nat(s(x))} &\leftarrow \texttt{nat(x)} \\
\texttt{list(nil)} &\leftarrow \\
\texttt{list(cons(x,y))} &\leftarrow \texttt{nat(x), list(y)}
\end{aligned}
$$

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 á 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.

Logic Programming...

SLD resolution = Unification + Search

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.$\texttt{nat(0)} \leftarrow$

2.$\texttt{nat(s(x))} \leftarrow \texttt{nat(x)}$

3.$\texttt{list(nil)} \leftarrow$

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

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

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

SLD-resolution

Example

1.nat(0) $\leftarrow$

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

3.list(nil) $\leftarrow$

4.list(cons(x,y)) $\leftarrow$

nat(x), list(y)

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

SLD-resolution

Example

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

`nat(x), list(y)`

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

SLD-resolution

Example

1.$\mathtt{nat(0)} \leftarrow$

2.$\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$

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

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

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

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

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

SLD-refutation = finite successful SLD-derivation.

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?

Neat connection to intuitionistic 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)

Relation of type classes to Horn Clause logic

```
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 = primtiveIntEq x y
```

Relation of type classes to Horn Clause logic

```
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 = primtiveIntEq x y
```

This translates into the following logic program:

$Eq\ (x), Eq\ (y) \Rightarrow Eq(x, y)$
$\Rightarrow Eq\ (Int)$

Relation of type classes to Horn Clause logic

```
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 = primtiveIntEq x y
```

This translates into the following logic program:

$Eq\ (x), Eq\ (y) \Rightarrow Eq(x, y)$
$\Rightarrow Eq\ (Int)$

Resolve the query ? $Eq\ (Int, Int)$.

- We have the following reduction by SLD-resolution:

$$\Phi \vdash Eq\ (Int, Int) \rightarrow Eq\ (Int), Eq\ (Int) \rightarrow Eq\ (Int) \rightarrow \emptyset$$

Outline

Motivation

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

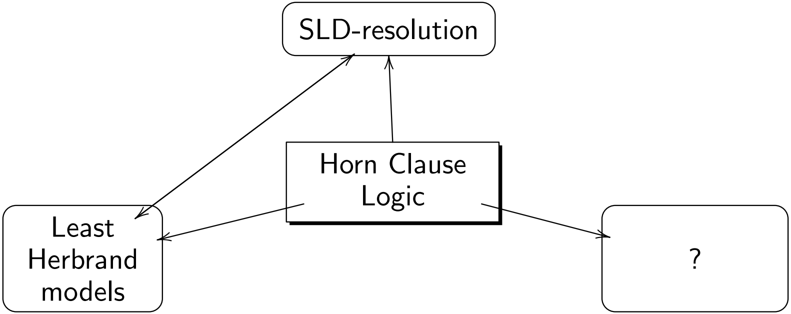
The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work

Inductive semantics of LP, 70s



Big Step semantics

Inductive Semantics of LP

Definition (Big step rule)

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

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

Definition

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

Example

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

Least Herbrand models as a least fixed point construction

Definition

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

Definition

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

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

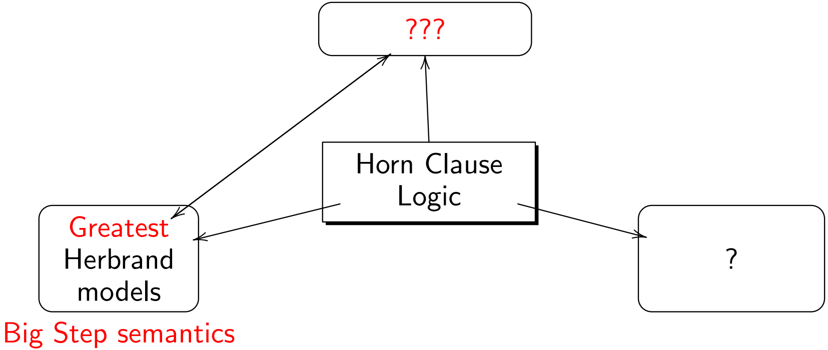
Soundness and Completeness

[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 θ, 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.*

CoInductive semantics



CoInductive Semantics of LP

Definition (Big step rule)

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

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

Definition

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

Example

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

Coinductive programs

Some programs have only one natural interpretation:

Example

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

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

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

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

Coinductive programs

Some programs have only one natural interpretation:

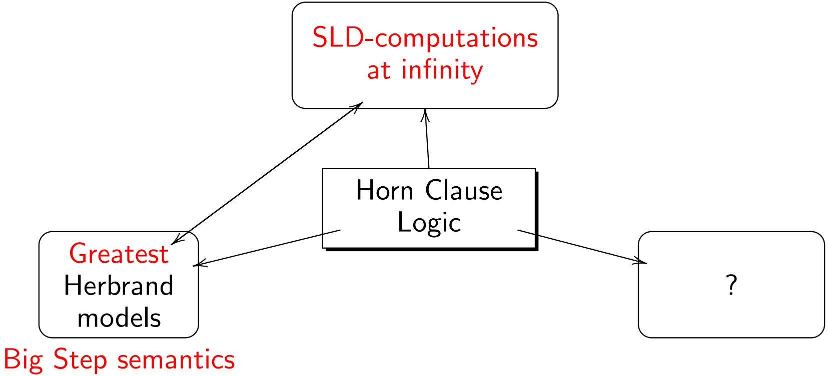
Example

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

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

... now the coinductive semantics does not match with the actual SLD-refutations...

CoInductive semantics, the 80s



" 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 $G_0 = (? \leftarrow t'), G_1, G_2, \ldots G_k \ldots$ with mgus $\theta_1, \theta_2, \ldots \theta_k \ldots$ such that $d(t, \theta_k \ldots \theta_1(t')) \rightarrow 0$ as $k \rightarrow \infty$.

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

Example

Program **Stream**:

Example

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

`bit(x), stream(y)`

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

So far, we have computed that $x \mapsto 0$, $y \mapsto \mathtt{scons(x_1,y_1)}$, and $x_1 \mapsto 0$. At infinity, the term $\mathtt{scons(0,scons(0,\ldots))}$ is computed.

Computations at infinity as globally productive computations

Program definition	For query $? \leftarrow \mathtt{p(x)}$, computes the answer:	Models
$\mathtt{p(x)} \leftarrow \mathtt{p(x)}$	*id*	$\{p(a), p(f(a), p(f(f(a)), \ldots p(f^{\omega}))\}$
$\mathtt{p(x)} \leftarrow \mathtt{p(f(x))}$	*id*	$\{p(a), p(f(a), p(f(f(a)), \ldots p(f^{\omega}))\}$
$\mathtt{p(f(x))} \leftarrow \mathtt{p(x)}$	$\{\mathtt{x} \mapsto \mathtt{f(f\ldots)}\}$	$\{p(f^{\omega})\}$

Computations at infinity are sound

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

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

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

Outline

Motivation

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

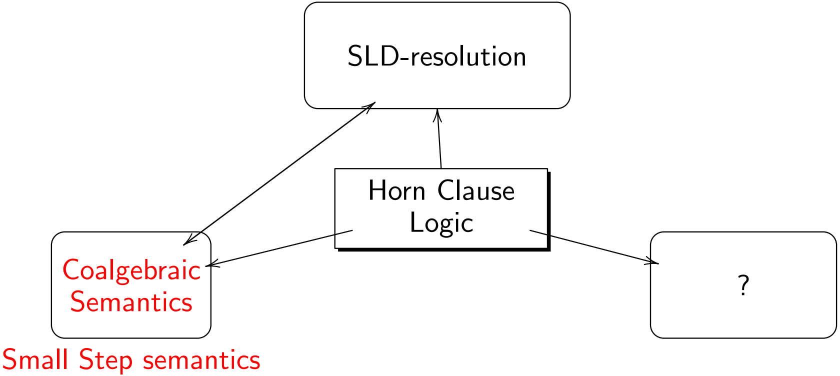
The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work

Coalgebraic operational semantics, K&Power, 2010-14



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 $c : U \to F(U)$.

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 as Comonads

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 $C(H)$.

Fibrational Coalgebraic Semantics of CoALP in 3 ideas

Idea 2: Derivations as Comonads

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

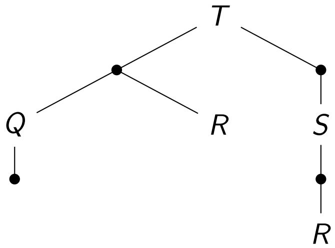
$$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

This gives a "tree-like" structure: $\&V$-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

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* $\mathscr{L}_\Sigma$ 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

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* $\mathscr{L}_\Sigma$ to model the terms over Σ
  - $\texttt{ob}(\mathscr{L}_\Sigma)$ is $\mathbb{N}$.
  - For each $n \in Nat$, let $x_1, \ldots, x_n$ be a specified list of distinct variables.
  - $\texttt{ob}(\mathscr{L}_\Sigma)(n, m)$ 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 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 $I : \mathbb{N}^{op} \to L$.

- 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 vars.
- model a program P by the $[\mathscr{L}_\Sigma^{op}, P_f P_f]$-coalgebra $p : At \longrightarrow P_f P_f At$ on the category $[\mathscr{L}_\Sigma^{op}, Set]$.

Final remarks

Actually, some modifications are needed:

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

Then $p : At \longrightarrow P_c P_f At$ gives a $Lax(\mathscr{L}_\Sigma^{op}, P_c P_f)$-coalgebra structure on At.

Examples

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

Examples

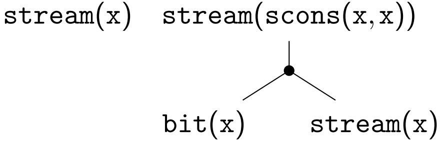
Program **Stream**: "fibers" given by term arities. Take the fiber of 1. $\& V$-trees:

```
stream(x)
```

Examples

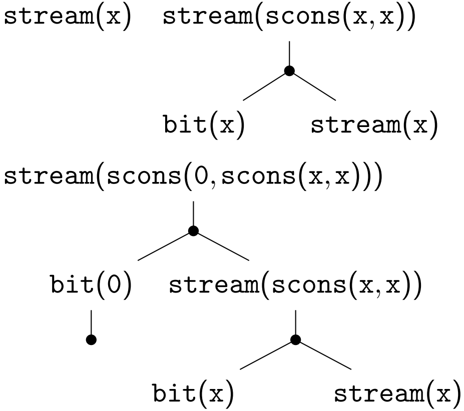
Program **Stream**: "fibers" given by term arities. Take the fiber of

1. &V-trees:



Examples

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

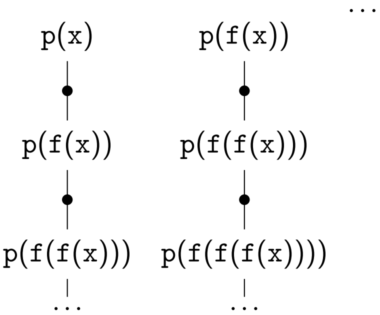


Examples

Program $p(x) \leftarrow p(f(x))$: “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:

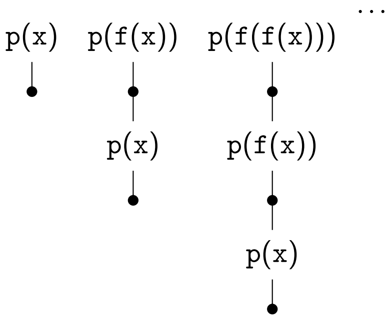


Examples

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

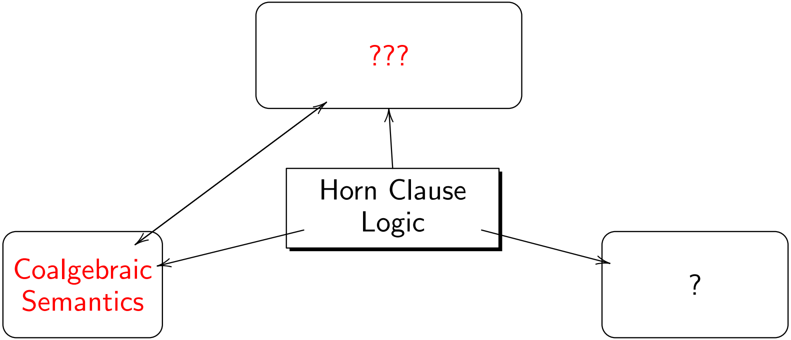
Examples

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



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

Coalgebraic operational semantics, K & Power, 2010-14



Small Step semantics

Outline

Motivation

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

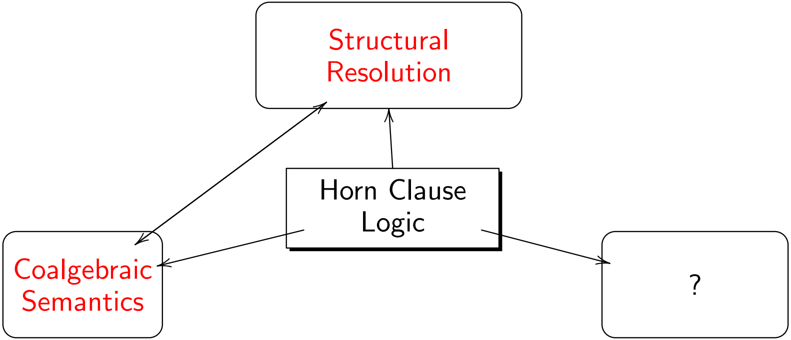
The “known unknown”:
Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
Structural resolution

Structural resolution: impact and scientific value

New “unknowns”: the future work

Coalgebraic operational semantics, K&Power, 2010-2014



Small Step semantics

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

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

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

Example

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

1. SLD-resolution reduction:
   $[stream(x)] \rightsquigarrow [bit(x'), stream(y)] \rightsquigarrow [stream(y)] \rightsquigarrow$
   $[bit(x''), stream(y')] \rightsquigarrow \ldots$
2. Term-matching reduction: $[stream(x)] \not\rightarrow$
3. S-resolution reduction:
   $[stream(x)] \hookrightarrow^1 [stream(scons(x', y))] \rightarrow^\mu$
   $[bit(x'), stream(y)] \hookrightarrow^1 [bit(0), stream(y)] \rightarrow^\mu [stream(y)] \hookrightarrow^1$
   $[stream(scons(x'', y'))] \rightarrow^\mu [bit(x''), stream(y')] \ldots$

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

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 $? \leftarrow \mathtt{p(x)}$, computes the answer:
$\mathtt{p(x)} \leftarrow \mathtt{p(x)}$	*id*
$\mathtt{p(x)} \leftarrow \mathtt{p(f(x))}$	*id*
$\mathtt{p(f(x))} \leftarrow \mathtt{p(x)}$	$\{\mathtt{x} \mapsto \mathtt{f(f \ldots)}\}$

Implementation of the productivity checker is available at `https://github.com/coalp`

What did it teach us?

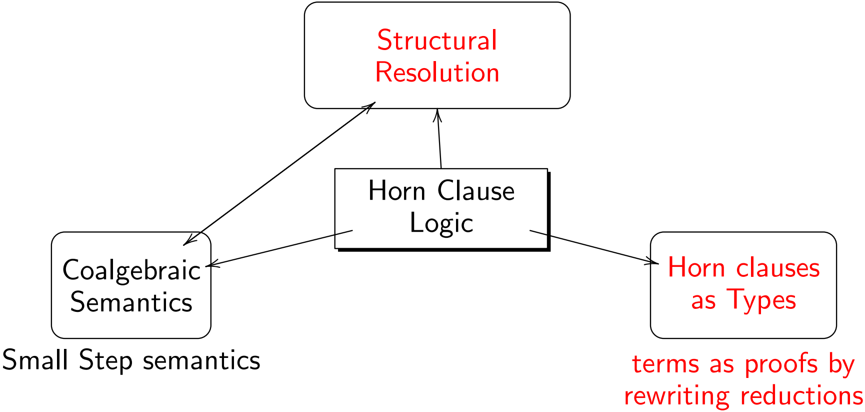
- First Ever notion of productivity for LP
- Connection of LP to term-rewriting
- ...

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



Type System for S-resolution

Horn Formulas as Types

Proof evidence as Terms

Term $t \ ::= \ x \mid K(t_1,...,t_n)$
Atomic Formula $A,B,C,D \ ::= \ P(t_1,...,t_n)$
Formula $F \ ::= A \mid F \Rightarrow F' \mid \forall x.F$
Horn Formula/Horn Clause $H \ ::= \forall \underline{x}.A_1,...,A_n \Rightarrow B$
Proof Evidence $p,e \ ::= \ \kappa \mid a \mid e \ e' \mid \lambda a.e$
Axioms/Logic Programs $\Phi \ ::= \cdot \mid \kappa : H, \Phi \mid a : F, \Phi$

Type System for S-resolution

Horn Formulas as Types

Proof evidence as Terms

Term t $::=$ $x \mid K(t_1,...,t_n)$
Atomic Formula A, B, C, D $::=$ $P(t_1,...,t_n)$
Formula F $::= A \mid F \Rightarrow F' \mid \forall x.F$
Horn Formula/Horn Clause H $::= \forall \underline{x}.A_1,...,A_n \Rightarrow B$
Proof Evidence p, e $::=$ $\kappa \mid a \mid e\ e' \mid \lambda a.e$
Axioms/Logic Programs Φ $::= \cdot \mid \kappa : H, \Phi \mid a : F, \Phi$

Note: finite formulas only

Simply Typed λ calculus

$$\frac{(\kappa : H) \in \Phi}{\Phi \vdash \kappa : H}\ \textsc{Axiom} \qquad \frac{(a : F) \in \Phi}{\Phi \vdash a : F}\ \textsc{Var} \qquad \frac{\Phi \vdash e_1 : F_1 \Rightarrow F_2 \quad \Phi \vdash e_2 : F_1}{\Phi \vdash e_1\ e_2 : F_2}\ \textsc{App}$$

$$\frac{\Phi \vdash e : \forall x.F}{\Phi \vdash e : [t/x]F}\ \textsc{Inst} \qquad \frac{\Phi \vdash e : F}{\Phi \vdash e : \forall x.F}\ \textsc{Gen} \qquad \frac{\Phi, a : F_1 \vdash e : F_2}{\Phi \vdash \lambda a.e : F_1 \Rightarrow F_2}\ \textsc{Abs}$$

The cut rule that models Horn clause resolution is admissible (We can use rules $\textsc{Abs}$ and $\textsc{App}$ to emulate $\textsc{Cut}$ rule):

$$\frac{\Phi \vdash e_1 : \underline{A} \Rightarrow D \quad \Phi \vdash e_2 : \underline{B}, D \Rightarrow C}{\Phi \vdash \lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C}\ \textsc{Cut}$$

Soundness of SLD and term-matching reductions

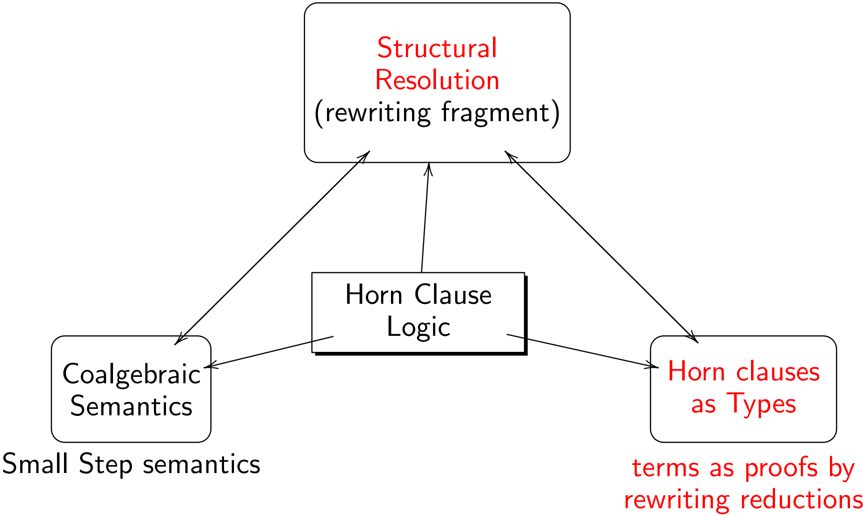
- If $P \vdash \{A\} \rightsquigarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$.
- If $P \vdash \{A\} \rightarrow^n \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$.

Example

$\kappa_1 : \texttt{nat(0)} \leftarrow$
$\kappa_2 : \texttt{nat(s(x))} \leftarrow \texttt{nat(x)}$
$\kappa_3 : \texttt{list(nil)} \leftarrow$
$\kappa_4 : \texttt{list(cons(x,y))} \leftarrow \texttt{nat(x)}, \texttt{list(y)}$

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

Type Theoretic semantics, Fu and K, 2015-2016



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

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash e : \underline{A} \Rightarrow B \quad HNF(e)}{\Phi \vdash \nu\alpha.e : \underline{A} \Rightarrow B} \ (\textsc{Nu})$$

If we add a fixpoint rule

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash e : \underline{A} \Rightarrow B \quad HNF(e)}{\Phi \vdash \nu\alpha.e : \underline{A} \Rightarrow B}\ (\textsc{Nu})$$

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

$$\kappa : p(x) \Rightarrow p(x)$$

:

$$\dfrac{\dfrac{\overline{P; \alpha : p(a) \vdash \alpha : p(a)}}{P; \alpha : p(a) \vdash \kappa\alpha : p(a)}\ \text{CUT}}{P \vdash \nu\alpha.\kappa\alpha : p(a)}\ \text{NU}$$

If we add a fixpoint rule

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash e : \underline{A} \Rightarrow B \quad HNF(e)}{\Phi \vdash \nu\alpha.e : \underline{A} \Rightarrow B} \text{(Nu)}$$

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

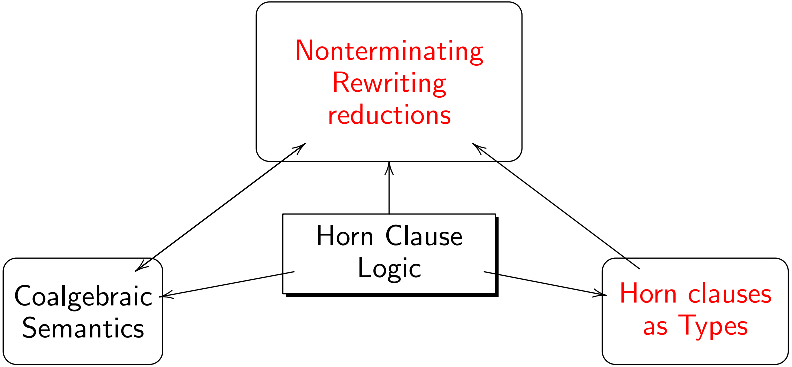
$$\kappa : p(x) \Rightarrow p(x)$$

:

$$\dfrac{\dfrac{\overline{P; \alpha : p(a) \vdash \alpha : p(a)}}{P; \alpha : p(a) \vdash \kappa\alpha : p(a)} \text{CUT}}{P \vdash \nu\alpha.\kappa\alpha : p(a)} \text{NU}$$

Note: still finite formulas only

Coinductive Type Theoretic semantics, Fu and K, 2015-2016



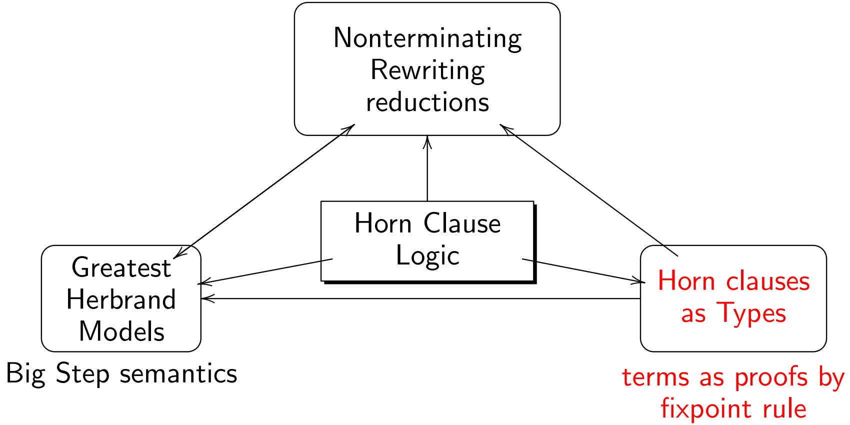
Small Step semantics

terms as proofs by fixpoint rule

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



Outline

Motivation

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

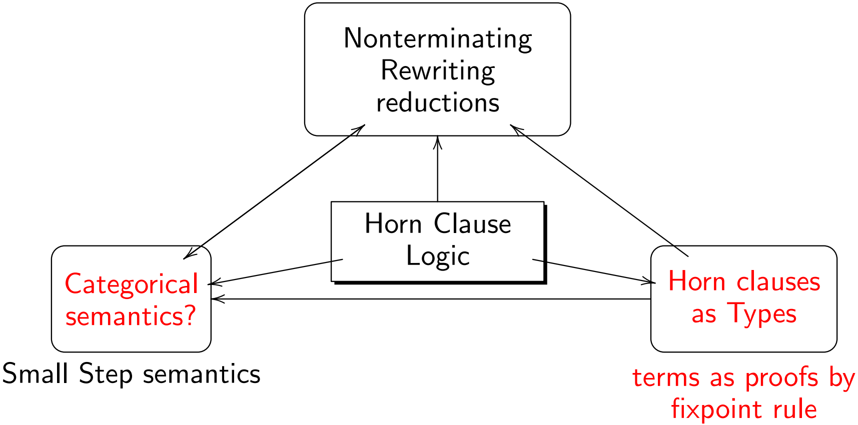
The “known unknown”:
  Small Step (Co)algebraic Semantics for LP

The “unknown unknown”:
  Structural resolution

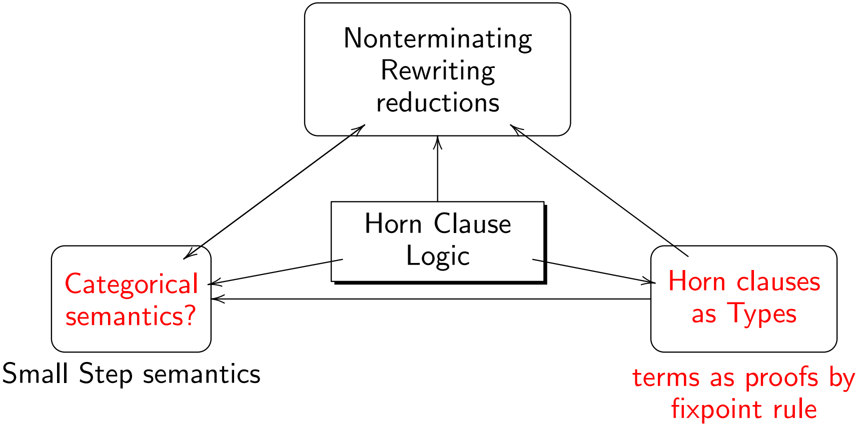
Structural resolution: impact and scientific value

New “unknowns”: the future work

Small-step categorical semantics for looping derivations?

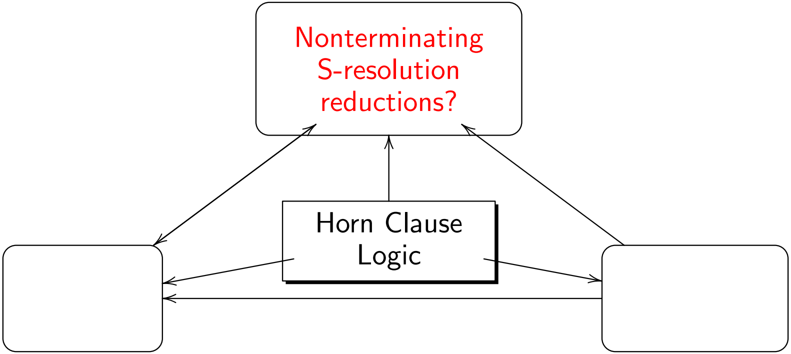


Small-step categorical semantics for looping derivations?



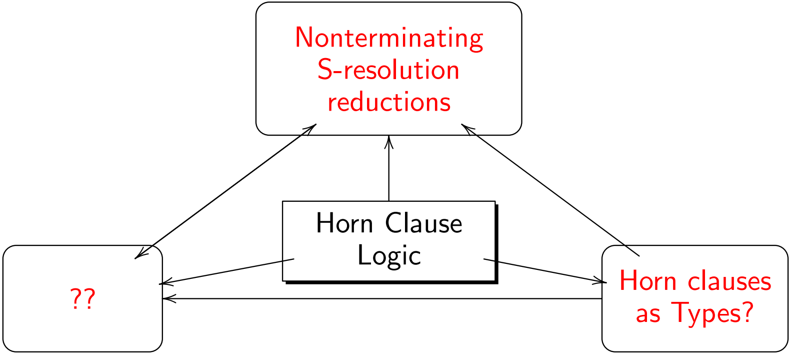
- Rewriting reductions give only universal fragment of Horn clause logic
- Infinite computations result in computation of finite terms (non-productive computations)

Type theoretic semantics for full fragment of S-resolution?



Note: existential fragment and infinite terms (productive computations)

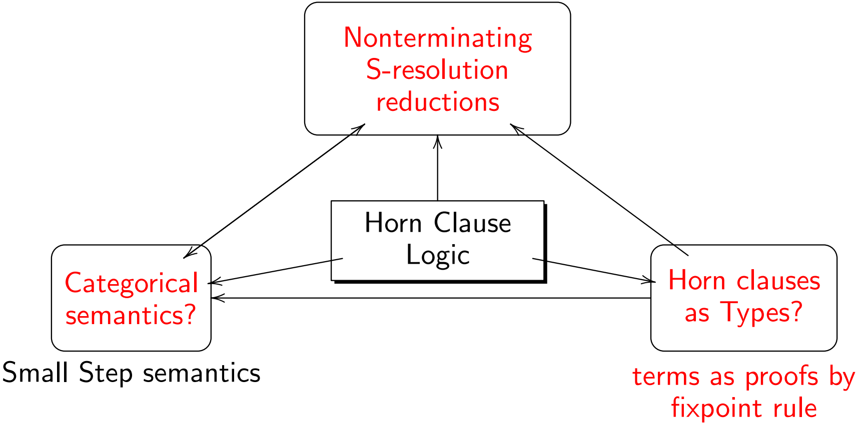
Type theoretic semantics for full fragment of S-resolution?



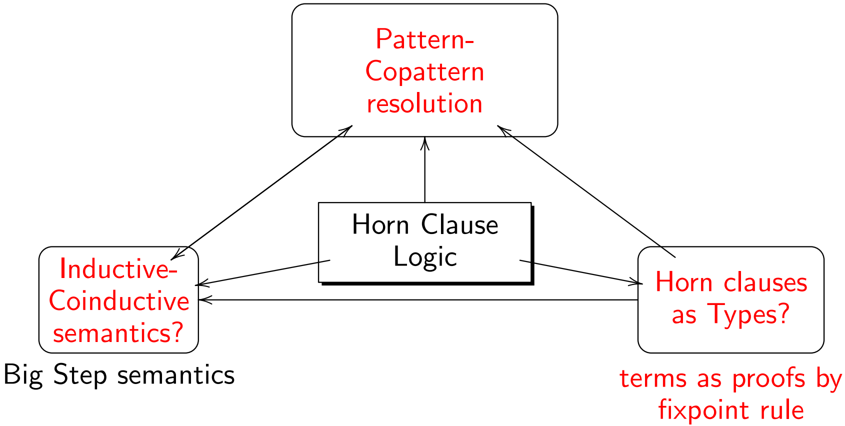
terms as proofs by fixpoint rule

Note: existential fragment and infinite terms (productive computations)

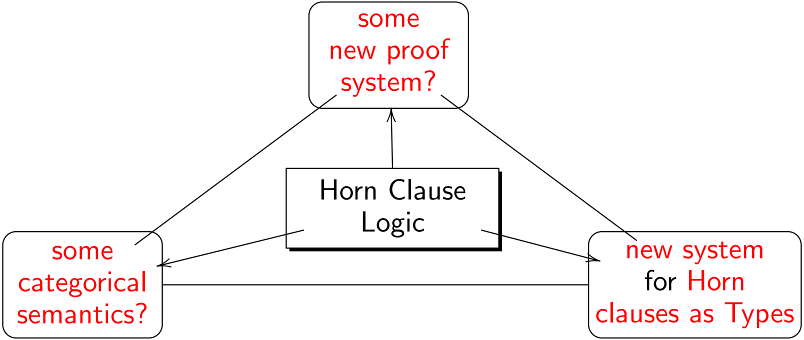
Small-step categorical semantics for looping S-resolution?



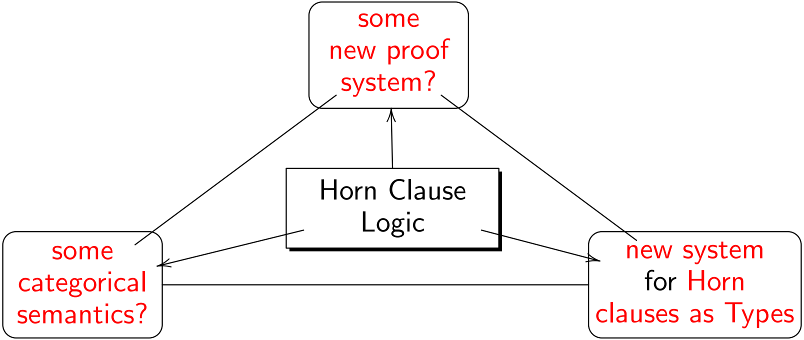
Patterns and copatterns?



Your own game?



Your own game?



- 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!