Structural Automated Proving

Katya Komendantskaya

School of Computing, University of Dundee, UK

11 December 2014

About myself

- 1998 – 2003 Undergraduate degree in Logic, Moscow State University; (1st class honours, gold medal for excellency).

About myself

- 1998 – 2003 Undergraduate degree in Logic, Moscow State University; (1st class honours, gold medal for excellency).
- 2004 – 2007 PhD in the UCC, Ireland. (The University (and department!) of the famous George Boole)

About myself

- 1998 – 2003 Undergraduate degree in Logic, Moscow State University; (1st class honours, gold medal for excellency).
- 2004 – 2007 PhD in the UCC, Ireland. (The University (and department!) of the famous George Boole)
- 2007 – 2008 First postdoc project with Yves Bertot in INRIA, France - on guardedness of corecursive functions in Coq.

About myself

- 1998 – 2003 Undergraduate degree in Logic, Moscow State University; (1st class honours, gold medal for excellency).
- 2004 – 2007 PhD in the UCC, Ireland. (The University (and department!) of the famous George Boole)
- 2007 – 2008 First postdoc project with Yves Bertot in INRIA, France - on guardedness of corecursive functions in Coq.
- 2008 – 2011 EPSRC TCS research fellowship first in St Andrews, later transferred to the School of Computing in Dundee.

About myself

- 1998 – 2003 Undergraduate degree in Logic, Moscow State University; (1st class honours, gold medal for excellency).
- 2004 – 2007 PhD in the UCC, Ireland. (The University (and department!) of the famous George Boole)
- 2007 – 2008 First postdoc project with Yves Bertot in INRIA, France - on guardedness of corecursive functions in Coq.
- 2008 – 2011 EPSRC TCS research fellowship first in St Andrews, later transferred to the School of Computing in Dundee.
- 2010 – now – Senior Lecturer, School of Computing, University of Dundee. We are growing a new "LiCS" group in Dundee...

Outline

Big Picture: Proofs and structures

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

- Structural Recursion in ITPs: Types give Structure
- Structural Recursion without structure in LP?

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

- Structural Recursion in ITPs: Types give Structure
- Structural Recursion without structure in LP?

Solution: New Structural Resolution for ATP

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

- Structural Recursion in ITPs: Types give Structure
- Structural Recursion without structure in LP?

Solution: New Structural Resolution for ATP

Solution's Effect: Universal Productivity for Structural Resolution, for free

Automated Reasoning and Automated Theorem Proving

... from Hilbert to our times:

$$\Gamma \vdash A$$

Automated Reasoning and Automated Theorem Proving

... from Hilbert to our times:

$$\Gamma \vdash A$$

$\uparrow$

Theory,

Program,

Set of axioms...

Automated Reasoning and Automated Theorem Proving

... from Hilbert to our times:

$$\Gamma \vdash A$$

$$\begin{array}{ccc} \uparrow & & \uparrow \\ \text{Theory,} & & \text{Theorem} \\ \text{Program,} & & \text{Derivation Goal} \\ \text{Set of axioms...} & & \text{Proposition} \end{array}$$

Automated Reasoning and Automated Theorem Proving

... research directions:

$$\Gamma \vdash A$$

Automated Reasoning and Automated Theorem Proving

... research directions:

$$\Gamma \vdash A$$

(i) Syntax/language: First-order, Higher-order, Typeful, untyped?

Automated Reasoning and Automated Theorem Proving

... research directions:

$$\Gamma \vdash A$$

(i) Syntax/language: First-order, Higher-order, Typeful, untyped?

(ii) Derivation/Inference methods: resolution, automated proof-search, interactive proofs, ...

Automated Reasoning and Automated Theorem Proving

... research directions:

$$\Gamma \vdash A$$

(i) Syntax/language: First-order, Higher-order, Typeful, untyped?
(ii) Derivation/Inference methods: resolution, automated proof-search, interactive proofs, ...

My focus is on (ii).

Proof inference methods

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

more or less Structural...

$$\Gamma \quad \vdash \quad A$$

Proof inference methods

Constructive Type theory

is more Structural...

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

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

Proof inference methods

Constructive Type theory

is more Structural...

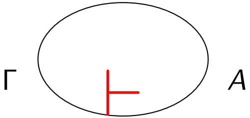
$$\Gamma \vdash p : A$$

To prove $\Gamma \vdash A$, we need to show that type A has inhabitant p; namely, we have to conSTRUCT it.
And hence the 3 related problems:
$\Gamma \vdash p : A?$ – Type Checking Problem;
$\Gamma \vdash p :?$ – Type Inference Problem;
$\Gamma \vdash ? : A$ – Type Inhabitation Problem.
Usually, the latter is not decidable, hence we need Interactive Theorem Provers (ITPs).

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.

Two important remarks:

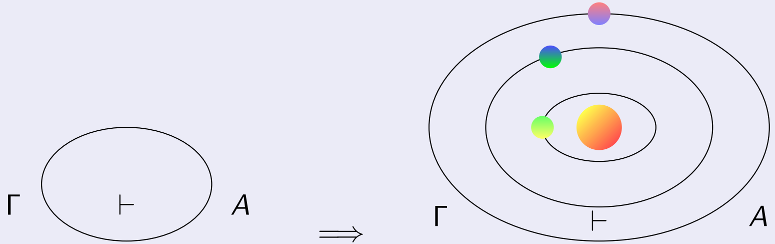
- ITPs use ATPs for type inference and type checking, so you are not "safe" from the Unstructural even if you work only with ITPs;
- it is often assumed that if we cannot impose types, we cannot have structural approach to proofs in ATPs.

Two important remarks:

- ITPs use ATPs for type inference and type checking, so you are not "safe" from the Unstructural even if you work only with ITPs;
- it is often assumed that if we cannot impose types, we cannot have structural approach to proofs in ATPs.

Is this true? May be we CAN discover new structural theory?

Analogy: discovery of atomic structures of particles



In search of a new theory

If a fundamental theory of Structures in ATPs is missing, we should have noticed by now some irresolvable problems in our old theory... Have we?

In search of a new theory

If a fundamental theory of Structures in ATPs is missing, we should have noticed by now some irresolvable problems in our old theory... Have we?

1. Recursion behaves badly with resolution, even structural recursion (which can be handled easily in ITPs)!

In search of a new theory

If a fundamental theory of Structures in ATPs is missing, we should have noticed by now some irresolvable problems in our old theory... Have we?

1. Recursion behaves badly with resolution, even structural recursion (which can be handled easily in ITPs)!
2. Coinduction is clunky in ATPs, if at all possible (again, problem "solved" in ITPs)!

In search of a new theory

If a fundamental theory of Structures in ATPs is missing, we should have noticed by now some irresolvable problems in our old theory... Have we?

1. Recursion behaves badly with resolution, even structural recursion (which can be handled easily in ITPs)!
2. Coinduction is clunky in ATPs, if at all possible (again, problem “solved” in ITPs)!
3. Parallelism is very restricted, due to variable dependencies (much worse state than in e.g. parallel Haskell).

In search of a new theory

If a fundamental theory of Structures in ATPs is missing, we should have noticed by now some irresolvable problems in our old theory... Have we?

1. Recursion behaves badly with resolution, even structural recursion (which can be handled easily in ITPs)!
2. Coinduction is clunky in ATPs, if at all possible (again, problem "solved" in ITPs)!
3. Parallelism is very restricted, due to variable dependencies (much worse state than in e.g. parallel Haskell).

In the rest of this talk, I'll focus only on (1) and in (less detail) – (2).
So, lets look under the bonnet...

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

- Structural Recursion in ITPs: Types give Structure
- Structural Recursion without structure in LP?

Solution: New Structural Resolution for ATP

Solution's Effect: Universal Productivity for Structural Resolution, for free

Inductive Types and Recursive Functions

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

Recursive functions have arguments of inductive types.

```
Fixpoint length (A:Type) (l: list A) : nat :=
match l with
 | nil => O
 | cons _ l' => S (length l')
end.
```

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions as types; proofs as programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking, we need to reduce expressions to normal form.

Universal Termination

A recursive function is terminating, if it terminates for all possible (legal) inputs.

Semi-deciding universal termination: Structural recursion

As function input is of inductive type, we can use constructors to reason about termination. Checking for structural recursion is one elegant way to decide termination.

```
Fixpoint length (A:Type) (l: list A) : nat :=
```

```
match l with
 | nil => O
 | cons _ l' => S (length l')
end.
```

```
Fixpoint plus (n m:nat) : nat :=
```

```
match n with
 | O => m
 | S p => S (p + m)
end.
```

Semi-decision: If an inductive function is structurally recursive, it terminates for any (legal) input.

Coinductive Types and Corecursive Functions

```
CoInductive stream (A:Set) : Set :=
SCons: A -> stream A -> stream A.
```

Corecursive functions have outputs of coinductive types. (Type of input arguments is not important.)

```
CoFixpoint repeat (a: A): stream A :=
SCons a (repeat a).
```

Productivity

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

Productivity

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.
The element of the stream at position n can be found by good old `nth`:

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

A given stream `s` is productive if the computation of `nth n s` is guaranteed to terminate, whatever the value of `n` is.

Universal Productivity

We call a function *productive*, if, for any given input, it outputs a productive value.

Semi-deciding Universal Productivity: Guardedness

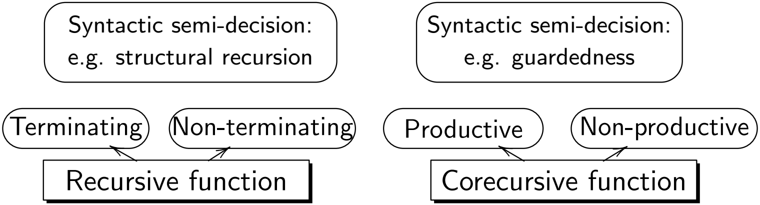
Guardedness checks:

- whether each corecursive call is made under at least one type (co)constructor, and
- if a recursive call is under a (co)constructor, then it does not appear as an argument of any function.

```
CoFixpoint repeat (a:  A): str A :=
SCons a (repeat a).
```

Semi-decision: If a coinductive function is guarded, it is productive.

Elegant picture:



Note:

- The role of inductive and coinductive types in definition of recursive and corecursive functions
- The role of constructors and (co)-pattern matching

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

- Structural Recursion in ITPs: Types give Structure
- Structural Recursion without structure in LP?

Solution: New Structural Resolution for ATP

Solution's Effect: Universal Productivity for Structural Resolution, for free

Logic Programming...

SLD resolution = Unification + Search

SLD-resolution + unification in LP derivations.

Program **NatList**:

Example

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

`nat(x), list(y)`

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

SLD-resolution + unification in LP derivations.

Example

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

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

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

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

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

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

SLD-resolution (+ unification) in LP derivations.

Example

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

`nat(x), list(y)`

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

$$|$$

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

$$|$$

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

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)`

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

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)` $\leftarrow$

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

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

NatList2:

Example

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

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

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

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

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

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

Alas, unlike ITP/FPs, the "function definition" is not localised, any clause can recursively call any other, in any order.

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

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

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

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

Example

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

2.$\mathtt{bit(1)} \leftarrow$

3.$\mathtt{stream(scons(x,y))} \leftarrow$

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

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...

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

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

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

Program **Stream**:

Example

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

   `bit(x), stream(y)`

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

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

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

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

Structurally recursive addition:

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

If the third argument in `add` is "thought of" as the "output", and the other arguments – as "inputs", then, giving queries with variable-free "inputs" will guarantee termination by structural recursion on the first argument. But otherwise, there will be non-terminating derivations for queries to `add`.

There is a range of solutions:

- use "modes" to distinguish termination cases for annotated input and output arguments.
- impose measures of reduction on terms, in order to formulate termination conditions in derivations.

As a consequence, in LP, it is common to talk about existential termination (only for some derivations, for queries of certain kinds, or satisfying certain conditions/measures), not programs in general.

Problems...

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

What does it mean if your program does not terminate?

Problems...

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

What does it mean if your program does not terminate?

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

Problems...

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

What does it mean if your program does not terminate?

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

Problems...

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

What does it mean if your program does not terminate?

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

Problems...

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

What does it mean if your program does not terminate?

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

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

Problems...

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

What does it mean if your program does not terminate?

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

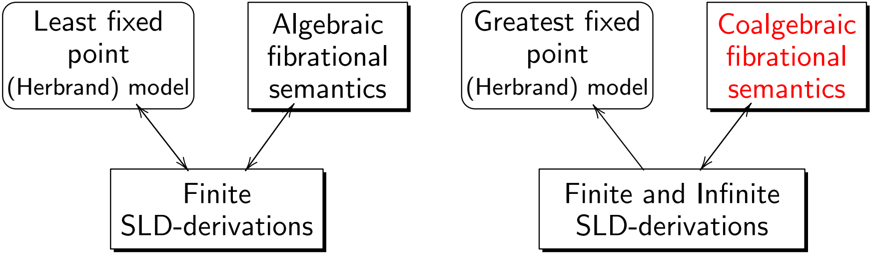
$$badstream(scons(x, y)) \leftarrow badstream(scons(x, y))$$

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

Outline

Coalgebraic Logic programming...

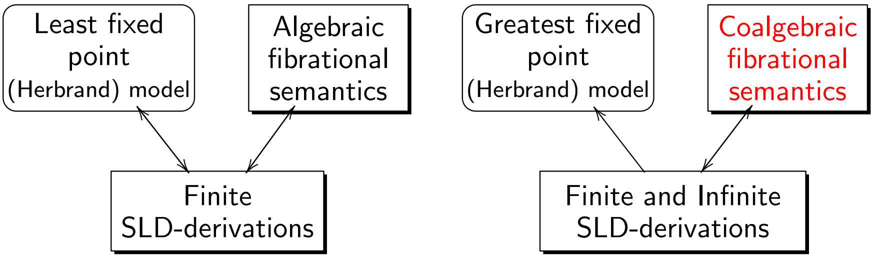
[AMAST2010, CSL2011, JLC2014], with John Power et al.



Noticed: There is more structure in this fibrational coalgebraic semantics than in SLD-resolution!

Coalgebraic Logic programming...

[AMAST2010, CSL2011, JLC2014], with John Power et al.



Noticed: There is more structure in this fibrational coalgebraic semantics than in SLD-resolution!

After a few years of study,

we propose to completely reform resolution in ATPs!

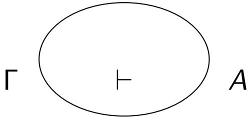
Defining structural resolution from first principles...

We now work on a new Three-tier Calculus of LP, providing a new structural approach to automated proofs.

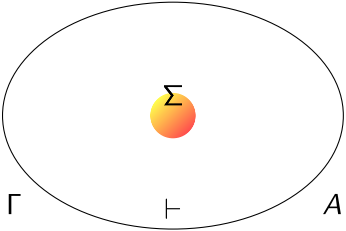
Defining structural resolution from first principles...

We now work on a new Three-tier Calculus of LP, providing a new structural approach to automated proofs.

At the start, there is a first-order signature Σ...



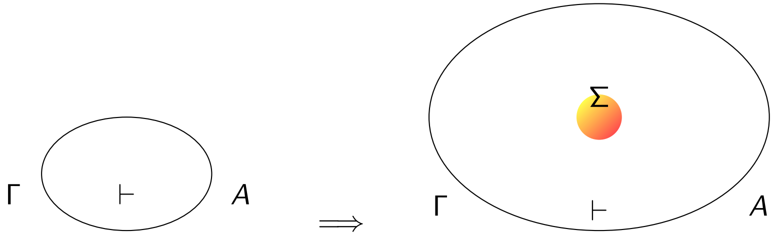
$\Longrightarrow$



Defining structural resolution from first principles...

We now work on a new Three-tier Calculus of LP, providing a new structural approach to automated proofs.

At the start, there is a first-order signature Σ...



Example

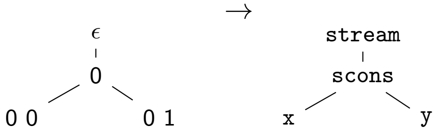
1.$\texttt{bit(0)} \leftarrow$

2.$\texttt{bit(1)} \leftarrow$

3.$\texttt{stream(scons(x,y))} \leftarrow \texttt{bit(x), stream(y)}$

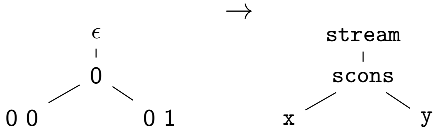
Tier-1: Term-trees, given Σ:

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



Tier-1: Term-trees, given Σ:

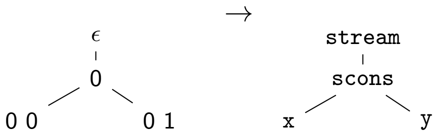
Let $\mathbb{N}^*$ denote the set of all finite words (sequences) over $\mathbb{N}$.
A set $L \subseteq \mathbb{N}^*$ is a *(finitely branching) tree language*, subject to prefix closedness.
A term tree is a map $L \to \Sigma$, satisfying term arity restrictions.



Arity: given by Σ
Operation: – first-order substitution
Calculus: – first-order unification and term-matching.

Tier-1: Term-trees, given Σ:

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



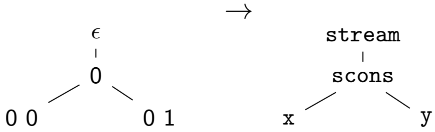
Arity: given by Σ
Operation: – first-order substitution
Calculus: – first-order unification and term-matching.

Notation:

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

Tier-1: Term-trees, given Σ:

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



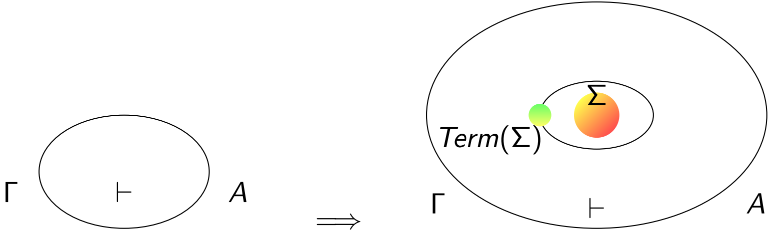
Arity: given by Σ
Operation: – first-order substitution
Calculus: – first-order unification and term-matching.

Notation:

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

Constructing the structural resolution from first principles...

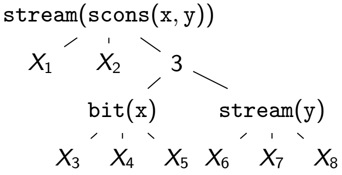
- At the start, there is a first-order signature Σ.
- First tier of Terms builds on it...



Tier-2: Coinductive trees

A coinductive tree is a map $L \to \textbf{Term}(\Sigma)$, subject to conditions.

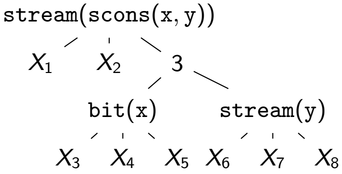
Note the size!



Tier-2: Coinductive trees

A coinductive tree is a map $L \to$ **Term(Σ)**, subject to conditions.

Note the size!

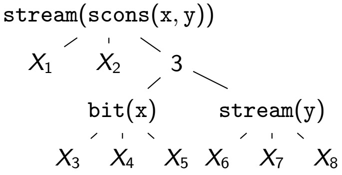


Arity: Number of clauses in the program and number of terms in clauses (modulo term-matching)
Operation: – coinductive tree substitution via mgu with clauses
Calculus: – coinductive derivations.

Tier-2: Coinductive trees

A coinductive tree is a map $L \to \mathbf{Term}(\Sigma)$, subject to conditions.

Note the size!



Arity: Number of clauses in the program and number of terms in clauses (modulo term-matching)
Operation: – coinductive tree substitution via mgu with clauses
Calculus: – coinductive derivations.

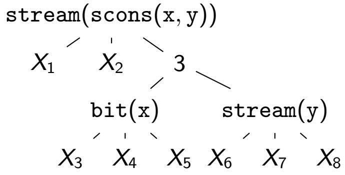
Notation:

$\mathbf{CTree}(\Sigma)$	all *finite* coinductive trees over $\mathbf{Term}(\Sigma)$
$\mathbf{CTree}^{\infty}(\Sigma)$	all *infinite* coinductive trees over $\mathbf{Term}(\Sigma)$
$\mathbf{CTree}^{\omega}(\Sigma)$	all *finite and infinite* coinductive trees over $\mathbf{Term}(\Sigma)$

Tier-2: Coinductive trees

A coinductive tree is a map $L \to$ **Term**(Σ), subject to conditions.

Note the size!



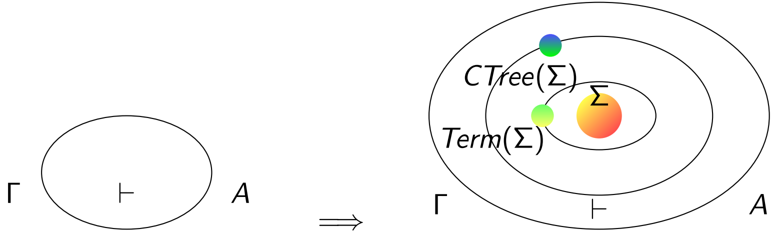
Arity: Number of clauses in the program and number of terms in clauses (modulo term-matching)
Operation: – coinductive tree substitution via mgu with clauses
Calculus: – coinductive derivations.

Notation:

CTree(Σ)	all *finite* coinductive trees over **Term**(Σ)
CTree$^{\infty}(\Sigma)$	all *infinite* coinductive trees over **Term**(Σ)
CTree$^{\omega}(\Sigma)$	all *finite and infinite* coinductive trees over **Term**(Σ)

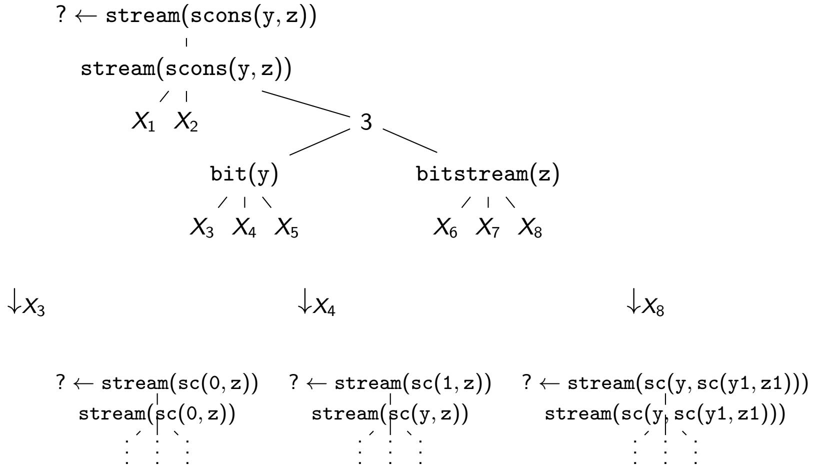
Constructing the structural resolution from first principles...

- At the start, there is a first-order signature Σ.
- First tier of Terms builds on it...
- Term-trees gave rise to a new tier of Coinductive trees...



Tier-3: Derivation trees

A derivation tree is a map $L \to$ **CTree(Σ)**.



Tier-3 laws and notation

Arity: Number of Coinductive tree variables (modulo unification with program clauses)
Operation: – coinductive observations
Calculus: – Guardedness checks.

Tier-3 laws and notation

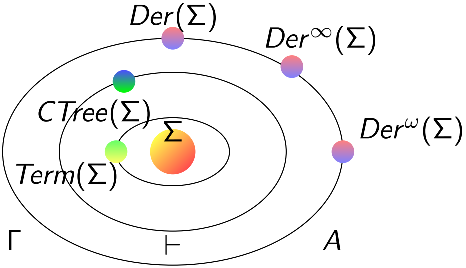
Arity: Number of Coinductive tree variables (modulo unification with program clauses)
Operation: – coinductive observations
Calculus: – Guardedness checks.

Notation:

$\mathbf{CDer}(\Sigma)$	all *finite* derivation trees over $(\mathbf{CTree}(\Sigma)))$
$\mathbf{CDer}^{\infty}(\Sigma)$	all *infinite* derivation trees over $(\mathbf{CTree}(\Sigma))$
$\mathbf{CDer}^{\omega}(\Sigma)$	all *finite and infinite* derivation trees over $(\mathbf{CTree}(\Sigma))$

Constructing the structural resolution from first principles...

- At the start, there is a first-order signature Σ.
- First tier of Terms builds on it...
- Term-trees gave rise to a new tier of Coinductive trees...
- And then, derivations by Structural resolution emerged!



Formal results

New structural resolution can perform the same computations as SLD-resolution.

Theorem

Structural resolution is sound and complete, inductively: every finite successful branch of a derivation tree for A and program P corresponds to SLD-refutation for $P \vdash A$.
...and vice versa...

Formal results

New structural resolution can perform the same computations as SLD-resolution.

Theorem

Structural resolution is sound and complete, inductively: every finite successful branch of a derivation tree for A and program P corresponds to SLD-refutation for $P \vdash A$.
...and vice versa...

Importantly, we have discovered plenty of structure to allow automated proof and program analysis.

Outline

Big Picture: Proofs and structures

Problem evidence: Structural Recursion without structure

Structural Recursion in ITPs: Types give Structure

Structural Recursion without structure in LP?

Solution: New Structural Resolution for ATP

Solution's Effect: Universal Productivity for Structural Resolution, for free

Theory of Universal Productivity, for free

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\Sigma)$.

Theory of Universal Productivity, for free

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\Sigma)$.

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

- finite LP that give rise to derivations in $\mathbf{CDer}(\Sigma)$,
  E.g. **bit**.

Theory of Universal Productivity, for free

A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\Sigma)$.

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

- finite LP that give rise to derivations in $\mathbf{CDer}(\Sigma)$, E.g. **bit**.
- inductive LPs all derivations for which are in $\mathbf{CDer}^{\omega}(\Sigma)$; E.g. **NatList**.

Theory of Universal Productivity, for free

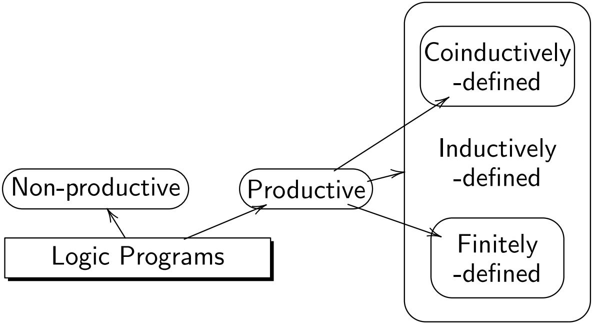
A first-order logic program P is *productive* if

for any term $t \in \mathbf{Term}(\Sigma)$, the coinductive tree with the root t belongs to $\mathbf{CTree}(\Sigma)$.

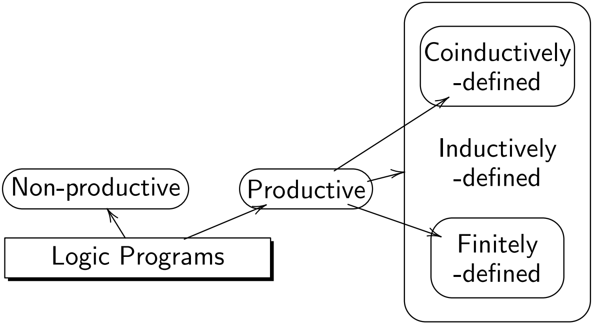
In the class of Productive LPs, we can further distinguish:

- finite LP that give rise to derivations in $\mathbf{CDer}(\Sigma)$, E.g. **bit**.
- inductive LPs all derivations for which are in $\mathbf{CDer}^{\omega}(\Sigma)$; E.g. **NatList**.
- coinductive LPs all derivations for which are in $\mathbf{CDer}^{\infty}(\Sigma)$ E.g. **Stream**.

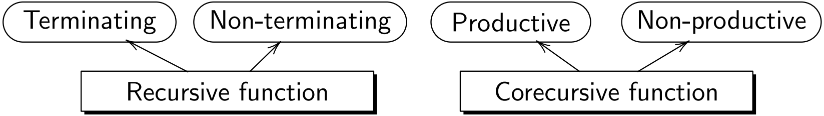
Theory of universal Productivity in LP, at last!



Theory of universal Productivity in LP, at last!



Compare with ITPs:



Deciding Productivity: Guardedness

- Tier 1. Measures of reduction on term trees;
- Tier 2. Use Tier-1 measures of reduction to identify unguarded coinductive tree loops;
- Tier 3. Use observation subtrees of derivation trees for semi-decidable search for unguarded coinductive trees and measures of reductions arising in the program.

Semi-deciding Universal Productivity: if a program is guarded, it is productive.

Current and future work

1. Coinduction by structured resolution.

Current and future work

1. Coinduction by structured resolution.
2. Coinductive Soundness and completeness of the 3-Tier calculus relative to models based on $\mathbf{CTree}^{\omega}(\Sigma)$.

Current and future work

1. Coinduction by structured resolution.
2. Coinductive Soundness and completeness of the 3-Tier calculus relative to models based on $\mathbf{CTree}^{\omega}(\Sigma)$.
3. Extensions, implementation, applications: structural resolution for type inference in functional languages

... join us, there is a lot more to it!

Thank you!

Download your copy of CoALP today:

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