Tree-like Grammars and Separation Logic

Christina Jansen **Christoph Matheja** **Thomas Noll**

Software Modeling and Verification Group

http://moves.rwth-aachen.de/

APLAS 2015

December 30, 2015; Pohang

Motivation

Typical programming errors

- Dereferencing null (or disposed) pointers
- Accidental invalidation of data structures
- Creation of memory leaks

$\Longrightarrow$ need to reason automatically about shared mutable data structures

Motivation

Typical programming errors

- Dereferencing null (or disposed) pointers
- Accidental invalidation of data structures
- Creation of memory leaks

$\Longrightarrow$ need to reason automatically about shared mutable data structures

Why separation logic?

- extension of Hoare-logic to reason about heaps
- Hoare-style proofs, shape analysis, symbolic execution...
- CYCLIST, INFER, VERIFAST...
- suffers from undecidable entailment problem

Motivation

Typical programming errors

- Dereferencing null (or disposed) pointers
- Accidental invalidation of data structures
- Creation of memory leaks

$\Longrightarrow$ need to reason automatically about shared mutable data structures

Why separation logic?

- extension of Hoare-logic to reason about heaps
- Hoare-style proofs, shape analysis, symbolic execution...
- CYCLIST, INFER, VERIFAST...
- suffers from undecidable entailment problem

Why graph grammars?

- extension of context-free grammars to describe graphs
- shape analysis, symbolic execution, natural language processing...
- JUGGRNAUT, GROOVE...
- suffers from undecidable inclusion problem

Motivation: Separation Logic Entailments

```
void addTwo(Node h) {
  Node u = new Node();
  u.next = h;
  h = u;
  u = new Node();
  u.next = h;
  h = u;

}
```

[1] J. Brotherston et al. "Automated cyclic entailment proofs in separation logic." CADE, 2011.

Motivation: Separation Logic Entailments

$\{ls(h, \texttt{null})\}$

```
void addTwo(Node h) {

  Node u = new Node();

  u.next = h;

  h = u;

  u = new Node();

  u.next = h;

  h = u;

}
```

$\{ls(h, \texttt{null})\}$

[1] J. Brotherston et al. "Automated cyclic entailment proofs in separation logic." CADE, 2011.

Motivation: Separation Logic Entailments

$\{ls(h, \texttt{null})\}$
`void addTwo(Node h) {`
$\{ls(h, \texttt{null})\}$
`Node u = new Node();`
$\{ls(h, \texttt{null}) * u \mapsto \_\}$
`u.next = h;`
$\{\exists x \,.\, ls(x, \texttt{null}) * u \mapsto x \wedge h = x\}$
`h = u;`
$\{\exists x, y \,.\, ls(x, \texttt{null}) * y \mapsto x \wedge h = y \wedge y = u\}$
`u = new Node();`
$\{\exists x, y \,.\, ls(x, \texttt{null}) * y \mapsto x * u \mapsto \_ \wedge h = y\}$
`u.next = h;`
$\{\exists x, y, z \,.\, ls(x, \texttt{null}) * y \mapsto x * u \mapsto z \wedge h = z\}$
`h = u;`
$\{\exists x, y, z \,.\, ls(x, \texttt{null}) * y \mapsto x * u \mapsto z \wedge h = u\}$
$\{ls(h, \texttt{null})\}$
`}`
$\{ls(h, \texttt{null})\}$

"Effective procedures for establishing entailments are at the foundation of automatic verification based on separation logic."[1]

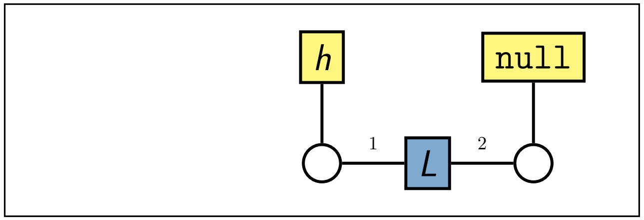
[1] J. Brotherston et al. "Automated cyclic entailment proofs in separation logic." CADE, 2011.

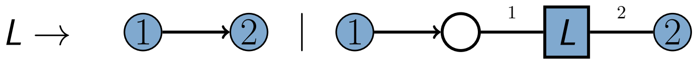
Motivation: Graph Grammar Language Inclusion

$\{ls(h, \texttt{null})\}$

```
▷ void addTwo(Node h) {
    Node u = new Node();
    u.next = h;
    h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
```

$\{ls(h, \texttt{null})\}$



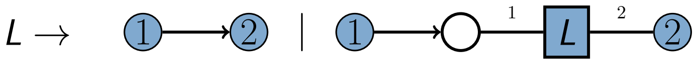


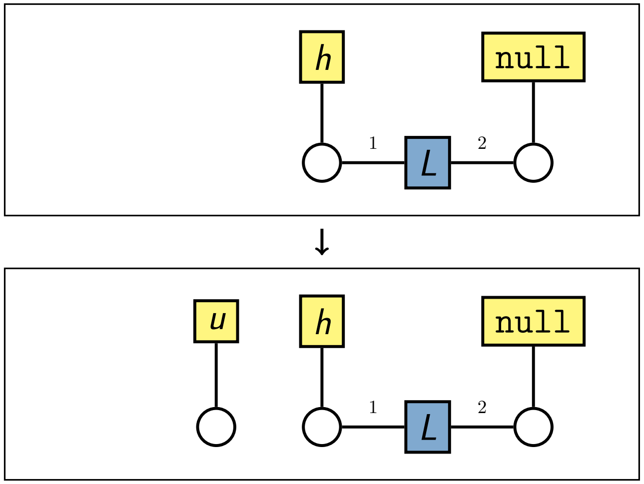
Motivation: Graph Grammar Language Inclusion

$\{ls(h, \texttt{null})\}$

```
void addTwo(Node h) {
▷ Node u = new Node();
  u.next = h;
  h = u;
  u = new Node();
  u.next = h;
  h = u;
}
```

$\{ls(h, \texttt{null})\}$



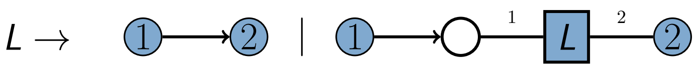


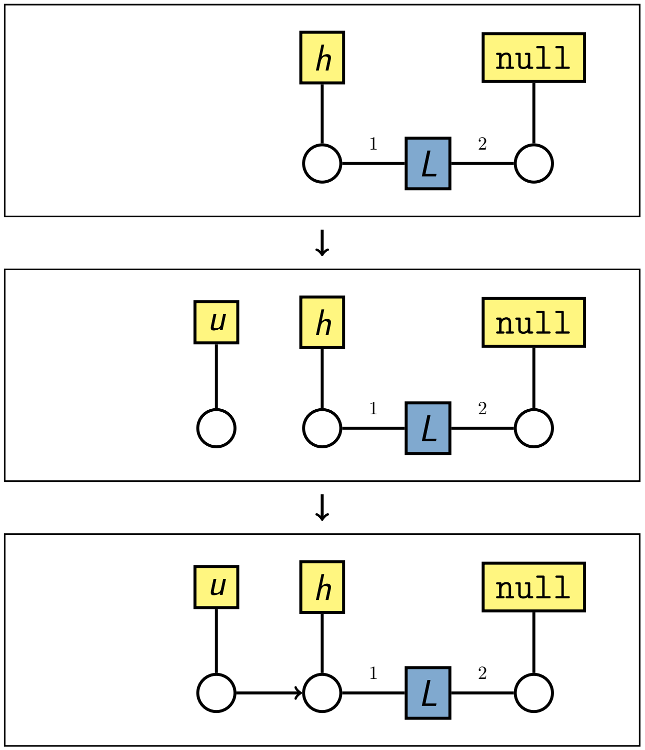
Motivation: Graph Grammar Language Inclusion

$\{ls(h, \texttt{null})\}$

```
  void addTwo(Node h) {
    Node u = new Node();
▷   u.next = h;
    h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
```

$\{ls(h, \texttt{null})\}$



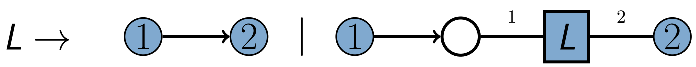


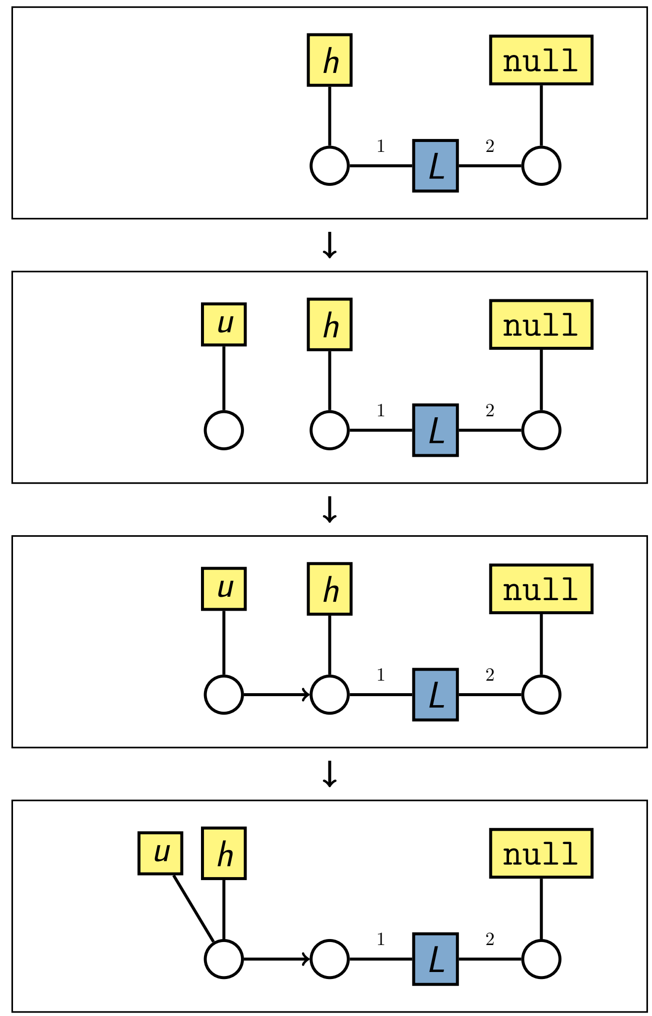
Motivation: Graph Grammar Language Inclusion

$\{ls(h, \texttt{null})\}$

```
  void addTwo(Node h) {
    Node u = new Node();
    u.next = h;
▷   h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
```

$\{ls(h, \texttt{null})\}$





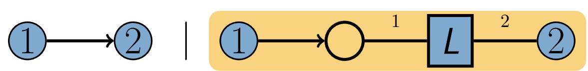
Motivation: Graph Grammar Language Inclusion

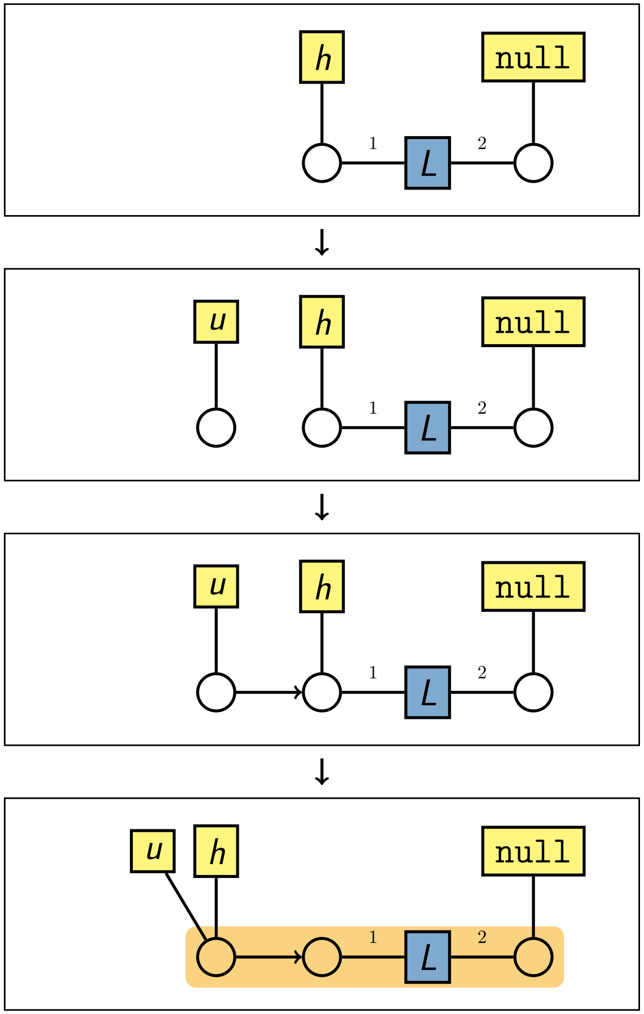
$\{ls(h, \texttt{null})\}$

```
  void addTwo(Node h) {
    Node u = new Node();
    u.next = h;
▷   h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
```

$\{ls(h, \texttt{null})\}$

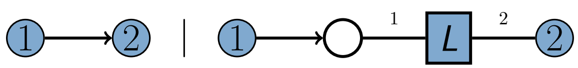
$L \rightarrow$

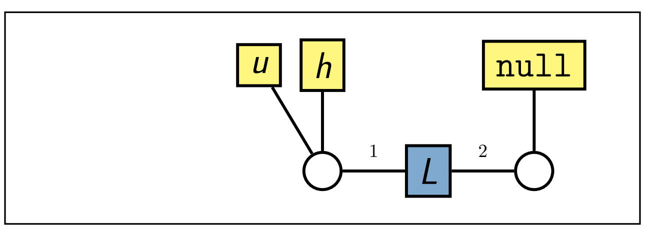


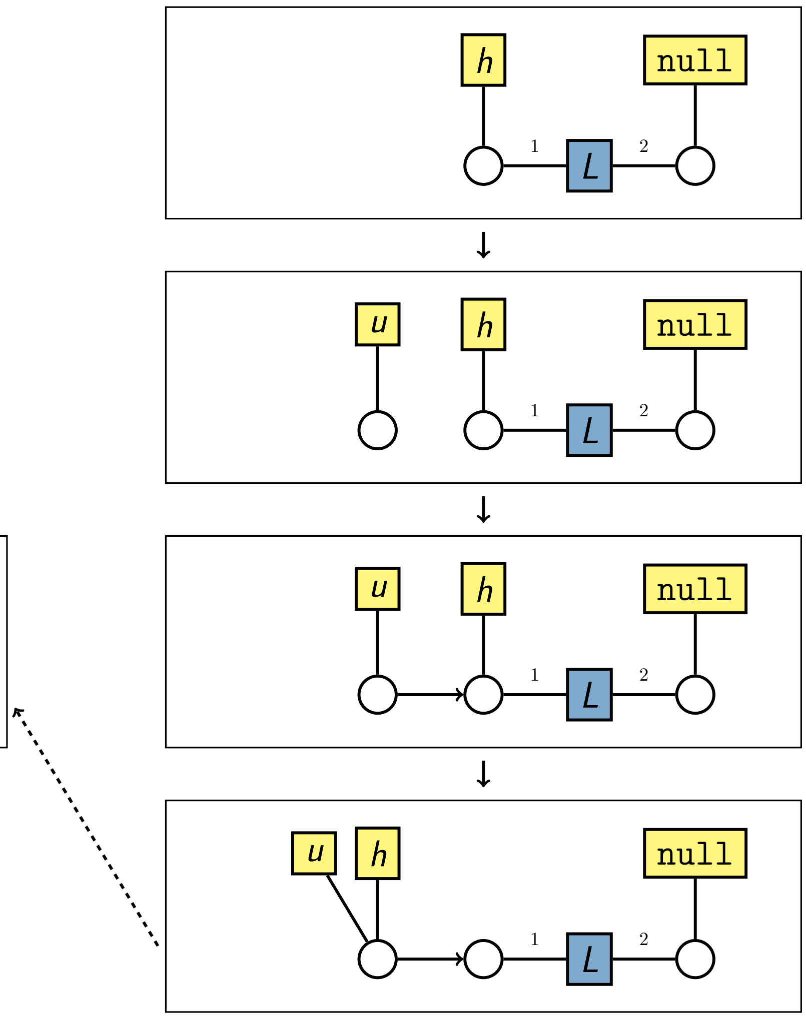


Motivation: Graph Grammar Language Inclusion

{$ls(h, \texttt{null})$}
```
  void addTwo(Node h) {
    Node u = new Node();
    u.next = h;
▷   h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
```
{$ls(h, \texttt{null})$}

$L \rightarrow$ 



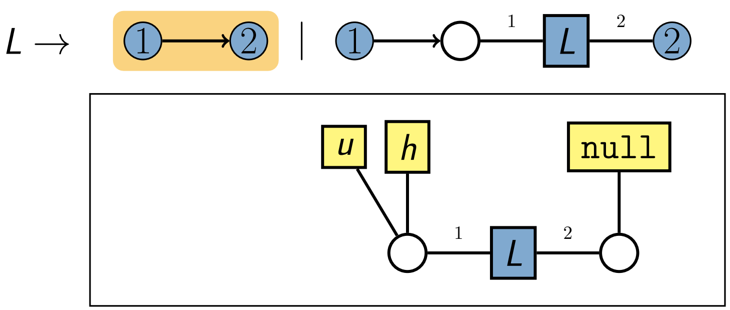


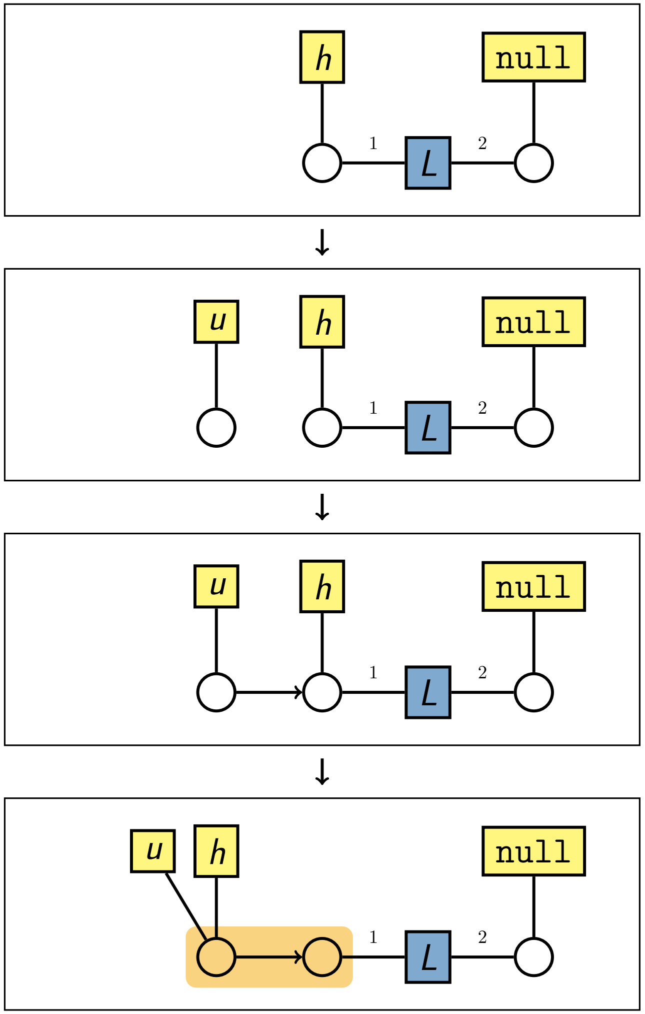
Motivation: Graph Grammar Language Inclusion

$\{ls(h, \texttt{null})\}$

```
void addTwo(Node h) {
  Node u = new Node();
  u.next = h;
▷ h = u;
  u = new Node();
  u.next = h;
  h = u;
}
```

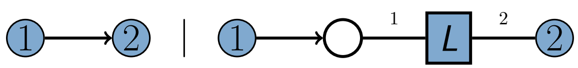
$\{ls(h, \texttt{null})\}$

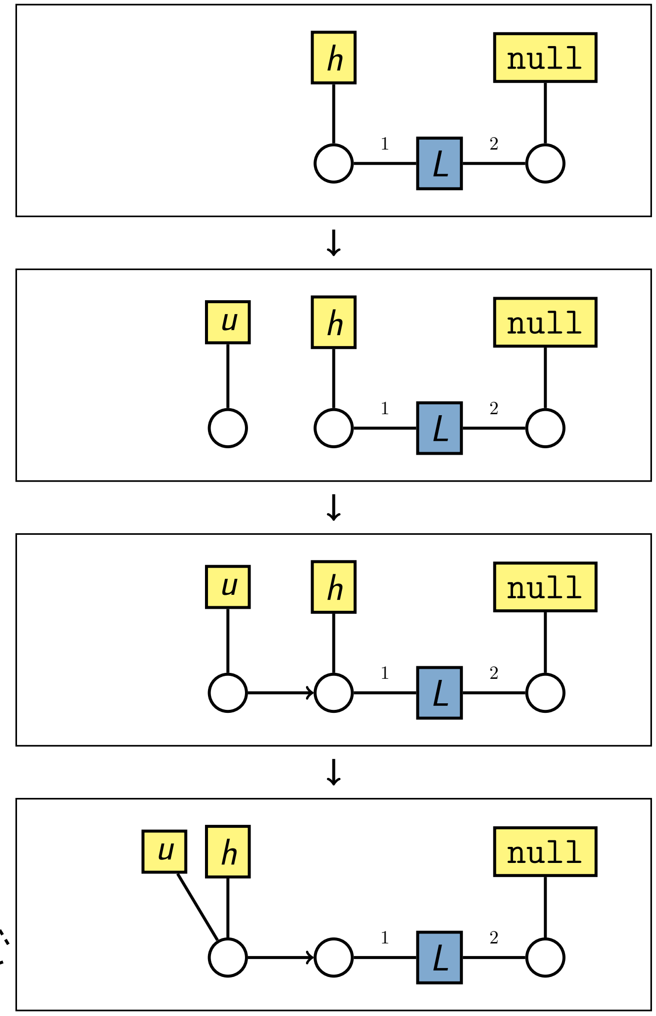


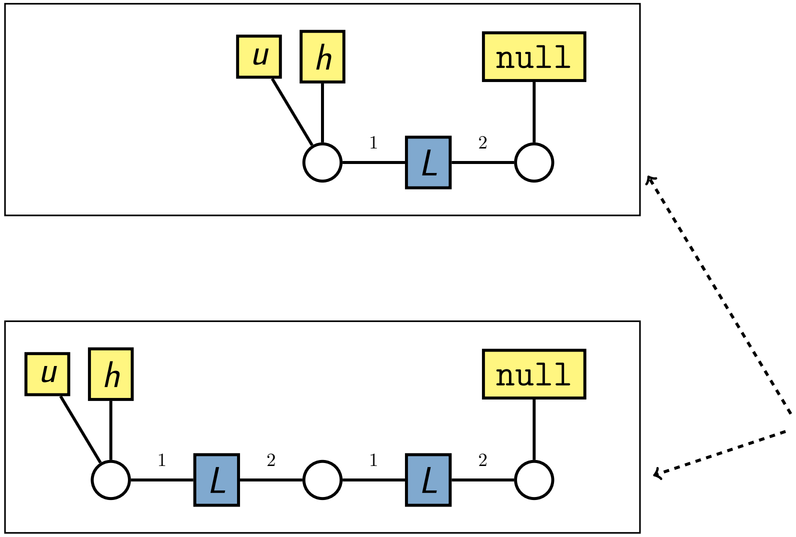


Motivation: Graph Grammar Language Inclusion

```
{ls(h, null)}
void addTwo(Node h) {
  Node u = new Node();
  u.next = h;
▷ h = u;
  u = new Node();
  u.next = h;
  h = u;
}
{ls(h, null)}
```

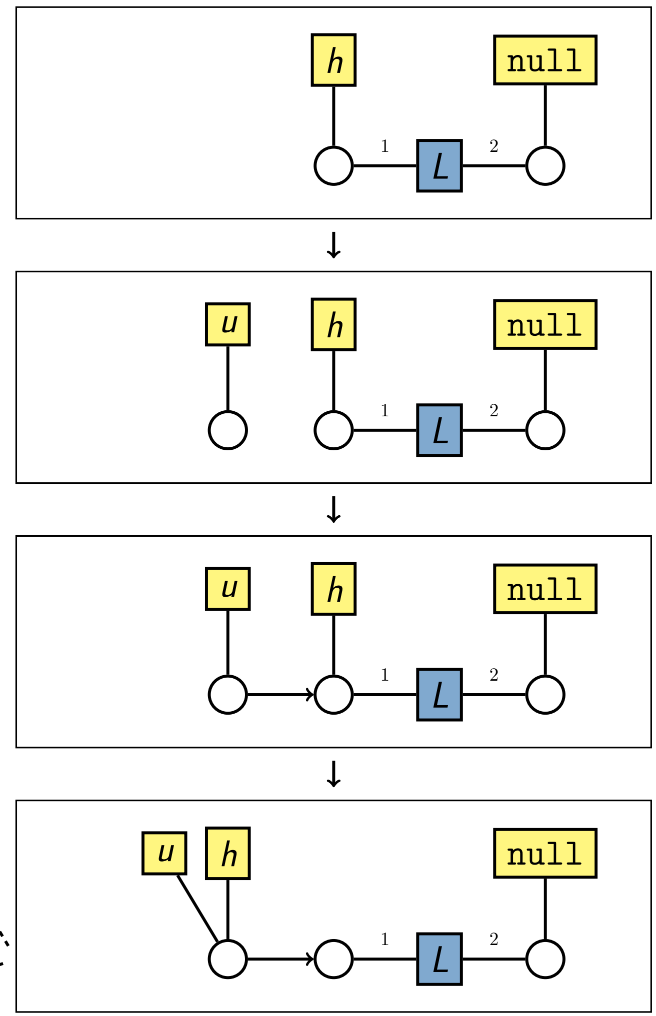
$L \rightarrow$ 

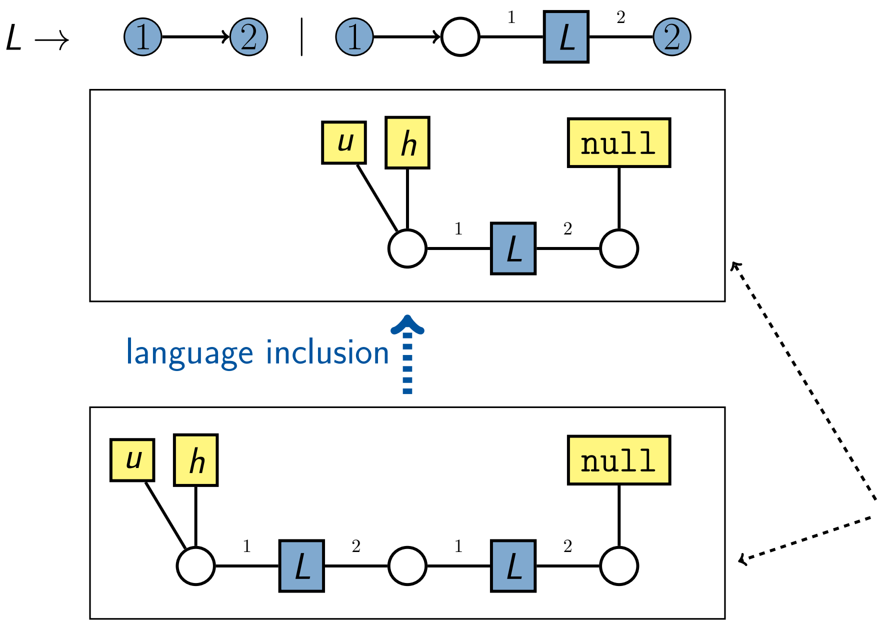




Motivation: Graph Grammar Language Inclusion

```
  {ls(h, null)}
  void addTwo(Node h) {
    Node u = new Node();
    u.next = h;
▷   h = u;
    u = new Node();
    u.next = h;
    h = u;
  }
  {ls(h, null)}
```

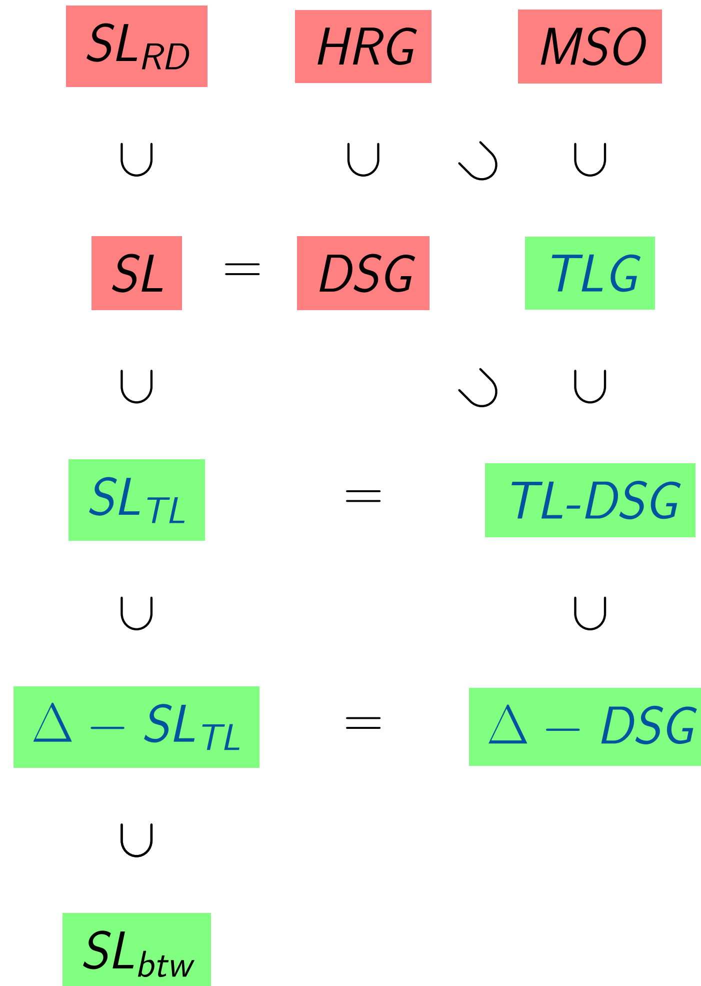




Overview

How are these problems related?

What are decidable fragments?



- undecidable entailment problem
- decidable entailment problem
- new fragments

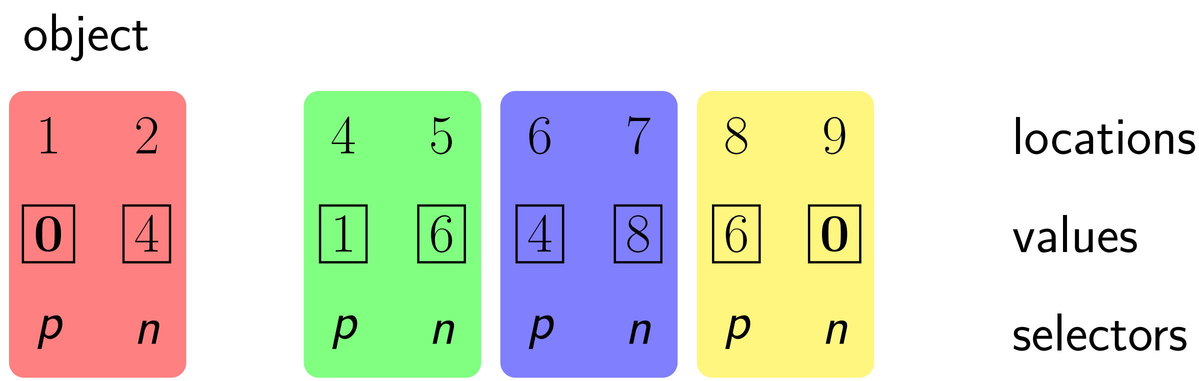
Heaps

$$h \;:\; \mathbb{N} \dashrightarrow_{\text{finite}} \mathbb{N}_0$$

1	2		4	5	6	7	8	9	locations
0	4		1	6	4	8	6	**0**	values

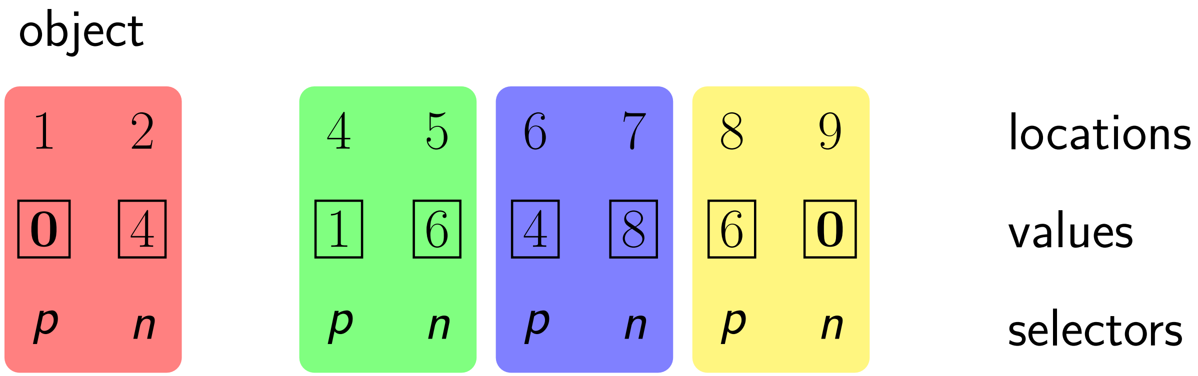
Heaps

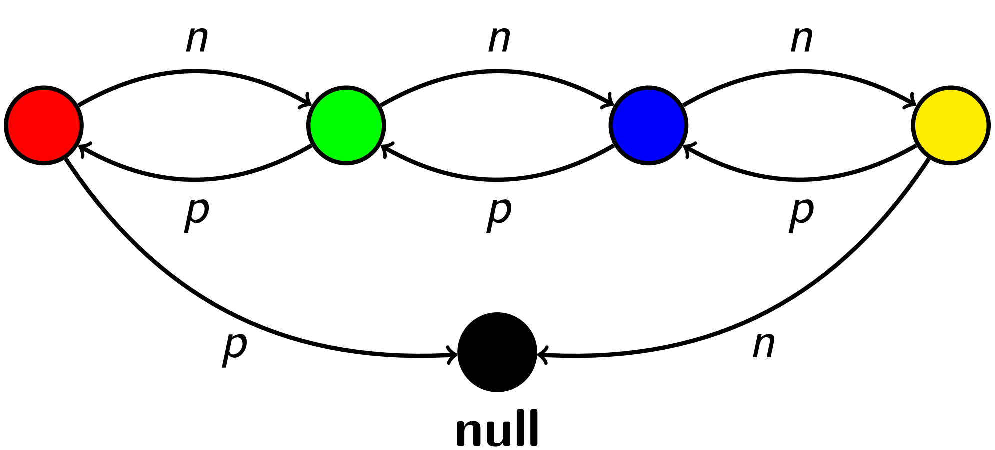
$$h : \mathbb{N} \dashrightarrow_{\text{finite}} \mathbb{N}_0$$



Heaps

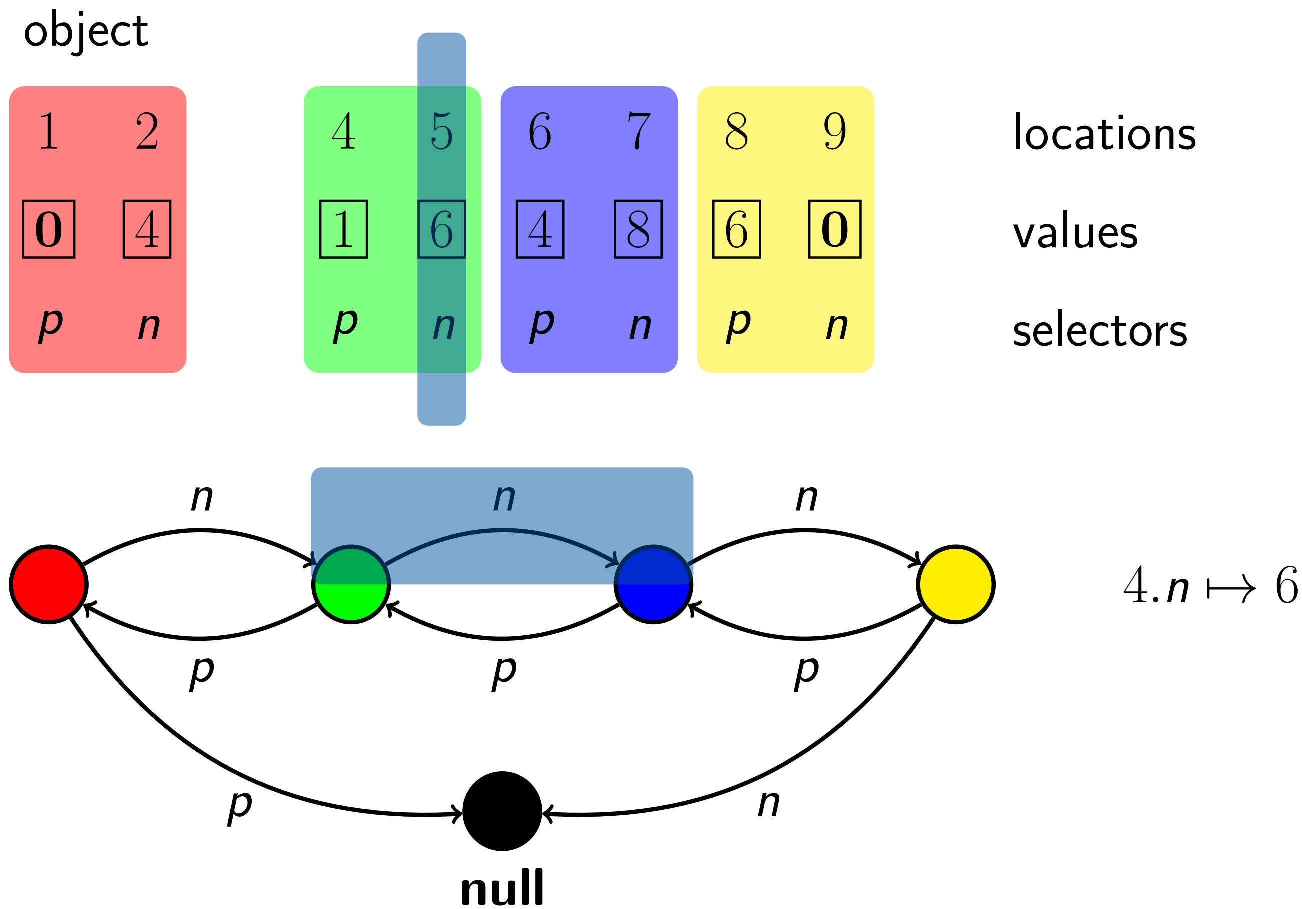
$$h \;:\; \mathbb{N} \dashrightarrow_{\text{finite}} \mathbb{N}_0$$





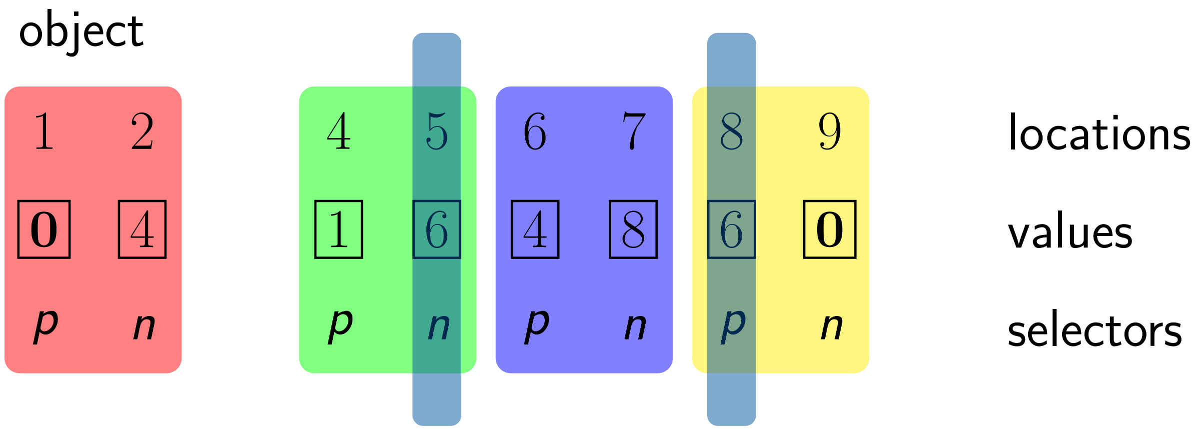
Heaps

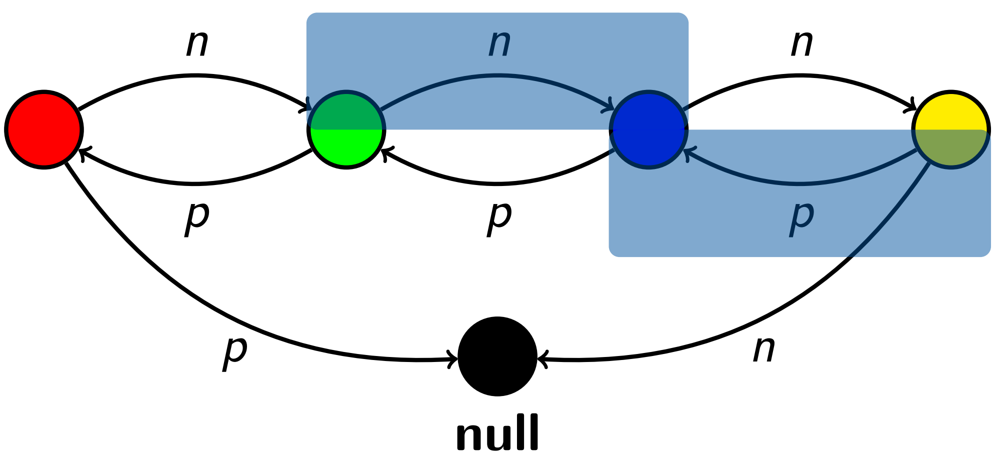
$$h : \mathbb{N} \dashrightarrow_{\text{finite}} \mathbb{N}_0$$



Heaps

$$h \ : \ \mathbb{N} \dashrightarrow_{\text{finite}} \mathbb{N}_0$$





$$4.n \mapsto 6 \ * \ 8.p \mapsto 6$$

Separation logic with recursive definitions

Separation logic formulae $\varphi(\vec{x})$

$$\varphi(\vec{x}) \ ::= \ \exists \vec{y} \, . \, \sigma(\vec{x}, \vec{y}) \wedge \pi(\vec{x}, \vec{y})$$

symbolic heaps

$$\sigma(\vec{z}) \ ::= \ z_i.s \mapsto z_j \mid P(\vec{z}) \mid \sigma * \sigma$$

spatial formulae

$$\pi(\vec{z}) \ ::= \ z_i = z_j \ \mid \ \pi \wedge \pi$$

pure formulae

Separation logic with recursive definitions

Separation logic formulae $\varphi(\vec{x})$

$$\varphi(\vec{x}) ::= \exists \vec{y} \,.\, \sigma(\vec{x}, \vec{y}) \wedge \pi(\vec{x}, \vec{y})$$ symbolic heaps

$$\sigma(\vec{z}) ::= z_i.s \mapsto z_j \mid P(\vec{z}) \mid \sigma * \sigma$$ spatial formulae

$$\pi(\vec{z}) ::= z_i = z_j \mid \pi \wedge \pi$$ pure formulae

Predicate definitions $P(\vec{x}) = \varphi_1(\vec{x}) \vee \ldots \vee \varphi_k(\vec{x})$

Example

$$ls(x_1, x_2) = (emp \wedge x_1 = x_2) \vee (\exists y \,.\, x_1.n \mapsto y * ls(y, x_2))$$

Separation logic with recursive definitions

Separation logic formulae $\varphi(\vec{x})$

$$\varphi(\vec{x}) \ ::= \ \exists \vec{y} \,.\, \sigma(\vec{x}, \vec{y}) \wedge \pi(\vec{x}, \vec{y})$$ symbolic heaps

$$\sigma(\vec{z}) \ ::= \ z_i.s \mapsto z_j \mid P(\vec{z}) \mid \sigma * \sigma$$ spatial formulae

$$\pi(\vec{z}) \ ::= \ z_i = z_j \mid \pi \wedge \pi$$ pure formulae

Predicate definitions $P(\vec{x}) \ = \ \varphi_1(\vec{x}) \vee \ldots \vee \varphi_k(\vec{x})$

Example

$$ls(x_1, x_2) \ = \ (emp \wedge x_1 = x_2) \ \vee \ (\exists y \,.\, x_1.n \mapsto y * ls(y, x_2))$$

Environments $\Gamma = \{P(\vec{x}) \mid P \in Pred\}$

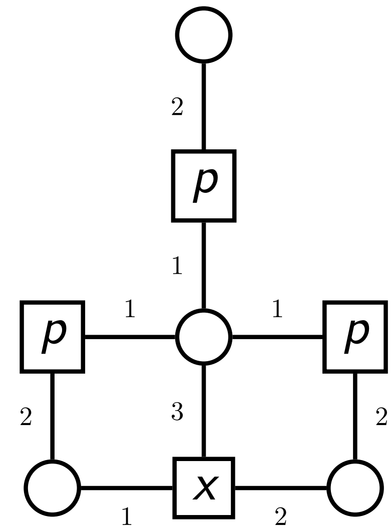
- set of predicate definitions
- every existentially quantified variable is eventually allocated

Graph grammars in a nutshell

Σ	finite alphabet
$rk : \Sigma \rightarrow \mathbb{N}$	ranking function

A hypergraph (HG) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \rightarrow \Sigma$, $\quad rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \rightarrow V^\star$ $\quad rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^\star$.

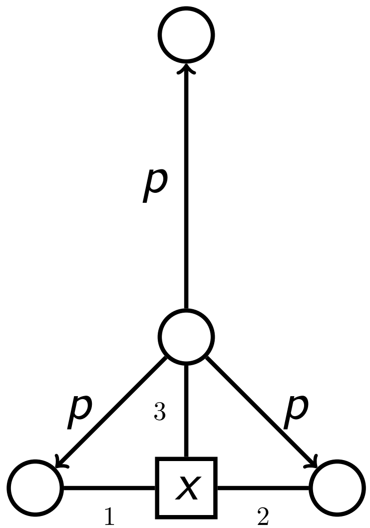


Graph grammars in a nutshell

Σ	finite alphabet
$rk : \Sigma \to \mathbb{N}$	ranking function

A hypergraph (HG) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $\quad rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \to V^\star$ $\quad rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^\star$.

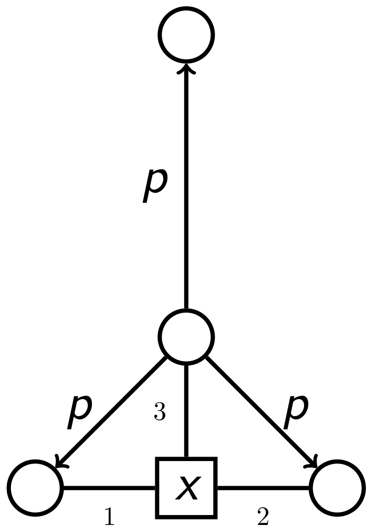


Graph grammars in a nutshell

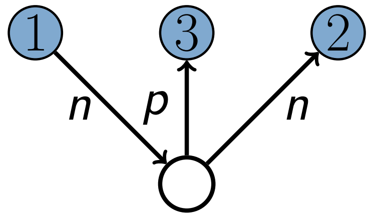
Σ finite alphabet
$rk : \Sigma \to \mathbb{N}$ ranking function

A hypergraph (*HG*) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \to V^\star$ $rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^\star$.



Hyperedge replacement



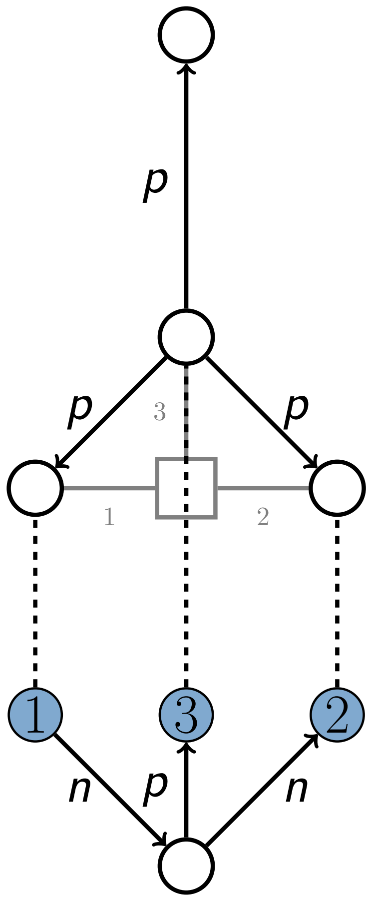
Graph grammars in a nutshell

Σ	finite alphabet
$rk : \Sigma \to \mathbb{N}$	ranking function

A hypergraph (HG) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $\quad rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \to V^{\star}$ $\quad rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^{\star}$.

Hyperedge replacement

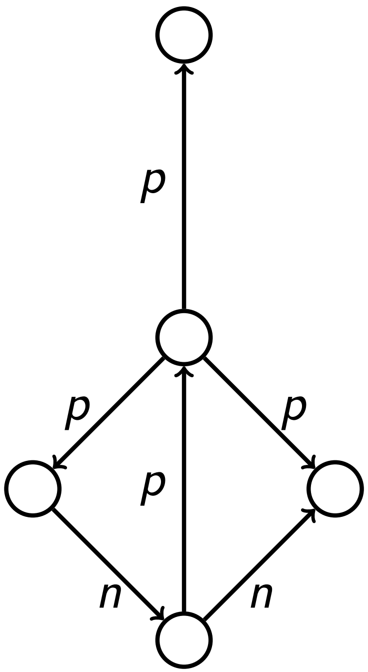


Graph grammars in a nutshell

Σ	finite alphabet
$rk : \Sigma \to \mathbb{N}$	ranking function

A hypergraph (HG) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \to V^{\star}$ $rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^{\star}$.



Hyperedge replacement

Graph grammars in a nutshell

Σ	finite alphabet
$rk : \Sigma \to \mathbb{N}$	ranking function

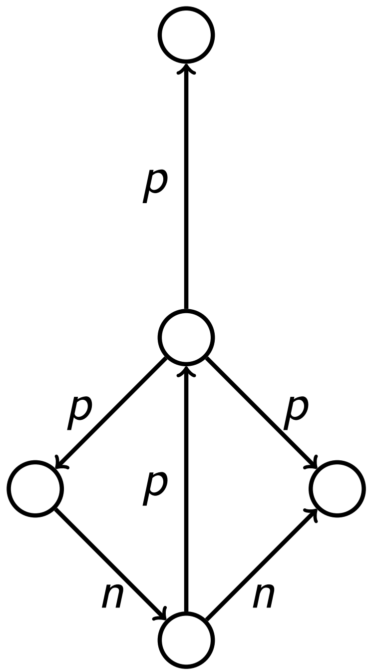
A hypergraph (*HG*) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $rk(e) = \mathrm{lab}(e)$,
- attachment $\mathrm{att} : E \to V^\star$ $rk(e) = |\mathrm{att}(e)|$,
- external nodes $\mathrm{ext} \in V^\star$.

Hyperedge replacement

A heap configuration (*HC*) is a hypergraph with

- $rk(e) = 2$ for each $e \in E$,
- at most one outgoing edge is labelled $s \in \Sigma$ for each $v \in V$.



Graph grammars in a nutshell

A hyperedge replacement grammar (HRG) is a tuple $G = (N, \Sigma, P, S)$ with

- disjoint sets of nonterminals N and terminals Σ,
- set of production rules $P \subseteq N \times HG$ of the form $X \to H$ $\qquad rk(X) = |\mathrm{ext}_H|$,
- initial symbol $S \in N$.

Derivations, derivation trees, languages are defined as for context-free grammars.

Graph grammars in a nutshell

A hyperedge replacement grammar (HRG) is a tuple $G = (N, \Sigma, P, S)$ with

- disjoint sets of nonterminals N and terminals Σ,
- set of production rules $P \subseteq N \times HG$ of the form $X \rightarrow H$ $\quad rk(X) = |\mathrm{ext}_H|$,
- initial symbol $S \in N$.

Derivations, derivation trees, languages are defined as for context-free grammars.

A data structure grammar (*DSG*) is an HRG generating heap configurations only.

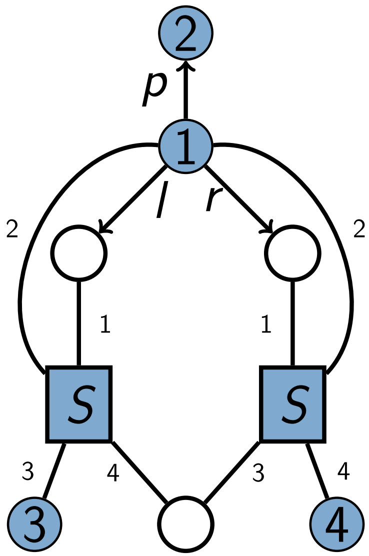
Theorem

For each HRG G one can construct a DSG K such that $L(K) = L(G) \cap HC$.

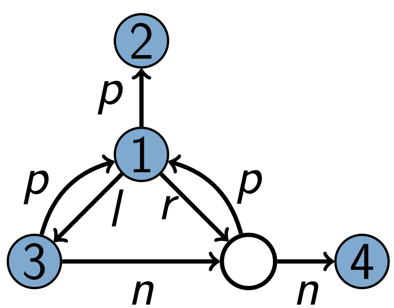
Data structure grammar for trees with linked leaves

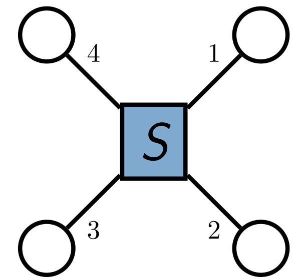
data structure grammar | derivation tree | derivation

$S \to S_1 \triangleq$



$S \to S_2 \triangleq$

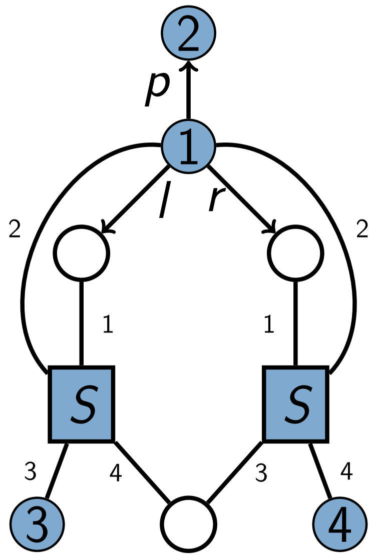




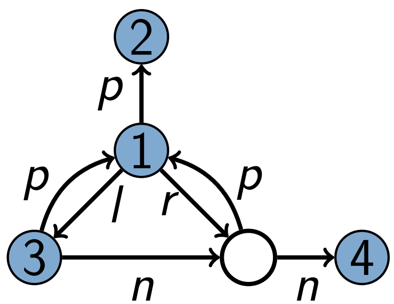
Data structure grammar for trees with linked leaves

data structure grammar	derivation tree	derivation

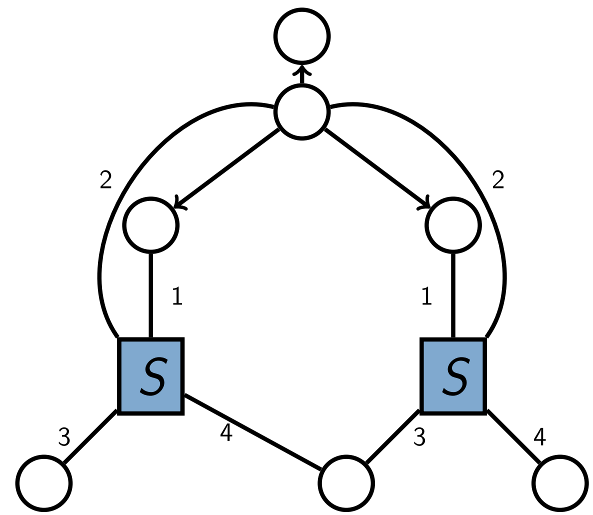
$S \rightarrow S_1 \triangleq$



$S \rightarrow S_2 \triangleq$



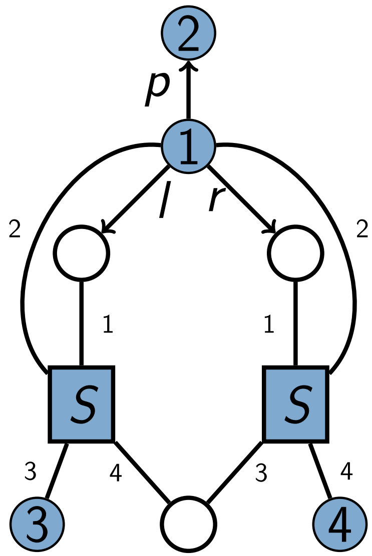
S_1



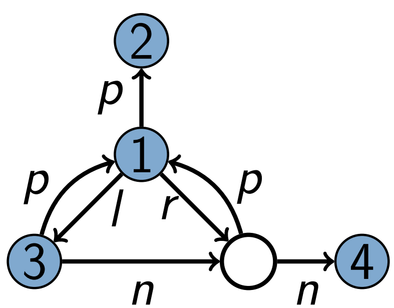
Data structure grammar for trees with linked leaves

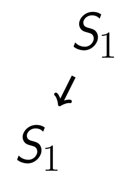
data structure grammar | derivation tree | derivation

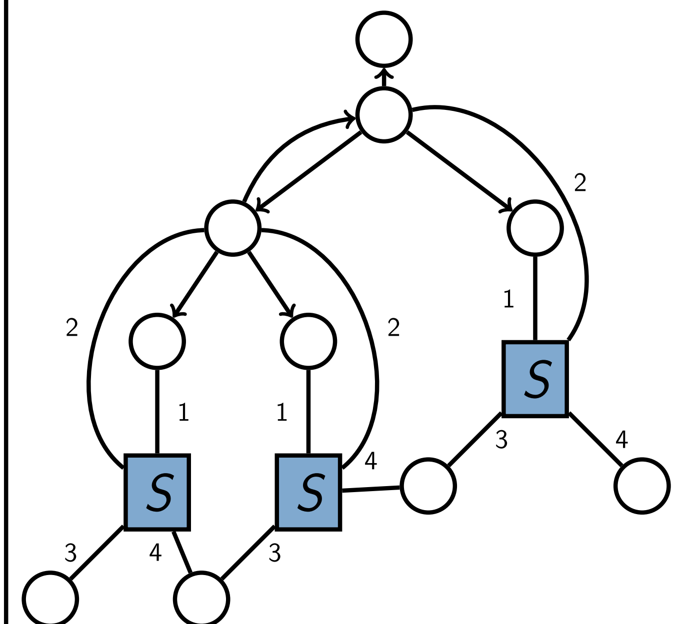
$S \to S_1 \triangleq$



$S \to S_2 \triangleq$



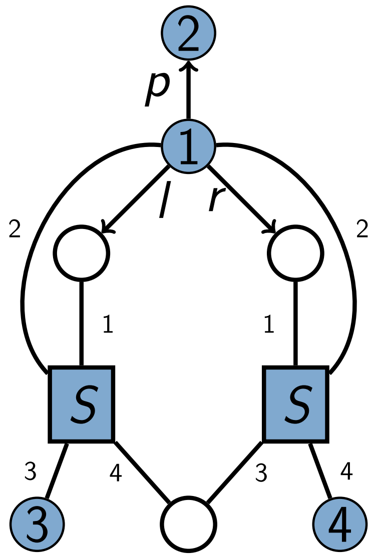




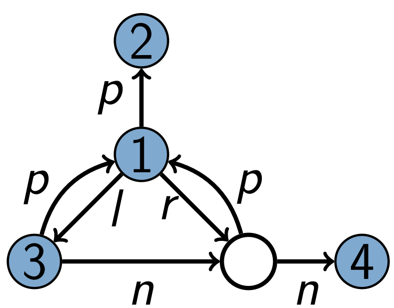
Data structure grammar for trees with linked leaves

data structure grammar

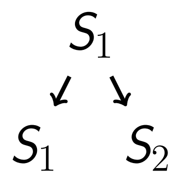
$S \to S_1 \triangleq$



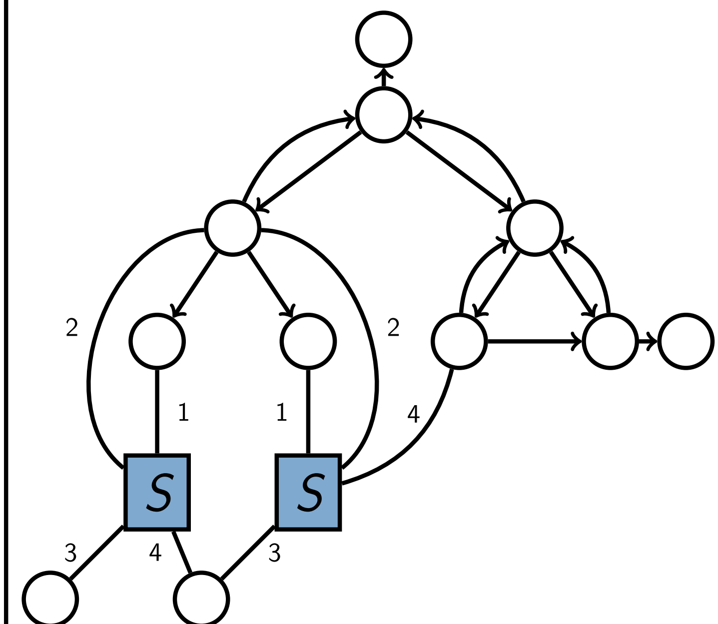
$S \to S_2 \triangleq$



derivation tree



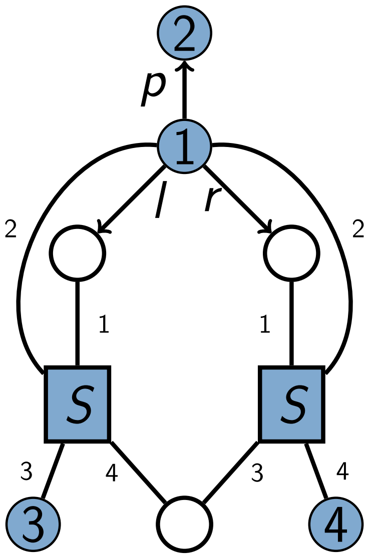
derivation



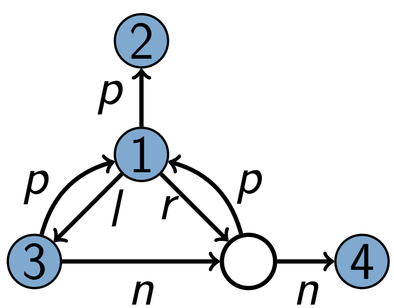
Data structure grammar for trees with linked leaves

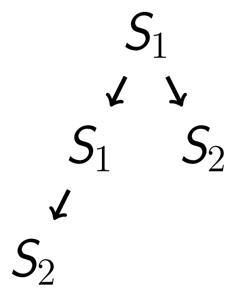
data structure grammar | derivation tree | derivation

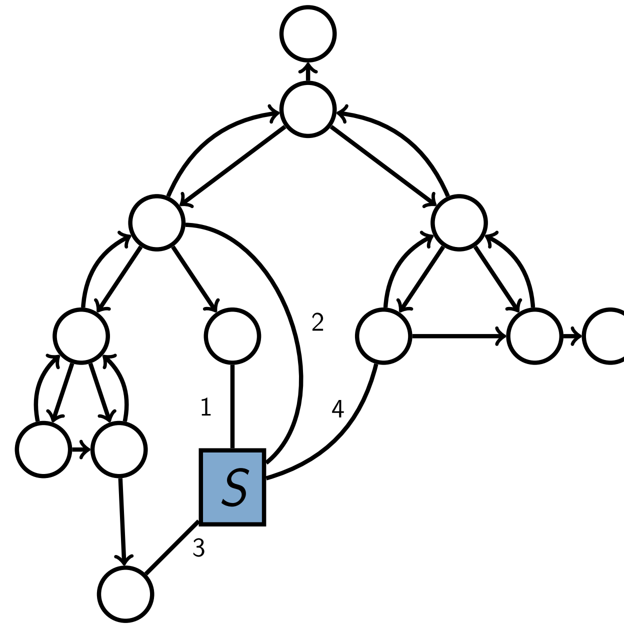
$S \to S_1 \triangleq$



$S \to S_2 \triangleq$



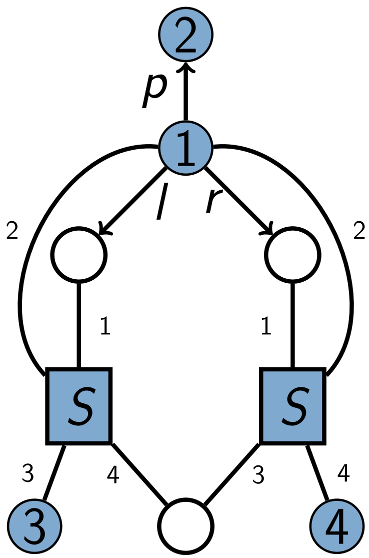




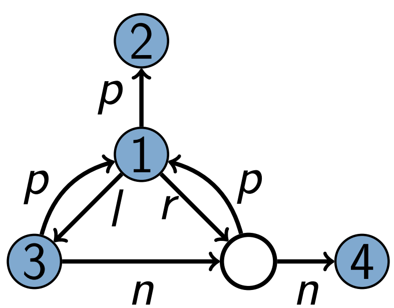
Data structure grammar for trees with linked leaves

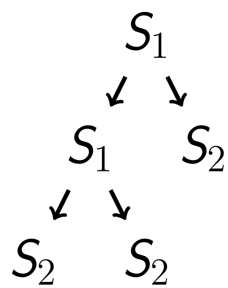
data structure grammar | derivation tree | derivation

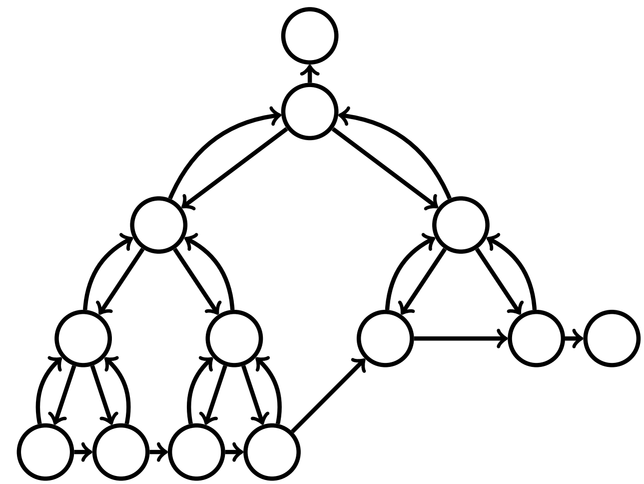
$S \to S_1 \triangleq$



$S \to S_2 \triangleq$



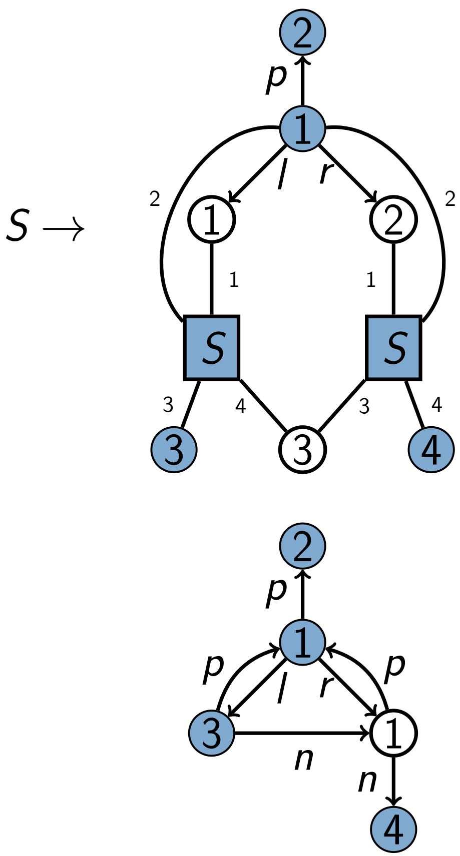




Separation logic and hyperedge replacement grammars

Theorem (Jansen et al.[1])

Every separation logic formula can be translated into a language-equivalent data structure grammar and vice versa.



$$S(x_1, x_2, x_3, x_4) \;=$$

$$\exists y_1, y_2, y_3 \;.\; x_1 \mapsto (y_1, y_2, x_2, \mathbf{null}) \;*\; S(y_1, x_1, x_3, y_3) \;*\; S(y_2, x_1, y_3, x_4)$$

$$\vee$$

$$\exists y_1 \;.\; x_1 \mapsto (x_3, y_1, x_2, \mathbf{null}) \;*\; x_3 \mapsto (\mathbf{null}, \mathbf{null}, x_1, y_1) \;*\; y_1 \mapsto (\mathbf{null}, \mathbf{null}, x_1, x_4)$$

[1] C. Jansen et al. "Generating inductive predicates for symbolic execution of pointer-manipulating programs." ICGT, 2014.

Towards a decidable inclusion problem

Theorem (Courcelle[2])

For each HRG G and MSO sentence φ, one can effectively construct an HRG K such that

$$L(K) \;=\; L(G) \cap L(\varphi) \;=\; \{H \in L(G) \mid H \models \varphi\}.$$

[2] Courcelle, B. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs." Information and computation, 1990.

Towards a decidable inclusion problem

Theorem (Courcelle[2])

For each HRG G and MSO sentence φ, one can effectively construct an HRG K such that

$$L(K) \; = \; L(G) \cap L(\varphi) \; = \; \{H \in L(G) \mid H \models \varphi\}.$$

Let G, K be data structure grammars.
Assume there exists *MSO* sentence φ with $L(K) = L(\varphi)$.

$$L(G) \subseteq L(K)$$

$$\Leftrightarrow \; L(G) \subseteq L(\varphi)$$

$$\Leftrightarrow \; L(G) \cap L(\neg\varphi) \; = \; \emptyset$$

[2] Courcelle, B. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs." Information and computation, 1990.

Towards a decidable inclusion problem

Theorem (Courcelle[2])

For each HRG G and MSO sentence φ, one can effectively construct an HRG K such that

$$L(K) \;=\; L(G) \cap L(\varphi) \;=\; \{H \in L(G) \mid H \models \varphi\}.$$

Let G, K be data structure grammars.
Assume there exists *MSO* sentence φ with $L(K) = L(\varphi)$.

$$L(G) \subseteq L(K)$$

$$\Leftrightarrow \; L(G) \subseteq L(\varphi)$$

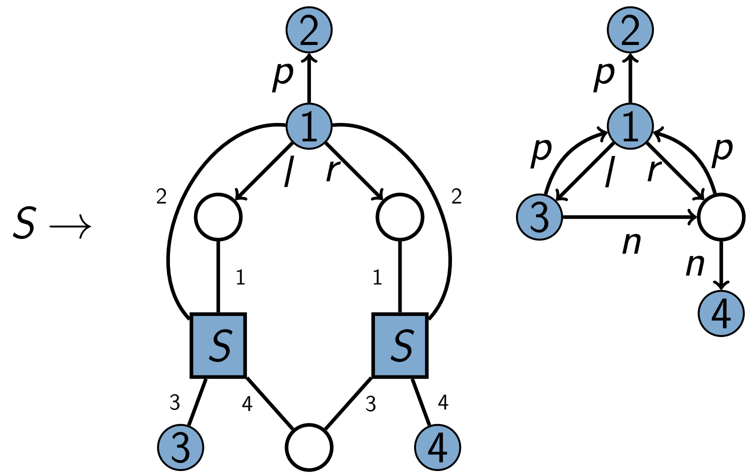
$$\Leftrightarrow \; L(G) \cap L(\neg\varphi) \;=\; \emptyset$$

G is an arbitrary data structure grammar!

[2] Courcelle, B. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs." Information and computation, 1990.

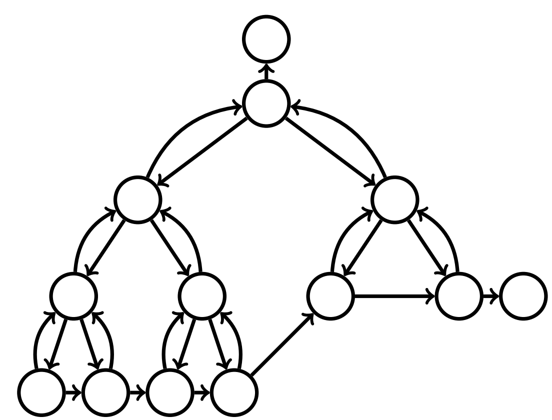
Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees



Derivation tree

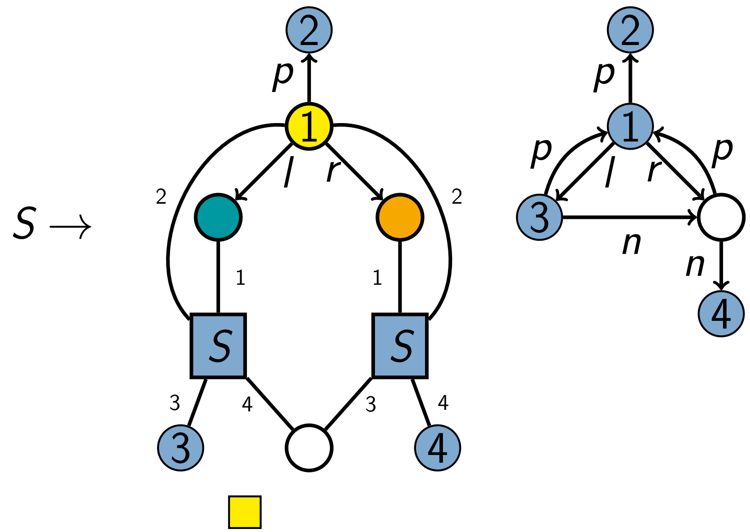
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



[1]Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

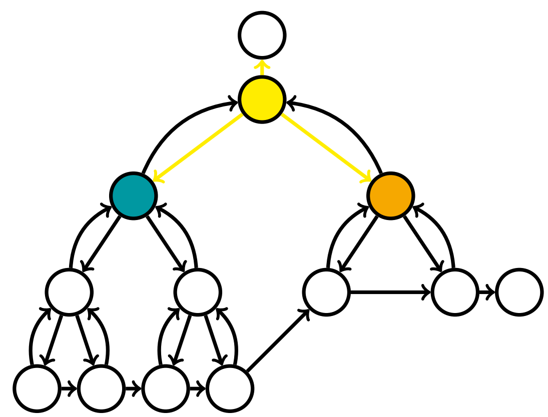
Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees



Derivation tree

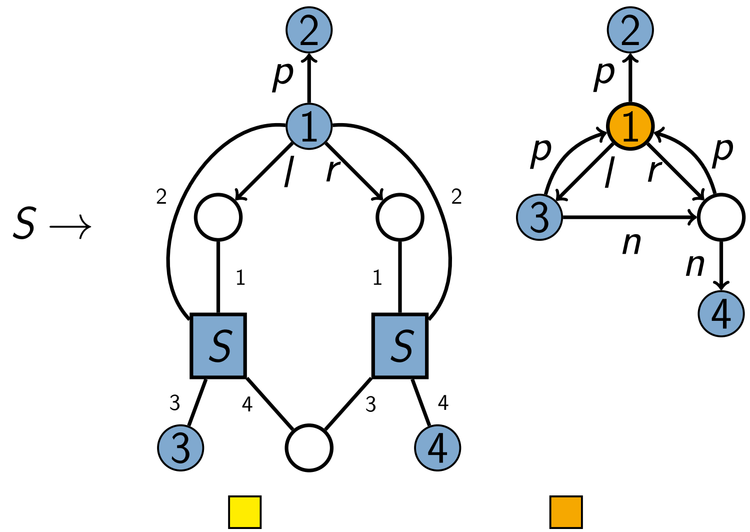
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



[1] Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

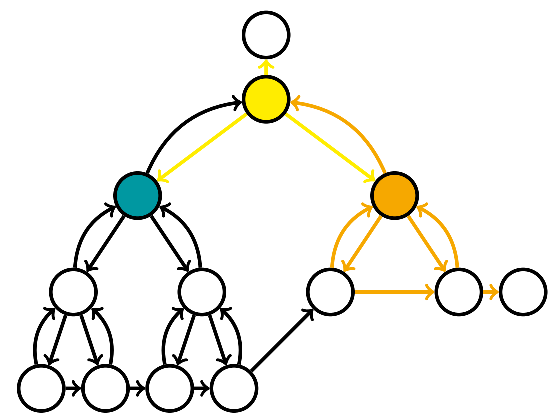
Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees



Derivation tree

- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



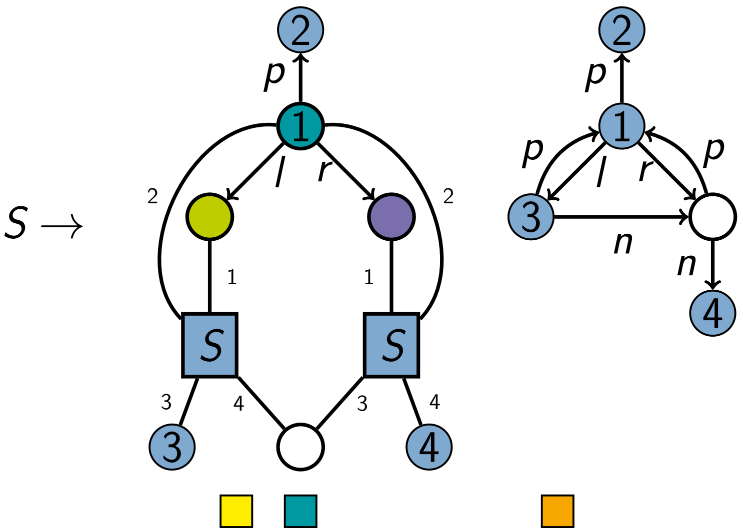
[1]Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

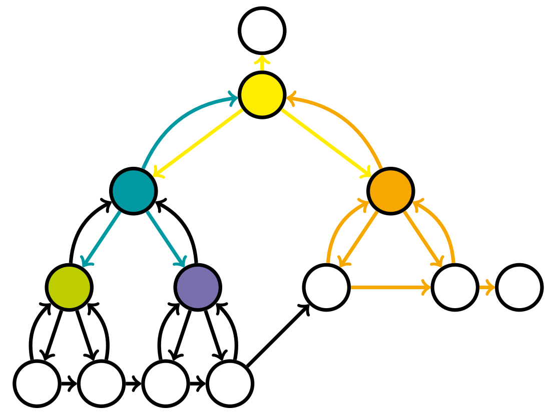
Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees

Derivation tree

- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$

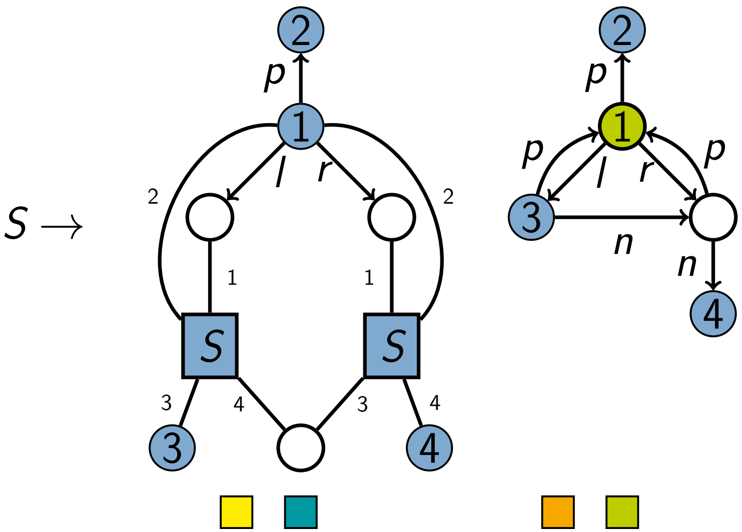




[1]Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

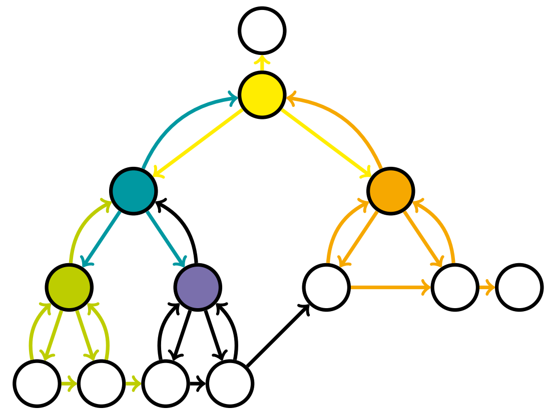
Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees



Derivation tree

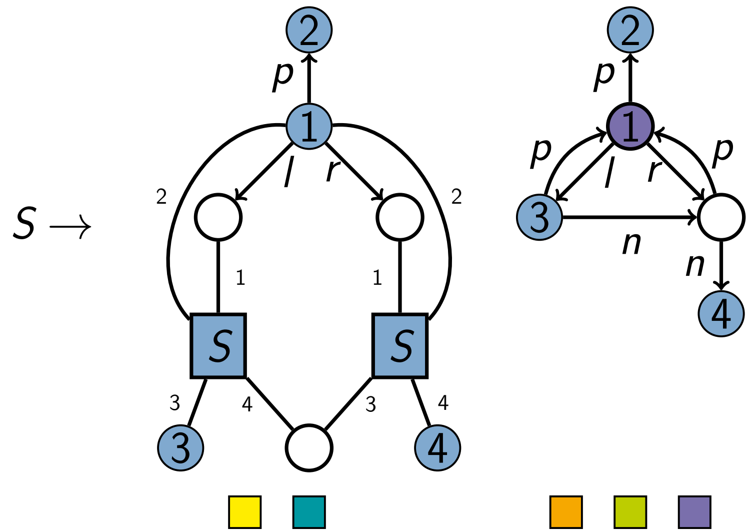
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



[1]Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

Towards MSO definable graph grammars

Courcelle[1]: *MSO* definable graph languages allow reconstruction of derivation trees

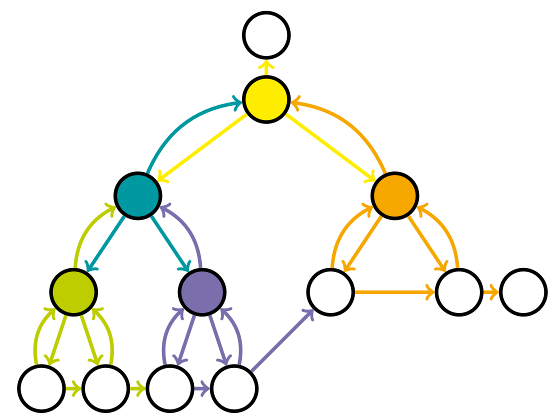


Derivation tree

- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$

MSO construction

1. Create witness for derivation of H by G
   i. Extract derivation tree t from H
   ii. Assign each edge to a node in t
2. $H \in L(G)$ iff witness specifies valid derivation of H by G



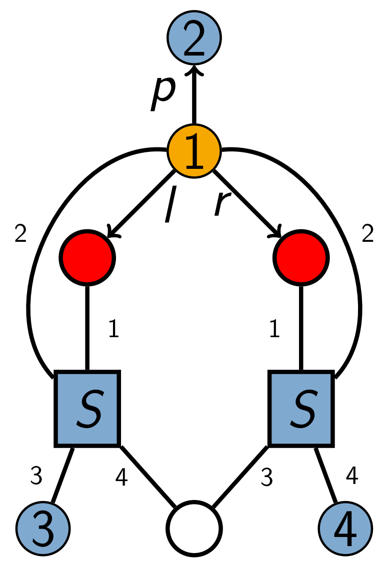
[1] Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

Tree-like hypergraphs

Definition

Hypergraph $H = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ is a tree-like hypergraph iff for each $e \in E$

1. $\mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$,
2. $\mathrm{lab}(e) \in N$ implies $\exists e' \,.\, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$.



anchor node $\mathrm{ext}(1)$

$\mathrm{att}(e)(1)$

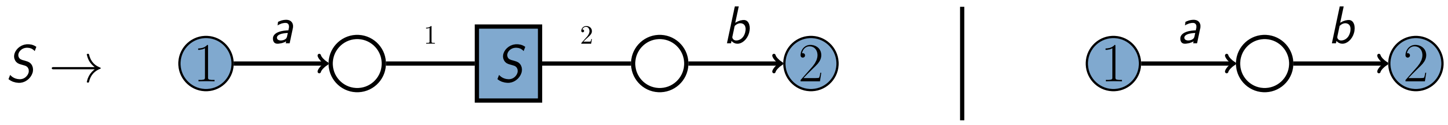
First approach: Every production rule maps to a tree-like hypergraph

Why tree-like hypergraphs?

$L = \{ a^n b^n \mid n \geq 1 \}$ is not *MSO* definable.

Why tree-like hypergraphs?

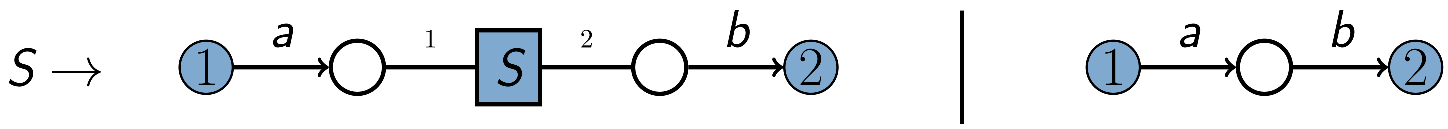
$L = \{ a^n b^n \mid n \geq 1 \}$ is not *MSO* definable.



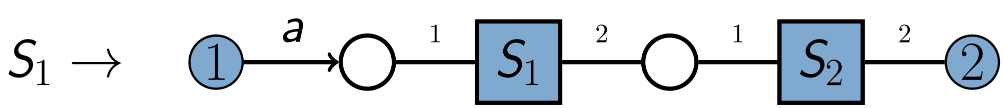
1. false $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. true $\quad \mathrm{lab}(e) \in N$ implies $\exists e' \,.\, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

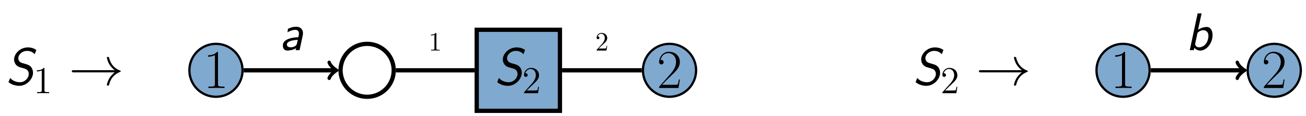
Why tree-like hypergraphs?

$L = \{ a^n b^n \mid n \geq 1 \}$ is not *MSO* definable.



1. false $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. true $\quad \mathrm{lab}(e) \in N$ implies $\exists e' \,.\, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$



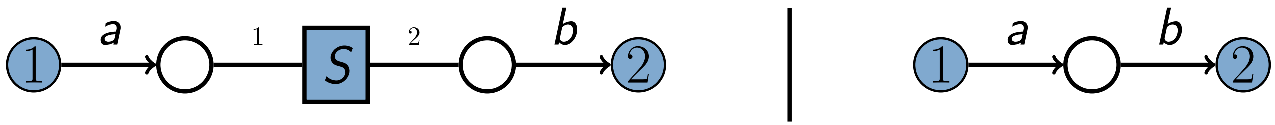


1. true $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. false $\quad \mathrm{lab}(e) \in N$ implies $\exists e' \,.\, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

Why tree-like hypergraphs?

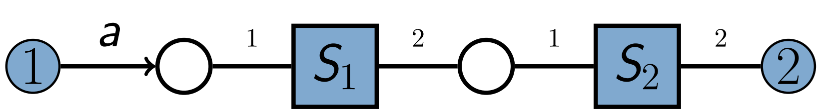
$L = \{ a^n b^n \mid n \geq 1 \}$ is not *MSO* definable.

$S \rightarrow$

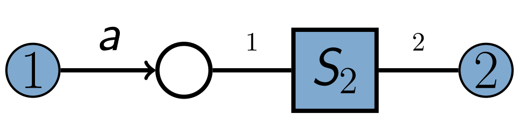


1. false $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. true $\quad \mathrm{lab}(e) \in N$ implies $\exists e' . \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

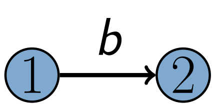
$S_1 \rightarrow$



$S_1 \rightarrow$



$S_2 \rightarrow$

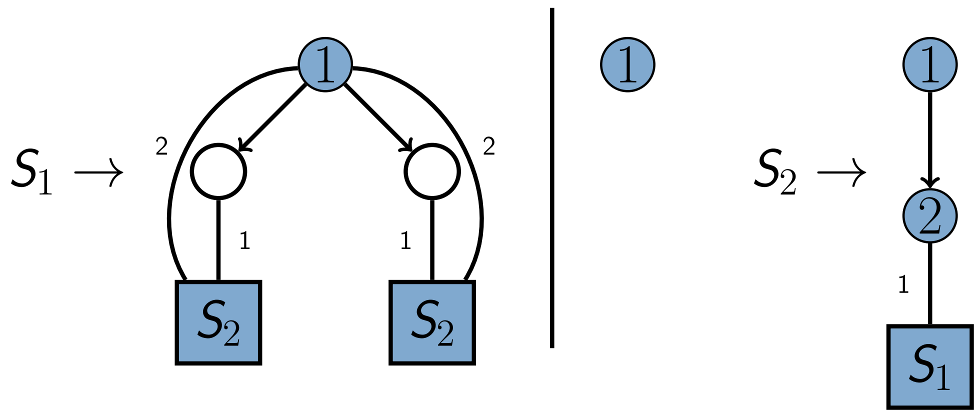


1. true $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. false $\quad \mathrm{lab}(e) \in N$ implies $\exists e' . \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

For context-free grammars our conditions yield right-linear grammars.

Tree-like hypergraphs are not enough

HRG

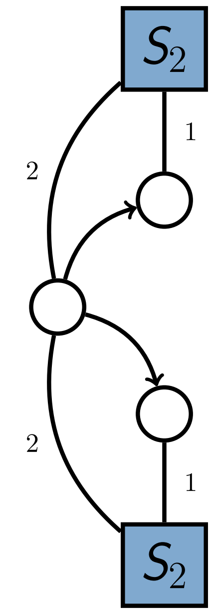


p *q* *r*

derivation tree

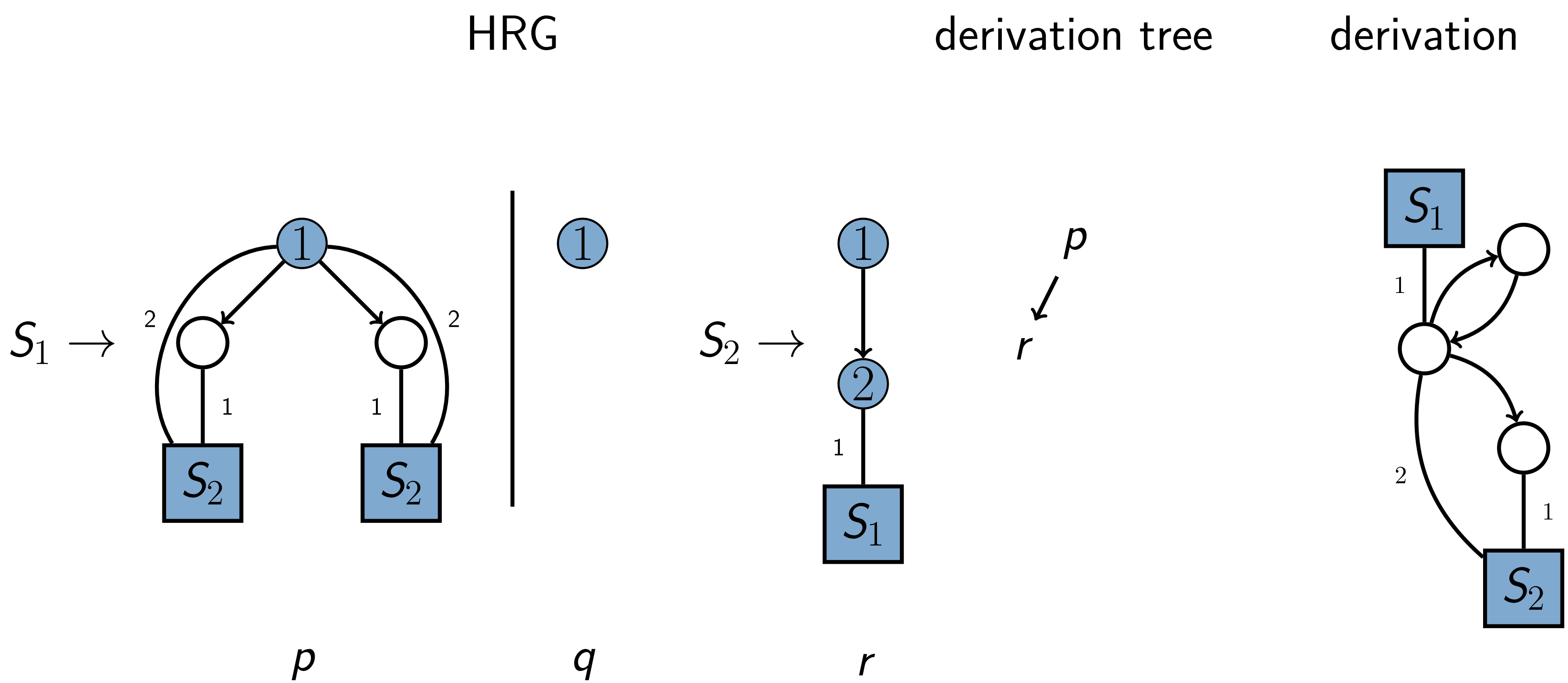
p

derivation



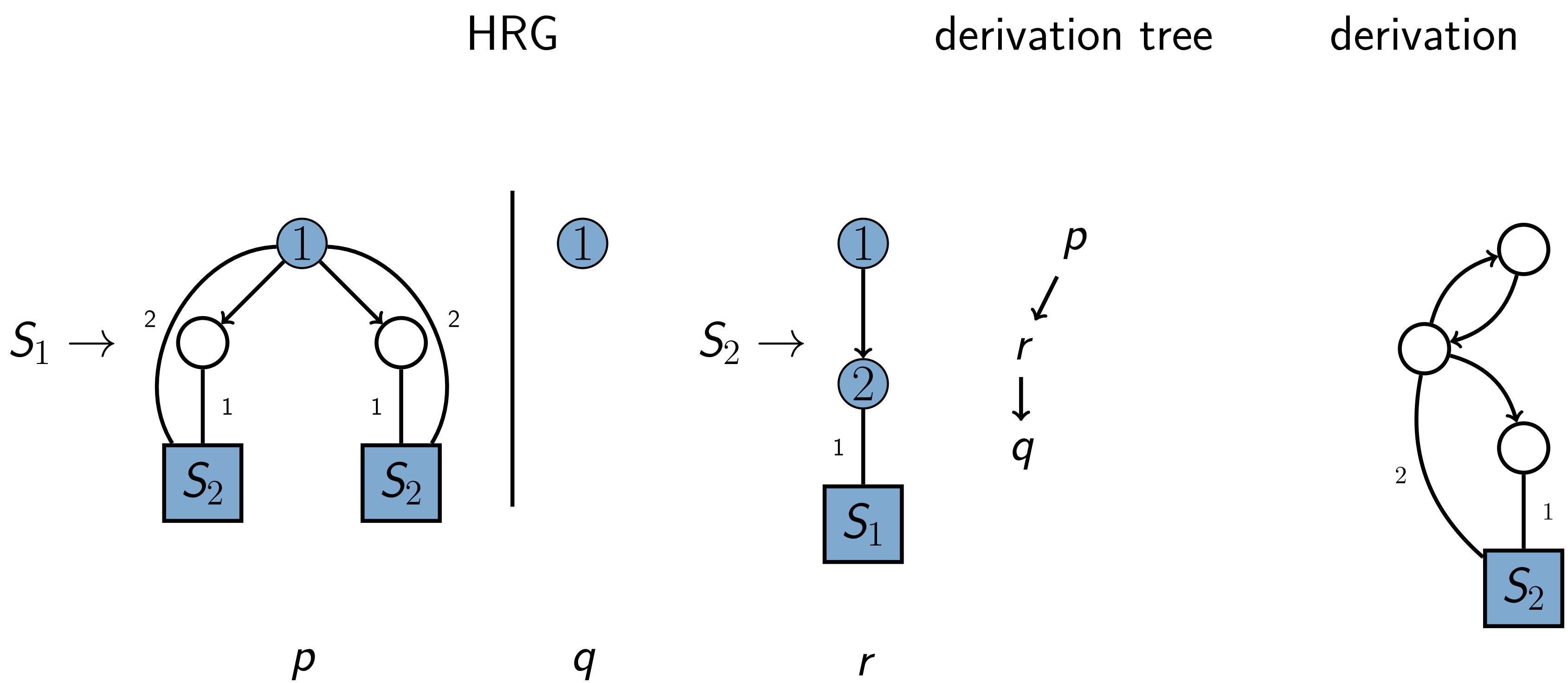
Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



Each production rule maps to a tree-like hypergraph.

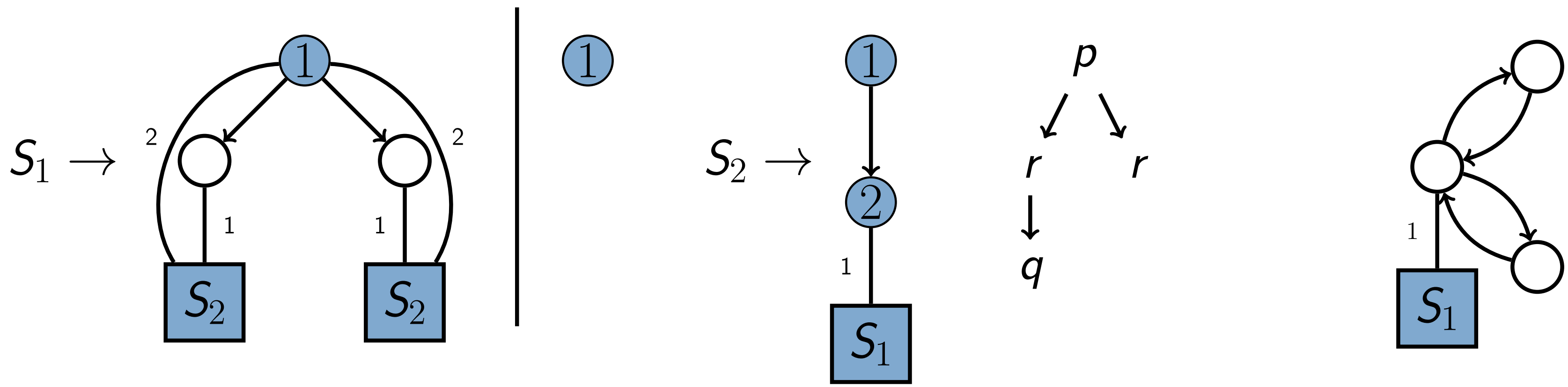
Tree-like hypergraphs are not enough



Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough

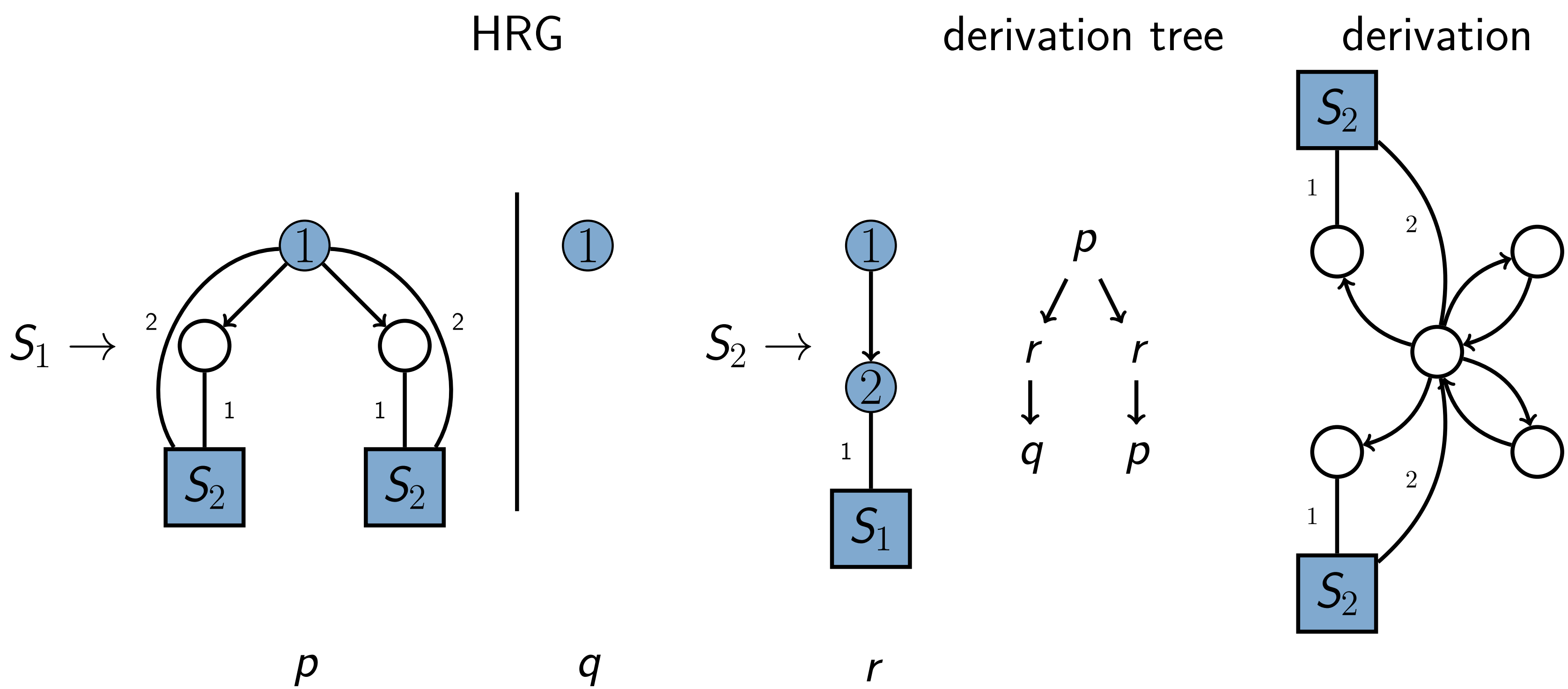
HRG derivation tree derivation



p q r

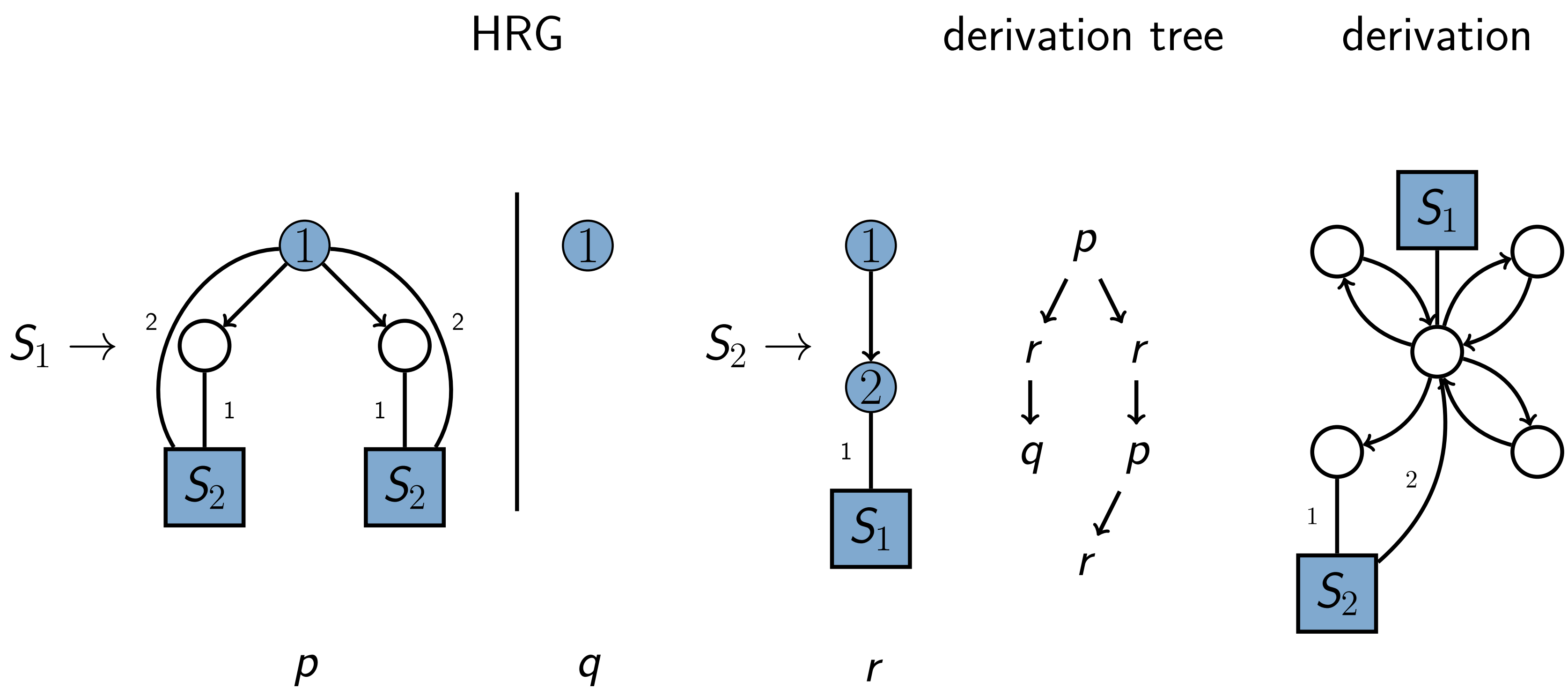
Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



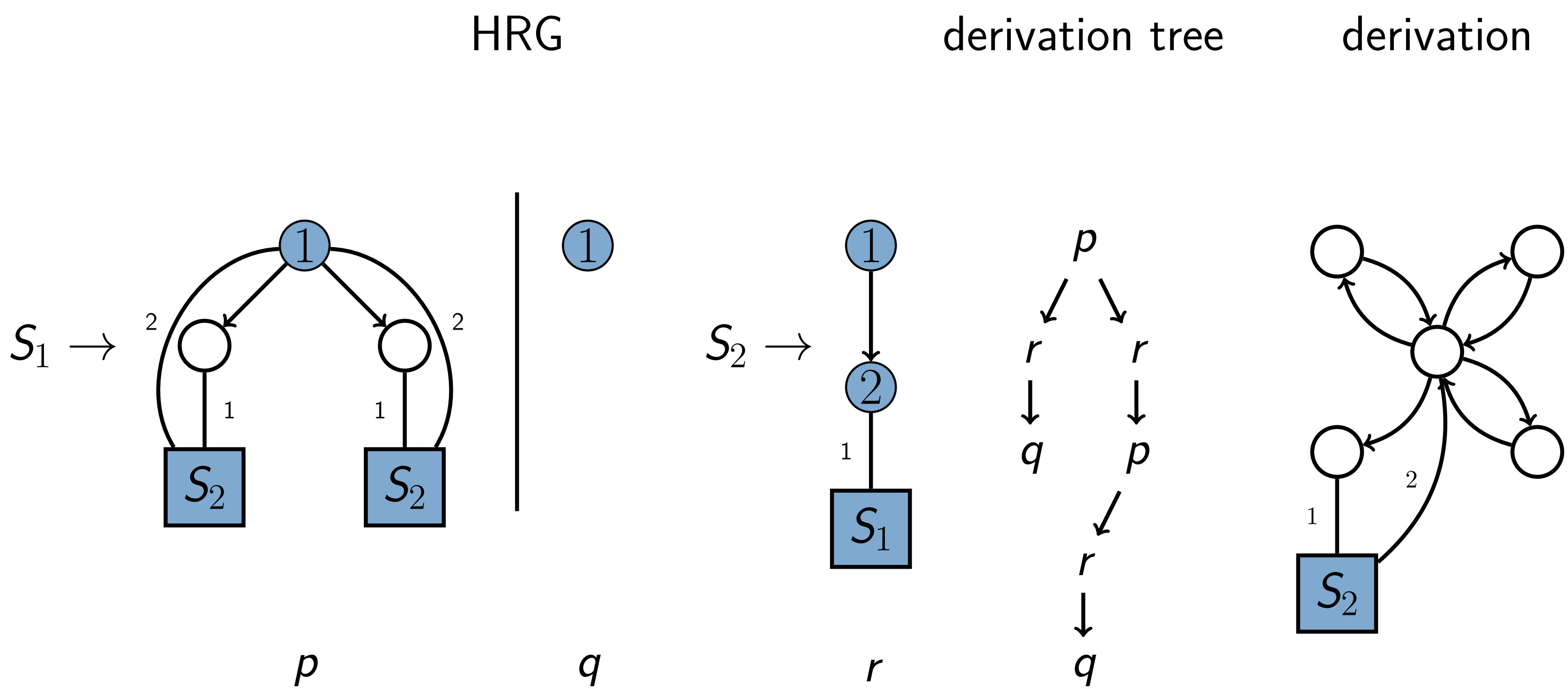
Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



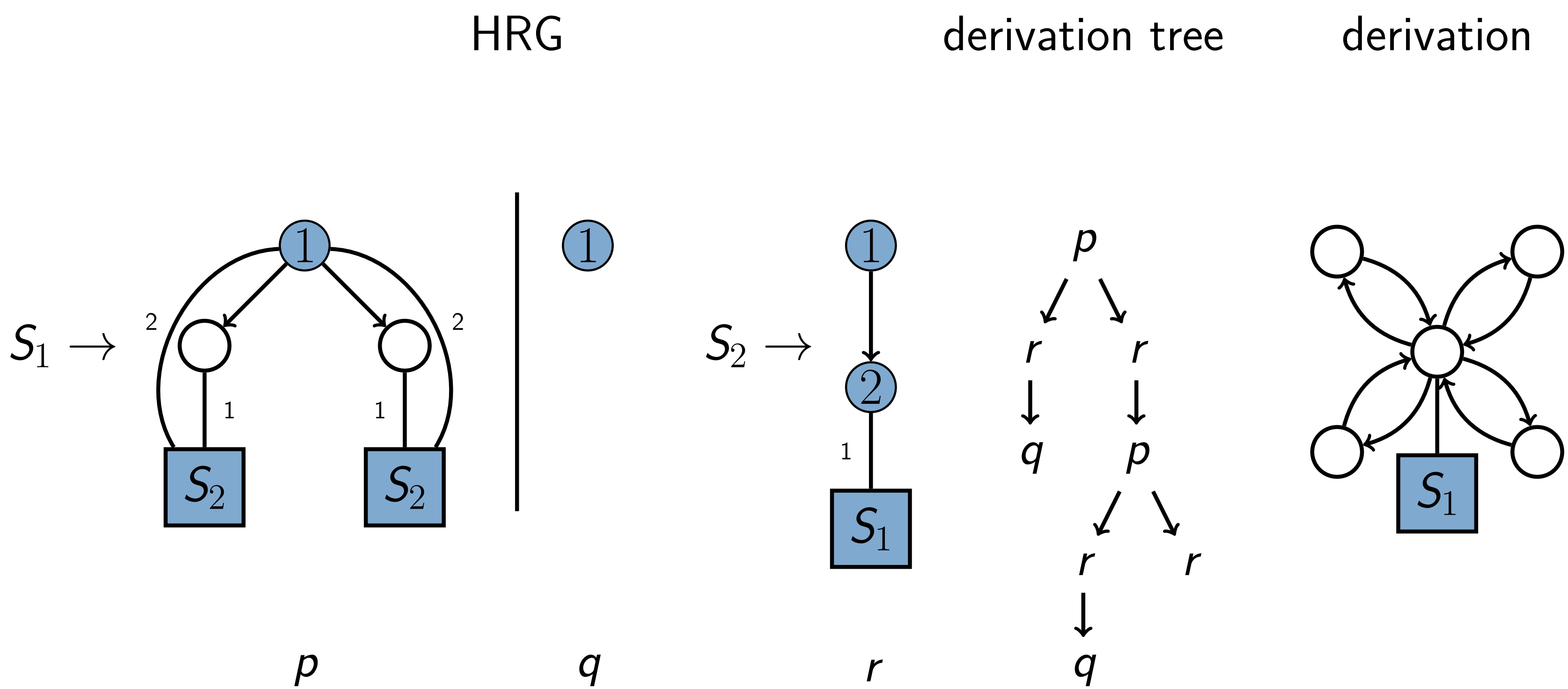
Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



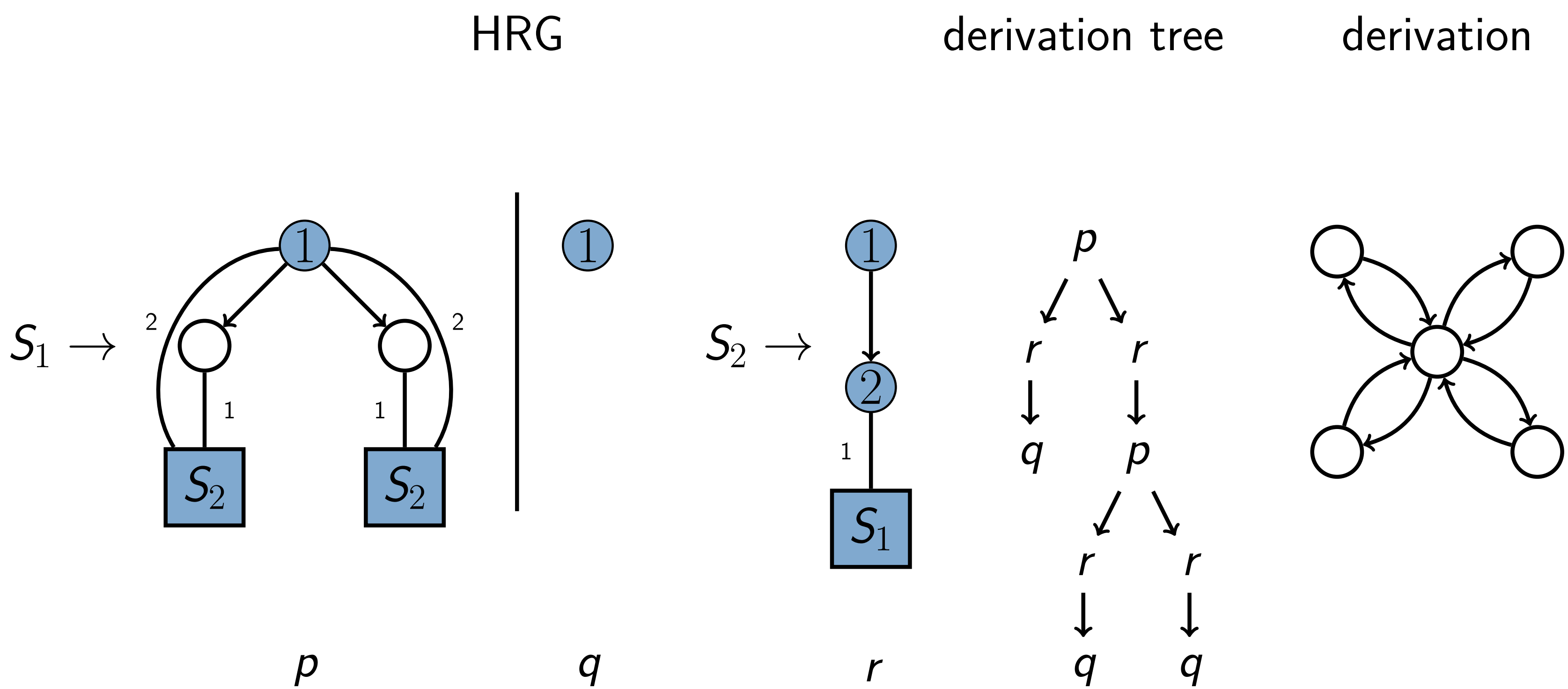
Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



Each production rule maps to a tree-like hypergraph.

Tree-like hypergraphs are not enough



Each production rule maps to a tree-like hypergraph.

Language of “even stars” is not *MSO* definable.

Observation: Anchor nodes are merged

Tree-like grammars

Let $\mathcal{M}(G) \triangleq \{H \in L(G) \mid \text{ two or more anchors are merged in a derivation of } H\}$.

Definition

A tree-like grammar is an HRG $G = (N, \Sigma, P, S)$ where

1. H is a tree-like hypergraph for each $(X, H) \in P$,
2. $\mathcal{M}(G) = \emptyset$.

Theorem

Let G be an HRG where each production rule maps to tree-like hypergraphs. Then one can construct a tree-like grammar K with $L(K) = L(G) \setminus \mathcal{M}(G)$.

Tree-like grammars

Theorem

For each tree-like grammar G there exists an MSO sentence φ_G such that for each hypergraph H

$$H \in L(G) \text{ if and only if } H \models \varphi_G.$$

Corollary

The class of languages generated by tree-like grammars is closed under union, intersection and difference.

Corollary

The inclusion problem for tree-like grammars is decidable.

Tree-like grammars

Theorem

For each tree-like grammar G there exists an MSO sentence φ_G such that for each hypergraph H

$$H \in L(G) \text{ if and only if } H \models \varphi_G.$$

Corollary

The class of languages generated by tree-like grammars is closed under union, intersection and difference.

Corollary

The inclusion problem for tree-like grammars is decidable.

What about separation logic?

Tree-like separation logic

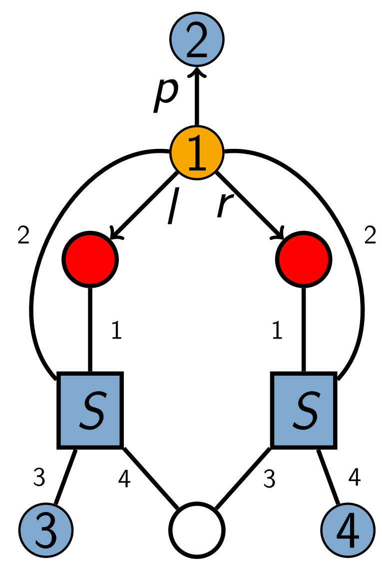
Let $PT(\varphi) \triangleq \{\{x, y\} \mid \exists s \in \Sigma \,.\, x.s \mapsto y \text{ occurs in } \varphi\}$.

Definition

Let $\varphi(\vec{x})$ be a separation logic formula. $\varphi(\vec{x})$ is tree-like iff

1. $x_1 \in A$ for each $A \in PT(\varphi)$,
2. there exists $A \in PT(\varphi)$ with $y_1 \in A$ for each predicate $P(\vec{y})$ in $\varphi(\vec{x})$.

$$
\begin{aligned}
S(x_1, x_2, x_3, x_4) \;&=\\
\exists y_1, y_2, y_3 \,.\, & x_1.l \mapsto y_1\\
&* \; x_1.r \mapsto y_2\\
&* \; x_1.p \mapsto x_2\\
&* \; x_1.n \mapsto \textbf{null}\\
&* \; S(y_1, x_1, x_3, y_3)\\
&* \; S(y_2, x_1, y_3, x_4)
\end{aligned}
$$



anchor node $\mathrm{ext}(1)$

$\mathrm{att}(e)(1)$

Tree-like separation logic

For $P(\vec{x}) = \varphi_1(\vec{x}) \vee \ldots \vee \varphi_n(\vec{x}) \in \Gamma$, let $\Gamma(P) = \{\varphi_1(\vec{x}), \ldots, \varphi_n(\vec{x})\}$.

Definition

Environment Γ is tree-like iff for each $P, Q \in \mathit{Pred}$

1. $\varphi(\vec{x})$ is tree-like for each $\varphi(\vec{x}) \in \Gamma(P)$.
2. $x_1 \neq y_1$ holds for each $\varphi(\vec{x}) \in \Gamma(P),\ \psi(\vec{y}) \in \Gamma(Q)$.

Tree-like separation logic

For $P(\vec{x}) = \varphi_1(\vec{x}) \vee \ldots \vee \varphi_n(\vec{x}) \in \Gamma$, let $\Gamma(P) = \{\varphi_1(\vec{x}), \ldots, \varphi_n(\vec{x})\}$.

Definition

Environment Γ is tree-like iff for each $P, Q \in \mathit{Pred}$

1. $\varphi(\vec{x})$ is tree-like for each $\varphi(\vec{x}) \in \Gamma(P)$.
2. $x_1 \neq y_1$ holds for each $\varphi(\vec{x}) \in \Gamma(P),\ \psi(\vec{y}) \in \Gamma(Q)$.

Theorem

Every tree-like separation logic formula can be translated into a language-equivalent tree-like data structure grammar and vice versa.

Corollary

The entailment problem for tree-like separation logic is decidable.

Tree-like separation logic

For $P(\vec{x}) = \varphi_1(\vec{x}) \vee \ldots \vee \varphi_n(\vec{x}) \in \Gamma$, let $\Gamma(P) = \{\varphi_1(\vec{x}), \ldots, \varphi_n(\vec{x})\}$.

Definition

Environment Γ is tree-like iff for each $P, Q \in Pred$

1. $\varphi(\vec{x})$ is tree-like for each $\varphi(\vec{x}) \in \Gamma(P)$.
2. $x_1 \neq y_1$ holds for each $\varphi(\vec{x}) \in \Gamma(P), \ \psi(\vec{y}) \in \Gamma(Q)$.

Theorem

Every tree-like separation logic formula can be translated into a language-equivalent tree-like data structure grammar and vice versa.

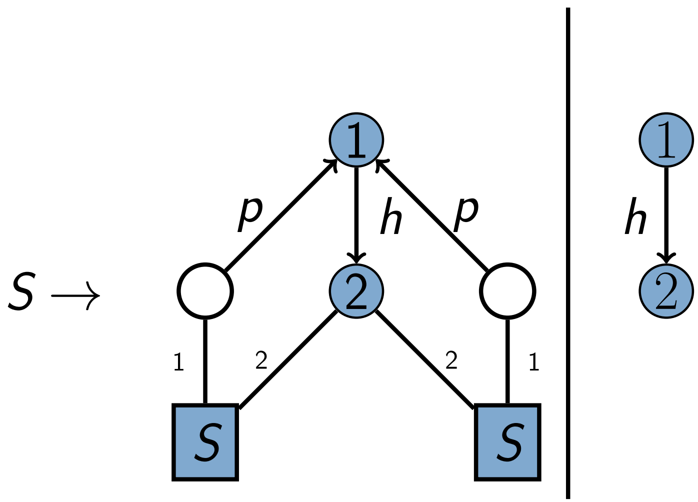
Corollary

The entailment problem for tree-like separation logic is decidable.

weak alternative to 2:
There exists $\emptyset \neq \Delta \subseteq \Sigma$ such that for each $\varphi(\vec{x}) \in \Gamma(P)$

$$\Delta \ \subseteq \ \{s \in \Sigma \ \mid \ x_1.s \mapsto y \text{ occurs in } \varphi(\vec{x}) \text{ for some } y\}.$$

Spaghetti stacks



$$
\begin{aligned}
S(x_1, x_2) \;=\; & \\
& \exists y_1, y_2 \,.\, x_1.h \mapsto x_2 \\
& \quad * \; y_1.p \mapsto x_1 \; * \; y_2.p \mapsto x_2 \\
& \quad * \; S(y_1, x_2) \\
& \quad * \; S(y_2, x_2) \\
& \vee \\
& x_1.h \mapsto x_2
\end{aligned}
$$

Theorem

Tree-like separation logic is strictly more expressive than separation logic with bounded tree width [3].

[3] Iosif, R. et al. "The tree width of separation logic with recursive definitions." CADE, 2013.

Conclusion

Wrap-up

- Close relationship between separation logic and data structure grammars
- (Extended) inclusion problem decidable for tree-like grammars
- (Extended) entailment problem decidable for tree-like separation logic
- Tree-like SL is more expressive than SL_{btw}

Future Work

- Complexity analysis?
- Tractable fragments of tree-like grammars?