Tree-like Grammars and Separation Logic

Christina Jansen Christoph Matheja Thomas Noll

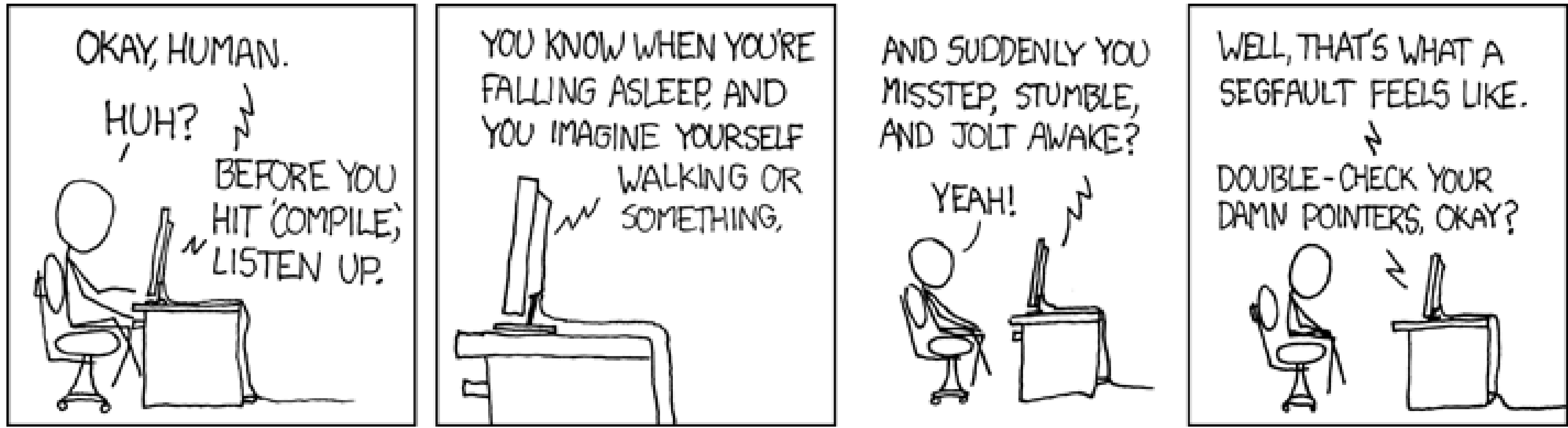
Software Modeling and Verification Group

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

RiSE seminar

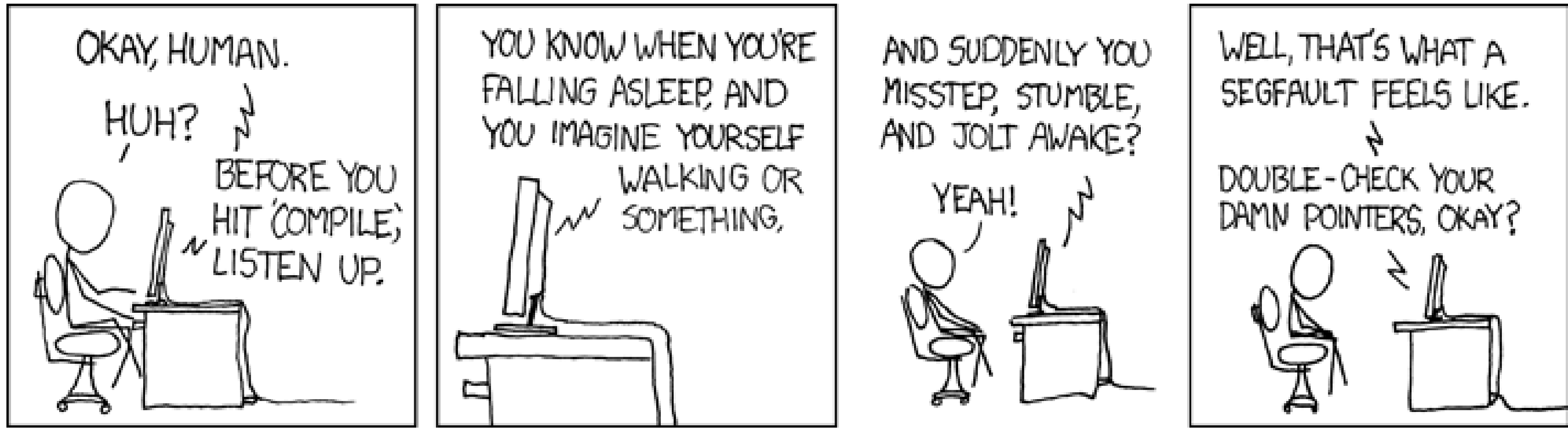
January 21, 2016; TU Wien

Motivation



https://xkcd.com/371

Motivation

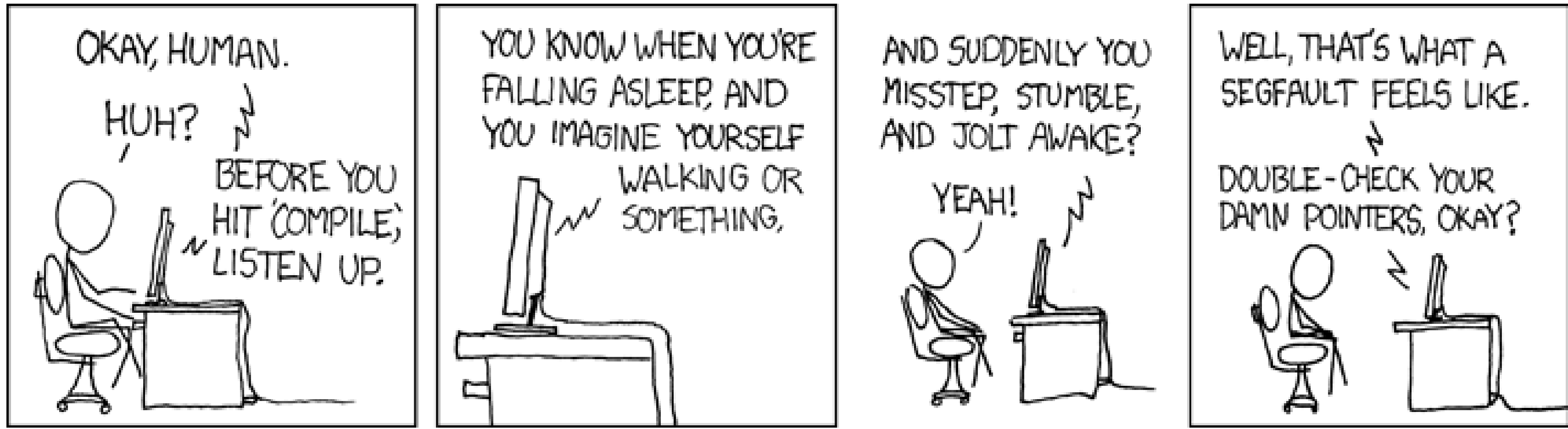


https://xkcd.com/371

Why Separation Logic?

- extension of Hoare-logic to reason about dynamic data structures
- Hoare-style proofs, shape analysis, symbolic execution...
- Cyclist, Infer, VeriFast, ...
- suffers from undecidable entailment problem

Motivation



https://xkcd.com/371

Why Separation Logic?

- extension of Hoare-logic to reason about dynamic data structures
- Hoare-style proofs, shape analysis, symbolic execution...
- Cyclist, Infer, VeriFast, ...
- suffers from undecidable entailment problem

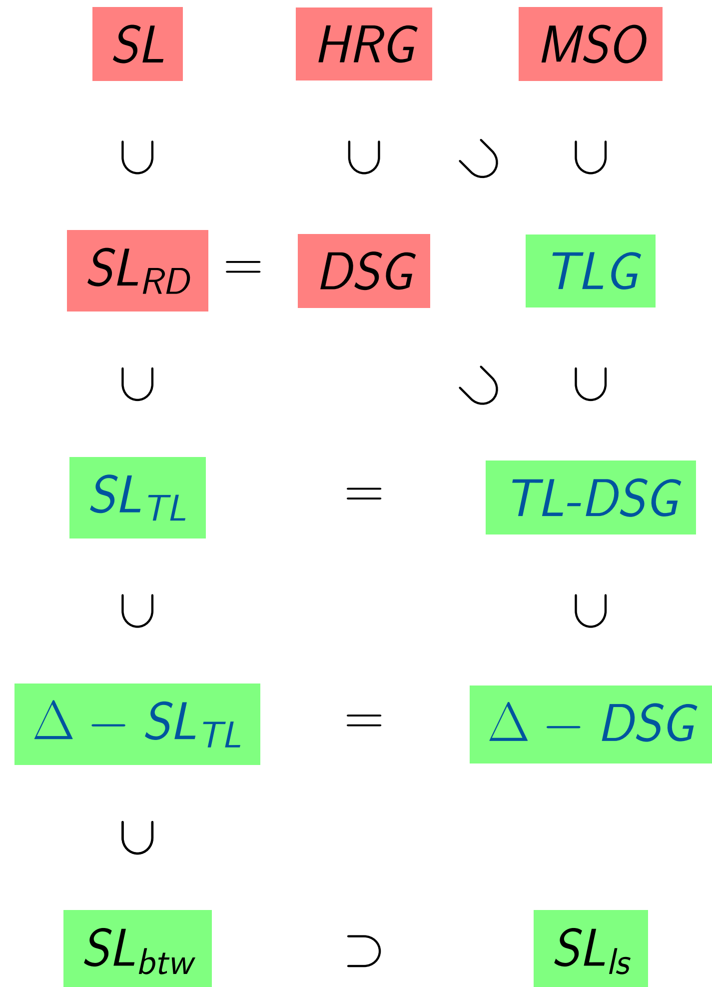
Why Graph Grammars?

- extension of context-free grammars to describe graphs
- program analysis, symbolic execution, natural language processing...
- Juggrnaut, Groove, ...
- suffers from undecidable inclusion problem

Motivation

How are these problems related?

What are decidable fragments?



- undecidable entailment problem
- decidable entailment problem
- new fragments

Outline

1. Introduction to Separation Logic

2. Symbolic Heaps with Recursive Definitions

3. Graph Grammars

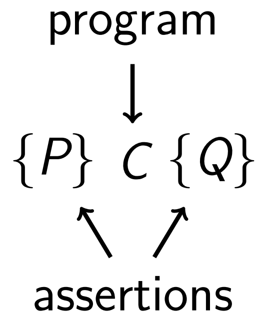
4. Tree-like Grammars

5. Tree-like Separation Logic

6. Conclusion

Hoare Logic

Classical program verification is based on Hoare triples:

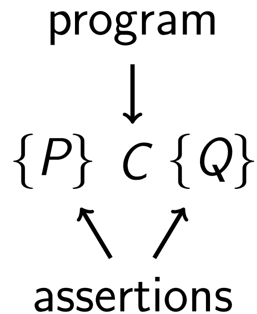


States: $\sigma : Var \dashrightarrow \mathbb{Z}$

$$\models \{P\}\ C\ \{Q\} \quad \Leftrightarrow \quad \forall \sigma \,.\, \sigma \models P \ \wedge\ \langle C, \sigma \rangle \rightarrow \sigma' \Rightarrow \sigma' \models Q$$

Hoare Logic

Classical program verification is based on Hoare triples:



States: $\sigma : Var \dashrightarrow \mathbb{Z}$

$$\models \{P\}\ C\ \{Q\} \quad \Leftrightarrow \quad \forall \sigma \,.\, \sigma \models P \,\wedge\, \langle C, \sigma \rangle \to \sigma' \Rightarrow \sigma' \models Q$$

Components

Programming language + semantics	(WHILE–language)
Assertion language + semantics	(first-order arithmetic)
Proof rules	(omitted)

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree}\, x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree}\, x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree}\, x \ \triangleq \ (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree}\, y * \mathrm{tree}\, z)$$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^{+}$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \ \triangleq \ (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P ::= emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x . P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \triangleq (emp \wedge x = 0) \vee (\exists y, z . x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {                                   {treex}
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}                                              {emp}
```

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \ \triangleq \ (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

$\{\mathrm{tree} x\}$

$\{\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z\}$

$\{emp\}$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \;::=\; emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree}\, x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree}\, x \;\triangleq\; (emp \wedge x = 0) \;\vee\; (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree}\, y * \mathrm{tree}\, z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

$\{\mathrm{tree}\, x\}$

$\{\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree}\, y * \mathrm{tree}\, z\}$

$\{\exists z \,.\, x \mapsto (l, z) * \mathrm{tree}\, l * \mathrm{tree}\, z\}$

$\{emp\}$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P ::= emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree}\,x \mid a < b \mid \exists x\,.\,P \mid \neg P \mid P \vee P$$

$$\mathrm{tree}\,x \triangleq (emp \wedge x = 0) \vee (\exists y, z\,.\,x \mapsto (y, z) * \mathrm{tree}\,y * \mathrm{tree}\,z)$$

```
cleanup(x) {                    {tree x}
  if (x != 0) {                 {∃y, z . x ↦ (y, z) * tree y * tree z}
    l := x.left;                {∃z . x ↦ (l, z) * tree l * tree z}
    r := x.right;               {x ↦ (l, r) * tree l * tree r}
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}                               {emp}
```

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree}\, x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree}\, x \ \triangleq \ (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree}\, y * \mathrm{tree}\, z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

$\{\mathrm{tree}\, x\}$

$\{\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree}\, y * \mathrm{tree}\, z\}$

$\{\exists z \,.\, x \mapsto (l, z) * \mathrm{tree}\, l * \mathrm{tree}\, z\}$

$\{x \mapsto (l, r) * \mathrm{tree}\, l * \mathrm{tree}\, r\}$

$\{x \mapsto (l, r) * emp * \mathrm{tree}\, r\}$

$\{emp\}$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \triangleq (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

$\{\mathrm{tree} x\}$

$\{\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z\}$

$\{\exists z \,.\, x \mapsto (l, z) * \mathrm{tree} l * \mathrm{tree} z\}$

$\{x \mapsto (l, r) * \mathrm{tree} l * \mathrm{tree} r\}$

$\{x \mapsto (l, r) * emp * \mathrm{tree} r\}$

$\{x \mapsto (l, r) * emp * emp\}$

$\{emp\}$

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \triangleq (emp \wedge x = 0) \ \vee \ (\exists y, z \,.\, x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {                    {treex}
  if (x != 0) {                 {∃y, z . x ↦ (y, z) * treey * treez}
    l := x.left;                {∃z . x ↦ (l, z) * treel * treez}
    r := x.right;               {x ↦ (l, r) * treel * treer}
    cleanup(x.left);            {x ↦ (l, r) * emp * treer}
    cleanup(x.right);           {x ↦ (l, r) * emp * emp}
    free(x);                    {emp * emp * emp}
  }
}                               {emp}
```

Separation Logic (with trees)

States: $\sigma = (s, h)$, stack $s : Var \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P ::= emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x . P \mid \neg P \mid P \vee P$$

$$\mathrm{tree} x \triangleq (emp \wedge x = 0) \vee (\exists y, z . x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z)$$

```
cleanup(x) {
  if (x != 0) {
    l := x.left;
    r := x.right;
    cleanup(x.left);
    cleanup(x.right);
    free(x);
  }
}
```

$\{\mathrm{tree} x\}$

$\{\exists y, z . x \mapsto (y, z) * \mathrm{tree} y * \mathrm{tree} z\}$

$\{\exists z . x \mapsto (l, z) * \mathrm{tree} l * \mathrm{tree} z\}$

$\{x \mapsto (l, r) * \mathrm{tree} l * \mathrm{tree} r\}$

$\{x \mapsto (l, r) * emp * \mathrm{tree} r\}$

$\{x \mapsto (l, r) * emp * emp\}$

$\{emp * emp * emp\}$

$\{emp\}$

$\{emp\}$

Local Reasoning

The proof relies on a frame property:

$$\frac{\{\mathrm{tree}\, l\}\, \mathit{cleanup}(l)\, \{\mathit{emp}\}}{\{\mathrm{tree}\, l * \mathrm{tree}\, r\}\, \mathit{cleanup}(l)\, \{\mathit{emp} * \mathrm{tree}\, r\}}$$

Local Reasoning

The proof relies on a frame property:

$$\frac{\{\mathrm{tree} l\} cleanup(l) \{emp\}}{\{\mathrm{tree} l * \mathrm{tree} r\} cleanup(l) \{emp * \mathrm{tree} r\}}$$

General frame rule:

$$[frame]\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \text{ if } Mod(C) \cap FV(R) = \emptyset$$

Local Reasoning

The proof relies on a frame property:

$$\frac{\{\mathrm{tree} \, l\} \, cleanup(l) \, \{emp\}}{\{\mathrm{tree} \, l * \mathrm{tree} \, r\} \, cleanup(l) \, \{emp * \mathrm{tree} \, r\}}$$

General frame rule:

$$[frame] \frac{\{P\} \; C \; \{Q\}}{\{P * R\} \; C \; \{Q * R\}} \text{ if } Mod(C) \cap FV(R) = \emptyset$$

But

$$\frac{\{x \mapsto (1,2)\} \; x.left := 2 \; \{x \mapsto (2,2)\}}{\{x \mapsto (1,2) \wedge y \mapsto (1,2)\} \; x.left := 2 \; \{x \mapsto (2,2) \wedge y \mapsto (1,2)\}}$$

is not valid if $x = y$.

Semantics

States: $\sigma = (s, h)$, stack $s : (Var \cup \mathbb{Z}) \dashrightarrow \mathbb{Z}$, heap $h : \mathbb{Z} \dashrightarrow \mathbb{Z}^+$

Assertions: $x \in Var$

$$P \ ::= \ emp \mid x \mapsto \vec{a} \mid P * P \mid \mathrm{tree} x \mid a < b \mid \exists x \,.\, P \mid \neg P \mid P \vee P$$

Semantics of Separation Logic

$$s, h \models emp \ \Leftrightarrow \ dom(h) = \emptyset$$

$$s, h \models x \mapsto \vec{a} \ \Leftrightarrow \ dom(h) = \{sx\}, \ h(sx) = s(\vec{a})$$

$$s, h \models P * Q \ \Leftrightarrow \ h = h_1 \uplus h_2 \text{ and } s, h_1 \models P \text{ and } s, h_2 \models Q$$

(h is partitioned into h_1, h_2)

$s, h \models \mathrm{tree} x$ via unfolding trees (later)

$$s, h \models \exists x \,.\, P \ \Leftrightarrow \ \exists a \in \mathbb{Z} \,.\, s[x \mapsto a], h \models P$$

...

Outline

1. Introduction to Separation Logic

2. Symbolic Heaps with Recursive Definitions

3. Graph Grammars

4. Tree-like Grammars

5. Tree-like Separation Logic

6. Conclusion

Symbolic Heaps with Recursive Definitions

Terms are variables $x, y, z, \ldots$ or the constant nil.

Spatial formulae Σ and pure formulae π are given by

$$\begin{aligned} \Sigma &::= \mathtt{emp} \mid x \mapsto \vec{a} \mid P\vec{a} \mid \Sigma * \Sigma \\ \pi &::= a = b \mid a \neq b \end{aligned}$$

where P is a predicate symbol and a, b are terms.

A symbolic heap is a formula $\varphi\vec{x} \triangleq \exists\vec{z} \,.\, \Sigma : \Pi$ where Π is a finite set of pure formulae.

Symbolic Heaps with Recursive Definitions

Terms are variables $x, y, z, \ldots$ or the constant nil.

Spatial formulae Σ and pure formulae π are given by

$$\begin{aligned} \Sigma &::= \mathtt{emp} \mid x \mapsto \vec{a} \mid P\vec{a} \mid \Sigma * \Sigma \\ \pi &::= a = b \mid a \neq b \end{aligned}$$

where P is a predicate symbol and a, b are terms.

A symbolic heap is a formula $\varphi\vec{x} \triangleq \exists\vec{z} \,.\, \Sigma : \Pi$ where Π is a finite set of pure formulae.

A system of recursive definitions Φ is a finite set of rules $P \Rightarrow \varphi\vec{x}$.

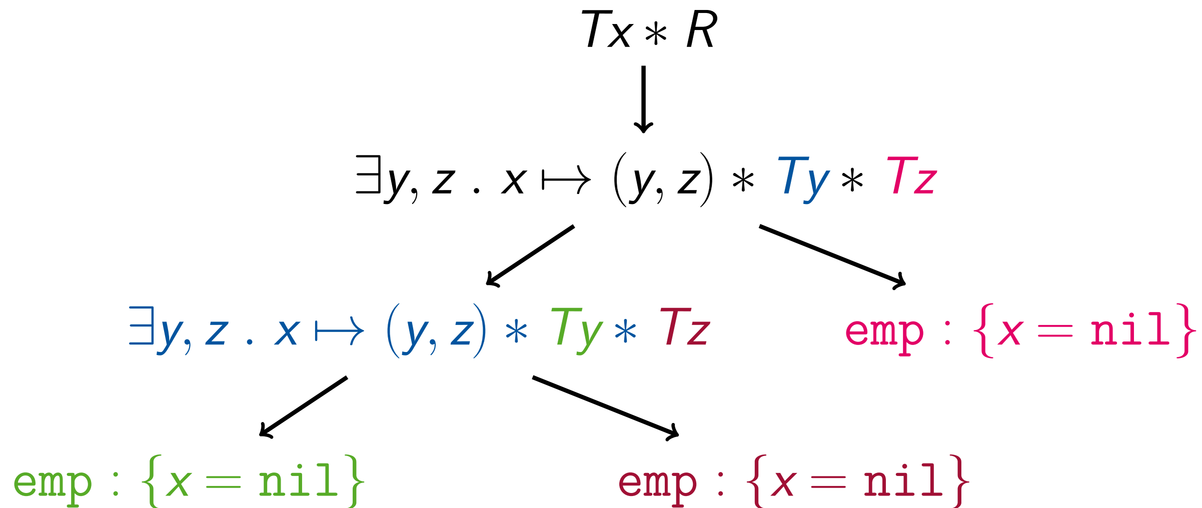
Example

$$T\,x \Rightarrow \mathtt{emp} : \{x = \mathtt{nil}\} \qquad LS\,x_1, x_2 \Rightarrow \mathtt{emp} : \{x_1 = \mathtt{nil}, x_2 = \mathtt{nil}\}$$

$$T\,x \Rightarrow \exists y, z \,.\, x \mapsto (y, z) * T\,y * T\,z \qquad LS\,x_1, x_2 \Rightarrow \exists y \,.\, x_1 \mapsto (y, x_2) * LS\,y, x_2$$

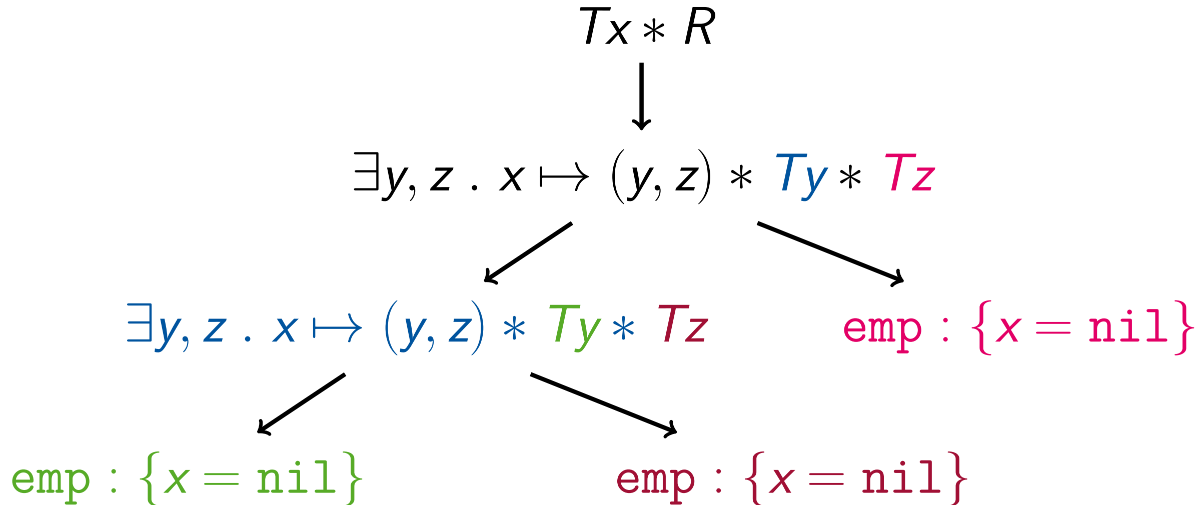
Semantics of Predicate Calls

An unfolding tree t captures an unrolling of predicates in a symbolic heap:



Semantics of Predicate Calls

An unfolding tree t captures an unrolling of predicates in a symbolic heap:



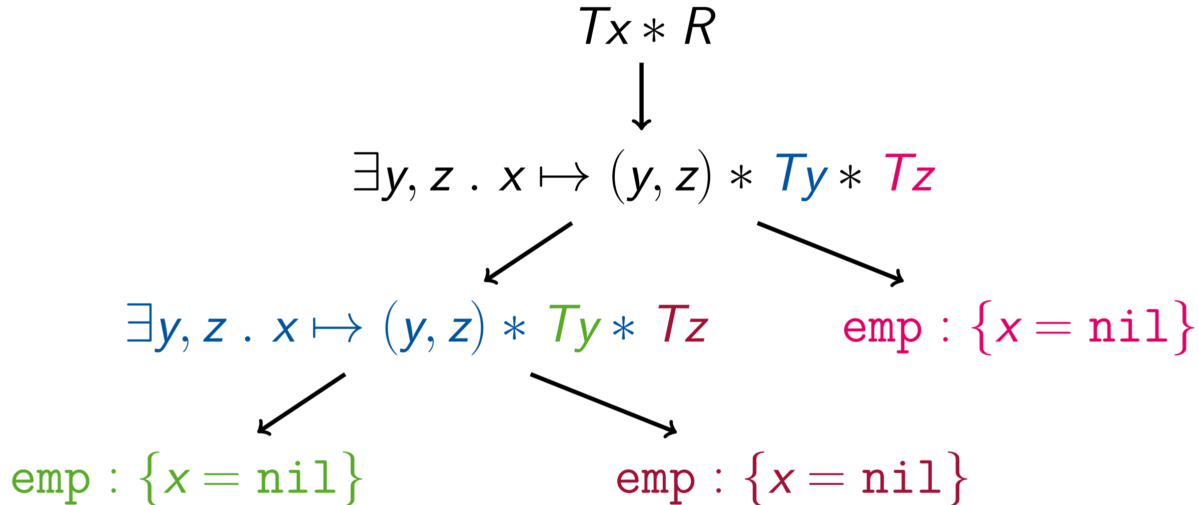
Corresponding unfolding $[\![t]\!]$:

$$\exists \vec{z} \,.\, x \mapsto (z_1, z_2) * z_1 \mapsto (z_3, z_4) * \mathtt{emp} * \mathtt{emp} * \mathtt{emp} * R \;:\; \{z_2 = z_3 = z_4 = \mathtt{nil}\}$$

$$\equiv \; \exists \vec{z} \,.\, x \mapsto (z_1, \mathtt{nil}) * z_1 \mapsto (\mathtt{emp}, \mathtt{emp}) * R$$

Semantics of Predicate Calls

An unfolding tree t captures an unrolling of predicates in a symbolic heap:



Corresponding unfolding $[\![t]\!]$:

$$\exists \vec{z} \,.\, x \mapsto (z_1, z_2) * z_1 \mapsto (z_3, z_4) * \mathtt{emp} * \mathtt{emp} * \mathtt{emp} * R \;:\; \{z_2 = z_3 = z_4 = \mathtt{nil}\}$$

$$\equiv\; \exists \vec{z} \,.\, x \mapsto (z_1, \mathtt{nil}) * z_1 \mapsto (\mathtt{emp}, \mathtt{emp}) * R$$

$$s, h \models_{\Phi} P\vec{x} \;\Leftrightarrow\; \exists \text{ unfolding tree } t \text{ of } P\vec{x} \,.\, s, h \models [\![t]\!]$$

Semantics of Predicate Calls

1. Introduction to Separation Logic

2. Symbolic Heaps with Recursive Definitions

3. Graph Grammars

4. Tree-like Grammars

5. Tree-like Separation Logic

6. Conclusion

From Heaps to Hypergraphs

$$h : \mathbb{Z} \dashrightarrow_{\text{finite}} \mathbb{Z}^{+}$$

$$u \mapsto (\texttt{nil}, v) \ * \ v \mapsto (u, w) \ * \ w \mapsto (v, x) \ * \ x \mapsto (w, \texttt{nil})$$

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

From Heaps to Hypergraphs

$$h : \mathbb{Z} \dashrightarrow_{\text{finite}} \mathbb{Z}^{+}$$

$$u \mapsto (\texttt{nil}, v) \ast v \mapsto (u, w) \ast w \mapsto (v, x) \ast x \mapsto (w, \texttt{nil})$$

object

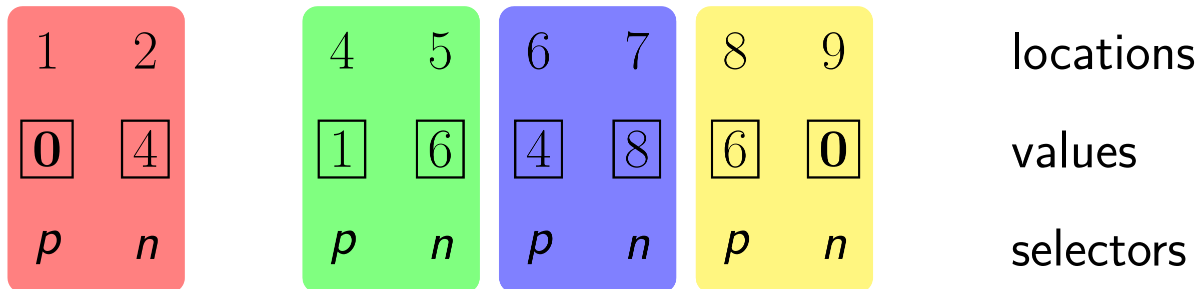
1	2		4	5	6	7	8	9	locations
0	4		1	6	4	8	6	**0**	values
p	n		p	n	p	n	p	n	selectors

From Heaps to Hypergraphs

$$h \; : \; \mathbb{Z} \dashrightarrow_{\text{finite}} \mathbb{Z}^{+}$$

$$u \mapsto (\texttt{nil}, v) \; * \; v \mapsto (u, w) \; * \; w \mapsto (v, x) \; * \; x \mapsto (w, \texttt{nil})$$

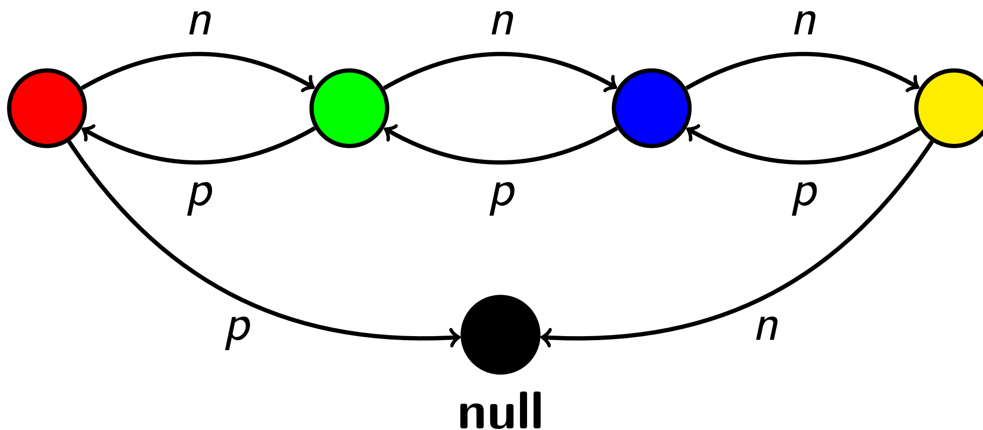
object



locations

values

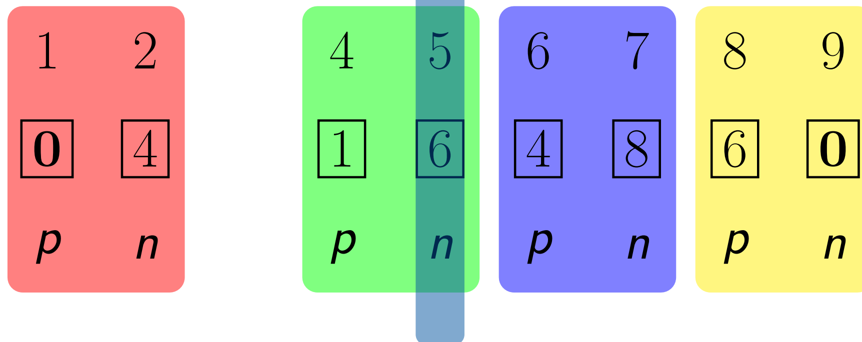
selectors



From Heaps to Hypergraphs

$$h \; : \; \mathbb{Z} \dashrightarrow_{\text{finite}} \mathbb{Z}^{+}$$

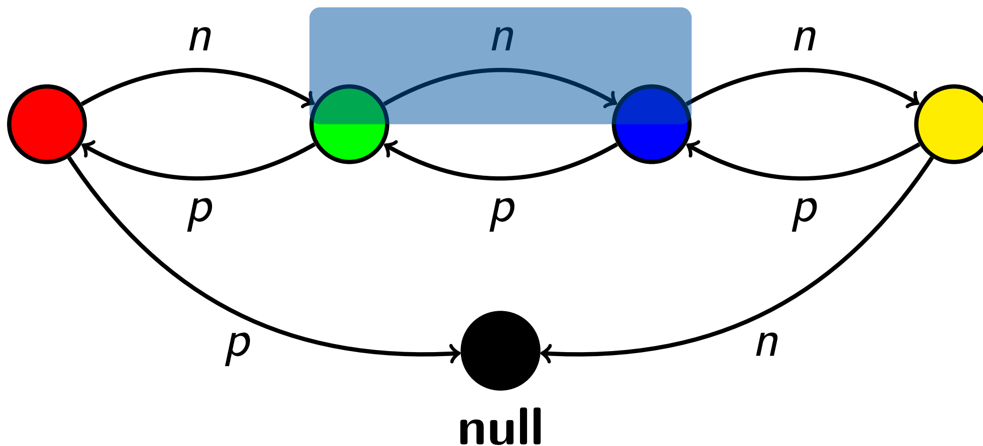
$$u \mapsto (\texttt{nil}, v) \; * \; v \mapsto (u, w) \; * \; w \mapsto (v, x) \; * \; x \mapsto (w, \texttt{nil})$$



locations

values

selectors



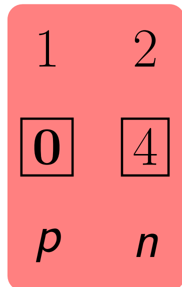
$4.n \mapsto 6$

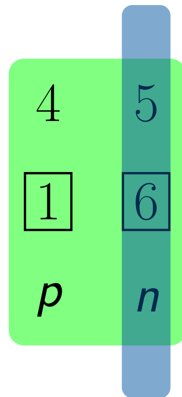
From Heaps to Hypergraphs

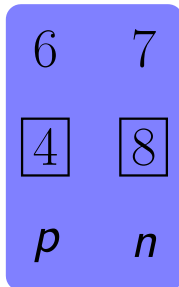
$$h : \mathbb{Z} \dashrightarrow_{\text{finite}} \mathbb{Z}^+$$

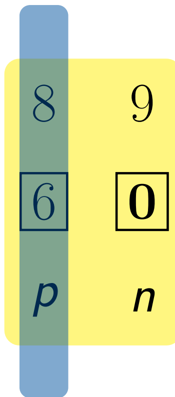
$$u \mapsto (\texttt{nil}, v) \ * \ v \mapsto (u, w) \ * \ w \mapsto (v, x) \ * \ x \mapsto (w, \texttt{nil})$$

object





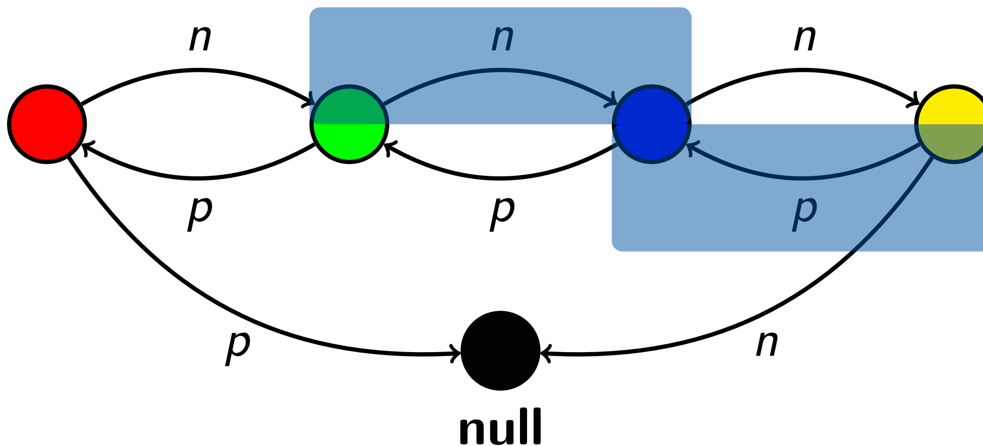




locations

values

selectors



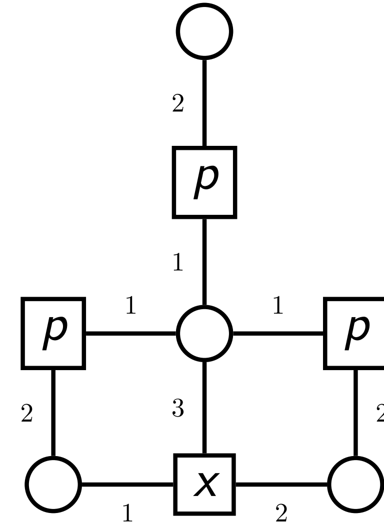
$$4.n \mapsto 6 \ * \ 8.p \mapsto 6$$

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 $\mathrm{ext} \in V^\star$.

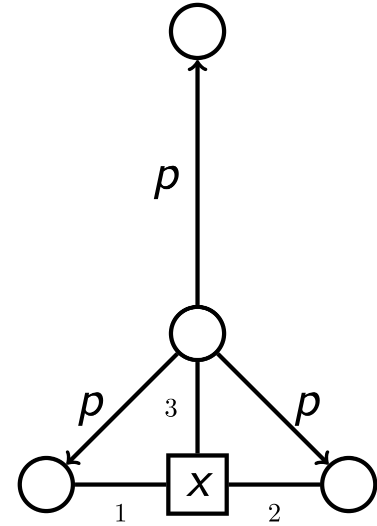


Graph Grammars in a Nutshell

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

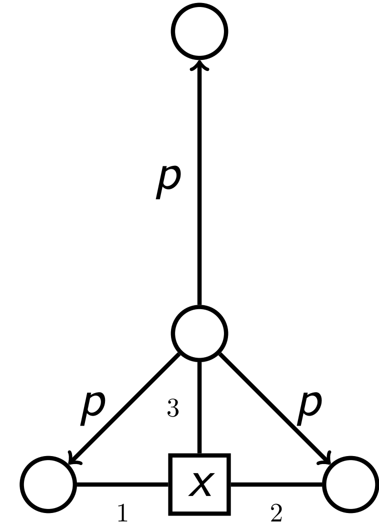
A hypergraph (HG) is a tuple $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with

- set of nodes V, set of hyperedges E,
- labelling $\mathrm{lab} : E \to \Sigma$, $\quad rk(e) = \mathrm{lab}(e)$,
- 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

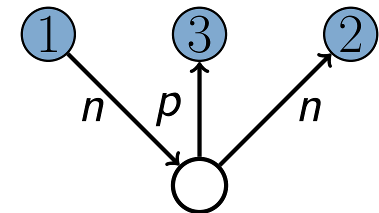
Σ	finite alphabet
$rk : \Sigma \to \mathbb{N}$	ranking function



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

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

Hyperedge replacement



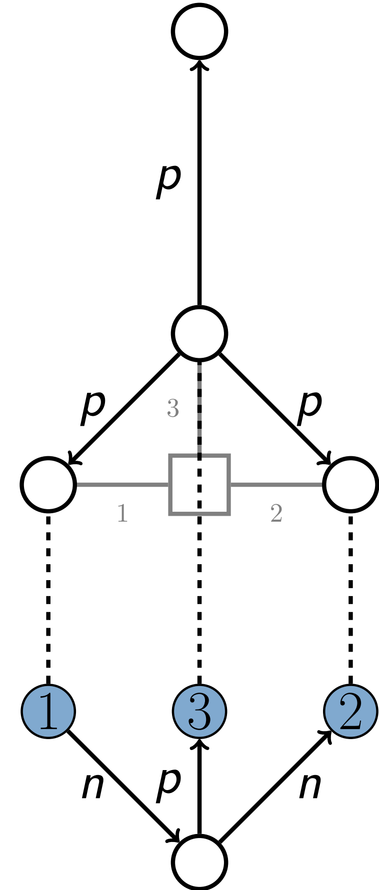
Graph Grammars in a Nutshell

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

Hyperedge replacement



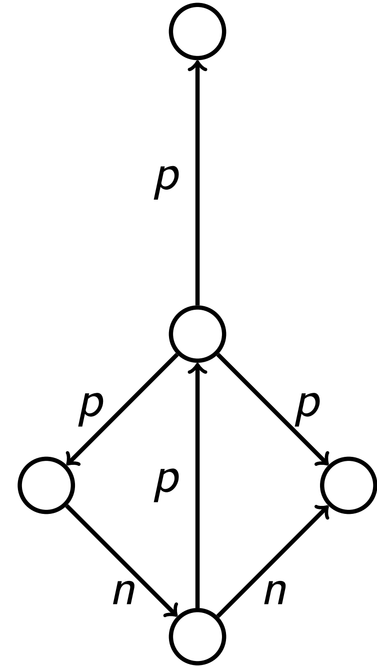
Graph Grammars in a Nutshell

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

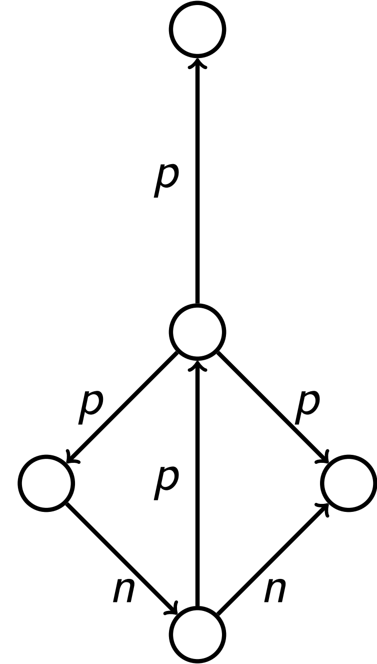


Graph Grammars in a Nutshell

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 $X \to H$ $rk(X) = |\mathrm{ext}_H|$,
- 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 $X \to H$ $\quad rk(X) = |\mathrm{ext}_H|$,
- 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.

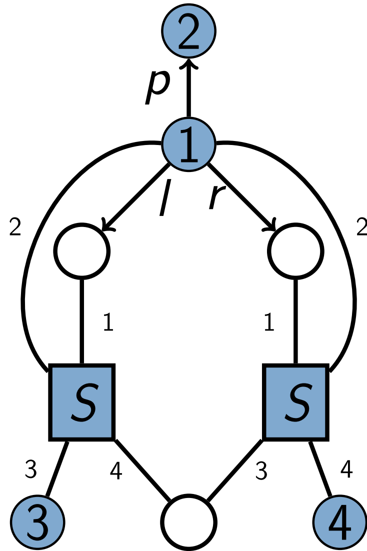
Theorem (APLAS, 2015)

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

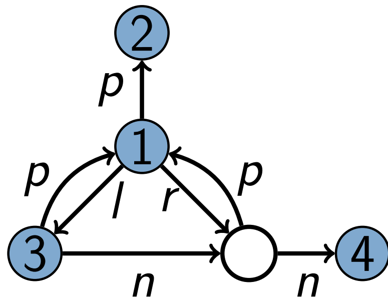
Data Structure Grammar for Trees with Linked Leaves

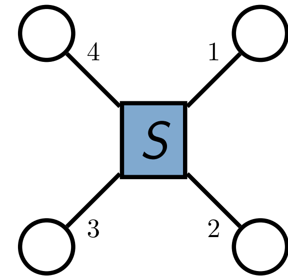
data structure grammar | derivation tree | derivation

$S \to S_1 \triangleq$



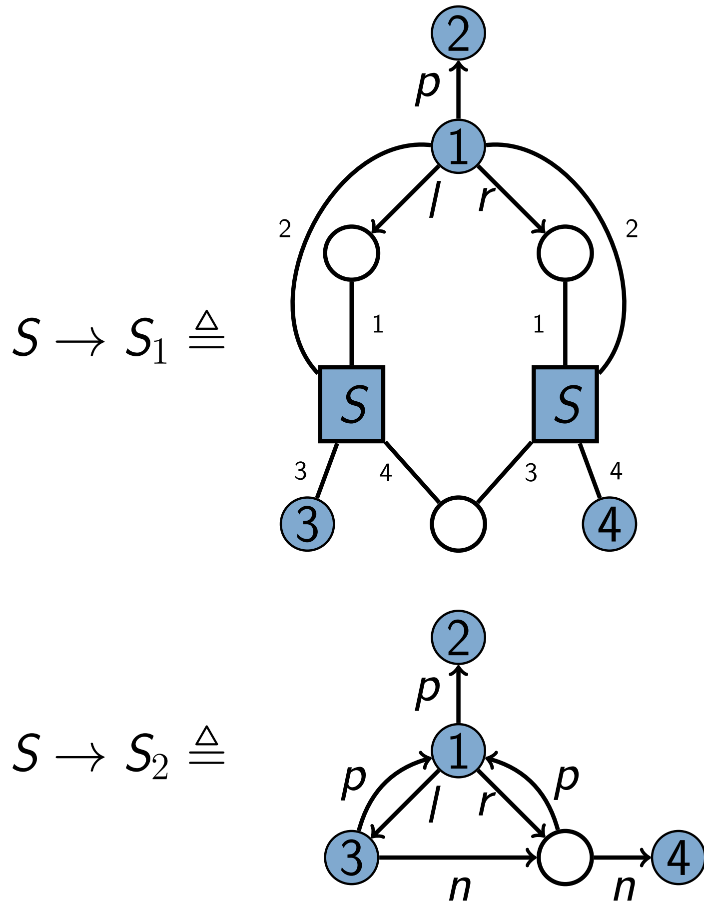
$S \to S_2 \triangleq$





Data Structure Grammar for Trees with Linked Leaves

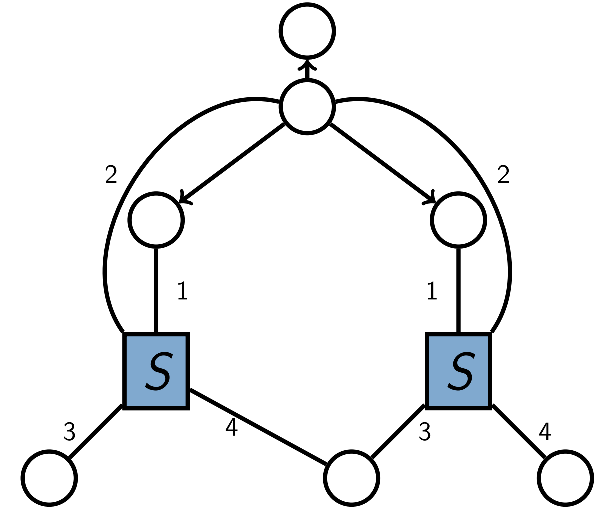
data structure grammar



derivation tree

S_1

derivation



Data Structure Grammar for Trees with Linked Leaves

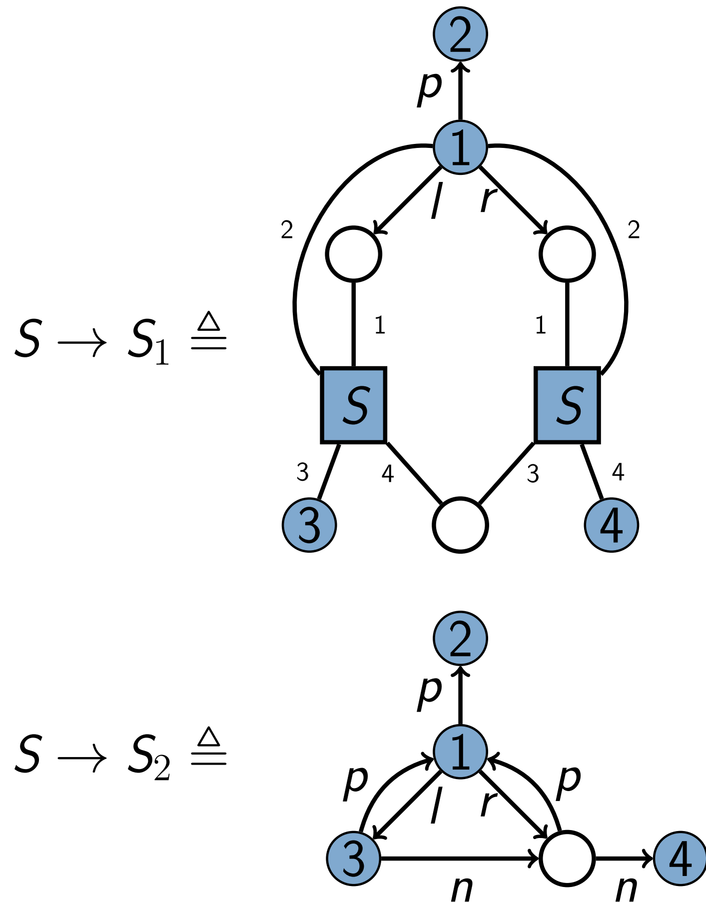
data structure grammar

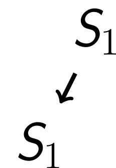
derivation tree

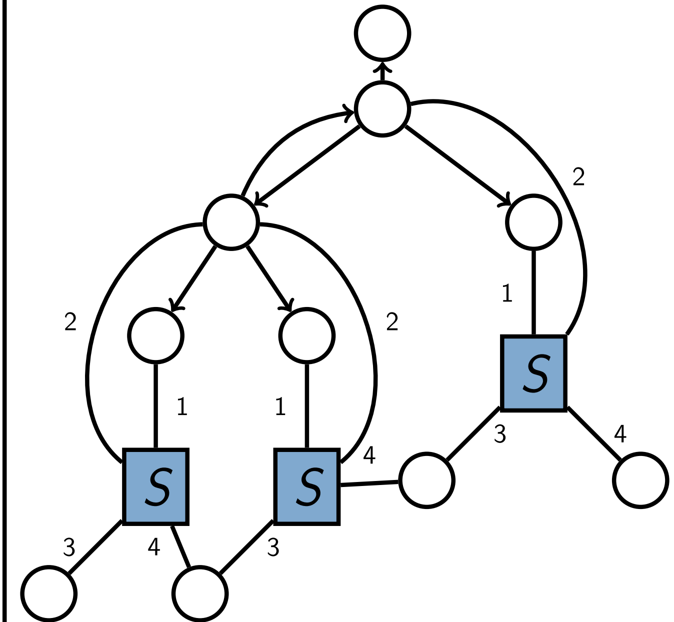
derivation

$S \to S_1 \triangleq$

$S \to S_2 \triangleq$





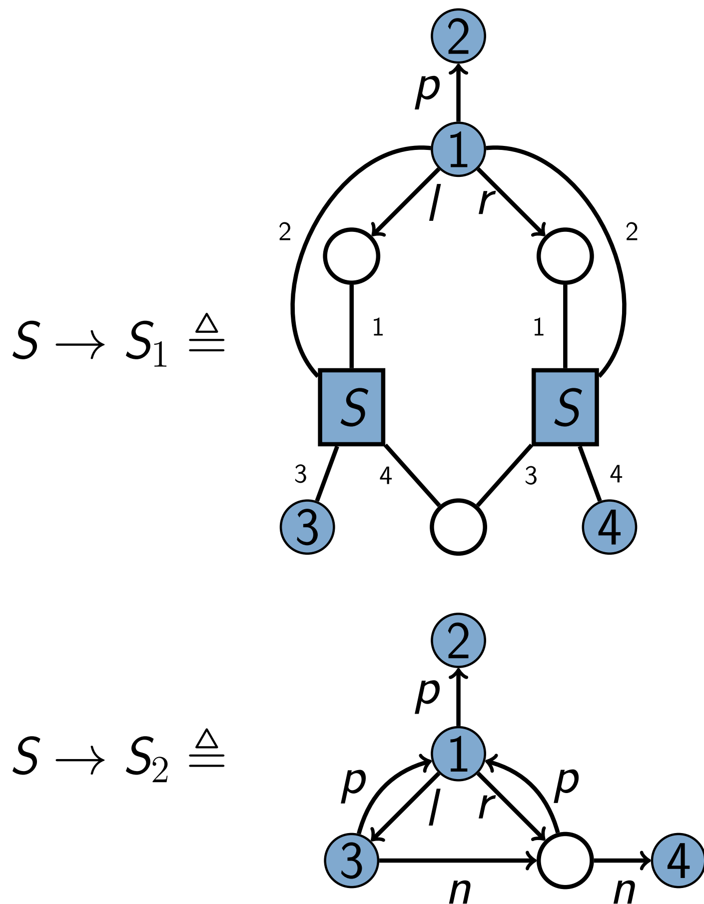


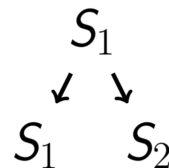
Data Structure Grammar for Trees with Linked Leaves

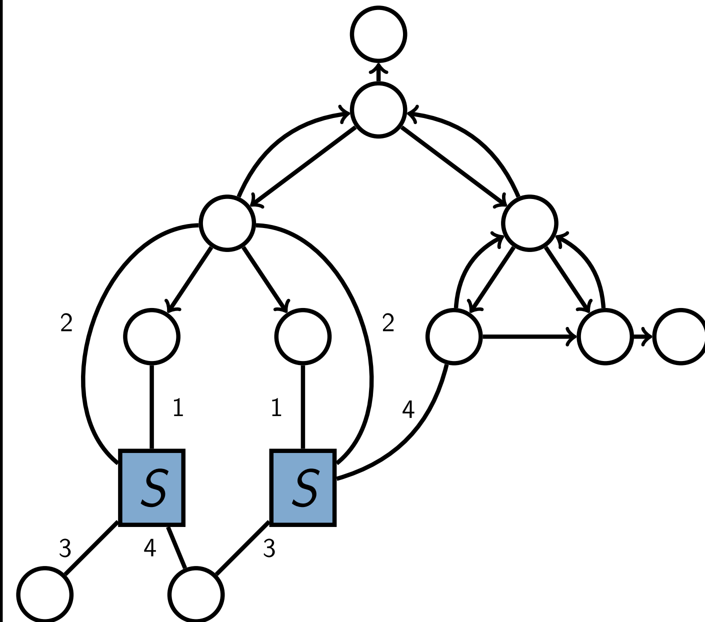
data structure grammar

derivation tree

derivation







Data Structure Grammar for Trees with Linked Leaves

data structure grammar | derivation tree | derivation

$S \to S_1 \triangleq$

$S \to S_2 \triangleq$

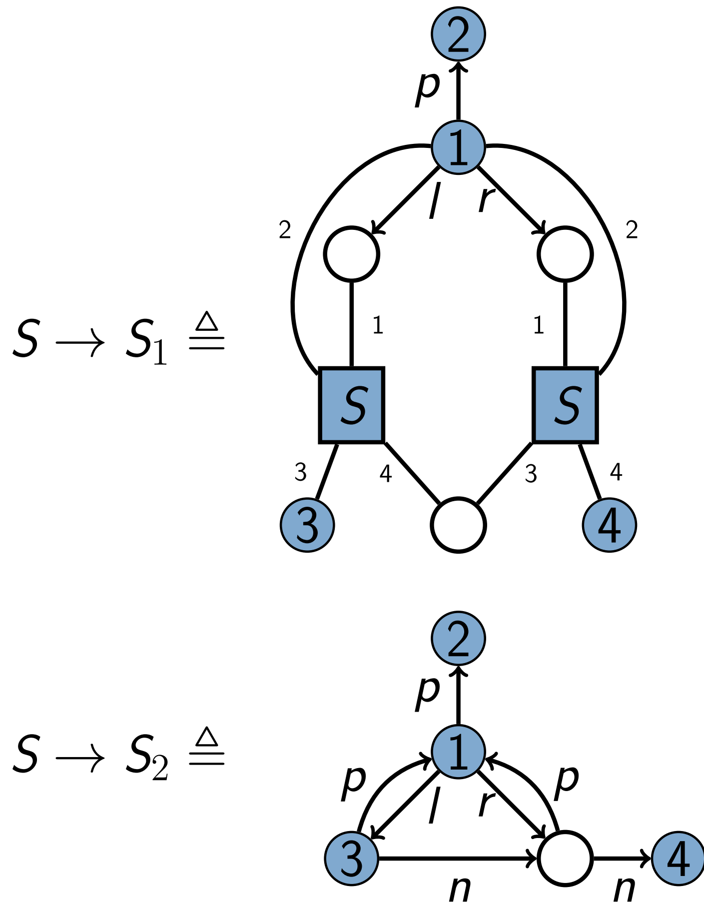
S_1
S_1 S_2
S_2

Data Structure Grammar for Trees with Linked Leaves

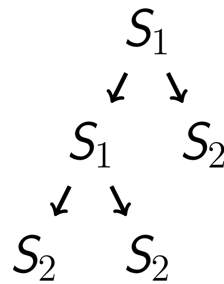
data structure grammar

$S \to S_1 \triangleq$

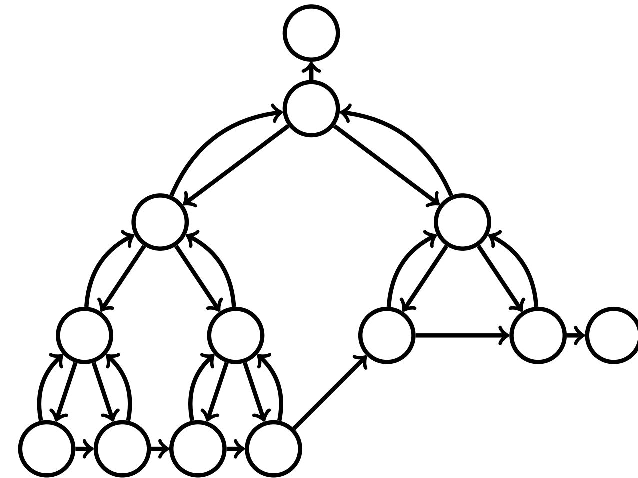
$S \to S_2 \triangleq$



derivation tree



derivation

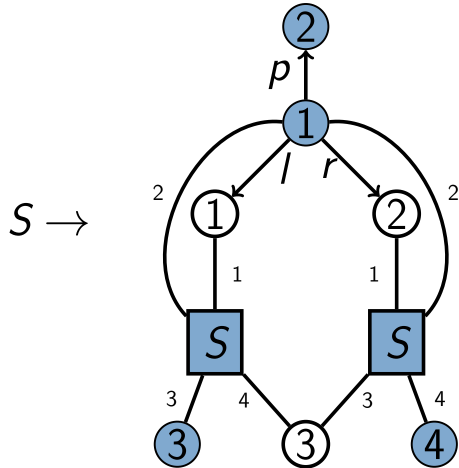


Separation Logic vs. Data Structure Grammars

Theorem (Jansen et al.[1])

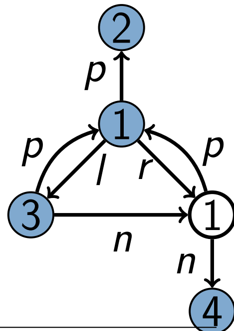
Every established system of recursive definitions can be translated into a language-equivalent data structure grammar and vice versa.

generalized memory model: $x.1 \mapsto y * x.2 \mapsto z$ instead of $x \mapsto (y, z)$



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

$$\exists y_1, y_2, y_3 \,.\, x_1 \mapsto (y_1, y_2, x_2, \texttt{nil}) * S(y_1, x_1, x_3, y_3) * S(y_2, x_1, y_3, x_4)$$



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

$$\exists y_1 \,.\, x_1 \mapsto (x_3, y_1, x_2, \texttt{nil}) * x_3 \mapsto (\texttt{nil}, \texttt{nil}, x_1, y_1) * y_1 \mapsto (\texttt{nil}, \texttt{nil}, x_1, x_4)$$

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

Outline

1. Introduction to Separation Logic

2. Symbolic Heaps with Recursive Definitions

3. Graph Grammars

4. Tree-like Grammars

5. Tree-like Separation Logic

6. Conclusion

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

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

Assume there exists *MSO* sentence φ with $L(K) = L(\varphi)$.

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

[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

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

Assume there exists *MSO* sentence φ with $L(K) = L(\varphi)$.

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

G is an arbitrary hyperedge replacement grammar!

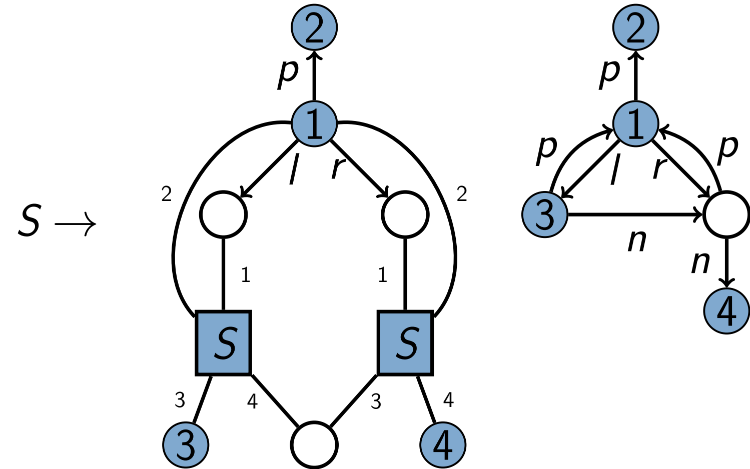
[2] Courcelle, B. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs." Information and computation, 1990.

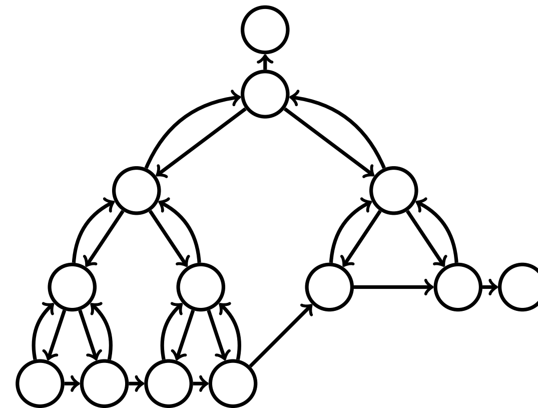
Towards MSO Definable Graph Grammars

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 $\mathrm{lab}(e) \in N$

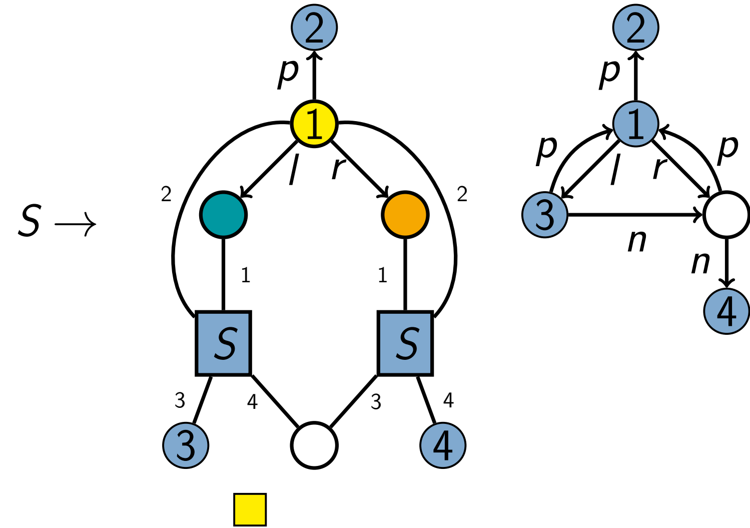




[1] Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

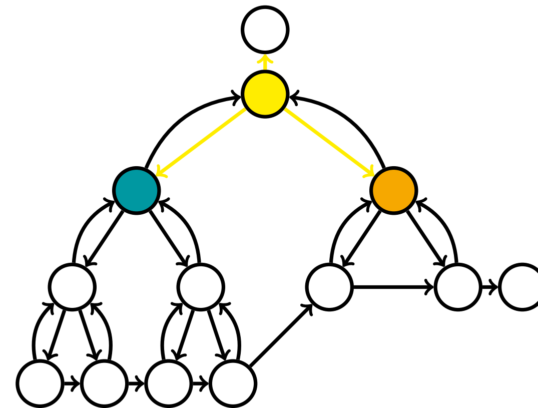
Towards MSO Definable Graph Grammars

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



Derivation tree

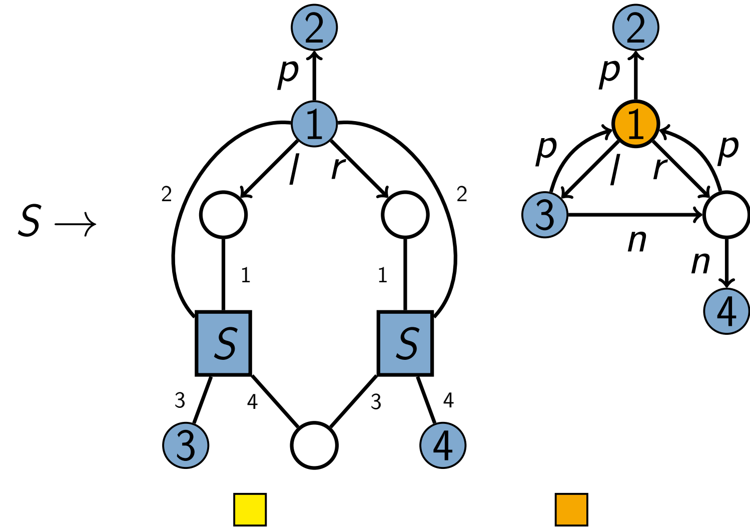
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



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

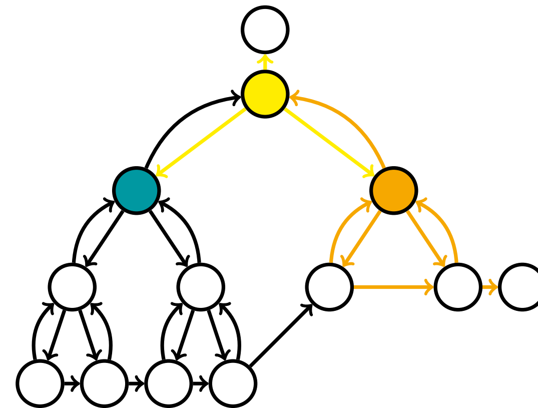
Towards MSO Definable Graph Grammars

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

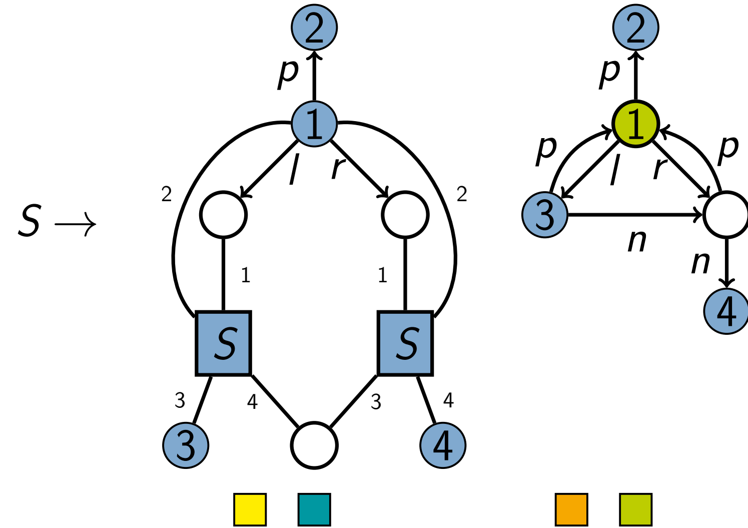
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$

$S \rightarrow$

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

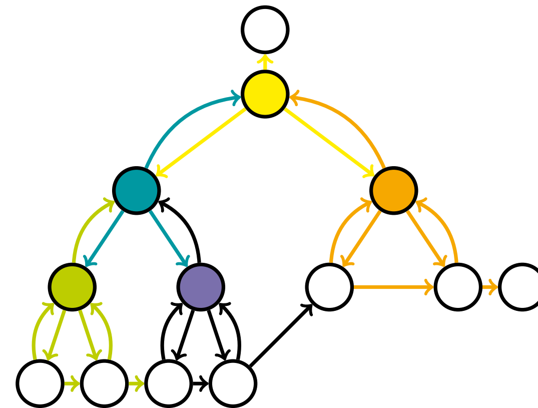
Towards MSO Definable Graph Grammars

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



Derivation tree

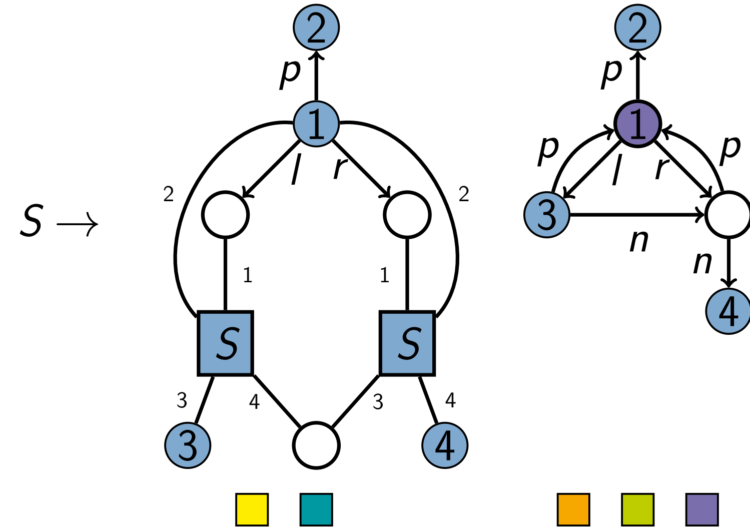
- Nodes: all anchor nodes $\mathrm{ext}(1)$
- Children: $\mathrm{att}(e)(1)$ if $\mathrm{lab}(e) \in N$



[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

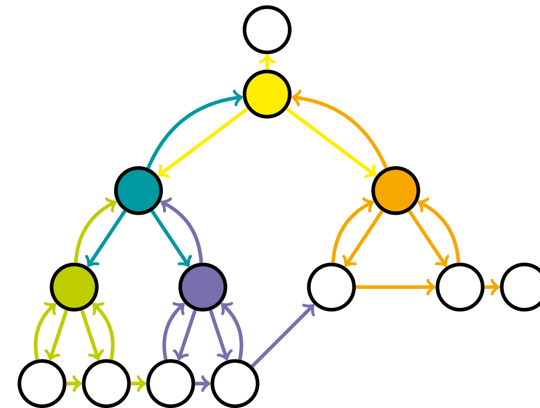


Derivation tree

- 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



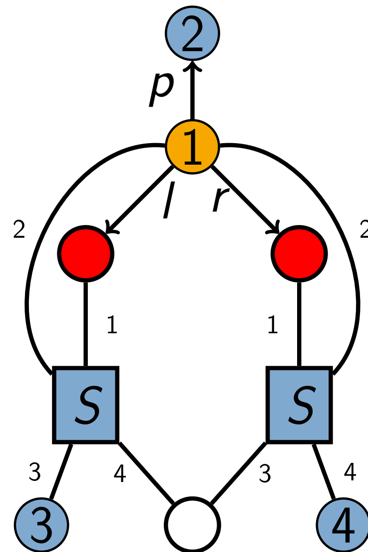
[1] Courcelle, B. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability." Theoretical Computer Science, 1991.

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 $\mathrm{ext}(1)$

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

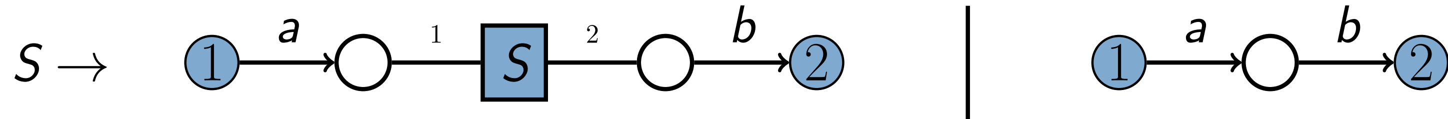
First approach: Every production rule maps to a tree-like hypergraph

Why Tree-Like Hypergraphs?

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

Why Tree-Like Hypergraphs?

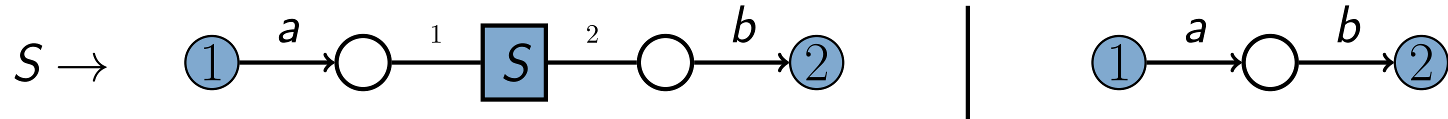
$L = \{ a^n b^n \mid n \geq 1 \}$ is not *MSO* definable.



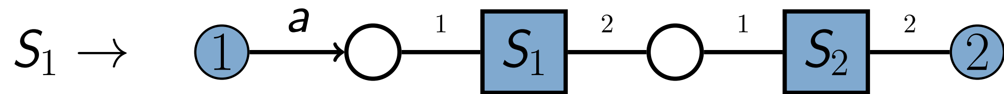
1. false $\mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. true $\mathrm{lab}(e) \in N$ implies $\exists e' . \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

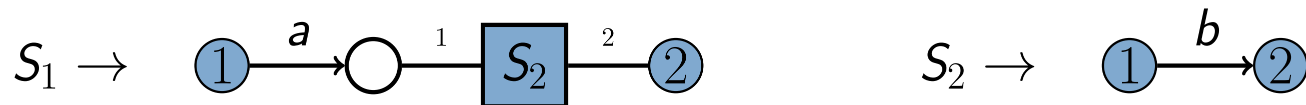
Why Tree-Like Hypergraphs?

$L = \{ a^n b^n \mid n \geq 1 \}$ 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 $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

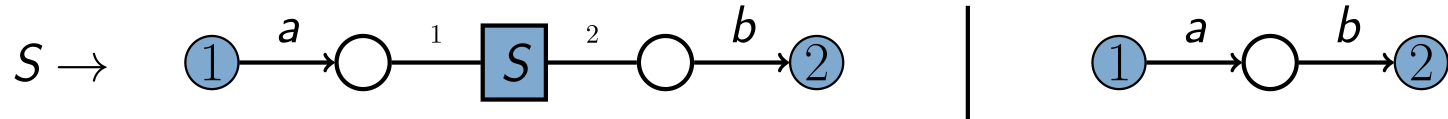




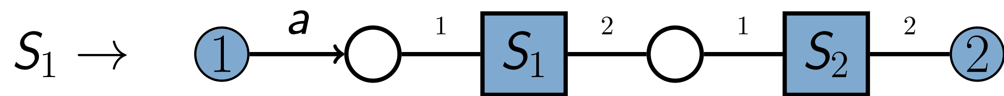
1. true $\quad \mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. false $\quad \mathrm{lab}(e) \in N$ implies $\exists e' \,.\, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

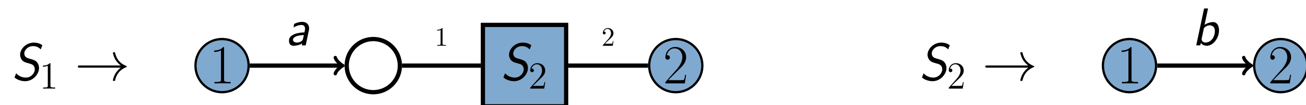
Why Tree-Like Hypergraphs?

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



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

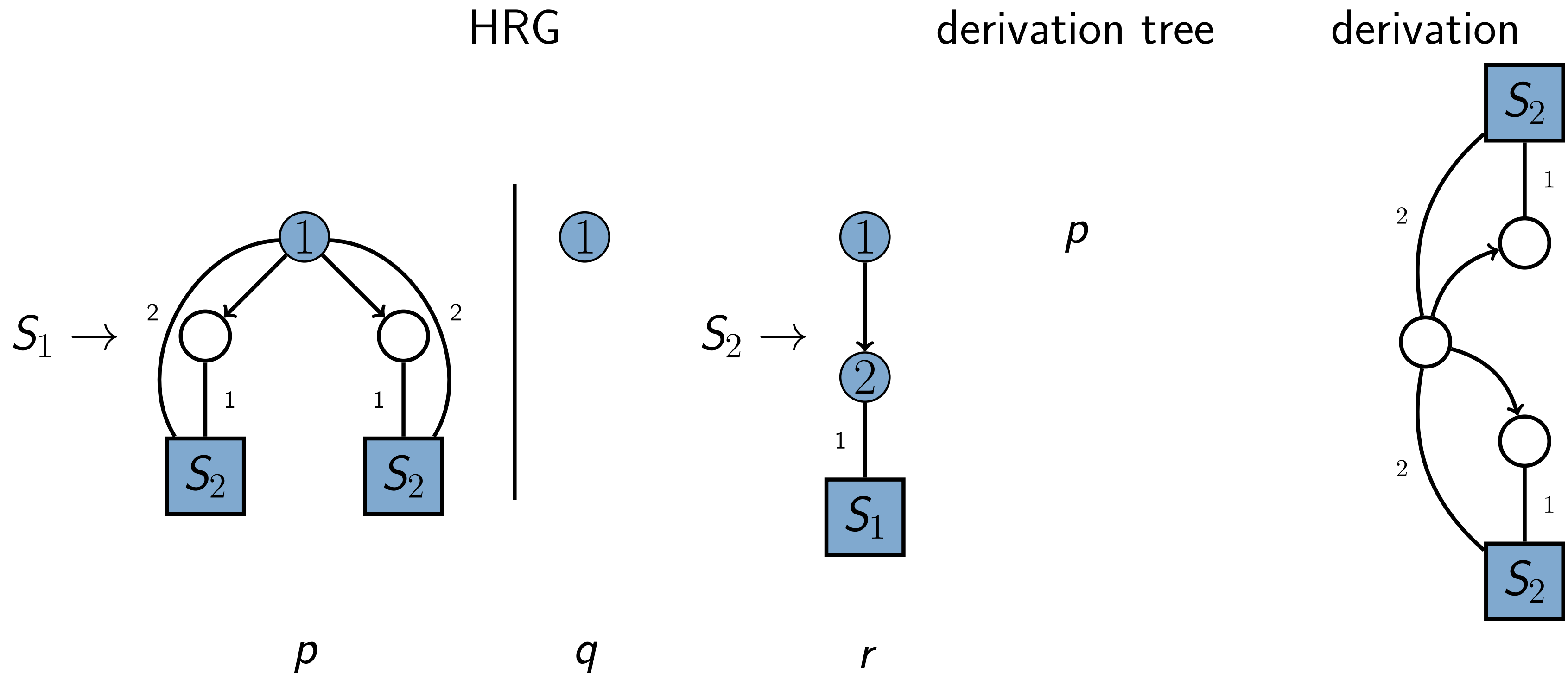




1. true $\mathrm{lab}(e) \in \Sigma$ implies $\mathrm{ext}(1) \in [\mathrm{att}(e)]$
2. false $\mathrm{lab}(e) \in N$ implies $\exists e' \, . \, \mathrm{lab}(e') \in \Sigma$ and $\mathrm{att}(e)(1) \in [\mathrm{att}(e')]$

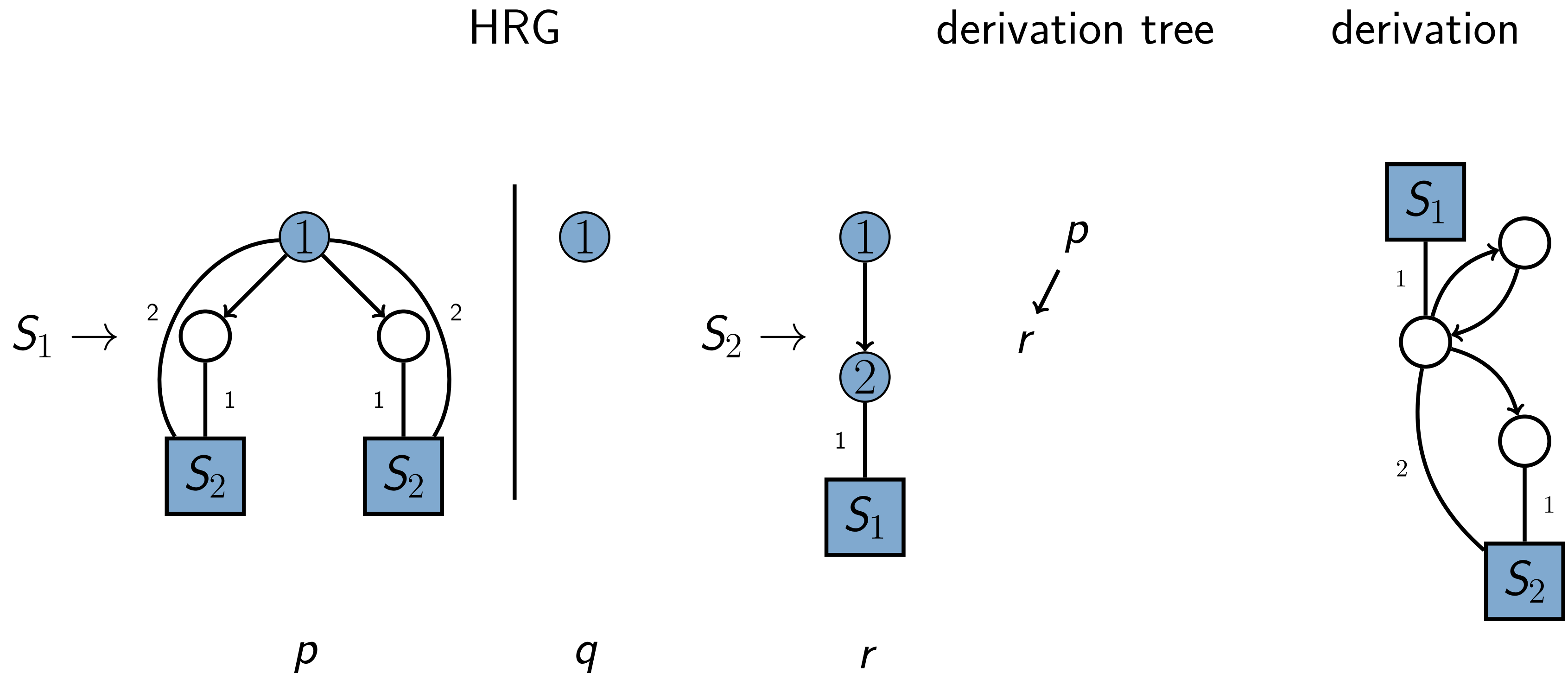
For context-free grammars our conditions yield right-linear grammars.

Tree-Like Hypergraphs are Insufficient



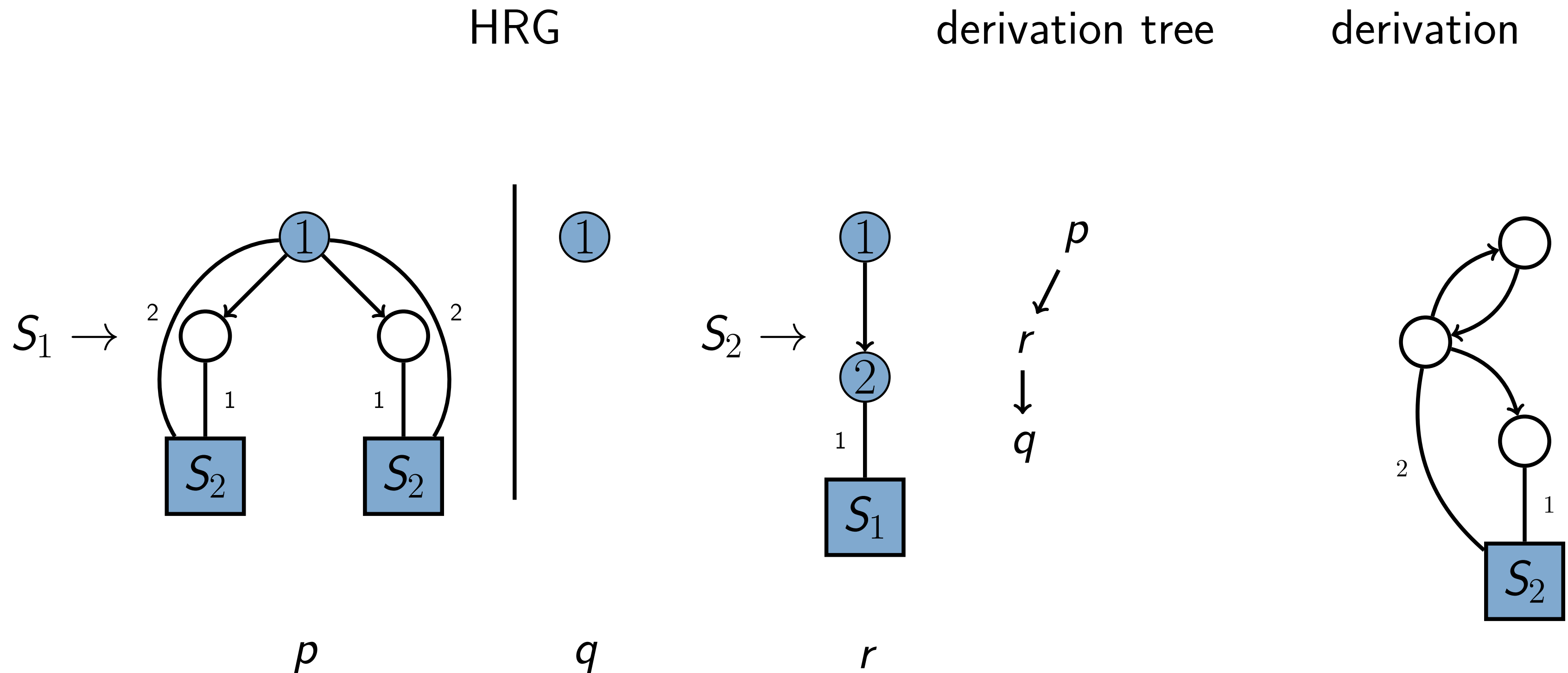
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



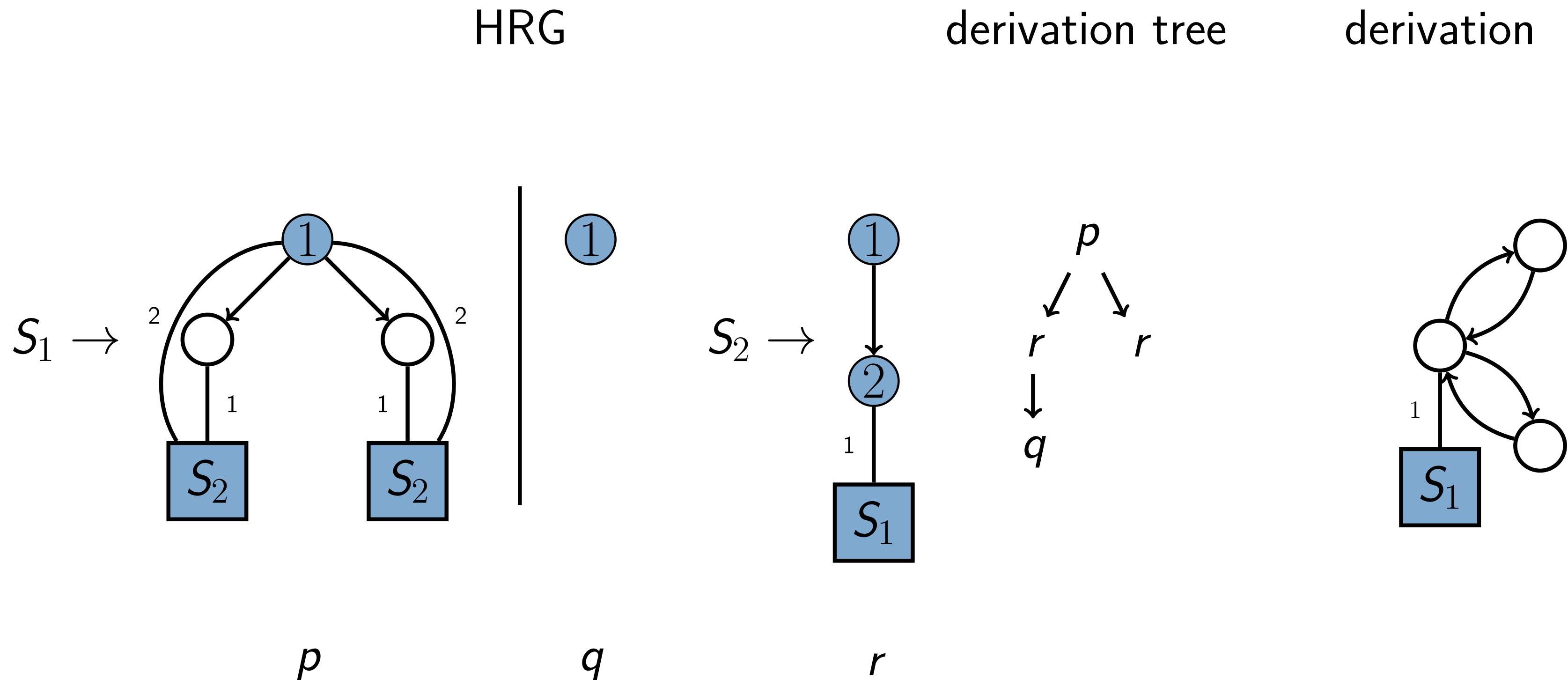
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



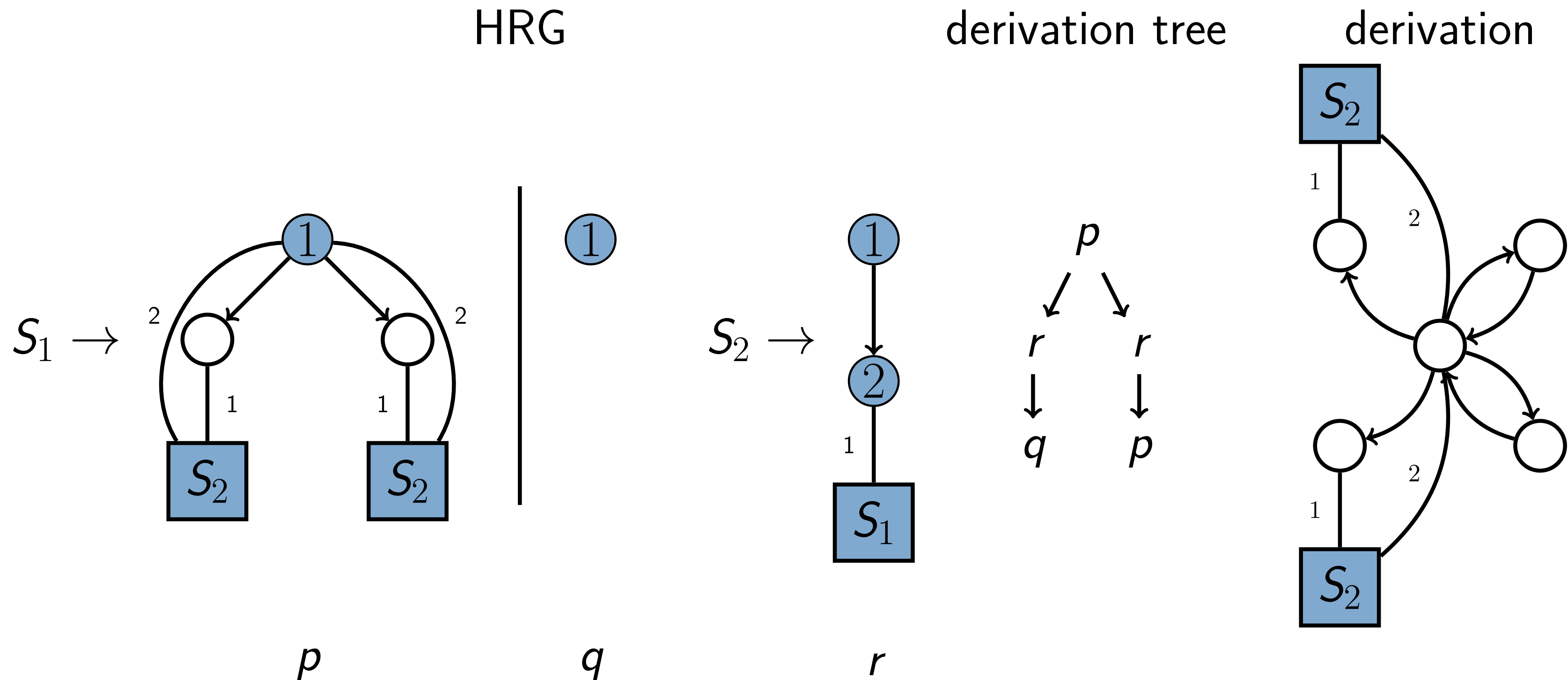
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



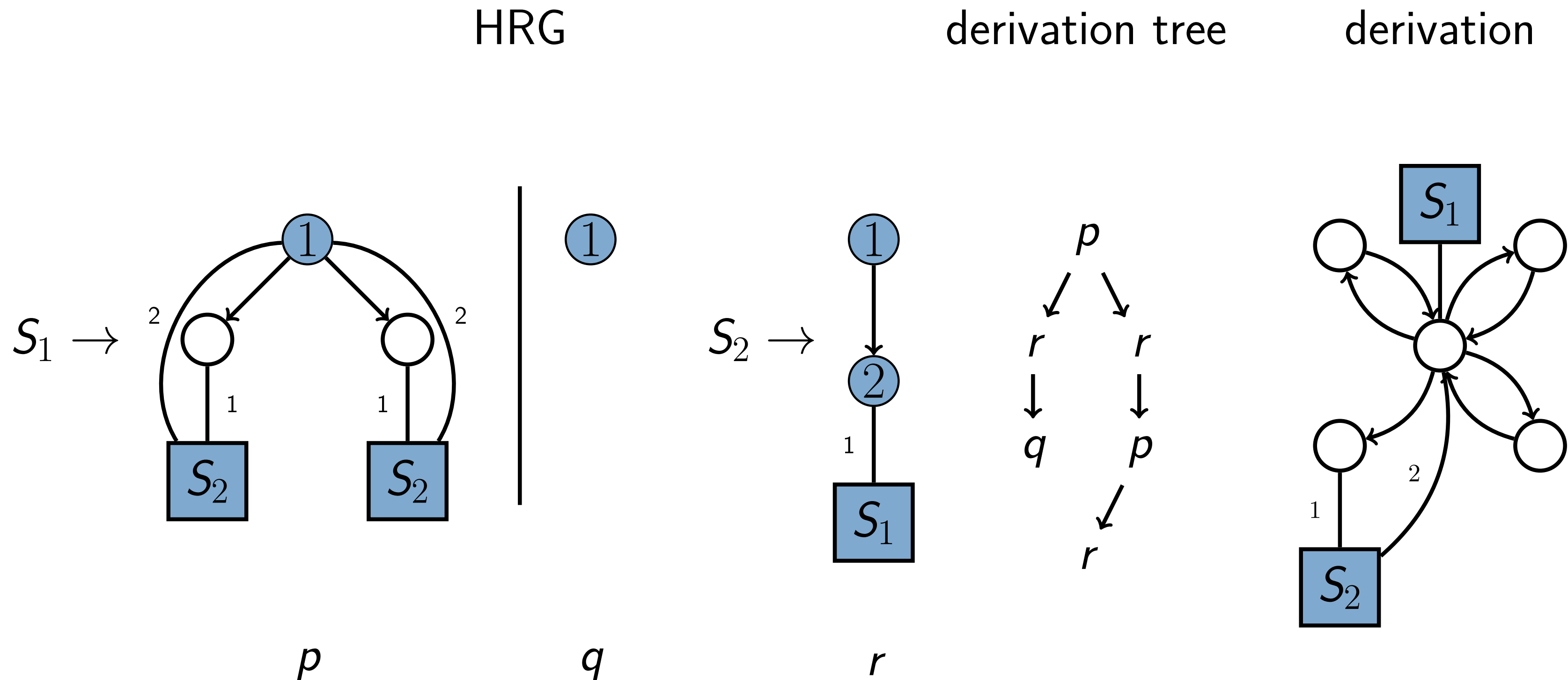
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



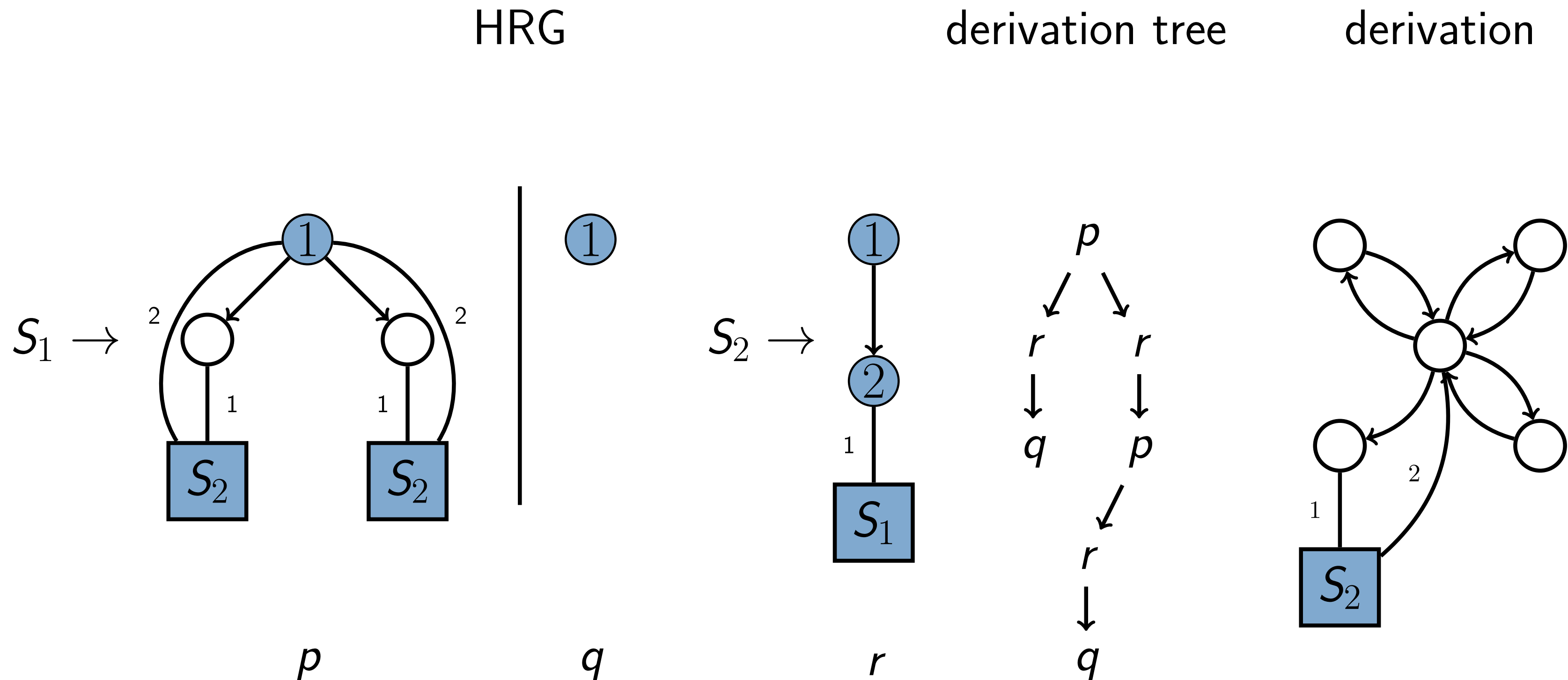
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



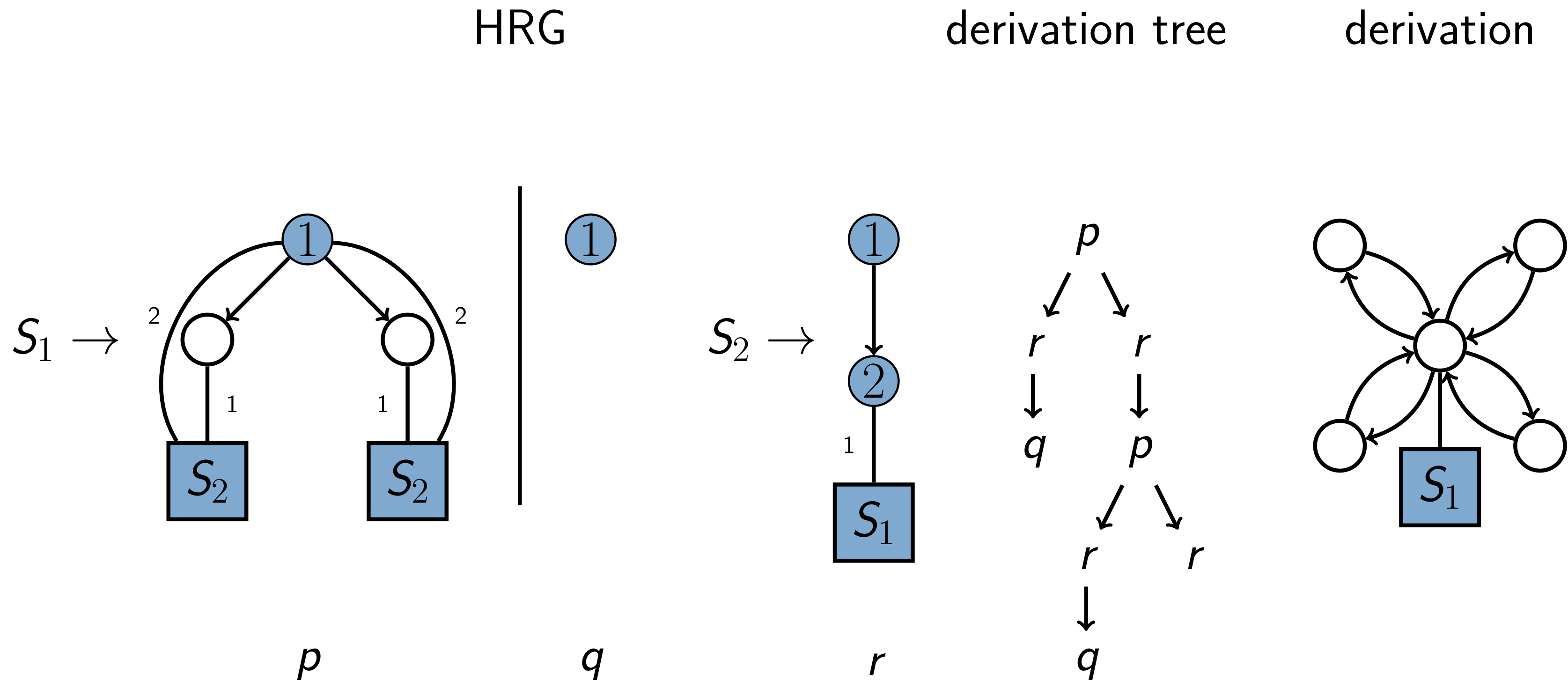
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



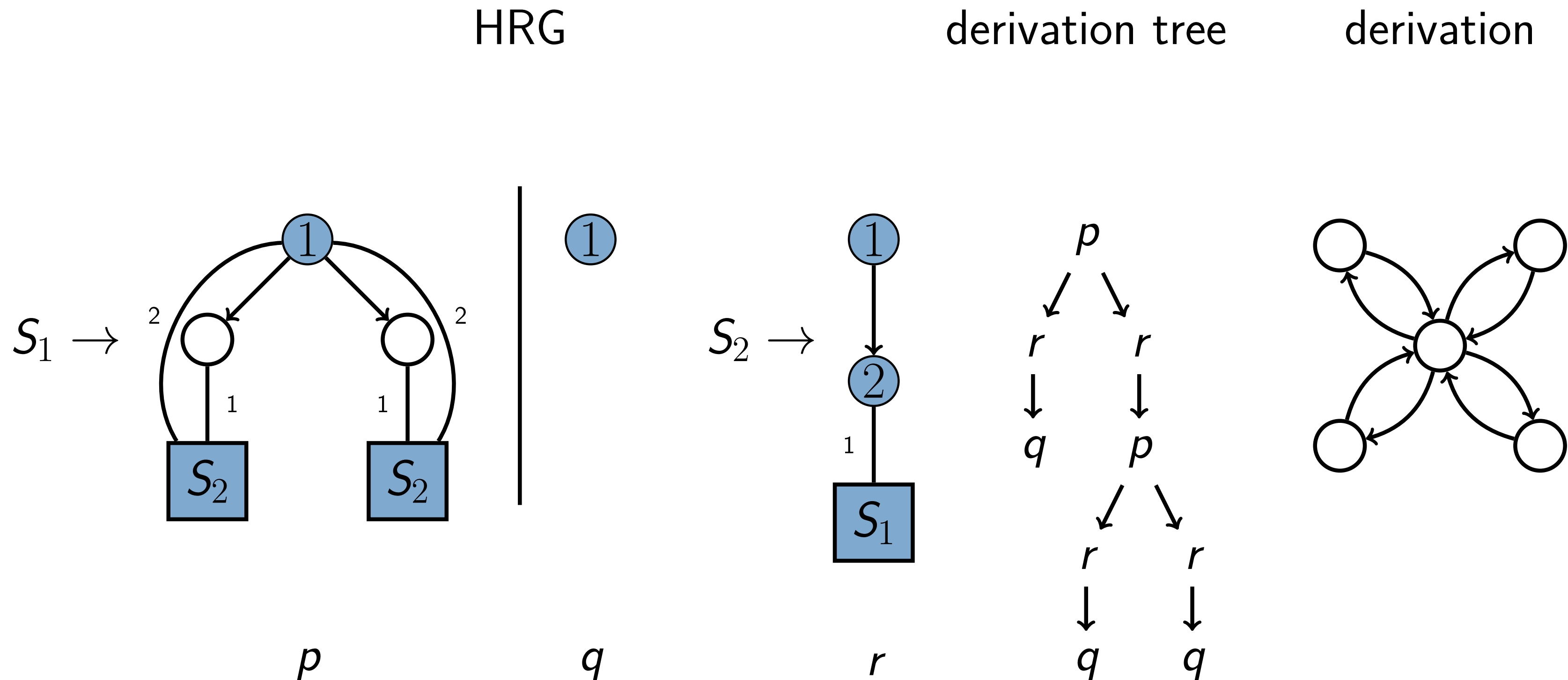
Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



Each production rule maps to a tree-like hypergraph.

Tree-Like Hypergraphs are Insufficient



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. $\mathcal{M}(G) = \emptyset$.

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 (APLAS'15)

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

Results [APLAS'15]

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.

Results [APLAS'15]

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?

Outline

1. Introduction to Separation Logic
2. Symbolic Heaps with Recursive Definitions
3. Graph Grammars
4. Tree-like Grammars
5. Tree-like Separation Logic
6. Conclusion

Tree-Like Separation Logic

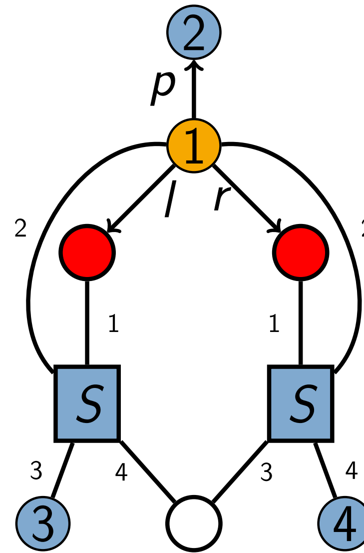
Let $PT(\varphi) \triangleq \{\{x, y\} \mid x.s \mapsto y \text{ occurs in } \varphi\}$.

Definition

A symbolic heap $\varphi\vec{x}$ 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 call $P\vec{y}$ in $\varphi\vec{x}$.

$$
\begin{aligned}
S(x_1, x_2, x_3, x_4) \; = \\
\exists y_1, y_2, y_3 \,.\, x_1 \mapsto (y_1, y_2, x_2, \texttt{nil}) \\
\quad * \; S(y_1, x_1, x_3, y_3) \\
\quad * \; S(y_2, x_1, y_3, x_4)
\end{aligned}
$$



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

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

Tree-Like Separation Logic

Definition

A system of recursive definitions Φ is tree-like if

1. Φ is established.
2. $\varphi\vec{x}$ is tree-like for each $P \Rightarrow \varphi\vec{x} \in \Phi$.
3. For each unfolding tree t and $t(i) = \varphi\vec{x}$, $t(j) = \psi\vec{y}$, $i \neq j$, $s, h \models [\![t]\!]$ implies $s(x_1) \neq s(y_1)$.

Tree-Like Separation Logic

Definition

A system of recursive definitions Φ is tree-like if

1. Φ is established.
2. $\varphi\vec{x}$ is tree-like for each $P \Rightarrow \varphi\vec{x} \in \Phi$.
3. For each unfolding tree t and $t(i) = \varphi\vec{x}$, $t(j) = \psi\vec{y}$, $i \neq j$, $s, h \models [\![t]\!]$ implies $s(x_1) \neq s(y_1)$.

Theorem (APLAS'15)

Every tree-like system of recursive definitions 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.

Conclusion

Wrap-up

- close relationship between Separation Logic and graph grammars
- (extended) inclusion problem decidable for tree-like grammars
- (extended) entailment problem decidable for tree-like Separation Logic

Questions

- Is tree-like Separation Logic as expressive as MSO over heaps of bounded tree width?
- What are tractable fragments of tree-like grammars?