

Simplifying Verification of Unbounded Structures

Alexander Dominik Weinert

RWTH Aachen University

September 3, 2012

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states ✓

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states ✓

Require: List l

Element $e := l.first()$

while $e \neq null$ **do**

$e := l.next(e)$

end while

Infinite number of states

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states ✓

Require: List l

Element $e := l.first()$

while $e \neq null$ **do**

$e := l.next(e)$

end while

Infinite number of states ✗

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states ✓

Require: List l

Element $e := l.first()$

while $e \neq null$ **do**

$e := l.next(e)$

end while

Infinite number of states ✗

Idea [Heinen et al., 2009]

Model heap configurations as graphs

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Require: int i_1 , int i_2

while $i_1 \neq 0$ **do**

$i_1 := i_1 - 1$

$i_2 := i_2 + 1$

end while

Finite number of states ✓

Require: List l

Element $e := l.first()$

while $e \neq null$ **do**

$e := l.next(e)$

end while

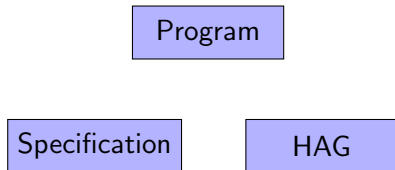
Infinite number of states ✗

Idea [Heinen et al., 2009]

Model heap configurations as graphs

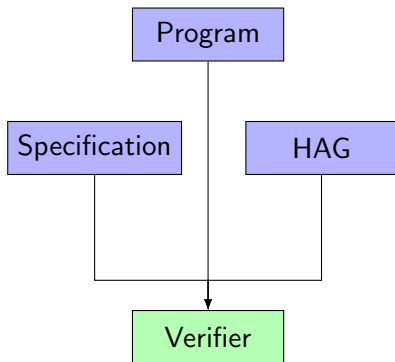
Use *Heap Abstraction Grammars (HAG)* \Rightarrow Finitely many heap states

Workflow in the Juggernaut-Framework



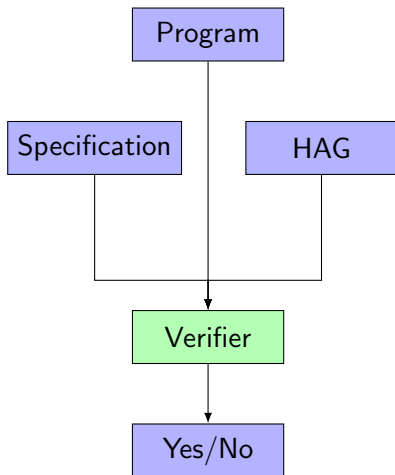
Current Workflow

Workflow in the Juggernaut-Framework



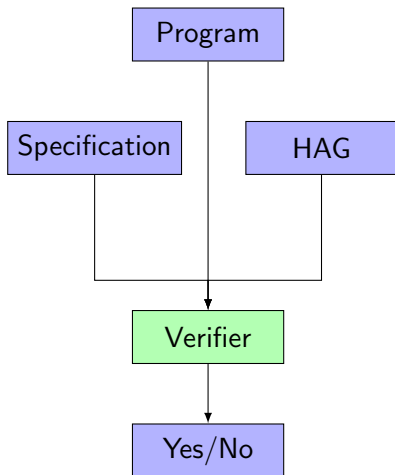
Current Workflow

Workflow in the Juggernaut-Framework

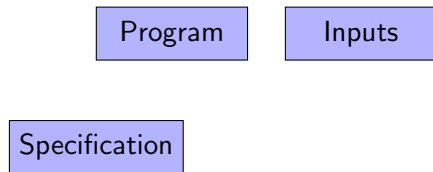


Current Workflow

Workflow in the Juggernaut-Framework

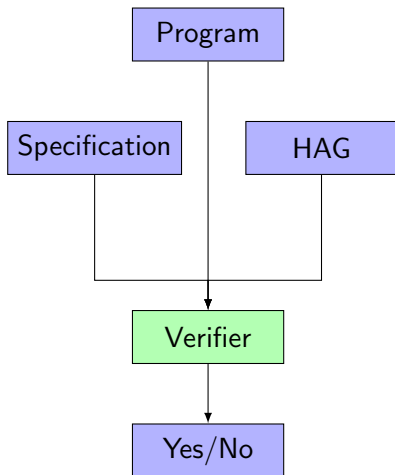


Current Workflow

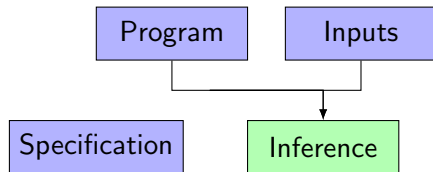


Desired Workflow

Workflow in the Juggernaut-Framework

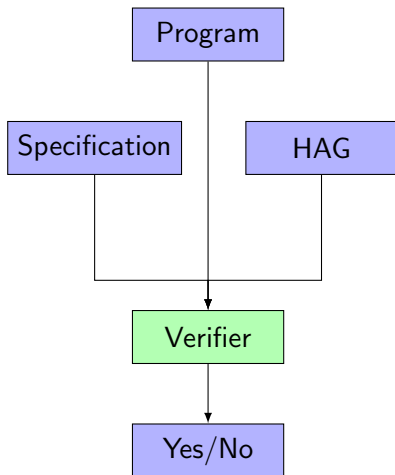


Current Workflow

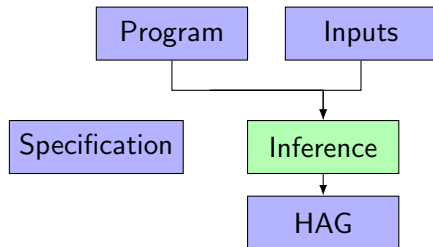


Desired Workflow

Workflow in the Juggernaut-Framework

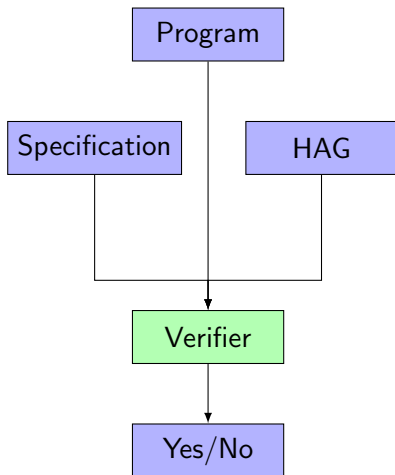


Current Workflow

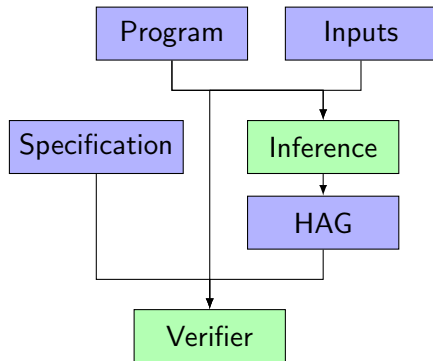


Desired Workflow

Workflow in the Juggernaut-Framework

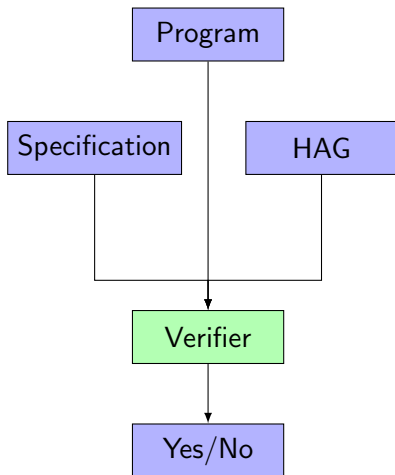


Current Workflow

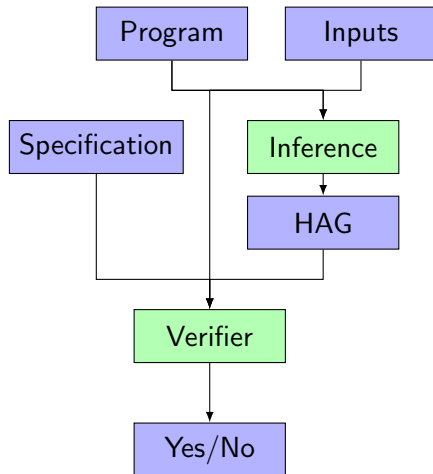


Desired Workflow

Workflow in the Juggernaut-Framework



Current Workflow

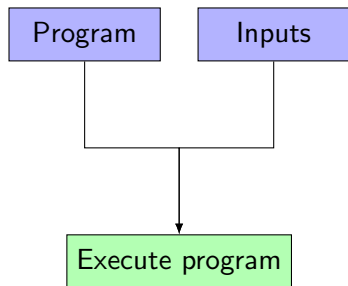


Desired Workflow

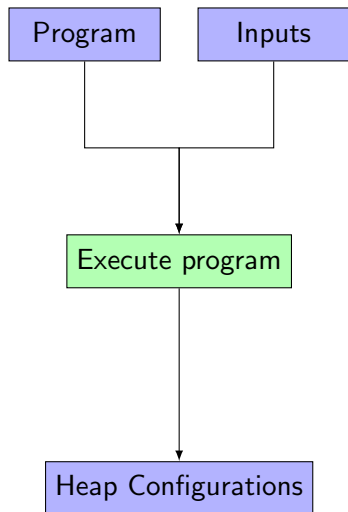
Program

Inputs

HAG

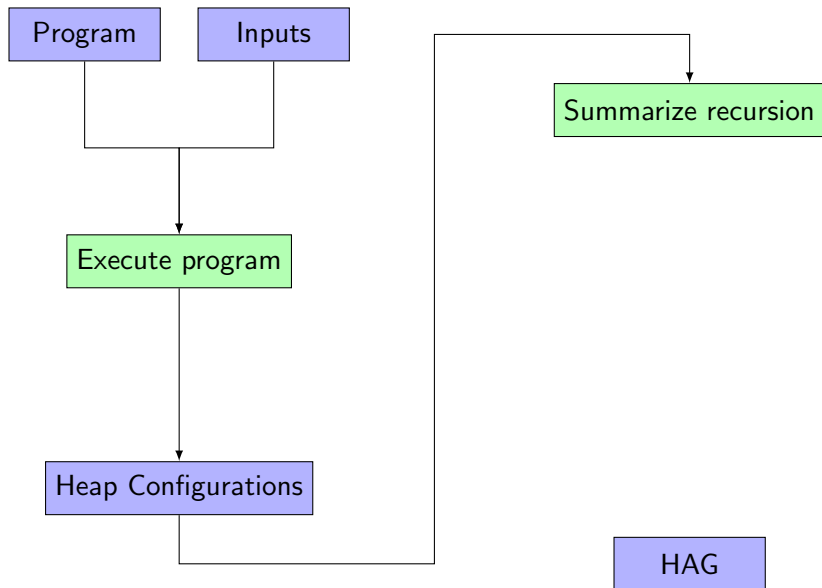


HAG

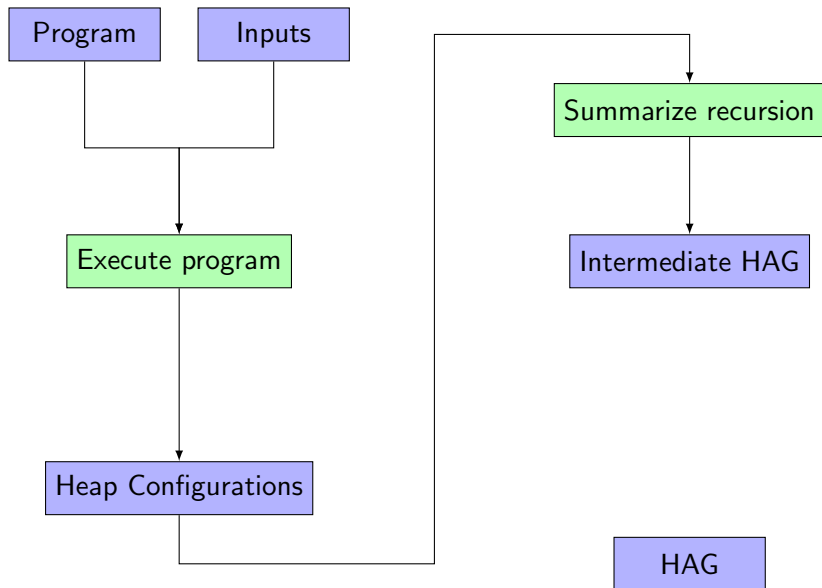


HAG

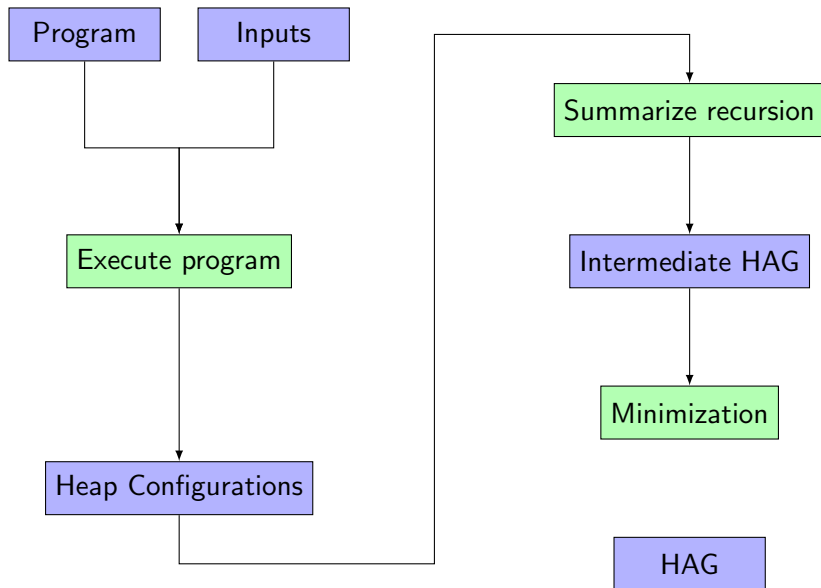
Inference



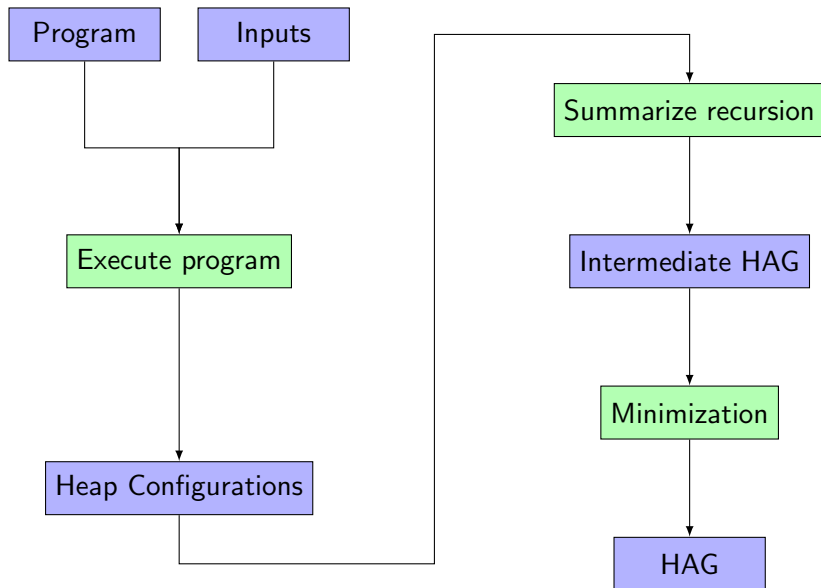
Inference



Inference



Inference



Minimization Problem

Given: Set S of rules $X \rightarrow Y$.

Task: Find “cheaper” set that describes the same set of graphs.

Minimization Problem

Given: Set S of rules $X \rightarrow Y$.

Task: Find “cheaper” set that describes the same set of graphs.

Solution: Minimum Description Length [Rissanen, 1978]

Define gain function for each subgraphs and cost function for rules:

Minimization Problem

Given: Set S of rules $X \rightarrow Y$.

Task: Find “cheaper” set that describes the same set of graphs.

Solution: Minimum Description Length [Rissanen, 1978]

Define gain function for each subgraphs and cost function for rules:

$$\text{gain}(G) = \text{gain}(S \mid Y \rightarrow G) - \text{cost}(Y \rightarrow G)$$

Minimization Problem

Given: Set S of rules $X \rightarrow Y$.

Task: Find “cheaper” set that describes the same set of graphs.

Solution: Minimum Description Length [Rissanen, 1978]

Define gain function for each subgraphs and cost function for rules:

$$\text{gain}(G) = \text{gain}(S \mid Y \rightarrow G) - \text{cost}(Y \rightarrow G)$$

Finding optimal graph for minimization

\Leftrightarrow

Maximization of gain function over all subgraphs

Minimization Problem

Given: Set S of rules $X \rightarrow Y$.

Task: Find “cheaper” set that describes the same set of graphs.

Solution: Minimum Description Length [Rissanen, 1978]

Define gain function for each subgraphs and cost function for rules:

$$\text{gain}(G) = \text{gain}(S \mid Y \rightarrow G) - \text{cost}(Y \rightarrow G)$$

Finding optimal graph for minimization

\Leftrightarrow

Maximization of gain function over all subgraphs

Cost- and Gain-functions are heuristics

Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs

Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ Exponential runtime

Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ Exponential runtime

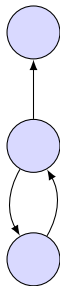
Solution [Jonker et al., 2002]: Grow only connected subgraphs.

Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ Exponential runtime

Solution [Jonker et al., 2002]: Grow only connected subgraphs.

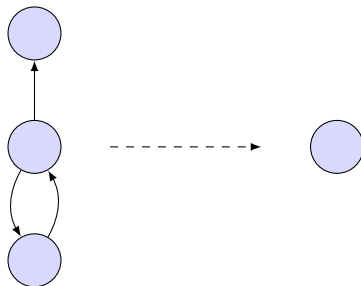


Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ **Exponential runtime**

Solution [Jonker et al., 2002]: Grow only connected subgraphs.

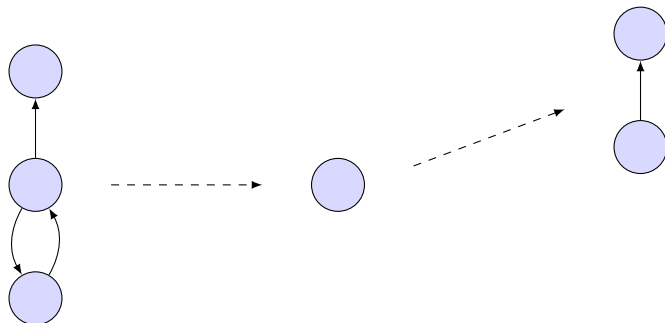


Enumeration of Subgraphs

Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ **Exponential runtime**

Solution [Jonker et al., 2002]: Grow only connected subgraphs.

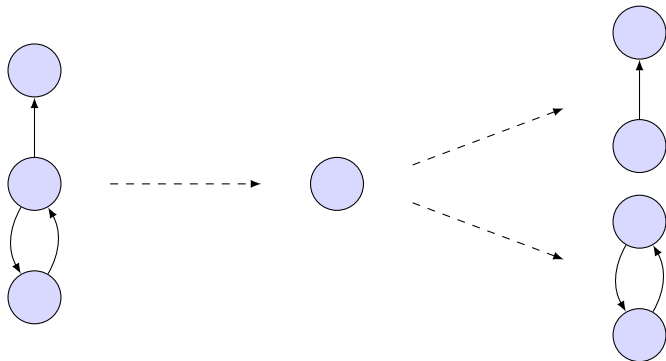


Enumeration of Subgraphs

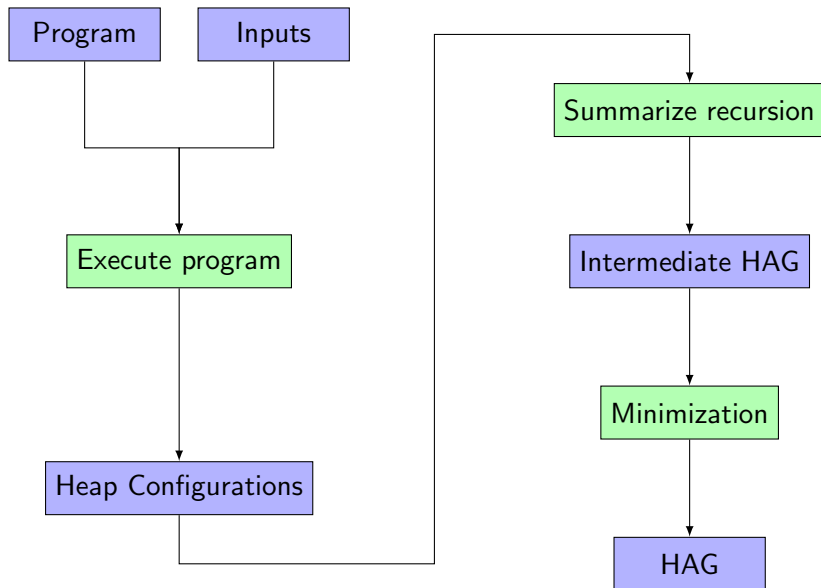
Minimum-Description-Length-approach

Problem: Minimum Description Length: enumeration over all subgraphs
⇒ **Exponential runtime**

Solution [Jonker et al., 2002]: Grow only connected subgraphs.



Inference (Reminder)



Conditions for Recursion

A data structure is recursive, if

Conditions for Recursion

A data structure is recursive, if

- it is found in at least two places,

Conditions for Recursion

A data structure is recursive, if

- it is found in at least two places,
- these two embeddings overlap and

Conditions for Recursion

A data structure is recursive, if

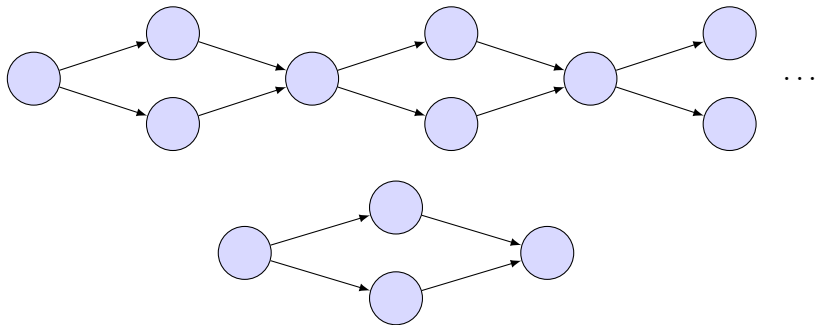
- it is found in at least two places,
- these two embeddings overlap and
- a concatenation of these embeddings is itself a subgraph.

Recursive Data Structures

Conditions for Recursion

A data structure is recursive, if

- it is found in at least two places,
- these two embeddings overlap and
- a concatenation of these embeddings is itself a subgraph.

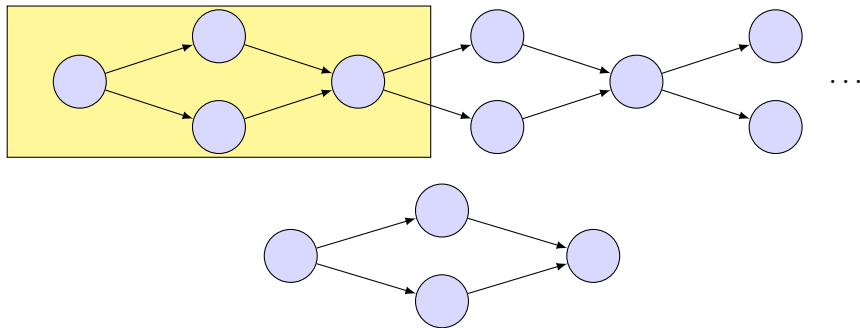


Recursive Data Structures

Conditions for Recursion

A data structure is recursive, if

- it is found in at least two places,
- these two embeddings overlap and
- a concatenation of these embeddings is itself a subgraph.

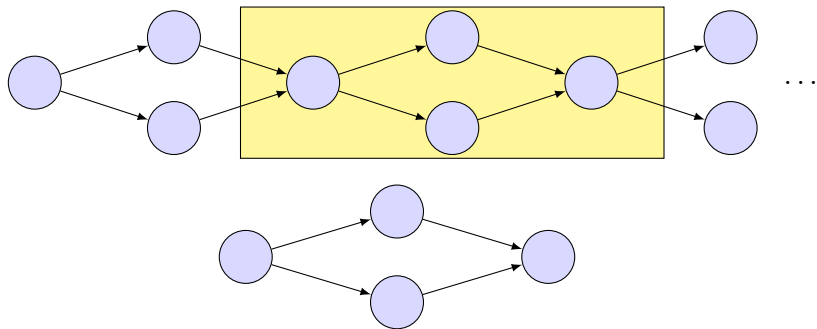


Recursive Data Structures

Conditions for Recursion

A data structure is recursive, if

- it is found in at least two places,
- these two embeddings overlap and
- a concatenation of these embeddings is itself a subgraph.



Representation of Recursive Data Structures

When finding a recursive data structure:

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Recursive Data Structures (cont.)

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Example in String Case

Recursive Data Structures (cont.)

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Example in String Case

String under consideration:

xyzabcabcyx

Recursive Data Structures (cont.)

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Example in String Case

String under consideration:

$xyzabcabcyx$ { Rule for concatenations: $X \rightarrow abcX$

Recursive Data Structures (cont.)

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Example in String Case

String under consideration:

$$xyzabcabcyx \left\{ \begin{array}{l} \text{Rule for concatenations: } X \rightarrow abcX \\ \text{Rule for stopping: } X \rightarrow abc \end{array} \right.$$

Recursive Data Structures (cont.)

Representation of Recursive Data Structures

When finding a recursive data structure:

- Add rule for concatenations of arbitrary length
- Add rule for stopping concatenation
- Remove concatenated structure from original graph and replace it with new nonterminal

Example in String Case

String under consideration:

$$xyzabcabcyx \left\{ \begin{array}{l} \text{Rule for concatenations: } X \rightarrow abcX \\ \text{Rule for stopping: } X \rightarrow abc \\ \text{Resulting string: } xyzXzxx \end{array} \right.$$

Results for Singly Linked List

Input: Singly linked lists with 25 to 200 nodes

Results for Singly Linked List

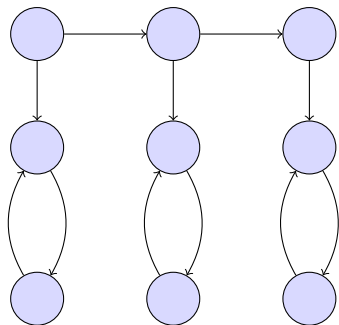
Input: Singly linked lists with 25 to 200 nodes

Nodes	Subgraphs [ms]	Complete [ms]
25	90	102
50	285	305
75	437	500
100	642	682
125	1 001	1 040
150	1 455	1 526
175	1 884	2 000
200	2 895	3 028

Input: Singly linked nested lists

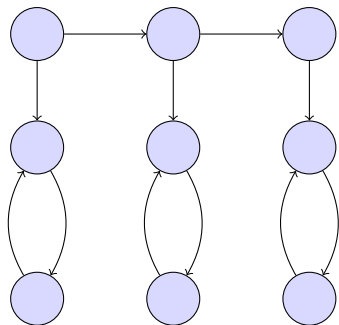
Results for Singly Linked Nested Lists

Input: Singly linked nested lists



Results for Singly Linked Nested Lists

Input: Singly linked nested lists



Outer	Inner	Inner [ms]	Outer [ms]
2	2	9	8
2	4	19	5
4	2	31	16
4	4	394	11
4	5	1 280	20
5	2	96	23
5	4	2 701	22
5	5	51 608	23

Thank you for your attention

`www.mpi-sws.org/~aweinert`

 Heinen, J., Noll, T., and Rieger, S. (2009).

Juggernaut: Graph Grammar Abstraction for Unbounded Heap Structures.

In Johnsen, E. B. and Stolz, V., editors, *Harnessing Theories for Tool Support in Software, Preliminary Proceedings*, pages 53–67. United Nations University - International Institute for Software Technology.

 Jonyer, I., Holder, L. B., and Cook, D. J. (2002).

Concept Formation Using Graph Grammars.

In *Proceedings of the KDD Workshop on Multi-Relational Data Mining*.

 Rissanen, J. (1978).

Modeling by shortest data description.

Automatica, 14(5):465–471.

 Weinert, A. (2012).

Inferring Heap Abstraction Grammars.

Bachelor's Thesis, RWTH Aachen University, Aachen.