Linearly Ordered Attribute Grammar scheduling using SAT-solving

Jeroen Bransen [1] L. Thomas van Binsbergen [2]
Atze Dijkstra [1] Koen Claessen [3]

[1]Utrecht University

[2]Royal Holloway, University of London

[3]Chalmers University of Technology

TACAS 2015, ETAPS, London

Overview

- Motivation.
- LOAG scheduling problem.
- SAT formulation.
- Constraint generation using chordal graphs.
  - Based on an approach by Bryant & Velev.

Attribute Grammars

- Attribute Grammars describe computations over trees.
- Useful in compiler construction, e.g. for:
  Code generation, static analysis, semantic evaluation.
- An AG compiler generates an evaluator from a description.
  - UUAGC (Utrecht University Attribute Grammar Compiler).
- To generate a *strict* evaluator, scheduling is required.
- The scheduling problem is NP-hard.

Compile-time scheduling

- No compile-time scheduling:
  - Generate lazy code.
- Some compile-time scheduling:
  - Find multiple schedules, covering all possible trees.
  - The actual schedule depends on the input tree.
  - Absolutely Non-Circular Attribute Grammars (ANCAGs).
  - Kennedy-Warren algorithm.
- Full compile-time scheduling:
  - Find a single evaluation order.
  - It needs to be compatible with all possible trees.
  - Linearly Ordered Attribute Grammars (LOAGs).
  - Kastens' algorithm (schedules subclass OAG).

Motivation

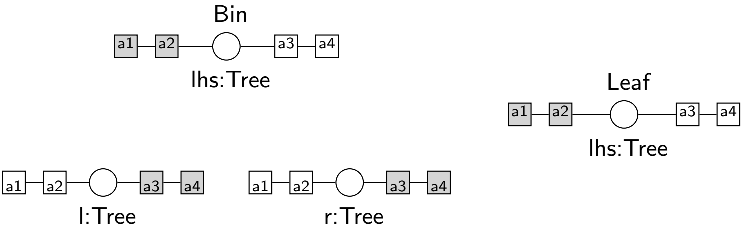
- Many tools at Utrecht University are developed using AGs.
- Large projects require efficient and strict code.
- Main motivation is the Utrecht Haskell Compiler.

AG descriptions

- An AG descriptions contains three components:
  1. A context-free grammar (describing all possible input trees).
  2. A set of attribute declarations (for every non-terminal).
  3. A definition for every attribute (in each context it appears).
- From the AG description, the AG compiler obtains a dependency graph for every production.

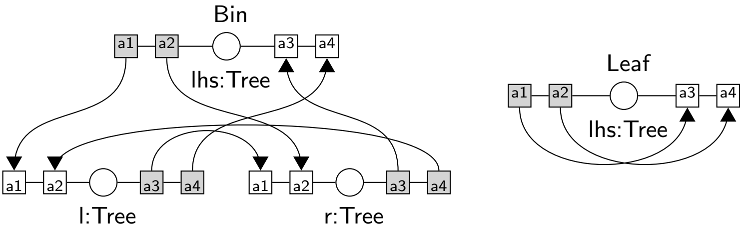
Production dependency graph

- A production graph contains:
  - A parent node for the production's left-hand side (*lhs*).
  - Children for all non-terminal occurrences of the right-hand side.
  - All attributes of the occurring non-terminals as vertices.
- The children are named by the programmer (*l* and *r* for Bin).
- The vertices are also called *attribute occurrences*.



Direct dependencies

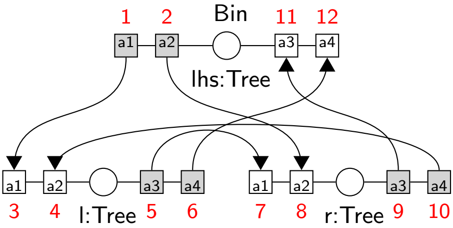
- From the AG description *direct dependencies* are obtained.
- If attribute a is used in the definition of b, then $a \rightarrow b$.

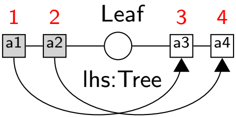


LOAG scheduling

- Find a linear order for every production graph, such that:
  1) The direct dependencies are 'respected' (local).
  2) Same relative ordering for non-terminal occurrences (global):

     If $a3 < a2$ holds for $a3$ and $a2$ attached to the same occurrence of non-terminal X,
     then $a3 < a2$ at all occurrences of X.



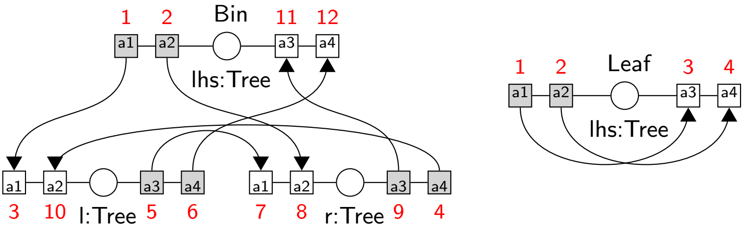


- Invalid schedule: $10 < 4$.

LOAG scheduling

- Find a linear order for every production graph, such that:
  1) The direct dependencies are 'respected' (local).
  2) Same relative ordering for non-terminal occurrences (global):

     If $a3 < a2$ holds for $a3$ and $a2$ attached to the same occurrence of non-terminal X,
     then $a3 < a2$ at all occurrences of X.

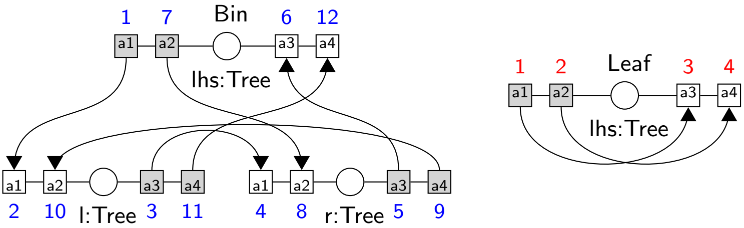


- Invalid schedule: *lhs* has different order.

LOAG scheduling

- Find a linear order for every production graph, such that:
  1) The direct dependencies are 'respected' (local).
  2) Same relative ordering for non-terminal occurrences (global):

     If $a3 < a2$ holds for $a3$ and $a2$ attached to the same occurrence of non-terminal X,
     then $a3 < a2$ at all occurrences of X.

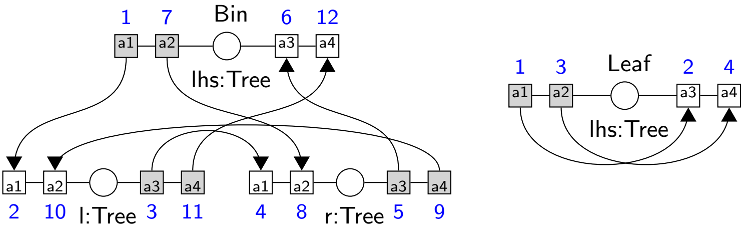


- No unsatisfied properties in Bin. *lhs* at Leaf still incorrect.

LOAG scheduling

- Find a linear order for every production graph, such that:
    1) The direct dependencies are ‘respected’ (local).
    2) Same relative ordering for non-terminal occurrences (global):

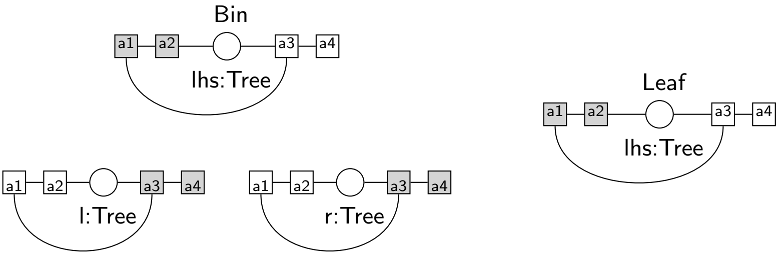
       If $a3 < a2$ holds for $a3$ and $a2$ attached to the same occurrence of non-terminal X,
       then $a3 < a2$ at all occurrences of X.



- Valid schedule.

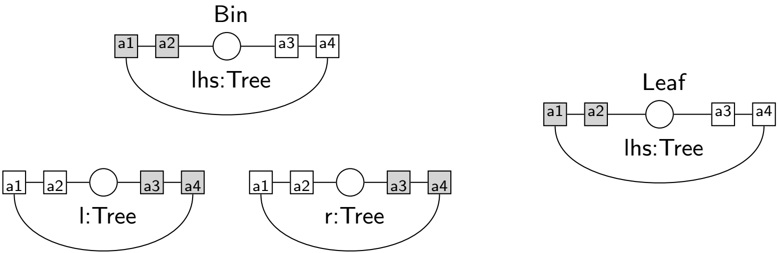
Sat formulation

- Let a variable correspond to an undirected edge.
- An assignment to that variable determines direction.
- By sharing variables we enforce the same relative ordering for occurrences of the same non-terminal.



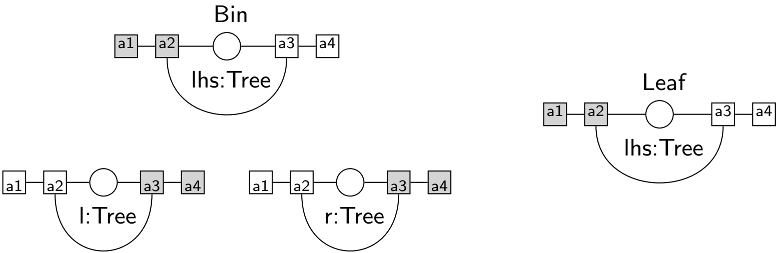
Sat formulation

- Let a variable correspond to an undirected edge.
- An assignment to that variable determines direction.
- By sharing variables we enforce the same relative ordering for occurrences of the same non-terminal.



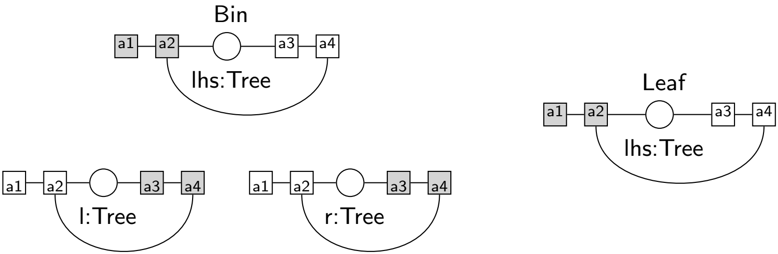
Sat formulation

- Let a variable correspond to an undirected edge.
- An assignment to that variable determines direction.
- By sharing variables we enforce the same relative ordering for occurrences of the same non-terminal.



Sat formulation

- Let a variable correspond to an undirected edge.
- An assignment to that variable determines direction.
- By sharing variables we enforce the same relative ordering for occurrences of the same non-terminal.



First attempt

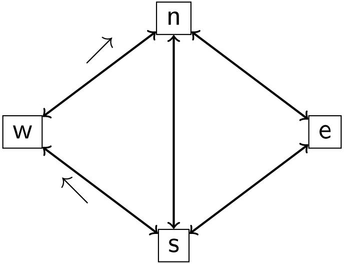
- Add every possible undirected edge to every production graph.
- Assign a variable to every undirected edge,
  where variables are shared as explained earlier.
- Add constraints corresponding to the direct dependencies.
- Add transitivity constraints.
- Number of constraints is cubic to the number of attribute occurrences.

Solution

- Less constraints are required when we:
  1. Observe not all possible undirected edges have to be considered.
  2. Triangulate the graphs,
     adding 2 clauses for all encountered triangles
     and removing vertices with a safe neighbourhood.
     - Based on work by Bryant & Velev on equality logic.
  3. Improving the heuristic for triangulating the graphs.
  4. Constrain non-terminal subgraphs separately.

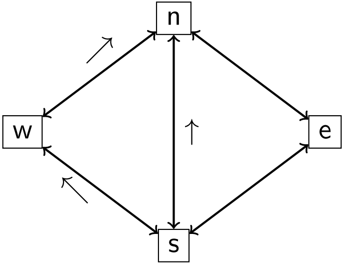
Constraining triangles

- In a triangulated graph every cycle has length 3 or has a chord.
- By ruling out 3-cycles we rule out all bigger cyles.



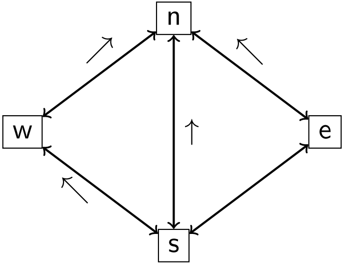
Constraining triangles

- In a triangulated graph every cycle has length 3 or has a chord.
- By ruling out 3-cycles we rule out all bigger cyles.



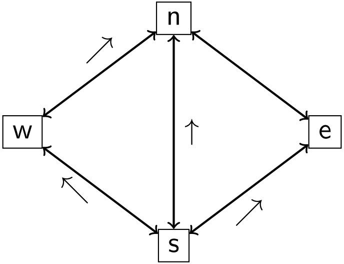
Constraining triangles

- In a triangulated graph every cycle has length 3 or has a chord.
- By ruling out 3-cycles we rule out all bigger cyles.



Constraining triangles

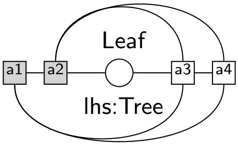
- In a triangulated graph every cycle has length 3 or has a chord.
- By ruling out 3-cycles we rule out all bigger cyles.



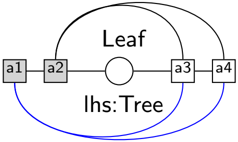
Triangulation

- A graph is triangulated by selecting *some* vertex v, and adding a chord for all pairs of unconnected neighbours.
- We add 2 clauses for every encountered triangle, one for the clockwise and one for the counter-clockwise cycle.
- After all neighbours are connected, v is removed.
- Based on the notion of a *perfect elimination order*.
- The order in which vertices are removed influences the number constraints and variables.
- And has a considerable impact on the time required to generate the SAT-instance.

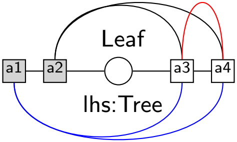
Triangulation



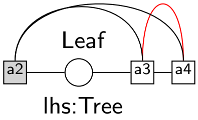
Triangulation



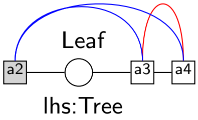
Triangulation



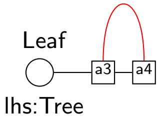
Triangulation



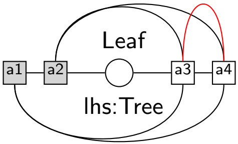
Triangulation



Triangulation



Triangulation



Experimental results

	K&W	LOAG-b	LOAG-SAT
UHC MainAG	33s	13s	9s
Asil Test	1.8s	4.4s	3.4s
Asil ByteCode	0.6s	29.4s	2.8s
Asil PrettyTree	390ms	536ms	585ms
Asil InsertLabels	314ms	440ms	452ms
UUAGC CodeGeneration	348ms	580ms	382ms
Pigeonhole principle^{+}	107ms	1970ms	191ms
Pigeonhole principle^{-}	111ms	60.2s	103ms

Schedule optimisations

- Encoding LOAG as SAT allows us to define arbitrary scheduling optimisations.
- For example:
    - Reducing overhead from performing *visits*.
    - Demanding certain attributes to be evaluated ASAP.
- Instead of specifying the demands up front, we interact with the SAT-solver.
- Based on the ideas of *sorting networks*.

Conclusion

- The LOAG scheduling problem is an instance of SAT.
- The problem is quite general:
  - Find a linear order on a number of graphs.
  - Where some pairs need to have the same ‘assignment’.
- Generating the SAT-instance takes up most of the work.
- We decreased the amount of work, by:
  - Adding constraints for triangles in a triangulated graph.
  - Selecting the right heuristic for triangulating the graph.
  - A number of domain specific improvements.

Scheduling the Utrecht Haskell Compiler (UHC)

- UHC is partly generated from of a large number of AGs.
- The "main AG" is large indeed:
  - 30 non-terminals
  - 134 productions
  - 1332 attributes (44.4 per non-terminal!)
  - 9766 dependencies
- Kastens' algorithm does not find a static evaluation order for the main AG.
- We know at least one exists, as Kastens' algorithm can be 'helped' to find one using 24 *augmenting dependencies*.

Triangulation heuristic

Order	#Clauses	#Vars	Ratio	Time
(d, s, c)	21,307,812	374,792	57.85	34s
(d, c, s)	8,301,557	220,690	37.62	17s
(s, d, c)	12,477,519	287,151	43.45	23s
(s, c, d)	8,910,379	241,853	36.84	18s
(c, d, s)	3,004,705	137,277	21.89	9s
(c, s, d)	3,359,910	156,795	21.43	10s
$(d + s, c)$	12,424,635	386,323	32.16	22s
$(d, s + c)$	8,244,600	219,869	37.50	17s
$(d + c, s)$	2,930,922	135,654	21.61	9s
$(s, d + c)$	8,574,307	236,348	36.28	17s
$(s + c, d)$	3,480,866	157,089	22.16	11s
$(c, d + s)$	3,392,930	157,568	21.53	11s
$(c + d + s)$	3,424,001	148,724	23.02	11s
$(3 * s * (d + c) + (d * c)^2)$	$\underline{2,679,772}$	$\underline{127,768}$	$\underline{20.97}$	9s