Linearly Ordered Attribute Grammars

with Automatic Augmenting Dependency Selection

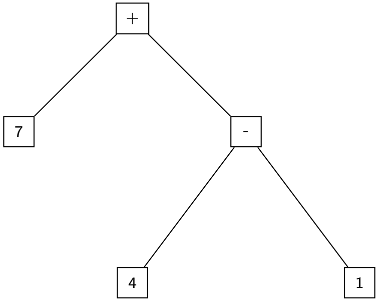
L. Thomas van Binsbergen [1]
Jeroen Bransen [2] Atze Dijkstra [2]

[1]ltvanbinsbergen@acm.org
Royal Holloway, University of London

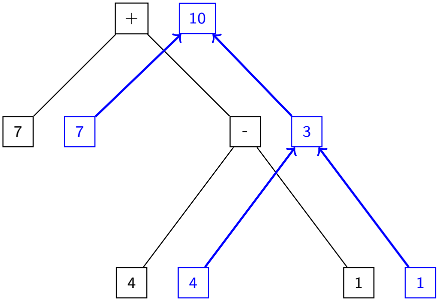
[2]{j.bransen,atze}@uu.nl
Utrecht University

PEPM'15, Mumbai, India

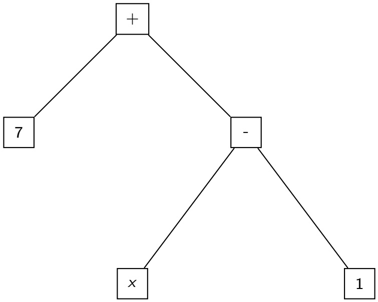
Evaluating Expressions



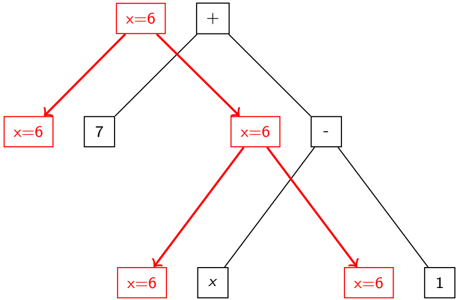
Evaluating Expressions



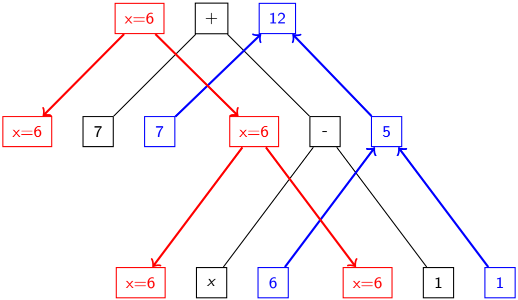
Evaluating Expressions



Evaluating Expressions

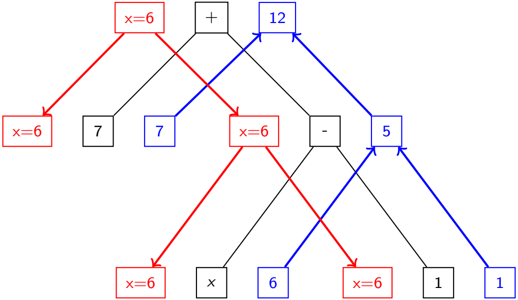


Evaluating Expressions



Evaluating Expressions using Attribute Grammars

- Attribute Grammars extend trees with attributes.
- Every node N represents one or more functions, that:
  - Receive a subset of the inherited attributes of N.
  - Produce a subset of the synthesized attributes of N.
- Attribute Grammars form a DSL for tree-based computations.



Modularity of Attribute Grammars

- Define multiple computations on the same tree separately.
- The AG compiler combines them and generates an evaluator.
- By generating code we abstract away from the problem of propagating changes.

```
  -- evalExpr :: Non-terminal → InhAttrs → SynAttrs
evalExpr :: Expr → Env → (String, Int)
evalExpr (Plus e1 e2) env =
  let (pp1, v1) = evalExpr e1 env
      (pp2, v2) = evalExpr e2 env
  in ("(" ++ pp1 ++ "+" ++ pp2 ++ ")", v1 + v2)
```

Utrecht University Attribute Grammar Compiler (UUAGC)

- The UUAGC generates Haskell code from UUAG descriptions.
- UUAG has experience-enhancing features such as *copy-rules* and *use-rules*.
- For different classes UUAGC generates different evaluators:
  - Lazy folds and algebras for any (cyclic) AG description.
  - Strict dynamic evaluators for Absolutely Non-Circular AGs.
  - Strict static evaluators for Ordered AGs.
- With *higher-order attributes*, UUAG allows looping computations by adding nodes to the tree on the fly.

Static Evaluation Orders

- We are interested in finding static evaluation orders as introduced by Kastens (1980).
- Static orders allow strict and efficient evaluators.
- To find a static evaluation order, we need to:
  - Find an interface for every non-terminal.
  - Show how every production implements it.
- AGs for which this is possible are linearly ordered (LOAG).
- Deciding whether an AG is linearly ordered is NP-hard.

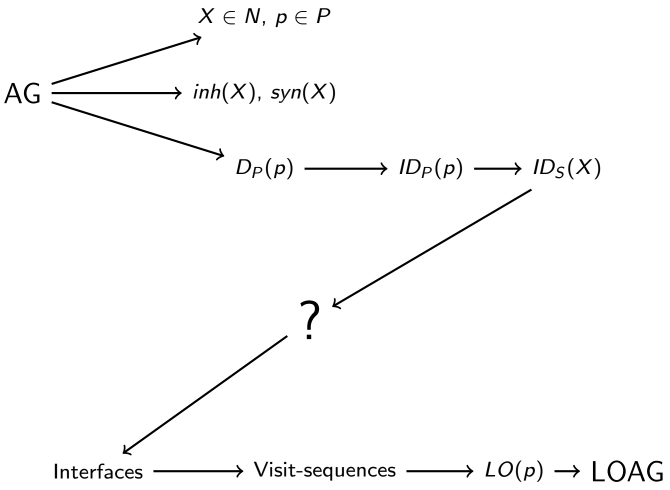
Scheduling the Utrecht Haskell Compiler (UHC)

- UHC is partly generated from of a large number of AGs.
- The “main AG” is very 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*.

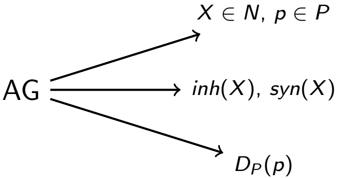
LOAG scheduling

- Kastens' algorithm recognises members of OAG $\subset$ LOAG.
- We have given two algorithms for LOAG:
  - AOAG: backtracking to find augmenting dependencies (paper).
  - LOAG: generate SAT-problem and give it to SAT-solver (future work).
- In the remainder of this talk we shall see:
  - A general method for determining whether an AG is a LOAG.
  - Why Kastens' algorithm does not implement this method.
  - Which dependencies are potential augmenting dependencies.
  - Our implementation that automatically selects augmenting dependencies.

Presentation overview



Presentation overview



UUAG - Non-terminals & Productions & Attributes

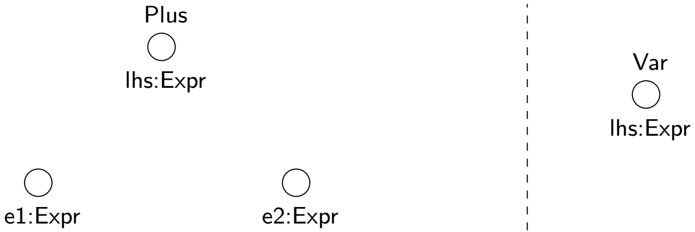
$$\begin{array}{llll}
\textbf{data Expr} & | & \textit{Plus} & e1 : \textbf{Expr}\ e2 : \textbf{Expr} \\
 & | & \textit{Min} & e1 : \textbf{Expr}\ e2 : \textbf{Expr} \\
 & | & \textit{Nat} & n : \texttt{Int} \\
 & | & \textit{Var} & id : \texttt{String}
\end{array}$$

$$\begin{array}{l}
\textbf{type}\ \texttt{Env} = [(\texttt{String}, \texttt{Int})] \\
\textbf{attr Expr} \\
\quad \textbf{inh}\ \textit{env} : \texttt{Env} \\
\quad \textbf{syn}\ \textit{val} : \texttt{Int} \\
\quad \textbf{syn}\ \textit{pp} : \texttt{String}
\end{array}$$

Terminology

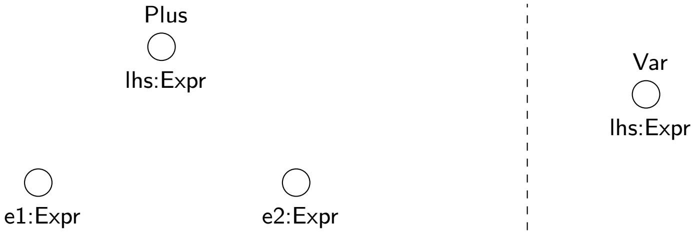
- We speak of three different kinds of attributes:
  - Attributes, assigned to a non-terminal.
  - Attribute occurrences, occurrences of attributes at productions.
  - Attribute instances, instances of occurrences in a parse-tree.
- Attribute occurrences are input- or output-occurrences:
  - Input: inherited of parent, synthesized of children.
  - Output: synthesized of parent, inherited of children.
- UUAGC requires descriptions to be *normalised*:
  - Every output-occurrence has a definition,
  - in terms of input-occurrences and terminals only.

UUAG - Production graphs



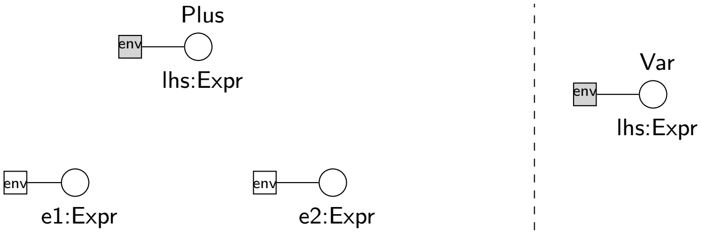
UUAG - Production graphs

attr Expr
  inh *env* : `Env`



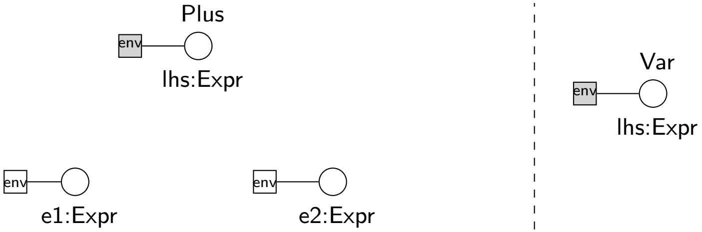
UUAG - Production graphs

attr Expr
  inh *env* : `Env`



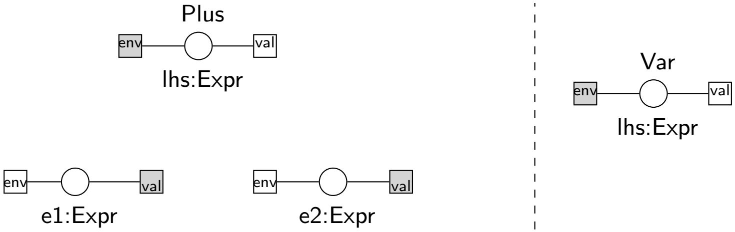
UUAG - Production graphs

attr Expr
  inh *env* : `Env`
  syn *val* : `Int`



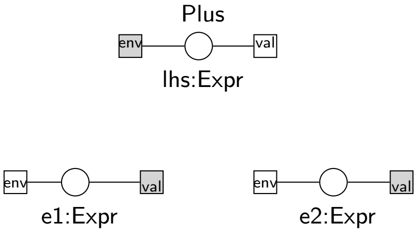
UUAG - Production graphs

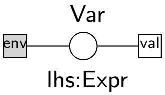
attr Expr
  inh *env* : `Env`
  syn *val* : `Int`



UUAG - Production graphs

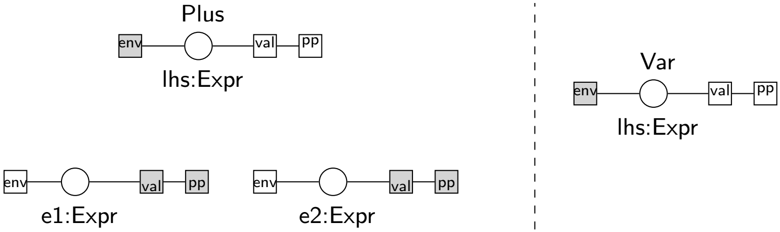
attr Expr
  inh *env* : `Env`
  syn *val* : `Int`
  syn *pp* : `String`





UUAG - Production graphs

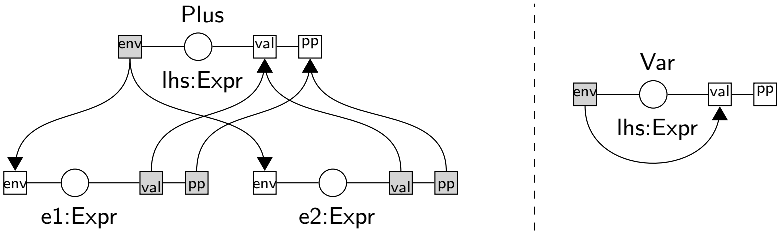
attr Expr
  inh *env* : `Env`
  syn *val* : `Int`
  syn *pp* : `String`



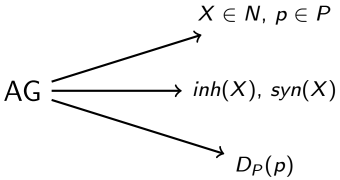
UUAG - Direct dependencies

```
sem Expr
  | Plus
    lhs.val = @e1.val + @e2.val
    lhs.pp  = "(" ++ @e1.pp ++ "+" ++ @e2.pp ++ ")"
  | Nat
    lhs.val = @n
    lhs.pp  = show @n
  | Var
    lhs.val = case lookup @id @lhs.env of
      Nothing → error ("Variable " ++ @id ++ " undefined")
      Just v  → v
    lhs.pp  = @id
```

Direct dependency graph - $D_P(p)$



Presentation overview



LOAG

LOAGs - Definition

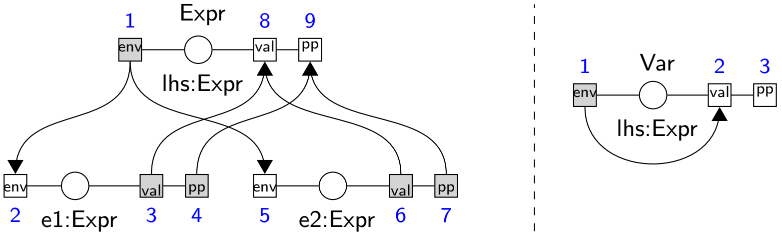
- $X_{p,i}$ is the i-th non-terminal in production p and is a *non-terminal occurrence* of some non-terminal $\mathcal{T}(X_{p,i}) \in N$.

Definition

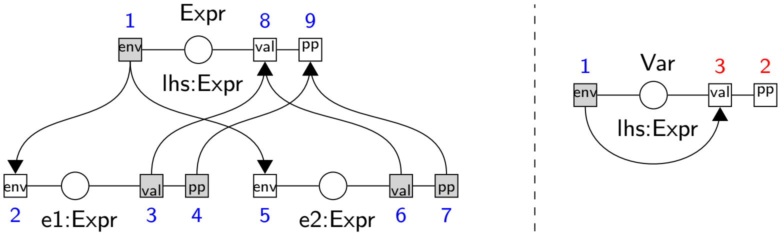
An AG is a *Linearly Ordered Attribute Grammar* (LOAG), if there exist linear orders $LO(p)$ for all $p \in P$ such that:

1. Every linear order $LO(p)$ respects the direct dependencies, i.e.
   if $(X_{p,i} \cdot a \rightarrow X_{p,j} \cdot b) \in D_P(p)$
   then $(X_{p,i} \cdot a < X_{p,j} \cdot b) \in LO(p)$.
2. The relative ordering of the attributes is the same for all occurrences of a non-terminal, i.e.
   if $\mathcal{T}(X_{p,i}) = \mathcal{T}(X_{q,j})$ and $(X_{p,i} \cdot a < X_{p,i} \cdot b) \in LO(p)$
   then $(X_{q,j} \cdot a < X_{q,j} \cdot b) \in LO(q)$ for all p, q, i and j.

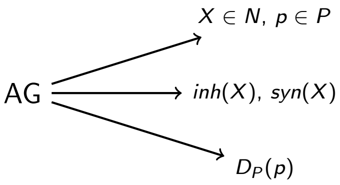
Linear Order for Expressions



Linear Order for Expressions



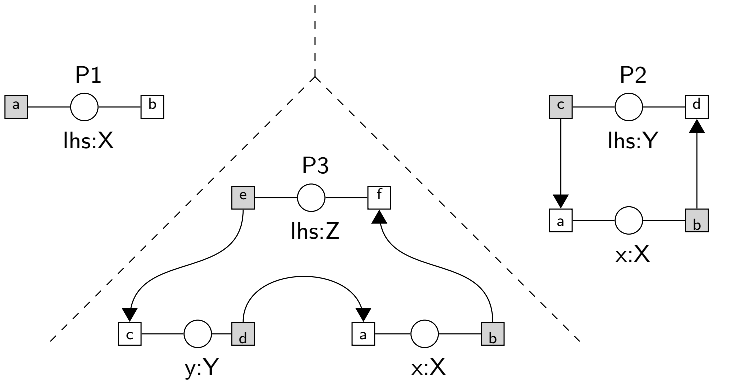
Presentation overview



$LO(p) \rightarrow$ LOAG

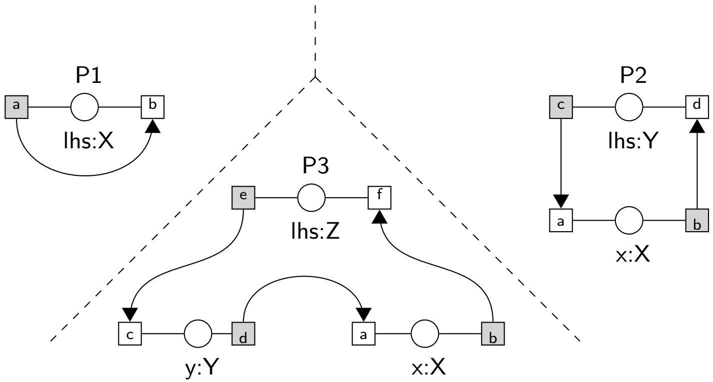
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \rightarrow X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \rightarrow Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \rightarrow X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



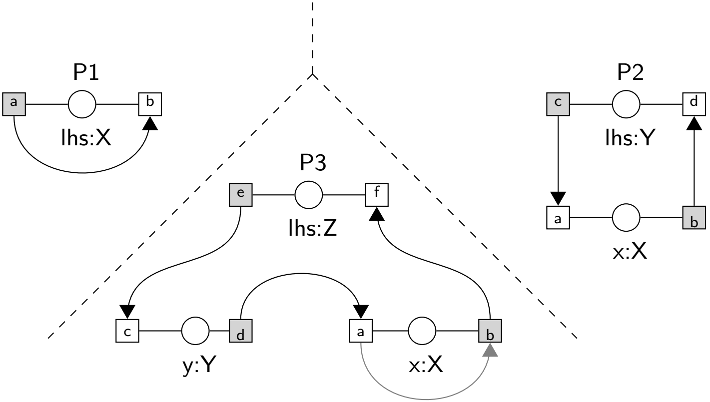
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \to Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



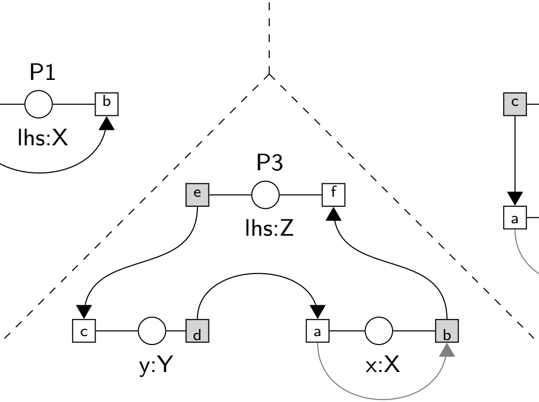
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \to Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



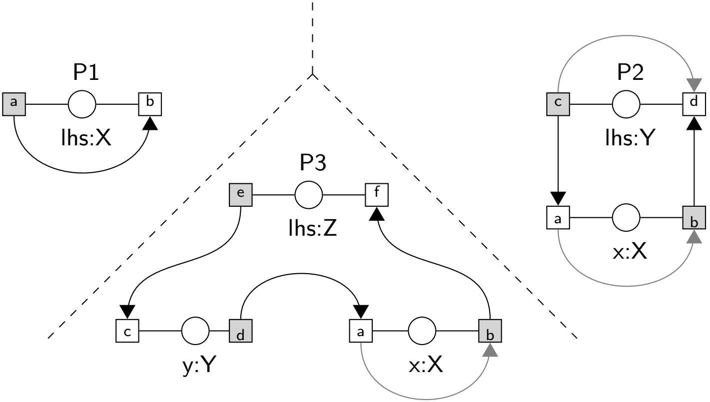
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \to Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



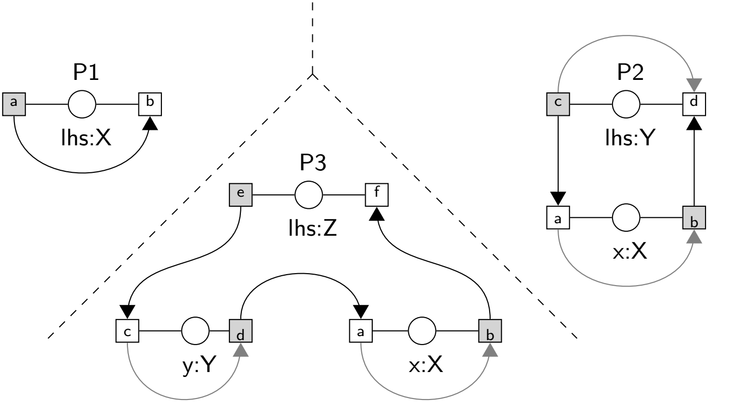
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \to Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



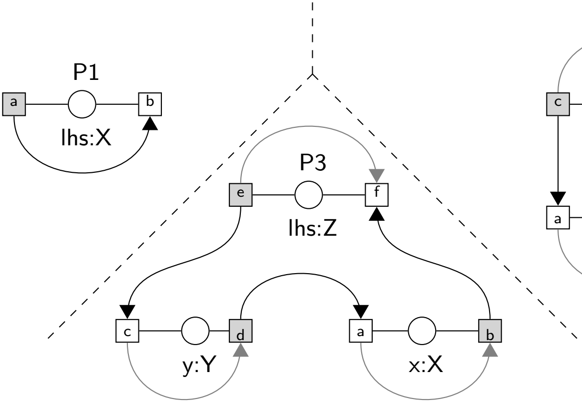
Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \rightarrow X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \rightarrow Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \rightarrow X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.

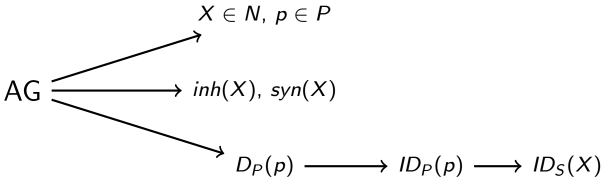


Induced dependency graphs - $ID_P(p)$, $ID_S(X)$

- Add all edges from $D_P(p)$ to $ID_P(p)$.
- If there is a path $(X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p)$
  - Add $(Y \cdot a \to Y \cdot b)$ to $ID_S(Y)$, where $Y = \mathcal{T}(X_{p,i})$
  - Add $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$ to $ID_P(q)$, for all $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$
- Continue until all paths have been propagated.



Presentation overview



$LO(p) \rightarrow$ LOAG

Presentation overview

AG $\longrightarrow$ $X \in N,\ p \in P$

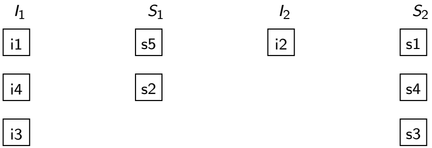
AG $\longrightarrow$ $inh(X),\ syn(X)$

AG $\longrightarrow$ $D_P(p) \longrightarrow ID_P(p) \longrightarrow ID_S(X)$

Interfaces $\longrightarrow$ Visit-sequences $\longrightarrow$ $LO(p)$ $\rightarrow$ LOAG

Interfaces

- An interface for $X \in N$ determines:
  - How many visits we use for X.
  - The inherited and synthesized attributes of every visit.
- An interface partitions all attributes of X in disjoint sets I_i, S_i such that (I_i, S_i) forms the i-th visit.



Visit-sequences

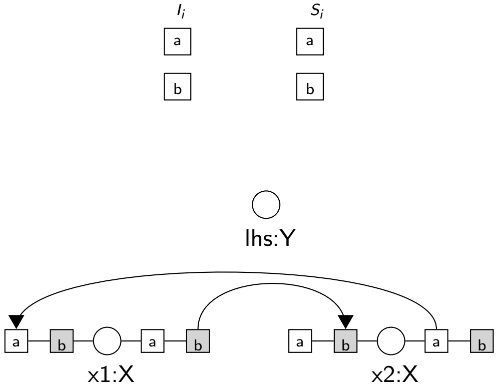
- Visit-sequences determine how every production of X executes every visit to X, such that:
  1. The j-th visit to X is executed after the i-th visit to X if $i < j$.
  2. Every synthesized attribute of a visit is evaluated.
  3. Every visit-instruction has to succeed the evaluation of the inherited attributes of the corresponding visit.
  4. If attribute a, depending on b, is evaluated in visit-sequence s:
     - 4.1 b is an inherited attribute of the visit, or
     - 4.2 b is produced by a visit-instruction in s before a.

1 : **eval** `e1`.*env*
2 : **visit** 1 `e1`
3 : **eval** `e2`.*env*
4 : **visit** 1 `e2`
5 : **eval** `lhs`.*val*
6 : **eval** `lhs`.*pp*

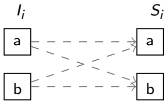
Visit-sequences

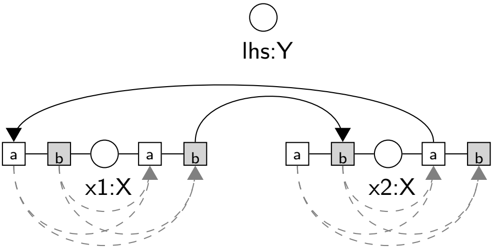
- The 4th property guarantees direct dependencies are respected.
- The first 3 properties guarantee interfaces are respected.
- Visit-sequences prove the AG is linearly ordered!
- However, creating interfaces introduces a third type of cycle.

Intra-visit dependencies

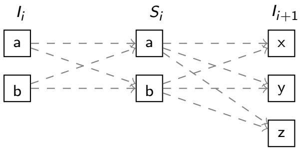


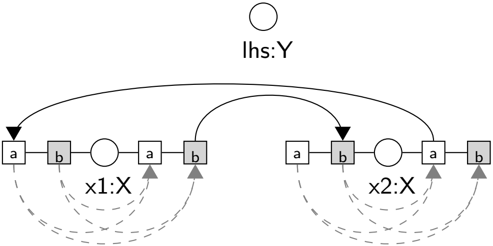
Intra-visit dependencies



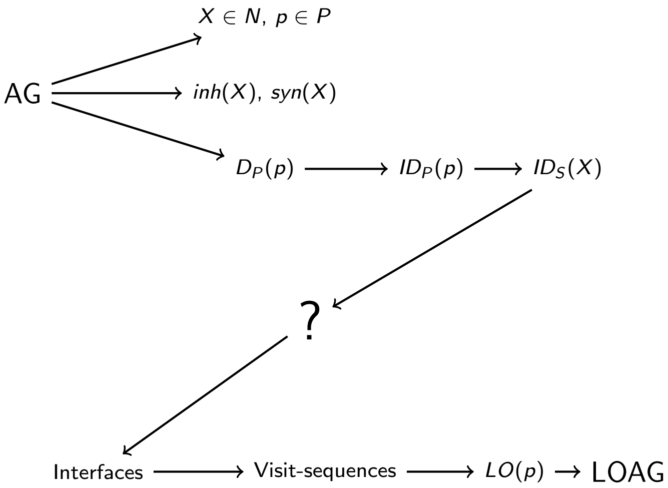


Intra-visit dependencies





Presentation overview



General procedure for LOAG

- Procedure:
  1. Build graphs $ID_S(X)$ from $D_P(p)$.
  2. Construct interfaces from $ID_S(X)$, such that the intra-visit dependencies do not contradict $D_P(p)$.
  3. Use the interfaces and $D_P(p)$ to build visit-sequences.
  4. Generate evaluation function for every visit-sequence.
- Step 2 is a combinatorial problem.

AOAG algorithm

- A candidate is a non-induced intra-visit dependency.

1. Compute interfaces like Kastens' algorithm.
2. Found a cycle without candidates: AG $\notin$ LOAG.
3. Found a cycle c with candidates:
   3.1 Select one, swap it, and propagate effects to interfaces.
      - Found a cycle without candidates: backtrack.
      - If all candidates in c are tried: failure or backtrack higher up.
      - Found a cycle with candidates: step.
      - Otherwise: AG $\in$ LOAG.
4. Otherwise: AG $\in$ LOAG.

Results

- We can now compile the UHC without manually adding augmenting dependencies.
- 10 corrections without backtracking.
- Most time is spent propagating dependencies: calculating and updating ID_S and interfaces.

Algorithm	Manual ADS?	main AG
Kastens'	Y	16.7s
AOAG	Y	5.9s
AOAG	N	12.6s
K&W	N	32.7s
LOAG	N	9.0s

- Backtracking will be costly.
- But we expect backtracking is not required for practical AGs.

Related Work

- Variants of OAG exist:
  - Chained Scheduling, Pennings 1994
  - The Eli System, Kastens et al. 1998
  - OAG*, Natori et al. 1999
- Polynomial algorithms for subclasses of LOAG.
- Can be combined with automatic augmenting dependency selection.

Future Work

Schedule optimisation

- With a static evaluation order it is possible to optimise.
- Optimise with respect to:
  - Runtime.
  - Space complexity.
  - Incremental evaluation.
  - etc.

Future Work

Code efficiency

- Formalise the costs of schedules by fixing an execution model.
- Possibly by developing a specialised virtual machine for AGs.
- Compare existing algorithms and the schedules they produce.
- Extend the algorithm(s) with user defined optimisations.