Attribute Evaluation

– Wilhelm/Maurer: Compiler Design, Chapter 9 –

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

Issues

- Separation into
  Strategy phase: Evaluation order is determined,
  Evaluation phase: Evaluation proper of the attribute instances directed by this evaluation strategy.
- Complexity of
  Generation: Runtime in terms of AG size,
  Evaluation: Size of evaluator, time optimality of evaluation.
- AG subclasses, hierarchy:
  Expressivity,
  Membership test,
  Generation algorithms,
  Complexity of generation and evaluation,
- Implementation issues.

Attribute Evaluation

Strategy phase: Determines the evaluation order, many approaches:

- Topological sorting of the individual dependency graph as in the dynamic evaluator,
- Fully predetermined at generation time, i.e. there is one fixed evaluation program for each production,

  pass oriented: Attributes are associated with passes over the tree,

  visit oriented: Attributes are associated with visits to production (instances),
- Selection between different precomputed evaluation orders, i.e. several precomputed evaluation programs for each production.

Evaluation phase: Alternatives,

data driven: Attribute instances are evaluated when arguments are available,

demand driven: demand for attribute values is recursively propagated, values are returned.

Implementation issues: Storage of attribute values:

- In the tree,
- On stacks,
- In global variables (shared by several instances of one attribute).

Attribute Grammar Classes

Membership test:

Dynamic: Evaluation for all trees is possible by a **defining evaluator**,

Static: Dependencies of the AG satisfy a **defining criterium**.

Example: Noncircular AGs,

dynamic criterium: defining evaluator is the dynamic evaluator, AG is noncircular iff topological sorting is possible for all individual dependency graphs,

static criterium: no cyclic graphs result from pasting lower char. graphs onto local graphs.

X–AG class of AGs with property X.
NC-AG class of noncircular AGs.
ANC-AG class of absolutely noncircular AGs.

Static Membership Tests

For all productions p:

- Paste graphs for $X_0, X_1, \dots, X_{n_p}$ onto $Dp(p)$,
- Check for cycles.
- Graphs (to be pasted) for smaller AG–classes
  - contain more edges, i.e. lead to cycles (and rejection) more often,
  - constrain more the evaluation strategy.

Complexity

Membership test:

- **NC–AG**: exponential,
- often same as that of evaluator generation, i.e. computation of global dependencies dominates evaluator generation.

Evaluation, time:

- no. of application of semantic rules plus
- tree walking effort plus
- construction of evaluation order.
- Optimality: at most one evaluation of each attribute instance + ?

Evaluation, space:

(static) size of the evaluator as function of the size of the AG,

(dynamic) space for attribute values and trees etc.

Space Complexity of the Dynamic Evaluator

Construction of evaluation order uses $Dt(t)$

Let

maxattr max. no. of attributes per non–terminal,

maxnont be max. no. of non–terminals in production right sides.

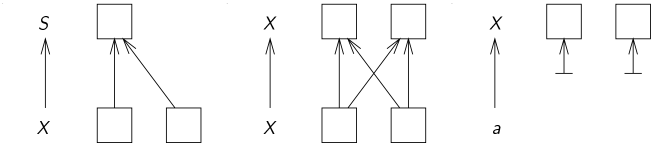
$$|Dp(p)| \leq ((maxnont + 1) \times \frac{1}{2} maxattr)^2$$

Let ap be no. of prod. applications in tree t,

$$|Dt(t)| \leq ap \times ((maxnont + 1) \times \frac{1}{2} maxattr)^2$$

Space complexity for topol. sorting is $O(maxattr^2)$

Dynamic Space



Demand driven evaluation,

- attribute values on a stack:
  needs a stack of depth $O(height(t))$ and t.
  Time complexity $O(4^{height(t)})$ or $O(2^{|V(t)|})$.
- atttribute values in the tree:
  Space complexity $O(|V(t)| + |t|)$ space and $O(|V(t)|)$ time.

Visit Oriented Evaluation

- Attribute (instance) evaluation happens during a sequence of visits to production instances,
- a **visit**
  - starts by descending from the upper context,
  - recursively visiting subtrees, and
  - ends by returning to the upper context.
- a (statically computed) **visit sequence** describes the evaluation of all attr. occ. of a production,
- there may be one or more visit sequences to a production,

one:	describes evaluation for all instances of the production in all trees,
several:	the right visit sequence for a production instance has to be determined from the context,

- the visit sequences (of productions) are computed from ordered partitions of the non–terminals occurring in the productions,
- an **ordered partition** for X splits $Attr(X)$ into a sequence of subsets associated with consecutive visits,
- ordered partitions for X are computed from a total order on $Attr(X)$,
- these total orders are computed from exact or approximate global dependency relations.

Total Orders on $Attr(X)$

- The first visit oriented evaluator is generated from a set of total orders $\{T_X\}_{X \in V_N}$.
- A total order T_X on $Attr(X)$ fixes the order of evaluation on $Attr(X)$,
- Total orders for different non–terminals (nodes in the tree) cannot be chosen independently, i.e., total orders at different nodes may be incompatible,
  $X \rightarrow Y$
  $Inh(X) = Inh(Y) = \{a, b\}$,
  $Syn(X) = Syn(Y) = \{c, d\}$
  $T_X = a\ c\ b\ d$, $T_Y = a\ d\ b\ c$

- An evaluation order $T(t)$ for a tree t **induces** at all nodes n total orders T_n on attributes, if
  for all $a, b \in \ Attr(symb(n)) \ a \ T_n \ b \ \Leftrightarrow \ a_n \ T(t) \ b_n$,
- Finding a set $\{T_X\}_{X \in V_N}$ of total orders as induced by trees is an NP–complete problem.

l–Ordered Attribute Grammars

AG is **l–ordered** (in **l–ordered–AG**) by a family of total orders $\{T_X\}_{X \in V_N}$ if

dynamic criterium: all trees t have an evaluation order $T(t)$ which induces T_X at nodes labelled with X, i.e. the dynamic evaluator can evaluate the attribute instances in all trees in the order given by the T_X,

static criterium: $Dp(p)[T_{p[0]}, T_{p[1]}, \ldots, T_{p[n_p]}]$ is acyclic for all productions p.

Testing for membership is as complex as constructing the total orders, namely NP–complete.

Ordered Attribute Grammars

Subset of the **l–ordered–AG**.
Use a polynomial heuristics to compute total orders $\{T_X\}_{X \in V_N}$
Step 1: Compute partial orders $\{R_X\}_{X \in V_N}$, the smallest relations satisfying

$$a_j \; Dp(p)[R_{X_0}, R_{X_1}, \ldots, R_{X_{n_p}}]^+ \; b_j \;\; \Rightarrow \;\; a \; R_{X_j} \; b$$

starting with $R_X = IO(X) \cup OI(X)$,
while changes **do**

1. Paste the R_X to the local dependency graphs,
2. Check whether new edges result for a non–terminal,
3. Add these new edges to the R_X.

This process terminates, since there are only finitely many attributes.

Ordered Attribute Grammars cont'd

Step 2: Compute the total orders $\{T_X\}$ from the $\{R_X\}$ by partitioning $Attr(X)$ into an alternating sequence $\iota^1\sigma^1\iota^2\sigma^2\ldots\iota^k\sigma^k$ of sets of inherited and synthesized attributes such that

- ι^j is (a total order on) the maximal set of the inherited attributes which can be evaluated when the attributes in $\iota^1\sigma^1\iota^2\sigma^2\ldots\iota^{j-1}\sigma^{j-1}$ are evaluated,
- σ^j is (a total order on) the maximal set of synthesized attributes which can be evaluated when the attributes in the $\iota^1\sigma^1\iota^2\sigma^2\ldots\iota^{j-1}\sigma^{j-1}$ are evaluated.

AG is **ordered** (is in **ordered-AG**),
if the relations $\{R_X\}_{X\in V_N}$ are all acyclic, and
if for all productions p:
$Dp(p)[T_{X_0}, T_{X_1}, \ldots, T_{X_{n_p}}]$ is acyclic,
where the $\{T_X\}_{X\in V_N}$ are computed as described above.

Evaluator Generation for Ordered AGs

Given: total orders T_X on $Attr(X)$,

1. Split T_X into an *ordered partition* of subsets of $Attr(X)$ to be evaluated during the same visit,
2. Local dependencies constrain how the visits at the non–terminals in a production may follow each other: From the ordered partitions of $X_0, X_1, \ldots, X_{n_p}$ and the local dependency graph of p generate a visit sequence for p,
3. From the set of visit sequences generate a recursive visit oriented evaluator rvE, a program performing the visits recursively traversing the trees.

Ordered Partitions in the scopes–AG

Attr(Decls)= Attr(Decl)= {it-env, e-env, st-env, ok}
The (only possible) total order is:

it-env st-env e-env ok

Splitting it into visits:

1. downward visit	*it-env*
1. upward visit	*st-env*
2. downward visit	*e-env*
2. upward visit	*ok*

Ordered partition:
it-env st-env e-env ok
Attr(Stms) = Attr(Stm) = {e-env, ok}
Total order: *e-env ok*

Ordered Partitions in the scopes–AG cont'd

Splitting it into visits:

1. downward visit	*e-env*
1. upward visit	*ok*

Ordered Partitions

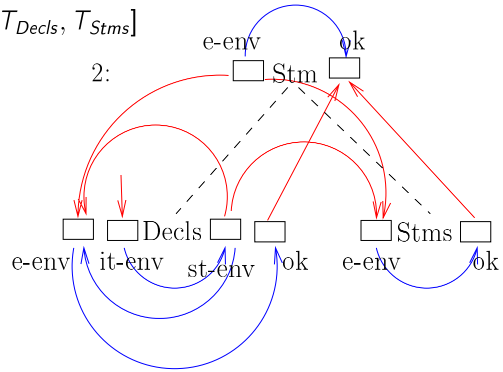
T total order on $Attr(X)$ seen as a word over $Attr(X)$.
An **ordered partition** for T is a dissection of T into a sequence $\iota^1\sigma^1\iota^2\sigma^2\ldots\iota^k\sigma^k$ where

- $\iota^j \in Inh(X)^*$, $\sigma^j \in Syn(X)^*$ for all $1 \leq j \leq k$,
- $\iota^j \neq \varepsilon$ for all $1 < j \leq k$,
- $\sigma^j \neq \varepsilon$ for all $1 \leq j < k$

- ι^j is the j-th **downward visit**,
- σ^j the j-th **upward visit**,
- $\iota^j\sigma^j$ the j-th **visit**.
- upper indices on ι and σ are **visit numbers**.
- the conditions $\iota^j \neq \varepsilon$ and $\sigma^j \neq \varepsilon$ guarantee maximal length of the substrings.

Visit Sequences for the Scopes-AG

$Dp(2)[T_{Stm}, T_{Decls}, T_{Stms}]$



A visit to production 2

1. starts with a downward visit from *Stm*, then
2. visits the *Decls*–subtree the first time, then either
    - visits the *Decls*–subtree the second time and then the *Stms*–subtree, or
    - visits the *Stms*–subtree and then the *Decls*–subtree the second time,
3. returns to the parent.

Visit Sequences

Let T_i be a total order on $Attr(X_i)$ such that $D = Dp(p)[T_0, T_1, \ldots, T_{n_p}]$ is acyclic.
Let $\iota_j^1 \sigma_j^1 \ldots \iota_j^{k_j} \sigma_j^{k_j}$ be the ord. partitions of T_j.
A **visit sequence** for p and $T_0, T_1, \ldots, T_{n_p}$ is an evaluation order for D of the following form:

$$V(p; T_0, T_1, \ldots, T_{n_p}) = \iota_0^1 \delta^1 \sigma_0^1 \; \iota_0^2 \delta^2 \sigma_0^2 \ldots \iota_0^k \delta^k \sigma_0^k$$

and δ^l is a sequence of visits $\iota_j^m \sigma_j^m$ at right side non–terminals X_j.
Thus, a visit sequence consists of a sequence of triples

1. a downward visit ι_0^l to X_0,
2. a sequence δ_l of visits $X_j (1 \leq j \leq n_p)$, and
3. an upwards visit σ_0^l to X_0.

Algorithm Visit Sequence

Input: local dependency graph $Dp(p)$,
total orders $\{T_i\}_{0\leq i\leq n_p}$ on $\{Attr(X_i)\})_{0\leq i\leq n_p}$ and their ordered partitions.

Output: a visit sequence $V(p; T_0, T_1, \ldots, T_{n_p})$

Method:

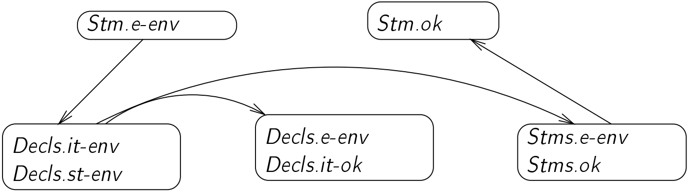
(1) construct a visit graph $\tilde{D}$ from $D = Dp(p)[T_0, T_1, \dots, T_{n_p}]$

- its vertices are:
  - $\iota_j^r \sigma_j^r$ $(1 \leq j \leq n_p)$, $\iota_j^r \sigma_j^r$ is the r-th visit of X_j (on the right side)
  - $\sigma_0^l \iota_0^{l+1}$ $(1 \leq l < k_0)$ (visit at parent), and
  - ι_0^1 und $\sigma_0^{k_0}$ first downwards from resp. last upwards visit to parent;
- there is an edge from x to y in $\tilde{D}$, if there are attribute occurrences a_i in x and b_j in y with $a_i \ D \ b_j$.

(2) Construct $V(p; T_0, T_1, \dots, T_{n_p})$ as an evaluation order for $\tilde{D}$, starting with ι_0^1 and ending with $\sigma_0^{k_0}$.

Executing Algorithm Visit Sequence

$\tilde{D} =$



Stm.e-env

Stm.ok

Decls.it-env
Decls.st-env

Decls.e-env
Decls.it-ok

Stms.e-env
Stms.ok

One visit sequence is:
Stms.e-env Decls.it-env Decls.st-env Decls.e-env Decls.ok
Stms.e-env Stms.ok Stm.ok

Recursive Visit Oriented Evaluator

- Evaluator as a program,
- Recursively traverses the trees,
- no. of visits to node n = length of ordered partition of $symb(n)$,
- At each production instance: executes the visits as indicated by the visit sequence.

The recursive visit oriented evaluator, **rvE**

```
program rvE;
proc visit_1(n : node);
  ⋮
proc visit_i(n : node);
begin
    case prod(n) of
        ⋮
    p : $V_i(p)$
        ⋮
    end case
end
  ⋮
begin
    visit_1($\varepsilon$)
end
```

Notation:

$V_i(p)$ program fragment for the i–th visit at p.

Let $\iota_0^i \iota_{j_1}^{i_1} \sigma_{j_1}^{i_1} \ldots \iota_{j_l}^{i_l} \sigma_{j_l}^{i_l} \sigma_0^i$ describe the i-th visit.
The following case-component $V_i(p)$ is constructed:

```
eval ($\iota_{nj_1}$); visit_$i_1$($nj_1$);
eval ($\iota_{nj_2}$); visit_$i_2$($nj_2$);
              ⋮
eval ($\iota_{nj_l}$); visit_$i_l$($nj_l$);
eval ($\sigma_0^i$)
```

Notation:

eval α is the sequence of semantic rules for the attribute occurrences in α.

rvE for the Scopes AG

```
program rvE_scopes;
proc visit_1(n : node);
begin
    case prod(n) of
        ⋮
    2 :         begin
                    eval($it\text{-}env_{n1}$);  visit_1(n1);
                    eval($e\text{-}env_{n1}$);  visit_2(n1);
                    eval($e\text{-}env_{n2}$);  visit_1(n2);
                    eval($ok_n$);
                end
    4 :         begin
                    eval($it\text{-}env_{n1}$);  visit_1(n1);
                    eval($it\text{-}env_{n2}$);  visit_1(n2);
                    eval($st\text{-}env_n$);
                end
        ⋮
    end case
end ;
```

```
proc  $visit\_2(n : \mathbf{node})$;
begin
    case  $prod(n)$ of
        ⋮
    2 :         begin
                    $eval(e\text{-}env_{n1})$;  $visit\_2(n1)$;
                    $eval(e\text{-}env_{n2})$;  $visit\_2(n2)$;
                    $eval(ok_n)$;
                end
        ⋮
    end case
end ;
begin
    $visit\_1(\varepsilon)$
end .
```

The recursive visit oriented evaluator, **rvE**

```
program rvE;
proc visit_1($n$ : node);
  ⋮
proc visit_$i$($n$ : node);
begin
    case $vs(n)$ of
        ⋮
    $V(p; T_0, T_1, \ldots, T_{n_p}) : V_i(p; T_0, T_1, \ldots, T_{n_p})$
        ⋮
    end case
end
  ⋮
begin
    visit_1($\varepsilon$)
end
```

Notation:

$V_i(p)$ program fragment for the i–th visit at p.

Parser Directed Attribute Evaluation

Method:

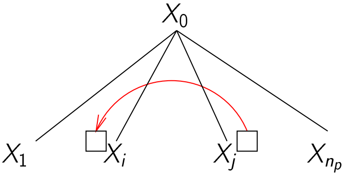
- Parser actions trigger attribute evaluation,
- Attribute values on a stack,
- No tree built.

Restrictions:

- Only “one pass” dependencies,
- “Horizontal” dependencies must correspond to parsing direction, i.e. no right-to-left dependencies,
- Inherited attributes and bottom up–parsing?

L–Attributed Grammars

- Parsers read/expand/reduce from left to right,
- Cannot trigger atttribute evaluation along right-to-left dependencies,



Right-to-Left Dependency

L–AG

- Superclass of all AGs with parser directed evaluation,
- Attributes can be evaluated in one left-to-right traversal of the tree,
- **S–AG** allow only synthesized attributes
  - subclass of **L–AG**,
  - fits bottom up parsing, e.g. BISON

L–AG, Defining Evaluator

```
program L-AE;
proc    visit ($n$ : node)
        case    prod ($n$) of
            ⋮
            $p$ :   begin
                      eval (Inh ($X_1$)); visit ($n1$);
                      eval (Inh ($X_2$)); visit ($n2$);
                      ⋮
                      eval (Inh ($X_{n_p}$)); visit ($nn_p$);
                      eval (Syn ($X_0$));
                    end ;
            ⋮
        endcase
end ;
begin
visit($\varepsilon$)          (*Start at root; inh. attr. of the root,
                        if existing, must have given values*)
end .
```

L–AG Definition

dynamic criterium: all attributes instances must be evluable by the defining interpreter,

static criterium: "no right-to-left dependencies",
formally for each $p : X_0 \longrightarrow X_1 \ldots X_{n_p}$
and each semantic rule $a_i = f_{p,a,i}(b^1_{j_1}, \ldots, b^k_{j_k})$:
$a \in Inh(X_i)$ and $1 \leq i \leq n_p$, implies $j_l < i$ for all l $(1 \leq l \leq k)$,
inherited attributes on the right side may only depend on

- inherited attributes of the left side and
- synthesized attributes on the right side occurring "before" them.

Short-Circuit Evaluation of Boolean Expressions

The C language standard is very consequent about the order of evaluation of expressions:

- the order is undefined for most operators
- the order is left-to-right for **&&** , ||, and ,.
- evaluation of Boolean expressions formed with **&&** , || terminates as soon as the value of the whole (sub-)expression is determined, short-circuit evaluation.

The following attribute grammar describes optimal code generation for short-circuit evaluation.

attribute grammar BoolExp

nonterminals *IFSTAT, STATS, E, T, F;*

attributes inh *tsucc*, *fsucc* with *E*,*T*,*F* domain string*;*
syn *jcond* with *E*,*T*,*F* domain bool*;*
syn *code* with *IFSTAT*, *E*,*T*,*F* domain string*;*

```
rules
IFSTAT → if E then STATS else STATS fi
  E.tsucc = t
  E.fsucc = e
  IFSTAT.code = E.code ++ gencjump (not E.jcond, e) ++
  t: ++ $STATS_1$.code ++ genujump (f) ++ e: ++ $STATS_2$.code ++ f:
E → T
E → E or T
  $E_1$.fsucc = t
  $E_0$.jcond = T.jcond
  $E_0$.code = $E_1$.code ++ gencjump ($E_1$.jcond, $E_0$.tsucc ) ++ t: ++ T.code T → F
T → T and F
  $T_1$.tsucc = f
  $T_0$.jcond = F.jcond
  $T_0$.code = $T_1$.code ++ gencjump (not $T_1$.jcond, $T_0$.fsucc ) ++ f: ++ F.code
F → (E)
F → not F
  $F_1$.tsucc = $F_0$.fsucc
  $F_1$.fsucc = $F_0$.tsucc
  $F_0$.jcond = not $F_1$.jcond
F → id
  F.jcond = true
  F.code = LOAD id.identifier
```

AG BoolExp is in L–AG.

Parser Directed Evaluation

The necessary functions for attribute evaluation:

1.	$eval(Inh(X))$	when starting to analyze a word for X,
2.	$eval(Syn(X))$	after finishing to analyze a word for X, i.e. when reducing to X,
3.	$get(Syn(X))$	when reading a terminal X.

Can be triggered by an LL–parser

1. upon expansion,
2. upon reduction,
3. upon reading.

An AG in **L–AG** is **LL–AG** if the underlying CFG is LL-grammar.
AG BoolExp is not in **LL–AG**, since the underlying CFG is left recursive.

Implementation of LL–Attributed Grammars

For the assignment of stack addresses we list the sets $Attr(X)$.

$LInh(X)$	List of inherited attributes of X.
$LSyn(X)$	List of synthesized attributes of X.

Two Stacks,

- Parse stack, PS,
- Attribute stack, AS.

Invariant(PS,AS):
Contents(PS) $= [A_1 \rightarrow \alpha_1.\beta_1]\ [A_2 \rightarrow \alpha_2.\beta_2] \ldots [A_n \rightarrow \alpha_n.\beta_n]$
$\Rightarrow$ contents(AS) $=$
values($LInh(A_1)\ LSyn(\alpha_1)\ LInh(A_2)\ LSyn(\alpha_2) \ldots LInh(A_n)$ $LSyn(\alpha_n)$)

Stack Situations

Expansion of a non–terminal B

$[A \rightarrow \alpha.B\beta]$

$LInh(A)$ $LSyn(\alpha)$

$[A \rightarrow \alpha.B\beta]$ $[B \rightarrow .\gamma]$

$LInh(A)$ $LSyn(\alpha)$ $LInh(B)$

Reading a terminal symbol a

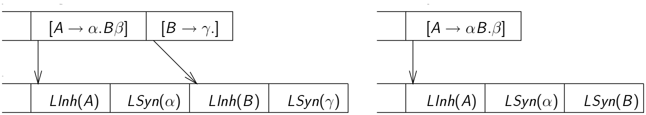
$[A \rightarrow \alpha.a\beta]$

$LInh(A)$ $LSyn(\alpha)$

$[A \rightarrow \alpha a.\beta]$

$LInh(A)$ $LSyn(\alpha)$ $LSyn(a)$

Reduction by $B \rightarrow \gamma$



$[A \rightarrow \alpha.B\beta]$ $[B \rightarrow \gamma.]$

$LInh(A)$ $LSyn(\alpha)$ $LInh(B)$ $LSyn(\gamma)$

$[A \rightarrow \alpha B.\beta]$

$LInh(A)$ $LSyn(\alpha)$ $LSyn(B)$

LR–Parser Directed Attribute Evaluation

- Calls to sematic rules triggered by reductions,
- Suffices for S–attributed grammars,
- For inherited attributes: Grammar transformation introduces "trigger non–terminals".

Trigger non–terminals N

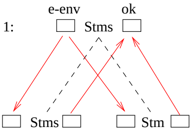
- have one production $N \longrightarrow \varepsilon$,
- are inserted in right production sides before a non–terminal with inherited attributes,
- this may change the grammar properties, e.g. LR(k),
- reduction to N triggers the evaluation of these attributes,

AG is **LR–Attributed (is in LR–AG)** if the underlying CFG of the transformed AG is LR.

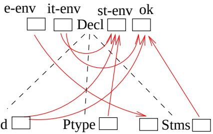
AG BoolExp is not LR–attributed, i.e. the transformation makes the underlying CFG non–LR.

Local Dependencies in the Scopes-AG

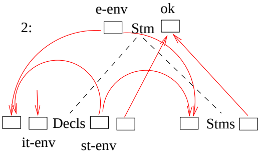
1:



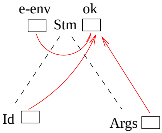
5:



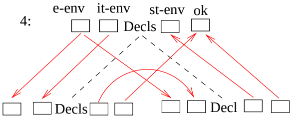
2:



6:



4:



Generation Time – Evaluation Time

Gen. Time	Eval. Time
tot. orders T_X on $Attr(X)$ for all $X \in V_N$	tree t mit $\{T_n\}_{n \in \text{nodes}\,(t)}$ $prod(n) = p, (T_0, T_1, \dots, T_{n_p})$
$\downarrow$	$\Downarrow$
ordered partition for $Attr(X)$	$B(p; T_{n0}, T_{n1}, \dots, T_{nn_p})$
$\downarrow$	⇣
visit sequences $B(p; T_0, T_1, \dots, T_{n_p})$ for $p \in P$, T_i tot. order on $Attr(p[i])$ $\rightarrow$	rbA, recursive visit-oriented evaluator

$B \;\rightarrow\; A$ stands for " A computed from B at gen. time",

$A \;\Rightarrow\; B$ stands for " A uniquely determines B ",

$A \cdots\!> B$ stands for " A is used in B ".