Storage Allocation Strategies for Recursive Attribute Evaluators

Kazunori Mizushima[1] and Takuya Katayama[2]

[1] PFU Limited, 658-1 Tsuruma Machida Tokyo 194-8510, Japan
[2] JAIST, 1-1 Asahidai Nomi Ishikawa 923-1292, Japan

1 Introduction

One of the features of attribute grammars is their evaluators can be generated automatically by mechanical transformations, and many research efforts have been focused on generating efficient evaluators. Generally, it is considered that the efficiency depends mainly on storage allocation for attributes. However there is a correlation between the storage allocation and evaluation order of attributes[1, 2]. This makes the allocation difficult.

Most researches on the storage allocation are divided into two categories: one gives priority to evaluation order and the other to storage allocation. The former method such as [1, 3, 4, 5, 11] first firms up the evaluation order, and then allocates attributes to storage. Conversely, the latter method such as [2, 7, 10, 13] first makes some allocation for attributes, and then examines whether an evaluable order exists under that allocation or not and chooses evaluable and the most optimized one. The latter allows us to get more optimized allocation than the former. However, the allocation problem of the latter is known to be NP-complete [2, 13].

In this paper we propose an allocation strategy in order to solve the problem within a practical amount of time. We suppose the class of attribute grammars is absolutely non-circular and their evaluator is a recursive attribute evaluator proposed in [6]. The structure of this paper is as follows. In Section 2, we explain notations used in this paper and the recursive evaluator. In Section 3, restrictions on the allocation are introduced and a formal definition of allocation problem is given. Reduction theorems to solve the allocation problem within a practical amount of time are proved in Section 4. Basic method called single-chain is suggested in Section 5 and two improved methods are suggested in Section 6. Section 7 is the conclusion.

2 Notations and Recursive Evaluators

Attribute grammars are extension of context-free grammars (abbreviated CFGs) and are defined by (G, A, F). $G = (V_N, V_T, P, S)$ is a CFG, where V_N, V_T, P, and S respectively stand for nonterminals, terminals, production rules, and a start symbol. Each production rule $p \in P$ is represented as $p : X_0 \to X_1X_2 \dots X_n$ neglecting terminal symbols.

A is a set of attributes. Each nonterminal $X \in V_N$ has a subset $A[X]$ of A. $A[X]$ is a disjoint union of the set $INH[X]$ of inherited attributes, which pass its value from the root of a derivation tree to the leaves, and the set $SYN[X]$ of synthesized attributes, which pass its value from the leaves to the root. When a nonterminal $X_k (0 \le k \le n)$ in a production rule $p : X_0 \to X_1 X_2 \ldots X_n$ has an attribute $a \in A[X_k]$, we say that p has an attribute occurrence $a.X_k$.

F is a set of semantic functions. We assume that semantic functions are in Bochmann normal form, and a semantic function $f_{p,a.X_k}$ is associated with attribute occurrence $a.X_k$ such that $a \in SYN[X_0]$ or $a \in INH[X_k]$ $(1 \le k \le n)$. $f_{p,a.X_k}$ specifies how to compute the value of $a.X_k$ from values of other attribute occurrences in the rule p. We call the special semantic function $v = w$ a copy rule where w is an attribute occurrence.

Semantic functions in the rule p yield dependency graph among attribute occurrences. This is called the attribute dependency graph $DG_p = (DV_p, DE_p)$ for the production rule p:

$$DV_p = \{a.X_k \mid 0 \le k \le n,\ a \in A[X_k]\}$$
$$DE_p = \{(v_1, v_2) \mid v_2 \text{ needs } v_1 \text{ for its evaluation}\}$$

We call the edge of copy rule a *copy edge* and call other edges *function edges*.

Let the production rule $p : X_0 \to X_1 X_2 \ldots X_n$ be applied at the root of derivation tree T and let T_k be the k-th subtree of T. A dependency graph $DG[T]$ for the T is recursively constructed in the following way:

$$DG[T] = (DV[T], DE[T]),$$
$$DV[T] = DV'_p \cup \bigcup_{k=1}^{n} DV[T_k],$$
$$DE[T] = DE'_p \cup \bigcup_{k=1}^{n} DE[T_k],$$

and $DG'_p = (DV'_p, DE'_p)$ is the graph obtained from DG_p by replacing every attribute occurrence $a.X_k$ in the p by the corresponding attribute instance $a.X_k.n_k$ in the tree T, where n_k is the root node of T_k.

For any tree T with the root labeled by $X_0 \in V_N$, if a path from $i.X_0.n_0$ $(i \in INH[X_0])$ to $s.X_0.n_0$ $(s \in SYN[X_0])$ exists, we write $(i, s) \in IO[X_0]$ and call it an *io edge*. The augmented dependency graph DG^*_p for the production rule p is defined as follows:

$$DG^*_p = (DV^*_p, DE^*_p),$$
$$DV^*_p = DV_p,$$
$$DE^*_p = DE_p \cup \{(a.X_k, b.X_k) \mid 1 \le k \le n,\ (a, b) \in IO[X_k]\}.$$

Example 1 (Attribute Grammar G1). Attribute grammar $G1$ and its augmented dependency graphs is illustrated in Fig. 1. This grammar computes the value and the length of fractional binary notation[6].

G = (V_N, V_T, P, S)
S = F
V_N = { F, L, B }
V_T = { 0, 1 }

Attributes
INH[F] = $\emptyset$
SYN[F] = {val, len}
INH[L] = {pos}
SYN[L] = {val, len}
INH[B] = {pos}
SYN[B] = {val}

Production Rules	*& Semantic Functions*
p0: F → L	pos.L = 1
	val.F = val.L
	len.F = len.L
p1 : L → B	pos.B = pos.L
	val.L = val.B
	len.L = pos.L
p2 : L → B L_1	pos.B = pos.L
	pos.L_1 = pos.L + 1
	val.L = val.B + val.L_1
	len.L = len.L_1
p3 : B → 0	val.B = 0
p4 : B → 1	val.B = $2^{-\text{pos.B}}$

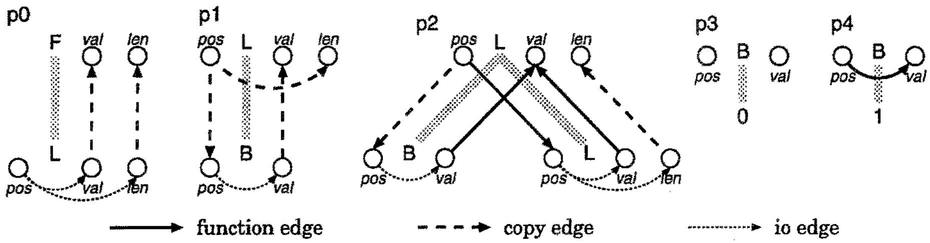


Fig. 1. An Attribute Grammar and Augmented Attribute Dependency Graphs DG_p^*.

2.1 Recursive Attribute Evaluators

The evaluator targeted in this paper is a recursive evaluator proposed in the literature [6]. The evaluator is able to evaluate the class of absolutely non-circular attribute grammars.

The recursive evaluator is constructed as a set of recursive procedures of the following form:

$$R_X(u_1, \ldots, u_m, T; v_1, \ldots, v_n),$$

where $X \in V_N$, $u_1, \ldots, u_m$ are input parameters corresponding to the inherited attributes for X, T is the derivation tree labeled X at its root, and $v_1, \ldots, v_n$ are output parameters corresponding to the synthesized attributes. These input and output parameters are determined from $IO[X]$. This procedure R_X is intended to evaluate the synthesized attributes $v_1, \ldots, v_n$ when it is supplied with the values of inherited attributes $u_1, \ldots, u_m$ and derivation tree T as its inputs. The procedure R_X takes the following form:

```
proc $R_X(u_1, \ldots, u_m, T; v_1, \ldots, v_n)$
    case production($T$) of
        $p_1$ : $H_{p_1}$
        $p_2$ : $H_{p_2}$
        ...
    end
end
```

```
proc $R_F(T; val.F, len.F)$
    var $pos.L, val.L, len.L$
    $pos.L \leftarrow 1$
    $R_L(pos.L, T_1; val.L, len.L)$
    $val.F \leftarrow val.L$
    $len.F \leftarrow len.L$
end

proc $R_B(pos.B, T; val.B)$
    case production($T$) of
    $p_3$ : $val.B \leftarrow 0$
        break
    $p_4$ : $val.B \leftarrow 2^{-pos.B}$
        break
    end
end

proc $R_L(pos.L, T; val.L, len.L)$
    case production($T$) of
    $p_1$ : var $pos.B, val.B$
        $pos.B \leftarrow pos.L$
        $R_B(pos.B, T_1; val.B)$
        $val.L \leftarrow val.B$
        $len.L \leftarrow pos.L$
        break
    $p_2$ : var $pos.B, val.B,$
                $pos.L_1, val.L_1, len.L_1$
        $pos.B \leftarrow pos.L$
        $R_B(pos.B, T_1; val.B)$
        $pos.L_1 \leftarrow pos.L + 1$
        $R_L(pos.L_1, T_2; val.L_1, len.L_1)$
        $val.L \leftarrow val.B + val.L_1$
        $len.L \leftarrow len.L_1$
        break
    end
end
```

T_k means k-th subtree of T.

Fig. 2. Recursive Evaluator for $G1$

where $p_1, p_2, \cdots$ are productions whose left-hand side symbol is X and $H_{p_1}, H_{p_2}, \cdots$ is a sequence of assignment or procedure call statements.

The procedure R_X determines the production rule p applied at the root of T by the function **production**, and it executes a sequence H_p of statements.

The sequence H_p of statements $ST[a]$'s computes the values of attribute occurrences in p in the topological order of DG_p^*. If a is an inherited attribute occurrence of X_i $(i > 0)$ or a synthesized occurrence of X_0, then $ST[a]$ is an assignment $a \leftarrow f_{p,a}(z_1, \ldots, z_r)$. If a is a synthesized attribute of X_i $(i > 0)$, then $ST[a]$ is a procedure call R_{X_i}.

The recursive evaluator keeps each value of attribute instance in its activation record as a local variable of the procedure.

Example 2 (Recursive Evaluator). Recursive procedures for $G1$ are illustrated in Fig. 2. In this form all attributes need their own storage as local variable.

3 Storage Allocation Problem

Attribute evaluation is to calculate synthesized attribute values in the root node of a derivation tree. In order to calculate them, it is necessary to determine other attribute values in the tree and to store these values somewhere for later references. Of course, we cannot blindly allocate storage to attributes because some values might be overwritten before their reference. Storage allocation is required to be at once evaluable and optimum. The difficulty arises here because these two requirements contradict one another. First we formulate the storage allocation problem in this section.

3.1 Restrictions on Allocation

Storage allocation for recursive evaluator studied in [7] and [10] takes exponential time for finding an optimum allocation. This is because these methods provide only evaluability test method and no allocation strategy is given, and it is necessary to examine all the cases exhaustively to get the allocation which is evaluable and optimum. This exponential time is an obstacle for practical use of these methods.

In order to get almost optimal allocation within a practical amount of time, we impose the following two restrictions on storage allocation:

1. *It's allowed to share storage with directly dependent attributes.*
   This restriction is acceptable because attribute values are propagated one after another along the dependency. This restriction will not only save storage spaces but also reduce attribute evaluation time since we may have chances of updating portions of big structured data instead of constructing the whole new values. Especially, if of a copy rule which just passes a value from one attribute to another, this effect is drastic.
2. *Even if some attributes directly depend on one attribute in multi-casting style, at most one directly depended pair is allowed to share.*
   Generally, when storage for an attribute is shared with another, the value of the attribute instance might be destroyed in the evaluation of other attributes. Therefore this restriction is natural and acceptable.

These two restrictions are convenient for recursive evaluator, because the evaluator prepares storage in activation record when it is needed and storage allocation is simply implemented by passing a pointer of the storage as an argument of procedures.

Formal definition of this restriction is given below. First storage allocation is formalized in terms of a shared edge to represent the former restriction. A shared edge means its both end nodes share the same storage.

Definition 1 (Shared Edge Set for Production S_p). *Shared edge set S_p for the dependency graph $DG_p = (DV_p, DE_p)$ of a production rule p is defined as*

$$S_p = \{\ (a,b) \mid a,b \in DV_p, a \textit{ and } b \textit{ share a common storage}\ \}$$

If attribute occurrences a and b share the same storage in a rule p $((a,b) \in S_p)$, it is natural to share the both ends of the corresponding attribute instances of (a',b') in any derivation tree T. Hence, a shared edge set $S[T]$ for derivation tree T is defined as follows.

Definition 2 (Shared Edge Set for Derivation Tree $S[T]$). *Let p be a production rule $X_0 \to X_1 \cdots X_n$ applied at the root of a derivation tree T, T_k be a k-th subtree of T, and S_p be a shared edge set for p. Shared edge set $S[T]$ for a derivation tree T is recursively constructed from $S_p, S[T_1], \cdots, S[T_n]$ as follows:*

$$S[T] = S'_p \cup \bigcup_{k=1}^{n} S[T_k]$$

where, $(a.X_i.n_i, b.X_j.n_j) \in S'_p \iff (a.X_i, b.X_j) \in S_p$.

Now we are ready to define the restrictions. These restrictions are for production rules and derivation trees, so that we write dependency graph as DG and shared edge set as S for both.

Definition 3 (Restrictions on Allocation). *Let $DG = (DV, DE)$ be an attribute dependency graph and S be a shared edge set. The following restriction is imposed on S:*

1. $S \subset DE$,
2. *at most only one y satisfies $(x, y) \in S$ for any x.*

3.2 Evaluation Order and Pebbling

On the basis of these restrictions, we will consider evaluable attribute order and optimal shared edge set.

To define evaluability we also apply the notion of "*pebbling*" to attribute grammars as in [7] and [13]. Pebbling is a computation sequence which never erroneously destroys any stored value. It has been introduced by Sethi [12] as a computation model for the storage allocation problem of directed acyclic graph (DAG).

As for attribute grammars, if there is a pebbling of attributes under a shared edge set, it is an evaluable sequence. As the shared edge set is defined for production rules and derivation trees, pebbling is also defined for both of them. In the following definitions, we write $a \prec b$ to denote that a is placed before b in a pebbling.

Definition 4 (Pebbling for Production Rules). *Let p be a production rule $X_0 \to X_1 \cdots X_n$. An evaluation sequence P_p for (DG^*_p, S_p) is a pebbling if the following three conditions are satisfied:*

$$P_p\langle DG^*_p\rangle = \langle v_1, v_2, \ldots, v_M\rangle$$

1. $DV^*_p = \{v_1, v_2, \ldots, v_M\}$,
2. *if $(v_i, v_j) \in DE^*_p$ then $v_i \prec v_j$,*
3. *if $(v_i, v_j) \in DE^*_p$ then there is no v_k such as*

$$v_i \prec v_k \prec v_j \ \wedge \ (v_i, v_k) \in S_p.$$

We call DG^*_p is evaluable under S_p if there is a pebbling which satisfies the above three conditions.

Definition 5 (Pebbling for Derivation Trees). *Let T be a derivation tree. An evaluation sequence P_T for $(DG[T], S[T])$ is a pebbling if the following three conditions are satisfied:*

$$P_T\langle DG[T]\rangle = \langle v_1, v_2, \ldots, v_M\rangle$$

1. *$DV[T] = \{v_1, v_2, \dots, v_M\}$,*
2. *if $(v_i, v_j) \in DE[T]$ then $v_i \prec v_j$,*
3. *if $(v_i, v_j) \in DE[T]$ then there is no v_k such as*

$$v_i \prec v_k \prec v_j \ \wedge \ (v_i, v_k) \in S[T]^+$$

where $S[T]^+$ is transitive closure of $S[T]$.

We call $DG[T]$ is evaluable under $S[T]$ if there is a pebbling which satisfies above three conditions.

3.3 Weighting Function

As a measure of allocation, we introduce a weighting function which is a mapping from edges to weight values. A weight of an edge represents a merit if both ends of the edge share a common storage.

Definition 6 (Weighting Function $w[T]$). *Let p be a production rule $X_0 \to X_1 \cdots X_n$ applied at the root of derivation tree T, T_k be a k-th subtree of T, $w_p : DE_p \to \mathbf{N}$ be a weighting function for p. Weighting function $w[T]$ for a derivation tree T is recursively constructed from $w_p, w[T_1], \cdots, w[T_n]$ as follows:*

$$w[T] = w'_p \cup \bigcup_{k=1}^{n} w[T_k]$$

where, $w'_p(\,(a.X_i.n_i, b.X_j.n_j)\,) \iff w_p(\,(a.X_i, b.X_j)\,)$.

Total weight is defined as follows:

$$W[T] = \sum_{e \in S[T]} w[T](e).$$

The heavier $W[T]$ becomes, the more optimized $S[T]$ becomes.

3.4 Formulation of Allocation Problem

Now we are ready to define storage allocation problem of attribute grammars.

Definition 7 (Storage Allocation Problem). *Storage allocation problem for $(DG[T], w[T])$ is to find a shared edge set $S[T]$ which has a pebbling P_T and maximizes the total weight $W[T]$.*

4 Reduction of Allocation Problem

A procedure of recursive evaluator is statically generated for each production rule, so that it is necessary to statically determine shared edge set at procedure generation time. This means that any derivation tree must be evaluable by static shared edge set for each production rule. This section gives reduction theorems that reduce the problem of derivation tree to that of production rules.

4.1 Two Reduction Theorems

Theorem 8 (Reduction Theorem of Evaluation). *If each DG_p^* is evaluable under S_p, any $DG[T]$ is evaluable under $S[T]$ which is recursively constructed from S_p by Definition 2.*

Proof. We prove this theorem by structural induction on the derivation tree T.
Basic step: This is trivial when derivation tree T is constructed from only one production rule p.
Induction step: Consider that derivation tree T is constructed from subtrees T_i $(1 \leq i \leq n)$ and a production rule $p : X_0 \rightarrow X_1X_2 \ldots X_n$ applied at the root. Assume each $DG[T_i]$ is evaluable by $S[T_i]$. The evaluation order of inherited and synthesized attributes of the root of $DG[T_i]$ is consistent with $IO[X_i]$ because evaluation sequence $P_T\langle DG[T_i]\rangle$ is consistent with dependency graph $DG[T_i]$. In Definition 5, pebbling is related only to the attribute dependency and shared edge set. When $DG[T]$ is recursively constructed from DG_p and $DG[T_1], DG[T_2], \ldots, DG[T_n]$, neither dependency nor shared edge is introduced and $IO[X_i]$ is still consistent with $P_T\langle DG[T_i]\rangle$.
Therefore, if DG_p^* is evaluable by S_p, $DG[T]$ is evaluable by $S[T]$. □

Theorem 8 means that allocation for each rule doesn't influence allocation for any other at all and each shared edge set can be determined independently.

Let the production rule $p : X_0 \rightarrow X_1X_2 \ldots X_n$ be applied at the root of derivation tree T and let T_k be the k-th subtree of T. Suppose S_p is an evaluable shared edge set for (DG_p, w_p) and total weight under the S_p is W_p, and $S[T_k]$ is an evaluable shared edge set for $(DG[T_k], w[T_k])$ and total weight under the $S[T_k]$ is $W[T_k]$. From this, the following relations are obvious by Definition 2 and Theorem 8:

$$\begin{aligned}(a.X, b.X) \in S_p \quad &\Longleftrightarrow (a.X.n, b.X.n) \in S[T],\\ (a.X.n, b.X.n) \in S[T_k] &\Longleftrightarrow (a.X.n, b.X.n) \in S[T].\end{aligned}$$

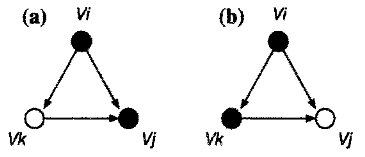
And total weight $W[T]$ under $S[T]$ for $(DG[T], w[T])$ is given by the next expression:

$$W[T] = W_p + \sum_{k=1}^{n} W[T_k].$$

Next theorem is an obvious consequence of this expression.

Theorem 9 (Reduction Theorem of Optimization). *For each production $p \in P$, let S_p be an evaluable shared edge set which makes W_p for (DG_p, w_p) maximize. Let $W[T]$ be constructed by Definition 6. $S[T]$ constructed by Definition 2 is the most optimized shared edge set which maximizes total weight $W[T]$ for $(DG[T], w[T])$.*

The above two theorems mean that storage allocation problem can be reduced from derivation tree level to production level under the restrictions in Definition 3.



Same pattern represents same storage.
(a) Evaluable by the sequence v_i, v_k, v_j.
(b) Not evaluable because before the evaluation of v_j the value of v_i is destroyed by the evaluation of v_k.

Fig. 3. An Example of Dependency and Shared Edge

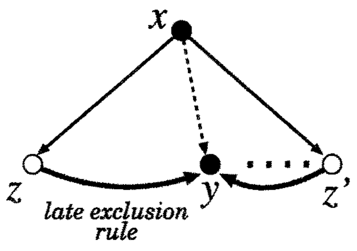


Fig. 4. Late Exclusion Rule

4.2 Decomposition to Bipartite Graphs

Usually, there are some superfluous edges which never become shared edges. Consider attribute dependencies $(v_i, v_j), (v_i, v_k), (v_k, v_j) \in DE_p^*$ in Fig. 3. Though it is possible for v_i and v_j to share, it is impossible for v_i and v_k to share because the value of v_i is destroyed by the evaluation of v_k before the evaluation of v_j. If these superfluous edges are removed from a dependency graph in advance, the graph becomes slimed and allocation will be simpler.

The slimed graph RDG_p is defined as follows. First we define RDG_p^*.

Definition 10 (RDG_p^*).

$$\begin{aligned} RDG_p^* &= (RDV_p^*, RDE_p^*) \\ RDV_p^* &= DV_p^* \\ RDE_p^* &= DE_p^* - \{\ (v_i, v_k)\ |(v_i, v_j), (v_i, v_k) \in DE_p,\ (v_k, v_j) \in DE_p^{*+}\ \} \end{aligned}$$

where DE_p^{*+} is transitive closure of DE_p^*.

Next, we define RDG_p by removing io edges from RDG_p^*.

Definition 11 (RDG_p).

$$\begin{aligned} RDG_p &= (RDV_p, RDE_p) \\ RDV_p &= RDV_p^* \\ RDE_p &= RDE_p^* - \{\ (i.X_k, s.X_k)\ |1 \le k \le n,\ i \in INH[X_k],\ s \in SYN[X_k]\ \} \end{aligned}$$

We write each connected element of RDG_p as $RDG_p^{(k)}$ $(k = 0, \ldots, N)$. $RDG_p^{(k)}$ is a bipartite graph because semantic function is assumed to be in Bochmann normal form. Shared edge set $S_p^{(k)}$ for $RDG_p^{(k)}$ is defined:

$$S_p^{(k)} = \{\ (x, y)|(x, y) \in RDE_p^{(k)}, x \text{ and } y \text{ share the same storage }\}.$$

The next reduction theorem says that storage allocation problem could be reduced to bipartite graph level.

Theorem 12 (Reduction Theorem for DG_p). *If each $S_p^{(k)}$ for $(RDG_p^{(k)}, w_p)$ is evaluable and optimum, S_p given below is also evaluable and optimum for (DG_p, w_p):*

$$S_p = \bigcup_{k=0}^{N} S_p^{(k)}$$

Proof. As for the optimization, it is obvious because each $RDG_p^{(k)}$ is disjoint and $S_p^{(k)}$ is a subset of $RDE_p^{(k)}$. As for the evaluability, we prove it by the construction of evaluation sequence. Let $P_p^{(k)}$ be a pebbling for $S_p^{(k)}$. We can construct evaluation sequence by $P_p\langle DG_p^*\rangle = \biguplus_{k=0}^{N} P_p^{(k)}$ where $A \uplus B$ is a merge of the sequences A and B in the order of DE_p^*. This merge is possible because each $P_p^{(k)}$ is consistent with DE_p^* from the definition of pebbling. Therefore this pebbling $P_p\langle DG_p^*\rangle$ satisfies three conditions of Definition 4, and evaluability is proved. □

5 An Allocation Method

5.1 Evaluability and Graph Transformation

If we share two attributes of an edge of bipartite graph $RDG_p^{(k)}$, new evaluation order is yielded. We represent this as a graph transformation and we call it late exclusion rule following [12](Fig. 4).

Definition 13 (Late Exclusion Rule and Closure). *Suppose we are interested in dependency graph $R = (DV, DE)$ and its shared edge set $S \in DE$. Let $x, y, z \in DV$ be distinct attributes and $(x,y)(x,z) \in DE$. If $(x,y) \in S$, then we say that DE transforms to $DE \cup \{(z,y)\}$ under the late exclusion rule, written $DE \Rightarrow DE \cup \{(z,y)\}$. When $R' = (DV, DE')$ and $DE \Rightarrow DE'$, we write $R \Rightarrow R'$.*

R is called late exclusion closed if $Q \Rightarrow R$ is false for any $R \neq Q$. We write $\Rightarrow^+$ for the transitive closure of $\Rightarrow$. If $R \Rightarrow^+ R^\sim$ and $R^\sim$ is late exclusion closed, we call $R^\sim$ the late exclusion closure of R. $R^\sim$ is the DAG which represents the evaluation sequence under the shared edge set S.

As for the late exclusion closure, Sethi[12] gives the following fact:

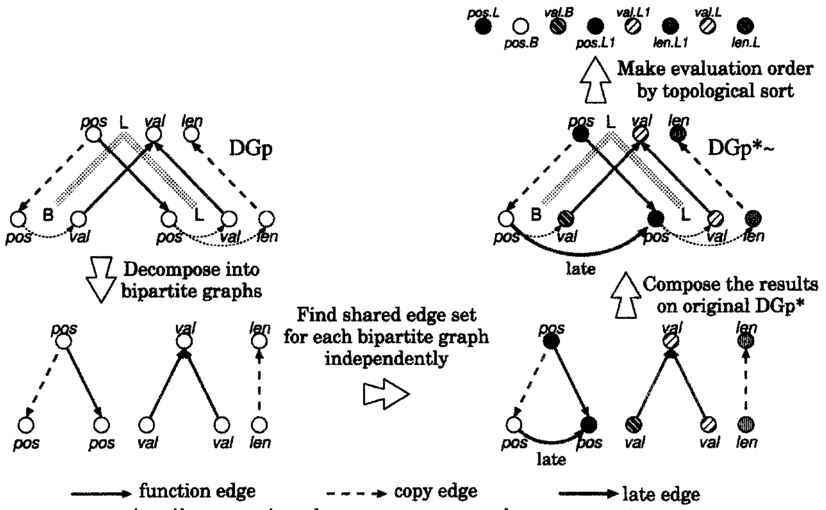
If $R^\sim$ is acyclic, a pebbling for (R, S) exists.

From this, we have only to examine whether closure $RDG_p^{(k)\sim}$ is acyclic or not instead of doing evaluability test of $RDG_p^{(k)}$.

5.2 Single-Chain Method

In this section, we introduce an allocation method called a single-chain method. This method finds shared edge set in the following way (see also Fig. 5):

1. For each production, decompose DG_p into bipartite graphs $RDG_p^{(k)}$.
2. For each $RDG_p^{(k)}$, compute shared edge sets for all the cases exhaustively, and then choose evaluable and the most optimized $S_p^{(k)}$.



pos.L
pos.B
val.B
pos.L1
val.L1
len.L1
val.L
len.L
Make evaluation order by topological sort
pos L val len
DGp
B
L
pos val pos val len
DGp*~
late
Decompose into bipartite graphs
Find shared edge set for each bipartite graph independently
Compose the results on original DGp*
pos val len
pos pos val val len
late
function edge
copy edge
late edge

Attributes painted a same pattern share same storage.

Fig. 5. Single-Chain Method

3. Construct evaluation order graph $DG_p^{*\sim}$ by augmenting new late exclusion rules to the original DG_p^*.

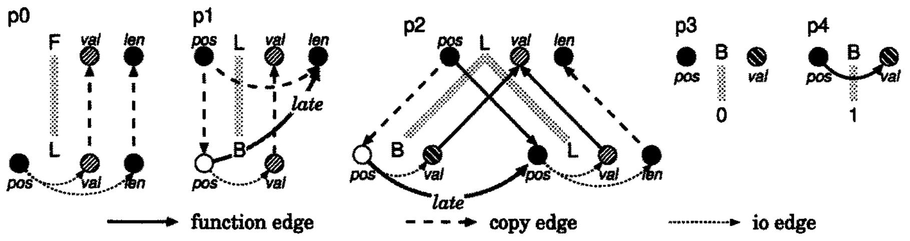
Although this method also test all the cases exhaustively as in [7] and [10], it finds shared edge set within a practical amount of time because decomposed bipartite graph $RDG_p^{(k)}$ is extremely small.

Example 3 (Allocation for $G1$*).* We applied single-chain method to $G1$ and constructed a recursive attribute evaluator. Comparing with Fig. 2, the number of storage is decreased and evaluator became simpler.

In the single-chain method, shared edge set for each production is determined independently. This might cause the case that shared pair of an inherited and a synthesized attributes of a nonterminal, which are input and output of a recursive procedure generated, differs from one production to another. As a single procedure is generated for the corresponding nonterminal, adjustment of these pairs is necessary to generate procedures consistently. The adjustment is done for Fig. 6.

6 Improvement of the Single-Chain Method

In allocating storage to attributes, we must not ignore copy rule which only passes a value from an attribute to another. In the description of language processors by attribute grammar, copy rules occupies high percentage of the description as reported in [3]. If we could share attributes of a copy rule, it will enable efficient evaluation because the time for copying values is saved. In this section, we propose some improved allocation methods which take copy rule into account.



```
proc $R_F(T, x1, x2)$
    $x1 \leftarrow 1$
    $R_L(T_1, x1, x2)$
end

proc $R_B(T, x1, x2)$
    case production$(T)$ of
    $p_3$ : $x2 \leftarrow 0$
        break
    $p_4$ : $x2 \leftarrow 2^{-x1}$
        break
    end
end

proc $R_L(T, x1, x2)$
    case production$(T)$ of
    $p_1$ : var $x3$
        $x3 \leftarrow x1$
        $R_B(T_1, x3, x2)$
        break
    $p_2$ : var $x3, x4$
        $x3 \leftarrow x1$
        $R_B(T_1, x3, x4)$
        $x1 \leftarrow x1 + 1$
        $R_L(T_2, x1, x2)$
        $x2 \leftarrow x4 + x2$
        break
    end
end
```

Fig. 6. Shared Edge Set and Recursive Evaluator

6.1 Two Necessary Conditions

Result of applying the single-chain method to production rule p2 of $G1$ is given in Fig. 7. In this case, pos.L and pos.L$_1$ are shared. Now we are interested in pos.B to be shared with them. In order to share more than one attributes in multi-cast style dependencies like this, the following two conditions must be satisfied:

Condition 1. The value in the storage allocated for pos.B is never changed in any subtree which has B as its root node. We call this unchanged attribute *invariant attribute.*

Because evaluation of pos.L$_1$ needs the value of pos.L, it is necessary to ensure that the value of pos.L is never destroyed when pos.L, pos.L$_1$ and pos.B share the same storage.

Condition 2. After 2-late edge is augmented as seen in Fig. 7, it is also evaluable. In recursive attribute evaluators, storage allocated for inputs can not be reused until the procedure returns: storage for an inherited attribute of a non-terminal become available after all synthesized attributes of the non-terminal are finished evaluating. This is represented by 2-late edge.

As for the evaluability, the following reduction theorem is also satisfied even if storage is shared in multi-cast style.

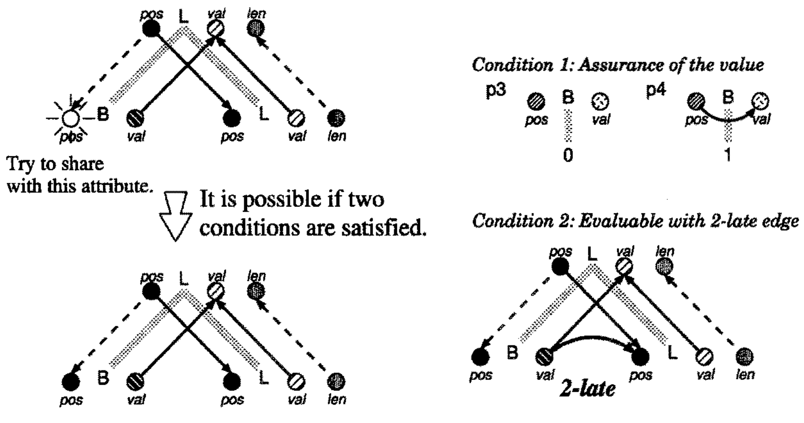


Fig. 7. Two Conditions to Share on Multi-casting

Theorem 14. *If each $DG_p^{*\sim}$ is evaluable under its shared edge set S_p, $DG[T]$ is evaluable under $S[T]$ recursively constructed from S_p by Definition 2.*

As space is limited, the proof cannot be presented here. See [9].

6.2 Multi-Chain Methods

As indicated in Condition 1, whether pos.B becomes invariant attribute or not depends on storage allocation in the subtree. Invariant attribute is closely related to shared edge set of other production rules. Therefore, there are some heuristic methods called multi-chain methods where storages are shared by some attributes in multi-cast dependencies. We suggest here two multi-chain methods and explain their outlines. For the details, see [9].

Multi-Chain 2-Step Method

We begin with finding shared edge set by the single-chain method, and then consider copy rules if possible. The following procedure is applied to all production rules simultaneously.

1. First, compute temporary shared edge set S'_p for each rule p by single-chain method.
2. Compute invariant attributes which never change their values under the temporary shared edge S'_p.
3. Finally, for each production rule p, try all evaluable shared edge set S_ps which are made from S'_p in combination with invariant attributes, and then choose the most optimized one.

Multi-Chain N-Step Method

In contrast, we apply the following procedure to each production rule one by one. To determine the shared edge set which is the most optimized for any derivation tree at evaluator generation time, it is considered advantageous to find shared edge set in the order from highly used production rules in most trees to less used ones.

Table 1. PL/0 Compiler

Productions	104
Nonterminals	43
Edges	403
Copy Edges (Ratio)	195 (48%)

Table 2. Comparison

	Time (msec)	Shared Edges $\sum_{p \in P} W_p$
Single-Chain	3,812	236
Multi-Chain 2-Step	11,925	261
Multi-Chain N-Step	64,069	267
Exhaustive Search	1,313,214	268

Assume $w_p(e) = 1$ for any $p \in P, e \in DE_p$.

1. First apply the single-chain method and get temporary shared edge set.
2. Next update invariant attributes under the above shared edge set.
3. Try all combinations of shared edge set and choose the most optimized and evaluable one.
4. Again update invariant attributes for the next allocation.

At the beginning of allocation, all inherited attributes are supposed invariant. Because it is impossible in practice to calculate the frequency in use of the rules, we suppose that recursive production rules such like p2 : L $\rightarrow$ B L_1 in $G1$ are frequently appeared in most trees and copy rules are frequently used in multi-cast style dependency. In our implementation, the above procedure is applied in the following order:

i. for all recursive production rules, apply the above procedure from root side to leaf side,
ii. for the rest , apply the above procedure from root side to leaf side.

Example 4 (PL/0 Compiler). We implemented these methods in Common Lisp and did an experiment on PL/0 compiler(Table 1). We compared the time needed to find shared edge set and degree of sharing of the set as shown in Table 2. As for the time, single-chain method gives the best result and multi-chain methods look to be practical. As for the degree, multi-chain N-step method is close to exhaustive search which gives the best result.

7 Conclusion

The allocation without fixed attribute evaluation order gives a much better optimization than the allocation with fixed order. This is especially true for the absolutely non-circular attribute grammar because the evaluation order is comparatively free. However, it takes exponential time for finding optimum allocation. We therefore proposed two restrictions which take advantage of the recursive attribute evaluator to the storage allocation strategy. These restrictions decompose the allocation problem for derivation trees into the problems for production rules and into for bipartite graphs in the production rules, so that the problem can be solved within a practical amount of time. Furthermore, we proposed two multichain methods which relax the restrictions and allow to share the attributes related to copy rule. We showed these methods make practical-time allocation possible and the degree of allocation is close to exhaustive search.

References

1. Rodney Farrow and Daniel M. Yellin. A comparison of storage optimizations in automatically-generated attribute evaluators. *Acta Informatica*, 23(4):393–427, 1986. See also: Technical Report, Department of Computer Science, Columbia University, New York, NY (January 1985).
2. Harald Ganzinger. On storage optimization for automatically generated compilers. In K. Weihrauch, editor, *4th GI Conf. on —THEC5—*, volume 67 of *Lecture Notes in Computer Science*, pages 132–141. Springer-Verlag, New York–Heidelberg–Berlin, March 1979. Aachen.
3. Mehdi Jazayeri and Diane Pozefsky. Space-efficient storage management in an attribute grammar evaluator. *ACM Trans. Progr. Languages and Systems*, 3(4):388–404, October 1981.
4. Catherine Julié and Didier Parigot. Space optimization in the FNC-2 attribute grammar system. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 29–45. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.
5. Uwe Kastens. Implementation of visit-oriented attribute evaluators. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 114–139. Springer-Verlag, New York–Heidelberg–Berlin, June 1991. Prague.
6. Takuya Katayama. Translation of attribute grammars into procedures. *ACM Transaction on Programming Languages and Systems*, 6(3):345–369, July 1984.
7. Takuya Katayama and Hisashi Sasaki. Global storage allocation in attribute evaluation. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages*, pages 26–37, St Petersburg Beach, Fl, January 1986.
8. Kazunori Mizushima and Takuya Katayama. A strategy for storage allocation in a recursive attribute evaluator. *Computer Software*, 12(6):50–66, 1995. (in Japanese).
9. Kazunori Mizushima and Takuya Katayama. Algorithms considering copy rule for storage allocation in an attribute grammar. *Computer Software*, 13(5):37–51, 1996. (in Japanese).
10. Takao Moriyama and Takuya Katayama. Attribute globalization by storage passing method. In *Proceedings of 3rd JSSST Conference*, pages 249–252, 1986. (in Japanese).
11. Rieks op den Akker and Erik Sluiman. Storage allocation for attribute evaluators using stack and queues. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 140–150. Springer-Verlag, New York–Heidelberg–Berlin, June 1991.
12. Sethi,R. Pebble games for studying storage sharing. *Theoretical Computer Science*, 19(1):69–84, July 1982.
13. Michael Sonnenschein. Global storage cells for attributes in an attribute grammar. *Acta Informatica*, 22:397–420, 1985.