PAGELEARN: Learning Semantic Functions of Attribute Grammars in Parallel

Gyöngyi Szilágyi¹ and Aggelos M. Thanos²

¹ Hungarian Academy of Sciences, Research Group on Artificial Intelligence, Szeged, Hungary
² National Technical University of Athens, Athens, Greece

Attribute Grammars (AGs) are a generalization of the concept of Context-Free Grammars (CFGs). The formalism of AGs has been widely used for the specification and implementation of programming languages. On the other hand, there is an intimate relationship between AGs and Logic Programming. The paper presents a parallel method for learning semantic functions of Attribute Grammars (AGs) based on AGLEARN [2], using PAGE system [12]. The method is more efficient in both execution time and interaction needed than the sequential one. The method presented is adequate for S-attributed grammars and for L-attributed grammars as well.

Keywords: attribute grammar, parallel evaluation of attribute grammars, machine learning, attribute value learner.

1. Introduction

In the framework of compilation-oriented language implementation, attribute grammars [3] are the most widely applied semantic formalism. The notion of an attribute grammar is an extension of the notion of a context-free grammar. The idea is to decorate parse trees of a context-free grammar by additional labels which provide a “semantics” for the grammar. Every node of a parse tree labeled by a nonterminal is to be additionally decorated by a tuple of semantic values called attribute values. The number of attribute values is fixed for any nonterminal symbol of the grammar. Their names are called attributes of nonterminal. Since the definition of an attribute grammar usually requires much work it would be a useful tool for inferring semantic rules in attribute grammars from examples.

In the case of inductive learning from examples, the learner is given some examples from which general rules or a theory underlying the examples can be derived. An inductive concept learner is given by a set of training examples, some background knowledge, a hypothesis description language, and an oracle willing to answer questions (in the case of interactive learner). The aim is to find a hypothesis such that the hypothesis is complete and consistent with respect to the examples and background knowledge.

To express efficiently the examples, the background knowledge, and the Hypothesis to be induced we need to use a language $L$ with sufficient expressive power. Many systems were developed for learning logic programs, using first order predicate logic language tools (Inductive Logic Programming (ILP)). Attribute Grammars merge the declarative power of predicate logic with the flexibility of a predefined interpretation of its terms. Complex objects and relations can be described in the framework of AGs. Introducing an AG-based description language $L$ in ILP implies the definition of an Attribute Grammar learner.

In the following sections we will see how this integration is carried out using the AGLEARN [2] methodology and we will give the description of an innovative technique for a parallel implementation.

AGLEARN is a method for learning semantic functions of attribute grammars, which infers
semantic rules of attribute grammars from examples. This is an interactive system, so during the execution the oracle has to answer a lot of questions, which needs a lot of work and much time. The parallel implementation of AGLEARN makes possible to decrease the number of oracles, so it is more efficient in both execution time and interaction needed. We notice that the execution time depends on the number of the user queries. Our method uses the PAGE system that is a general purpose parallel parser, augmented with a powerful semantic evaluator. We notice that this parallel method can be implemented using any general purpose parallel parser that has the appropriate facilities to store and handle the necessary information.

This paper is organized as follows. Section 2 gives an introduction to the concepts of Attribute Grammars. Section 3 provides a brief introduction to the AGLEARN method, and gives a typical example. Then Section 4 presents the computational model of PAGE in order to represent the necessary facilities to the parallelization. In Section 5 the PAGELEARN method is presented. We give a detailed description of the parallel method for S-attributed grammars, explaining how we can handle the circuity problem, and giving an illustrative example. Finally we show how the parallel method works for L-attributed grammars.

2. Preliminaries

2.1. Attribute Grammars

Attribute Grammars have been proposed by Knuth [3, 4] as an extension of context-free grammars. The original motivation was to facilitate compiler specification and development procedure.

While compilers were the initial area of research for AGs, they can also be used in a very wide research spectrum, where relations and dependences among structured and interpreted data are very valuable. Areas like software engineering [7, 5], visual programming [10], logic programming [1], distributed programming [15, 8], functional logic programming [6], pattern recognition [9] and signal processing are some notable examples.

**Definition 2.1.** A context-free grammar $G$ is a quadruple such that $G = < N, T, P, D >$, where $N$ is a finite set of nonterminal symbols, $T$ is a set of terminal symbols, $P$ is a finite set of productions, and $D \in N$ is the start symbol of $G$.

An element in $V = N \cup T$ is called grammar symbol. The production in $P$ are pairs of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in V^*$, i.e. the left hand side symbol (LHSS) $X$ is a nonterminal, and the right hand side symbol (RHSS) $\alpha$ is a string of grammar symbols. An empty RHSS (empty string) will be denoted by $\varepsilon$.

**Definition 2.2.** An Attribute Grammar consists of three elements, a context-free grammar $G$, a finite set of attributes $A$ and a finite set of semantic rules $R$. Thus $AG = < G, A, R >$.

A finite set of attributes $A(X)$ is associated with each symbol $X \in V$. The set $A(X)$ is partitioned into two disjoint subsets, the inherited attributes $I(X)$ and the synthesized attributes $S(X)$. Thus $A = I \cup A(X)$.

The production $p \in P$, $p : X_0 \rightarrow X_1 \cdots X_n$ ($n \geq 1$) has an attribute occurrence $X_i.a$, if $a \in A(X_i), 0 \leq i \leq n$. A finite set of semantic rules $R_p$ is associated with the production $p$, with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a$, $1 \leq i \leq n$.

Thus $R_p$ is a collection of semantic rules of the form $X_i.a = f(y_1, \cdots, y_k), k \geq 1$, where

1. either $i = 0$ and $a \in S(X_i)$, or $1 \leq i \leq n$ and $a \in I(X_i)$
2. each $y_j$, $1 \leq j \leq k$, is an attribute occurrence in $p$ and
3. $f$ is a function called semantic function, that maps the values of $y_1, \ldots, y_k$ to the value of $X_i.a$. In a semantic rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ depends on each occurrence $y_j, 1 \leq j \leq k$. 
Thus $R = \cup R_p$. By definition, synthesized attributes are output to the LHSS of the productions while inherited attributes are output to the RHSS. In other words synthesized attributes move the data flow upwards and inherited attributes move the data flow downwards in the derivation tree during the attribute evaluation procedure.

**Remark:** Notice that each semantic rule of an attribute grammar can be seen as a definition of the relation between local attribute values of the neighboring nodes of the parse tree. This relation is defined for a production rule and should hold for every occurrence of this production rule in any parse tree. In the original definition of AG (definition 2.2) the definition of the relation takes the form of the equation. It is possible to generalize the concept of semantic rules and allow them to be arbitrary formulae (not necessarily equalities) over a language $\mathcal{L}$.

**Definition 2.3.** A **Conditional Attribute Grammar** (CAG) is an attribute grammar having the concept of the semantic rules extended. Thus a CAG is a 5-tuple $< G, S, A, \Phi, I >$ where:

- $G$ is the underlying context-free grammar
- $S$ is a set of sorts (i.e. of domains where the attributes take values)
- $A$ is a finite set of attributes. Each attribute $a$ has a sort $s(a)$ in $S$.
- $\Phi$ is a map function of a logic formula $\Phi_p$ written in terms of an $S$-sorted logic language $\mathcal{L}$ to each production rule $p \in P$. The variables of a formula $\Phi_p$ include all output attribute occurrences of $A(p) = \bigcup x \in pA(X)$. The allowed form of the function $\Phi_p$ is either a function $f$ or a relation $c$.

  - in the case of a function, $f$ has the form $f : X_{p,k} \rightarrow f(X_{p,k_1}, a_1, \ldots, X_{p,k_m}, a_m)$ where $f : I(X_{p,k_1}, a_1) \times \cdots \times I(X_{p,k_m}, a_m) \rightarrow I(X_{p,k}, a)$.
  - in case of relation, $c$ has the form $c : c(X_{p,k_1}, a_1, \ldots, X_{p,k_m}, a_m)$ where $c : I(X_{p,k_1}, a_1) \times \cdots \times I(X_{p,k_m}, a_m) \rightarrow \{\text{true}, \text{false}\}$.
- $I$ is an interpretation of $\mathcal{L}$ in some $S$-sorted algebraic structure $\mathcal{A}$.

Semantic rules induce dependences between attributes. These dependences can be presented by a **dependency graph**, from which partial ordering relations are implied. From these partial orderings the evaluation order of the attribute occurrences can be determined. A decorated tree is a derivation tree in which all the attribute occurrences have been evaluated according to their associated semantic rules. The dependency graph characterizes all restrictions on the control of computations. The actual sequence of attribute evaluation must preserve this ordering which is called attribute evaluation strategy. Attribute grammars can be classified according to the attribute evaluation strategy used. A special class, introduced by Knuth [3], is the $S-\text{attributed}$ grammars in which only synthesized attributes are allowed. Due to the restrictive form of $S$-attributed grammars, $L-\text{ordered}$ grammars are used in practice.

**Definition 2.4.** An attribute grammar is said to be $L-\text{attributed}$ if and only if each inherited attribute of $X_{p,j}$ in the production $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p}$ depends only on the attributes in the set $\bigcup_{j \in \{1, \ldots, j-1\}} \text{Inh}(X_{p,k}) \cup \text{Syn}(X_{p,0})$ for $j = 1, \ldots, n_p$.

**Example 2.1.** We now present a typical example for the type checking of arithmetic expressions [16]. Consider the following attribute grammar:

- **Nonterminals:** $N = \{\text{Expression, Term, Factor, AddOp, MulOp}\}$
- **Terminals:** $T = \{\text{Real, Integer, +, -, \times, /, (, )}\}$
- **Start Symbol:** $D = \text{Expression}$
- **Sorts:**
  
  $S = \{S_{\text{mode, operator}}\}$ where
  
  - $S_{\text{mode}} = \{\text{int, real}\}$
  - $S_{\text{operator}} = \{\text{add, sub, mul, div}\}$
- $\Phi = \{\text{add, sub, mul, div, int, real, id, f1, f2}\}$ where
- *add, sub, mul, div, int* and *real* are constants
- *id : S_{mode} → S_{mode}* denotes the identity function
- *f_1 : S_{mode} × S_{mode} → S_{mode}*

\[
f_1(op_1, op_2) := \begin{cases} 
  \text{real} & \text{if } op_1 = \text{real or } op_2 = \text{real} \\
  \text{int} & \text{else}
\end{cases}
\]

- *f_2 : S_{operator} × S_{mode} × S_{mode} → S_{mode}*

\[
f_2(op_1, op_2, op_3) := \begin{cases} 
  \text{int} & \text{if } op_1 = \text{mul and } op_2 = \text{int and } op_3 = \text{int} \\
  \text{real} & \text{else}
\end{cases}
\]

**Attributes:** \( A = \text{Syn} = \{ \text{mode}, \text{operator} \} \)

so that

\[
\text{Syn(Expression)} = \text{Syn(Term)}
\]

\[
= \text{Syn(Factor)} = \text{mode}
\]

\[
\text{Syn(AddOp)} = \text{Syn(MulOp)} = \text{operator}
\]

**Productions and Semantic Rules:**

1. \( \text{Expression}_0 → \text{Expression}_1 \text{AddOp Term} \)
   \( R(1) : \text{Expression}_0.\text{mode} := f_1(\text{Expression}_1.\text{mode}, \text{Term}.\text{mode}) \)

2. \( \text{Expression} → \text{Term} \)
   \( R(2) : \text{Expression}.\text{mode} := \text{Term}.\text{mode} \)

3. \( \text{Term}_0 → \text{Term}_1 \text{MulOp Factor} \)
   \( R(3) : \text{Term}_0.\text{mode} := f_2(\text{MulOp}.\text{operator}, \text{Term}_1.\text{mode}, \text{Factor}.\text{mode}) \)

4. \( \text{Term} → \text{Factor} \)
   \( R(4) : \text{Term}.\text{mode} := \text{Factor}.\text{mode} \)

5. \( \text{Factor} → \text{Real} \)
   \( R(5) : \text{Factor}.\text{mode} := \text{real} \)

6. \( \text{Factor} → \text{Integer} \)
   \( R(6) : \text{Factor}.\text{mode} := \text{int} \)

7. \( \text{Factor} → (\text{Expression}) \)
   \( R(7) : \text{Factor}.\text{mode} := \text{Expression}.\text{mode} \)

8. \( \text{AddOp} → + \)
   \( R(8) : \text{AddOp}.\text{operator} := \text{add} \)

9. \( \text{AddOp} → − \)
   \( R(9) : \text{AddOp}.\text{operator} := \text{sub} \)

10. \( \text{MulOp} → \ast \)
    \( R(10) : \text{MulOp}.\text{operator} := \text{mul} \)

11. \( \text{MulOp} → / \)
    \( R(11) : \text{MulOp}.\text{operator} := \text{div} \)

\[\square\]

3. **The AGLEARN Method**

AGLEARN [2] is a method for learning semantic functions of attribute grammars. The method uses background knowledge for learning semantic functions of S-attributed and L-attributed grammars. The given context-free grammar and background knowledge allow one to restrict the space of relations and give a smaller representation of data. The basic idea of this method is that the learning problem of semantic functions is transformed to a propositional form and the hypothesis induced by a propositional learner is transformed back into semantic functions. This approach is motivated by the fact that there is a close relationship between attribute grammars and logic programs [1].

AGLEARN uses the same concept as Inductive Logic Programming (ILP) but has a different representation. The background knowledge and the concepts are represented in the form of attribute grammars. An example contains a string which can be derived from the target nonterminal. We suppose that the underlying context-free grammar is given. The task of AGLEARN is to infer the semantic functions associated with the production rule. In the learning process the grammar, the background semantic functions and the examples can be used.

**The input:**

- The AG in which:

  - The set of productions \( P \) is partitioned into two disjoint sets \( P_B \) (The background rules) and \( P_T \) (the target rules). The set of semantic functions \( R \) is fully defined for the rules belonging to \( P_B \) and there are no semantic functions in \( R \) associated with the rules belonging to \( P_T \).
- A partially defined set of semantic conditions $C$. $C$ is fully defined for $P_B$ and there is no condition for any rule in $P_T$.

- For each $p \in P_T$ and for each synthesized attribute occurrence $X_{p,0} a$ a set of examples $E_p(a)$, which is partitioned into two disjoint sets, the set of positive $E_p^+(a)$ and negative examples $E_p^-(a)$, respectively. Let us suppose that $p$ has the form of $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p}$. An example $e \in E_p(a)$ is given in the form: $e=(w, (a_1, v_1), \ldots, (a_m, v_m), (a_{m+1}, v_{m+1}))$, where $w \in T^*$, $X_{p,0} \Rightarrow^* w$, $(a_1, \ldots, a_m) = \text{Inh}(X_{p,0})$, and $a_{m+1} = a$. A pair $(a_p, v_j)(1 \leq j \leq m + 1)$ in the example $e$ denotes that the attribute $X_{p,0} a$ has the value $v_j$ in the evaluated attributed tree built for the word $w$.

The aim is to find a set of functions $F_t$ and for each production $p \in P_T$ a definition of the sets $R(p)$ and $C(p)$ such that the semantic functions and conditions in $R(p)$ and $C(p)$ are defined by the elements of $F_t \cup F_i$. The resultant attribute grammar must be complete and consistent with the examples.

### 3.1. Learning Semantic Functions of S-attributed Grammar

The method is based on the idea that the semantic functions of the background rules can introduce new columns in the table corresponding to the transformed learning problem.

Consider the production $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p} \in P_T$, and suppose that the symbols in this rule can possess only synthesized attributes. We would like to learn the semantic function associated with $X_{p,0} a$. For $X_{p,0} a$, a table $T(a)$ must be constructed. Each row of this table corresponds to an example from $E_p(a)$. The table has a set of columns. The key part is the computation of the column $U$ (see Table 1) which contains columns corresponding to the attribute instances $X_{p,j} b \in \text{Syn}(X_{p,j}), j = 1, \ldots, n_p$. Therefore we build the tree for the actual example from $E_p(a)$, and if the subtree derived from $X_{p,j}$ contains a node corresponding to a rule instance belonging to $P_T$ then the values of the attributes in $\text{Syn}(X_{p,j})$ are asked from the user for the given derivation. The if-rules are constructed from the table $T(a)$, and the semantic functions are generated from a set of accepted if-rules. (For a detailed description see [2]).

### 3.2. Learning L-attributed Semantic Functions

Suppose that the symbols in the rule $p$ can possess synthesized and inherited attributes. The task of an attribute learner is to define semantic functions for the set of defined occurrences $\bigcup_{k \in \{1, \ldots, n_p\}} \text{Inh}(X_{p,k}) \cup \text{Syn}(X_{p,0})$. The learning process of these attributes can be summarized as follows:

1. **Learning semantic functions for** $a \in \text{Inh}(X_{p,j})$, $j = 1, \ldots, n_p$.

The user is asked to provide examples. For the attribute $X_{p,j} a$ a table $T(a)$ must be constructed. The columns in the part $U$ of the table are all the attribute occurrences

<table>
<thead>
<tr>
<th>class</th>
<th>word</th>
<th>target</th>
<th>$T_{0,m}$</th>
<th>$U_1$</th>
<th>$U_2$</th>
<th>$U_3$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$F_{UCR}$</th>
<th>$F_{UCF}$</th>
<th>$F_{UF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3 * 2.5</td>
<td>real</td>
<td>int* mul</td>
<td>real</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>+</td>
<td>5 * 3</td>
<td>int</td>
<td>int* mul</td>
<td>int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>+</td>
<td>1.5 * 4</td>
<td>real</td>
<td>real* mul</td>
<td>int</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>+</td>
<td>2.5/3</td>
<td>real</td>
<td>real* div</td>
<td>int</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>+</td>
<td>2/3</td>
<td>real</td>
<td>int* div</td>
<td>int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>+</td>
<td>6/3.4</td>
<td>real</td>
<td>int* div</td>
<td>real</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>-</td>
<td>2 * 3.2</td>
<td>int</td>
<td>int* mul</td>
<td>real</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>-</td>
<td>4.3/2</td>
<td>int</td>
<td>real* div</td>
<td>int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>-</td>
<td>8/3</td>
<td>int</td>
<td>int* div</td>
<td>int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 1. The generated propositional table
on which the value of \( X_{p,j,a} \) depends on
\( \text{Inh}(X_{p,0}) \cup \bigcup_{k \in \{1, \ldots, j-1\}} \text{Syn}(X_{p,k}) \). Afterwards the propositional part can be
constructed in the same way as that presented for S-attributed grammars. When semantic functions for all
\( a \in \text{Inh}(X_{p,j}) \) have been learned, the attributes of \( \text{Syn}(X_{p,j}) \) can be computed by using the background rules.

2. **Learning semantic functions for**
\( a \in \text{Syn}(X_{p,0}) \)

The semantic functions for \( a \) can be determined in the same way as presented for S-attributed grammar.

**Example 3.1.** Example of AGLEARN method (taken from [2]).

Apply the above procedure to our example:

\[
\begin{align*}
P_T &= \{ R(1), R(2), R(3), R(4) \} \\
P_B &= \{ R(5), \ldots, R(11) \}.
\end{align*}
\]

We demonstrate the learning of the semantic function of \( R(3) \).

- \( E_3^+(\text{mode}) = \{ (3\times2.5, \text{real}), (5\times3, \text{int}), (1, 5\times4, \text{real}), (2.5/3, \text{real}), (2/3, \text{real}), (6/3.4, \text{real}) \} \)

- \( E_3^-(\text{mode}) = \{ (2\times3.2, \text{int}), (4.3/2, \text{int}), (8/3, \text{int}) \} \)

**Step 1.**

Table 1 shows the generated propositional table where:

\* : we have to ask this value from the user

- \( U_1 : \text{Term}_1\text{.mode} \)
- \( U_2 : \text{MulOp}\text{.Operator} \)
- \( U_3 : \text{Factor}\text{.mode} \)
- \( R_1 : \text{Term}_1\text{.mode} = \text{int} \)
- \( R_2 : \text{Term}_1\text{.mode} = \text{real} \)
- \( R_3 : \text{MulOp}\text{.Operator} = \text{mul} \)
- \( R_4 : \text{MulOp}\text{.Operator} = \text{div} \)
- \( R_5 : \text{Factor}\text{.mode} = \text{int} \)
- \( R_6 : \text{Factor}\text{.mode} = \text{real} \)
- \( F_1 : \text{Term}_0\text{.mode} = \text{int} \)
- \( F_2 : \text{Term}_0\text{.mode} = \text{real} \)
- \( C_1 : \text{Term}_0\text{.mode} = \text{Term}_1\text{.mode} \)
- \( C_2 : \text{Term}_0\text{.mode} = \text{Factor}\text{.mode} \)
- \( T : \text{true} , F : \text{false} \)

**Step 2.**

The if-rule:

\[
\begin{align*}
\text{if } R_1 = \text{true} & \text{ & } R_3 = \text{true} & \text{ & } R_5 = \text{true} & \text{ & } F_1 = \text{true} \text{ else } F_2 = \text{true} \\
\end{align*}
\]

**Step 3.**

The transformed semantic function \( R(3) \) takes the form of

\[
\begin{align*}
\text{if } \text{Term}_1\text{.mode} = \text{int} & \text{ & } \text{MulOp}\text{.operator} = \text{mul} & \text{ & } \text{Factor}\text{.mode} = \text{int} \\
\text{then } \text{Term}_0\text{.mode} := \text{int} \\
\text{else } \text{Term}_0\text{.mode} := \text{real} \\
\end{align*}
\]

\[ \square \]

4. The Computational Model of PAGE

One of the objectives of our research was to implement a tool which is portable and independent of the underlying system architecture. PAGE [11] is built on top of ORCHID kernel [13, 14] which encapsulates the machine dependencies providing a layer with parallel programming primitives. In PAGE a supervisor process maintains a pool of messages, and is responsible for supplying the processors with computational load (i.e. processes to execute). Each slave process handles a grammar production along with its semantic rules. It collects messages corresponding to the RHSS (body) of the production in which it is the head, and produces new messages which correspond to the head of the production in which it is a RHSS. All the static information of the grammar (grammar productions, semantic rules, atomic productions) is broadcast and kept in the network processors for faster access. All the information generated from the slave processes (i.e. partial solutions) is stored locally in the network processors, in a caching hierarchy, inducing an incremental attribute evaluation and achieving a controlled grained parallelism.

**Example 4.1.** Let us assume we have to evaluate the following grammar:

\[
\begin{align*}
1. & S(x, y, z) : - A(x, y), E(y, z). \\
2. & A(x, y) : - B(x), C(y). \\
3. & E(y, z) : - B(y), D(z). \\
\end{align*}
\]

where \( S \) is the start symbol

The Supervisor assigns to a new slave (for example slave \( s1 \)) the evaluation of rule (1). Slave
PAGELEARN: Learning Semantic Functions of Attribute Grammars in Parallel

Page 121

Fig. 1. Slave Processes generation.

$s1$ asks the Supervisor for solutions for the RHSS $A$ and $E$ along with their corresponding inherited attributes. Supervisor checks if there exist already solutions for $A(x,y)$ and $E(y,z)$. If not, it assigns the evaluation of $A(x,y)$ and $E(y,z)$ to $s1$ two new slaves (say $s2$ and $s3$), respectively. This scheme achieves AND parallelism. If there already exist solutions for some or for all of the RHSSs, then there is no need for new slaves corresponding to these RHSSs. As a result PAGE achieves controlled grained parallelism, because no unnecessary slaves are generated. Similarly, slave $s2$ asks the supervisor for solutions for the RHSS $B$ and $C$ along with their inherited attributes (i.e., $x$ and $y$ respectively). The Supervisor checks if there already exist solutions for $B(x)$ and $C(y)$. If not, it assigns the evaluation of $B(x)$ and $C(y)$ to two new slaves (say $s4$ and $s5$), respectively. The procedure goes on in the same manner unfolding a proof tree over the network. If we add one more rule to our example

\[ S(x,y,z) \rightarrow A(y,x), E(z,y). \]

then another similar proof sub-tree will be generated, establishing OR parallelism.

Fig. 1 illustrates the above example. Each slave may be located in different processors or in the same processor with another slave.

Fig. 2 shows the Network Supervisor processor structure in which the AG is stored. The AG is decomposed and broadcast to the network nodes along with the input string, which is sent to the network nodes. Moreover the network Supervisor process maintains a Process Allocation Table in order to prevent an explosion in the number of processes. The results of all the rules evaluated in the network are stored in a special-purpose structure.

Fig. 3 depicts the Node Processor Structure. There is a Node Supervisor Process controlling the underlying slave processes, which handles the grammar rules. In addition, the Node Processor keeps a list of the AG terminals, in order to prevent the communication overhead we would have if this list was maintained by the Network Supervisor Processor. The supervisor process handles local requests for solutions of rules evaluated in remote processes in a special-purpose queue. Besides this, every slave process keeps a similar queue of remote requests. The solutions are kept in a caching hierarchy in which every generated or remote accessed solution is stored in a special structure.

\[ Process\ Allocation\ Table \quad Results \quad Grammar \quad Input\ String \]

Fig. 2. Network Supervisor Structure.
In general the data-flow style of execution of PAGE assimilate to the computational model of Conery’s AND/OR process model, however it differs in the following important aspects: We use the above-mentioned structure sharing approach when referring to these structures, so there is no need for stack copying, which is a major source of overhead in the latter model. Solution caching for incremental attribute evaluation greatly improves our model. Moreover, PAGE tries to keep processes belonging to the same OR-branch locally in the same processor. Finally, we use an interleaved unification scheme where more than one solution of each AND parallel branch are unified at the same time.

5. PAGELEARN: A Parallel Approach to AGLEARN, using PAGE Technology

In this section a parallel method is presented for the implementation of AGLEARN using the PAGE general purpose multi-paradigm attribute grammar evaluator. The method is adequate for S-attributed and for L-attributed AGs as well. Parallel learning leads to a more efficient execution time and reduces the oracles that may be needed. In the following section a description of the method of using S-attribute grammars is supplied. We show how the circuity problem can be handled and give a detailed example. A description of the method using L-attribute grammars is also presented.

5.1. Parallel Learning of S-attribute Grammars

Description of the Method

Let \( \text{AG} = < G, S, A, \Phi, I > \) the given Attribute Grammar, \( P_B \) the set of background rules, \( P_T \) the set of target rules, and \( E \) the set of the training examples. The target nonterminals possess only synthesized attributes. We would like to learn the semantic functions of each \( P \in P_T \) in parallel using PAGE.

In this case the Supervisor processor and the processors have the AG. The rules of \( P_T \) are decomposed, and each processor handles one part of the target rules. Every process (slave) learns one semantic function of a rule \( R \). So every rule and every attribute of the rules in \( P_T \) is learned in parallel. The process has to build the table \( T \) (given in example 3.1). This means that for every example the corresponding process must compute the columns \( \text{class}, \text{word}, \text{target}, U, F_{UR}, F_{UCR}, F_{UF}, F_{UCF} \). It is clear from this procedure that the main issue is the computation of \( U \), since other columns can be derived from the column \( U \) or target or from the examples.

Let us suppose that \( R \) has the form of \( R : -X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,k} \).

To compute \( U \), in the previous section, we have to know the value of \( X_{p,j}, b \in \text{Syn}(X_{p,j}), j = 1, \ldots, k \) (AGLEARN method). For this we build the tree for the actual example, and if the subtree derived from \( X_{p,j} \) contains a node corresponding to a rule instance belonging to \( P_T \), then the values of the attributes of \( \text{Syn}(X_{p,j}) \) are asked from the user for the given derivation. Now we have the possibility of asking the other processes if the rule belonging to \( P_T \) is already learned or not (because of parallelism). If this is the case, we can use these semantic functions and we do not need an oracle. If it hasn’t been learned yet, we have again a possibility of waiting until it is learned (or we can ask the user).

At this point we have to handle a difficult problem. If we decide to wait for these semantic rules to be learned, we may face a circuity situation where two processes wait (perhaps transitively) for one another to produce a semantic rule.

In Fig. 4 we give the general procedure of PAGELEARN.
for each grammatical rule $p \in P_T$ and for each synthesized occurrence $X_{p,b,a}$ do spawn a process in parallel

Every process does the following
for each example $e \in E_p(a)$ do
build the attributed tree for the actual example on the input string $w$ using grammar $G$ and semantic functions $R$
if the subtree contains only nodes corresponding to rule instances $R_i$ belonging to $P_B$ then
the attributes of this subtree can be evaluated
so the columns of the table $T$ can be computed
else
for every $R_i \in P_T$ do
the process that evaluates $R_i$ asks the Supervisor if the unknown rule $R$ has already been learned
if learned then
the process can evaluate $R_i$
using the learned semantic rules
else if there isn’t any circuit in the waiting processes then
wait...
else ask the user for an oracle and kill the waiting rules belonging to this tree. (These rules are different from the other that are being learned. Moreover, the rules belonging to the tree are used only for their operational semantics, while the rules at the other processes may be used for inferring the semantics)
end { for } for example 
end { for } for example 

As soon as every column and row of table $T$ has been computed, $T$ is sent to the Supervisor, which sends it to an attribute value learner (to C4.5 in our case). When the Supervisor gets from the attribute value learner the learned semantic rules broadcasts it to the processes.

end { for }

Fig. 4. General Algorithm of the PAGELEARN Method.

Handling the Circuity Problem

The problem can be summarized as follows: Every process learns the semantic rule for an attribute $(R_i,a_j)$ in parallel. When the process creates the column $U$ for the given training example, it has to build the derivation tree for this example, and evaluate this tree. If this tree has a rule that belongs to $P_T$, then the process has to wait for the semantic rules of this rule. We want to avoid the circuity problem, that is, when two or more processes are waiting one for another.

Example 5.1. Assume that we have the processes $P_1$, $P_2$, $P_3$, $P_4$ learning the semantic rules $R_1.a$, $R_2.a$, $R_3.a$, $R_4.a$ respectively. Fig. 5 depicts the circuity problem when the processes are waiting for the learning of the following semantic rules:

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. The circuity problem: Neighbouring Matrix $M$. 
P1 is learning \( R1.a \) and in the subtree for the given example \( R3 \) and \( R4 \) are used. (It means that we need the semantic rules \( R3.a \) and \( R4.a \) to evaluate the value of the attributes of the derivation tree built for the actual example.)

P2 is learning \( R2.a \) and in the subtree for the given example \( R2 \) is used.

P3 is learning \( R3.a \) and in the subtree for the given example \( R1 \) is used.

P4 is learning \( R4.a \) and in the subtree for the given example \( R2 \) and \( R3 \) are used.

\[ \text{In the above example we saw that each learning process of a semantic rule may depend on the learning process of another semantic rule of another grammatical rule. These dependences form a dependency graph } G. \text{ Let } G = (V, E) \text{ a directed graph, where } V = (R_1, \ldots, R_k), \ R_j \in P_T, j = 1, \ldots, k, \ E = \{ (R_i, R_j) : \text{There is an edge from } R_i \text{ to } R_j \ \text{if and only if } R_j \text{ is waiting for } R_i \} \]

\[ \text{Fig. 6. The circuity problem: the dependency graph.} \]

\[ \text{In Fig. 6 the dependency graph of the example 5.1 is shown. In this graph we have three circuits:} \]

\[ R1 \rightarrow R3 \rightarrow R4 \rightarrow R1 \]

\[ R2 \rightarrow R2 \]

\[ \text{Fig. 5. The circuity problem: the waiting processes.} \]

\[ \text{In the general algorithm of PAGELEARN (see Fig. 4) we have to check for the circuity problem. For this purpose a neighbouring matrix } M \text{ is used, situated in the Supervisor, and is similar to that of the Table 2. Using this matrix it is easy to detect when a circuity problem is being faced or not. If a circuity problem is being faced then it is enough to cut the circuit only at one point (one rule) and ask the user. The check algorithm is listed in Fig. 7.} \]

**A Detailed Example**

This section presents a detailed example describing the proposed parallel method using PAGE. Some processes belonging to a processor, which have been assigned the task of learning the semantic rules of a target rule. Other processes belonging to the same processor work on the learning of the semantic rules of another target rule. Every process has to build a table \( Tij \) (from which the if-rules are generated). The Supervisor as well as the slave processors know the whole AG specification. To be more precise, each one of the PAGE processes has the following tasks assigned.

**The Supervisor:**

- Handle the table \( M \) that is used for dealing with the circuity problem.
- Decompose the target rules \( P \in P_T \).
- Store the learned semantic rules
- Serve the requested messages for the semantic rules or the circuits

**The slave processors:**

- Spawn and delete processes for the rules and for its attributes
- Forward the requests and the learned semantic functions to the Supervisor
Initialize matrix $M$: fill all elements with 0.

if $P_j$ (which is learn a semantic function of $R_j$) have
to wait for $R_k$ then
  Ask the Supervisor if there is a circuit or not
  (check if the addition of the new edge
  $(R_k, R_j)$ in the neighbouring matrix may cause a circuity problem.
if there is a circuit then
  the Supervisor sends a message informing that a circuit exists
else
  Set $M[k, j] = 1$ and the Supervisor sends a message that no circuit
  exists, so the process can wait for $R_k$ to learn the semantic
  functions
if all semantic rules for $R_k$ have been learned then
  they are sent to the waiting processes
  Set $Row(M_k) = 0$

Fig. 7. Checking Algorithm for the Circuity Problem

- Handle the requests of the processes for the AGs rules

The processes:

- compute the table $T_{ij}$.
- Build the derivation tree for the actual example { this is automatic in the PAGE system }
- Ask the Supervisor or other processes or
  the user for the unknown semantic rules if it is necessary
- Transform the table $T_{ij}$ to the Supervisor

The Supervisor decomposes the target rules and broadcasts them along with the corresponding training examples over the network. The processors spawn processes (slaves) for every semantic rule corresponding to each one of the synthesized attributes of the target rules. The processes start to learn the target semantic rules from the given examples. The core tasks of PAGELEARN are: creation of the table $T_{ij}$, building of the derivation tree for each of the given corresponding examples, and, if it is necessary, asking the Supervisor for the target rules for evaluation of the parse tree, or waiting for other processes to produce target semantic rules. When the table $T_{ij}$ is computed then the process generates the if-rules, transforms them into semantic rules, and sends them to the Supervisor.

Example 5.2. We demonstrate how the method works in parallel by continuing the previous example. Recall that $P_f = \{R(1), R(2), R(3), R(4)\}$.

- Processor 2 handles $R(1)$ and $R(2)$,
- Processor 3 handles $R(3)$ and $R(4)$.
- Processor 2 spawns 2 processes $P_{21}$ for
  $R(1)$ and $P_{22}$ for $R(2)$.
- Processor 3 spawns 2 processes $P_{31}$ for
  $R(3)$ and $P_{32}$ for $R(4)$.

Fig. 8 shows the mapping of the processes in the processor network.

The rules $R(1) - R(4)$ are learned in parallel. The positive and negative examples are:

- $E_1^+(mode) = \{(2.5 + 3, real), (5 + 3, int), (1.5 - 4, real), \ldots\}$
- $E_1^-(mode) = \{2 + 3.2, int), (4.3 - 2, int), \ldots\}$
- $E_2^+(mode) = \{(2, int), (3.5, real), \ldots\}$
- $E_2^-(mode) = \{(1.3, int), (2, real), \ldots\}$
- $E_3^+(mode) = \{(3 * 2.5, real), (5 * 3, int), (1, 5 * 4, real), (2.5/3, real), (2/3, real), (6/3.4, real)\}$
- $E_3^-(mode) = \{(2 * 3.2, int), (4.3/2, int), (8/3, int)\}$
Fig. 8. Processes mapping for the Example 5.2.

- $E_4^+(mode) = \{(2, \text{int}), (3.5, \text{real}), \ldots\}$
- $E_4^-(mode) = \{(1.3, \text{int}), (2, \text{real}), \ldots\}$

Table 3 shows the table $T(mode)$ which is computed in parallel.

| Class | Word | Target | $U_1$ | ... | ...
|-------|------|--------|------|-----|-----
| +     | 2.5+3 | Real   | int  | ... | ...
| +     | ...   | ...    | ...  |     | ...
| -     | ...   | ...    | ...  |     | ...

`Table 3. The table $T(a)$ for the example 5.2`

To compute the element of the column $U$ an attributed grammar tree is built on the input string $w$ of each example. Fig. 9 shows the attribute tree for the rule $R(1)$. Recall that the rules $R(2)$ and $R(4)$ are evaluated in parallel.

- $R(2)$ and $R(4) \in P_T$ are used in this derivation tree. If we build the derivation tree for the given examples we find that
  - To learn $R(2)$ we need to use $R(4)$
  - To learn $R(3)$ we need to use $R(4)$
  - To learn $R(4)$ we don’t need to use rules from $P_T$

The corresponding dependency graph for the detection of circuity problem is given in Fig. 10. We do not have a circuit in this graph so $P_{22}$ and $P_{31}$ can wait until $R(4)$ is learned, and $P_{21}$ can wait until $R(2)$ and $R(4)$ are learned. So we needn’t ask the user, but when we had to learn sequentially we had to ask the user for every example in $R(1)$, $R(2)$ and $R(3)$.

**Example 5.3.** We give an example to show how there can be a circuit in the dependency graph. Let $P_T = \{R(3), R(7)\}$. $E_3^+ = \{(3.2 \times (4.1+3), \text{real})\}$ and $E_7^+ = \{((4.1 \times 1.2), \text{real})\}$. When the process builds the attributed tree for $e_3$, then we have to wait for $R(7)$ to be learned, and when the other process builds the attributed tree for $e_7$ then we have to wait for $R(3)$ to be learned. So we have the circuity problem.

5.2. How the Method Works for L-attributed Grammars

We suppose that a target nonterminal can possess inherited attributes. Consider the production: $p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,n_p} \in P_T$. The symbols in this rule may possess synthesized and inherited attributes. We have to learn the
Fig. 9. Derivation tree for the Example 5.2.

Fig. 10. Dependency graph for detecting ciruicty problem for Example 5.2

semantic functions for the set of defined occurrences \( \bigcup_{0 < k \leq n_p} \text{Inh}(X_{p,k}) \cup \text{Syn}(X_{p,0}) \).

In our method we learn the semantic functions for every \( X_{p,j} \), \( j = 1, \ldots, n_p \), and then learn the semantic functions for \( X_{p,0} \) sequentially. However, we can learn in parallel the semantic functions for the attributes of a given \( X_{p,j} \), \( j = 1, \ldots, n_p \), and for \( X_{p,0} \) as well since both are independent.

The whole task of process mapping and decomposition of the rules is similar to the parallel learning of S-attributed grammar. The only difference is that we have to spawn processes for every inherited attribute \( \in \text{Inh}(X_{p,j}), j = 1, \ldots, n_p \), when their semantic functions are learned, and we have to spawn processes for every synthesized attribute \( \in \text{Syn}(X_{p,0}) \) when we learn the semantic functions of \( X_{p,0} \).

The procedure can be summarized as given in Figure 11.

So we learn in parallel the rules in \( P_T \) (the set of the target semantical rules).

For a given rule \( p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,n_p} \in P_T \) we learn
1. first semantic functions for the inherited attributes of \( X_{p,1} \) in parallel
2. then semantic functions for the inherited attributes of \( X_{p,2} \) in parallel
   
   * \( n_p \). finally semantic functions for the inherited attributes of \( X_{p,n_p} \) in parallel
   * \( n_p + 1 \). and then the synthesized attributes of \( X_{p,0} \) in parallel

For steps \( 1 - n_p \) to create table \( T_i \), we have to know the value of \( \bigcup_{k \in \{1, \ldots, i-1\}} \text{Syn}(X_{p,k}) \cup \text{Inh}(X_{p,0}) \) of which semantic functions are learned in the previous steps.

We learn in parallel the rules, so for a non-terminal both the inherited and the synthesized attributes are learned parallelly, because

- If it is in the right hand side, its inherited attributes are learned in the actual rule, and its synthesized attributes are learned in an-
for every rule in parallel do
   \{ p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,n_p} \in P_T \} \\
   \begin{enumerate}
   \item for j = 1, \ldots, n_p do \{ sequential \} \\
       \hspace{1cm} for every inherited attribute \in Inh(X_{p,j}) in parallel do spawn and run Process2
   \item for every synthesized attribute \in Syn(X_{p,0}) in parallel do spawn and run Process1
   \end{enumerate}
end \{ for \}

Process1

for each grammatical rule p \in P_T and for each synthesized occurrence X_{p,0} do
   spawn processes in parallel which do the following
   \begin{enumerate}
   \item for each example e \in E_p(a)
       build the attributed tree for the actual example on the input string w
       using grammar G and semantic functions R
       if the subtree contains only nodes corresponding to rule instances R_i belonging to P_B then
       the attributes of this subtree can be evaluated, and the columns of the table T can be computed
       else
       for every R_i \in P_T do
       the process that evaluates R_i asks the Supervisor if the unknown rule R has already been learned
       if learned then the process can evaluate R_i using the learned semantic rules
       else if there isn't any circuit in the waiting processes then wait...
       else ask the user for an oracle and kill the waiting rules belonging to this tree.
       end \{ for \}
   end \{ for example \}
end \{ for \}

Process2

for every example do
   \begin{enumerate}
   \item ask the user to give the example for the target attribute X_{p,j,a}
   \item build the attributed tree for the actual example on the input string w
   \item evaluate the part of the tree in order to get the value of \( \bigcup_{k \in \{1, \ldots, j-1\}} \text{Syn}(X_{p,k}) \cup \text{Inh}(X_{p,0}) \)
   \end{enumerate}
end \{ for \}
5. generate the if rule
6. generate the semantic rule

Fig. 11. General algorithm for L-attributed grammar

other rule of which it is the left hand side symbol ( or this rule belongs to P_B ).

- If it is in the left hand side, its synthesized attributes are learned in the actual rule, and its inherited attributes are learned in an other rule in that it is on the right hand side ( or this rule belongs to P_B ).

5.3. Analysis of the Method

Our method learns semantic functions of Attribute Grammars. During the execution an or-
The method presented here is based on the AGLERNAE method described in [2]. Parallelism improves the efficiency of the previous method both in execution time and in interaction needed. PAGE is capable to handle the learning of many semantic rules concurrently. OR parallelism PAGE explores gives the ability of reducing the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation occurs. While the main problem of the sequential system was the large number of the user queries, this method improves the efficiency of the previous method from this aspect too. The elimination’s percent of the user queries depends (on the training examples) on the number of the circuits appearing among the waiting rules during the learning method.

6. Discussion

The method presented here is based on the AGLERNAE method described in [2]. Parallelism improves the efficiency of the previous method both in execution time and in interaction needed. PAGE is capable to handle the learning of many semantic rules concurrently. OR parallelism PAGE explores gives the ability of reducing the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation occurs. Moreover, behind parallelism a “dual nature” of the proposed method is hidden. The concept of the method described is to give the user the capability of handling smaller specifications of the grammar describing the problem the user deals with. The natural order to do this is, firstly to learn the semantic rules from the given examples and secondly to execute the program. Now we can have both steps interleaving each other in the following fashion:

1. Every semantic rule of the target rules is learnt in parallel.
2. The PAGE system is a parallel parser, so the attributed tree built for the actual example is handled in parallel. Moreover, this facility makes it possible to learn many semantic rules concurrently.
3. The concurrent learning of many semantic rules gives the possibility to reduce the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation occurs. While the main problem of the sequential system was the large number of the user queries, this method improves the efficiency of the previous method from this aspect too. The elimination’s percent of the user queries depends (on the training examples) on the number of the circuits appearing among the waiting rules during the learning method.

Evaluate the grammar → if a semantic rule is needed, try to learn it → continue the execution.

References


Received: November, 1997
Revised: December, 1999
Accepted: January, 2000

Contact address:
Gyöngyi Szilágyi
Hungarian Academy of Sciences
Research Group on Artificial Intelligence
Szeged
Hungary
Aggelos M. Thanos
National Technical University of Athens
Athens
Greece

Gyöngyi Szilágyi is a research assistant at the Research Group of Artificial Intelligence of the Hungarian Academy of Sciences. From 1995 till 1998 she was a Ph.D. student in computer science at Jozsef Attila University. Her research interests include Logic Programming (LP), Inductive Logic Programming (ILP), Constraint Logic Programming (CLP), Learning of CLP, Attribute Grammars (AG), Learning of AG’s and Slicing of Logic Programs.

Aggelos M. Thanos received his M.S. degree (1993, in Computer Science) from the University of Crete at Heraklion. In 1993 he entered to the National Technical University of Athens Greece PhD. Program. His research interest focuses on Distributed Memory Systems, Declarative Languages, Parallel Programming, Parallel Logic, Constraint Logic Programming, Distributed O.S. and Telemedicine.