Giving mathematical semantics of nondeterministic and parallel programming structures by means of attribute grammars

By R. Alvarez Gil

1. Introduction

A formal definition of the semantics of the programming languages is a prerequisite for the verification of specific implementations. The definition of the semantics of a programming language can be formulated in different ways. Knuth [8] has introduced attribute grammars for this purpose, Scott and Strachey [10] has developed a mathematical method.

Many papers has been published about the relation between attribute grammars and mathematical semantics. Mayoh [9] has shown that for any attribute grammar it is possible to find an equivalent mathematical semantics. The reverse affirmation is true only with several restrictions [3].

In this paper after the introduction of the used notations and the concept of attribute grammar we give an example to show how it is possible to describe the mathematical semantics of programming languages with the help of attribute grammars in section 3.

In section 4 we describe the nondeterministic structures introduced by Dijkstra [2] and give their mathematical semantics by means of attribute grammars. For this purpose it is sufficient to employ the notion of the possible states used in the description of the semantics of sequential programs, but many-valued functions are necessary to give the semantics of statements and programs. In section 5 we have to extend the notion of the possible states, too, to give the mathematical semantics of a parallel programming language allowing communication of sequential processes through Hoare's monitors.

2. Attribute grammars

In this section we will follow in general the notations used in [7].

An attribute grammar is defined by a 4-tupel

$$AG = (CFG, A, SR, SC)$$

where $CFG=(N, T, P, S)$ is a reduced context-free grammar (N is the set of nonterminal symbols, T is the set of terminal symbols, P is the set of productions or syntactical rules and S is the start symbol), A is a finite set of attributes, SR is the set of semantic rules and SC is the set of semantic conditions.

A production $p \in P$ is denoted by $p: X_0 ::= X_1 X_2 \ldots X_{n_p}$, where $n_p \geqq 0$, $X_0 \in N$ and $X_i \in N \cup T$ for all i $(1 \leqq i \leqq n_p)$.

For each $X \in N$ there is a subset $A(X)$ of A. The set of attributes A is partitioned into two disjoint subsets A_S and A_I, the set of synthesized attributes and the set of inherited attributes: $A = A_S \cup A_I$ and $A_S \cap A_I = \emptyset$. Thus $A(X)$ is partitioned into two disjoint subsets $A_S(X)$ and $A_I(X)$, so that $A_S(X) \subseteqq A_S$, $A_I(X) \subseteqq A_I$ and $A(X) = A_S(X) \cup A_I(X)$.

If $p: X_0 ::= X_1 X_2 \ldots X_{n_p} \in P$ is a production, $X \in N$ occurs in p and $a \in A(X)$, then $X \cdot a$ denotes the attribute occurrence of a in p associated to X. The set A_p of attribute occurrences of a production p is defined by $A_p = \bigcup_{i=0}^{n_p} A_p(X_i)$, where $A_p(X_i) = \{X_i \cdot a : a \in A(X_i)\}$ if $X_i \in N$, and $A_p(X_i) = \emptyset$ if $X_i \in T$. The set OA_p of output attribute occurrences of a production p is defined by

$$OA_p = \{X_i \cdot a \in A_p \colon (i = 0 \text{ and } a \in A_S(X_i)) \text{ or }$$
$$(i > 0 \text{ and } X_i \in N \text{ and } a \in A_I(X_i))\},$$

and the set IA_p of input attribute occurrences of a production p is defined by

$$IA_p = \{X_i \cdot a \in A_p \colon (i = 0 \text{ and } a \in A_I(X_i)) \text{ or }$$
$$(i > 0 \text{ and } X_i \in N \text{ and } a \in A_S(X_i))\} = A_p \setminus OA_p.$$

For each $p \in P$ there is a subset SR_p of SR and a subset SC_p of SC, the set of the production semantic rules and the set of the production semantic conditions, so that

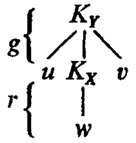
$$SR = \bigcup_{p \in P} SR_p \quad \text{and} \quad SC = \bigcup_{p \in P} SC_p.$$

For each production $p \in P$, for each attribute occurrence $X_i \cdot a \in OA_p$ there is one and only one semantic rule $f \in SR_p$ which determines the value of $X_i \cdot a$, and each semantic rule $f \in SR_p$ determines the value of some output attribute occurrence $X_i \cdot a \in OA_p$.

Let s be a sentence of $L(CFG)$ derived by

$$S \xrightarrow{*} xYy \xrightarrow[q]{} xuXvy \xrightarrow[r]{} xuwuy \xrightarrow{*} s$$

A node K_X represents the symbol X in the derivation tree t_s corresponding to that derivation and it is called an instance of X. For each attribute occurrence $X \cdot a$ an attribute instance $K_X \cdot a$ is associated to K_X. The values of the inherited attribute instances associated to K_X are defined by rules in SF_g, and the values of the synthesized attribute instances associated to K_X are defined by rules in SF_r.



A derivation tree augmented with the attribute instances is called an attributed derivation tree. An attribute evaluations strategy is an algorithm to calculate the value of each attribute instance. The single natural condition for the application of a semantic rule $f \in SR$ is that the value of the attribute instances which appear as arguments of f were calculated previously. This condition generates a dependence relation on the attribute instances of the attributed derivation tree.

An attribute grammar is well defined if and only if for each attributed derivation tree the graph belonging to the generated dependence relation is noncircular. A well defined attribute grammar is also called noncircular. The problem of the decision of attribute grammars noncircularity is NP-complete [5], but subclasses of the class of noncircular attribute grammars have been introduced in which we can decide in polynomial time whether an attribute grammar belongs to the subclass. Such subclasses are for example the LR [1], the ASE [6] and the OAG [7] attribute grammars.

3. Giving mathematical semantics by means of attribute grammars

As usual in the mathematical semantics we consider a program as a function on the set of the possible states

$$S_p = \{\{(v_1, t_1), \dots, (v_n, t_n)\} \colon t_1 \in T'_{v_1}, \dots, t_n \in T'_{v_n}\}$$

where $v_1, \dots, v_n$ are all the variables which appear in the program, $T'_{v_i} = T_{v_i} \cup \{\text{unvalued, undefined}\}$. T_{v_i} is the set from which the variable v_i takes its values. The variable v_i in a state is unvalued if it has no value and is undefined, if its name is not valid in that state. Later in section 5 we have to extend and consequently redefine the notion of the possible states.

The semantics of the statements and a program p too are functions $f_p \colon S_p \to S_p$. In this section we define such functions for the programs of a very simple sequential language. For this purpose we need the following attributes:
Synthesized attributes:

name — to give a unique identifier for each variable of the program
T — to give the type of each variable and arithmetical expression
V — to give the set of declared variables and their types
S — to give the set of the possible states
g — a function $g \colon S \to T'$ to give the value of an arithmetical expression in a given state, where T' is the type of the arithmetical expression

h — a function h: $S \to \{\text{true, false}\}$ to give the value of a logical expression in a given state
f — a function f: $S \to S$ to give the semantics of the statements and the programs of the language.

Inherited attributes:
V' — to give the variables valid in the environment and their types
S' — to transmit towards the levels of the tree the set of the possible states

Nonterminal symbols and their attributes:
program has V, S, f
declaration_statement has V, f
declaration_list has V
declaration has V
variable_list has V
variable has name
type has T
statement_list has V, f, V', S'
statement has V, f, V', S'
expression has g, T, V', S'
bool_expression has h, V', S'

Syntactical rules and their semantic rules and semantic conditions:
(In a production $X_0::=X_1X_2\ldots X_{n_p}$ $(n_p \geqq 0)$ we will omit the semantic rules of the form $X_0 \cdot a = X_i \cdot a$ $(1 \leqq i \leqq n_p)$ if there is no X_j $(1 \leqq j \leqq n_p$ and $i \neq j)$ which has the synthesized attribute a, and we will omit the semantic rules of the form $X_i \cdot a = X_0 \cdot a$ $(1 \leqq i \leqq n_p)$).

i) program::=**begin** declaration_statement; statement_list **end**
program. V=declaration_statement. V ∪ statement_list. V
program. S $=\{\{(v, t_v):(v, T) \in \text{program}.V\}: t_v \in T \cup \{\text{unvalued, undefined}\}\}$
program. $f(s)$=statement_list. f(declaration_statement. $f(s)$)
statement_list. V'=declaration_statement. V
statement_list. S'=program. S

ii) declaration_statement::=**var** declaration_list
declaration_statement. $f(s)=\{(v, s(v))$: there is not $(v_1, T_1) \in$ declaration_list. V for which $v_1=v\} \cup \{(v, \text{unvalued})$: there is $(v_1, T_1) \in$ declaration_list. V for which $v_1=v\}$

iii) declaration_list$_1$::=declaration; declaration_list$_2$
declaration_list$_1$ · V=declaration. V ∪ declaration_list$_2$. V
condition: if $(v_1, T_1) \in$ declaration. V and $(v_2, T_2) \in$ declaration_list$_2$. V then $v_1 \neq v_2$

iv) declaration_list::=declaration

v) declaration::=variable_list **of** type
declaration. $V=\{(v, \text{type}.T)$: $(v, \emptyset) \in$ variable_list. $V\}$

vi) variable_list$_1$::=variable, variable_list$_2$
variable_list$_1$. V=variable_list$_2$. V ∪ {(variable. name, $\emptyset$)}
condition: if $(v, \emptyset) \in$ variable_list$_2$. V then variable. name $\neq v$

vii) variable_list::=variable

$\quad$ variable_list. $V=\{(\text{variable.name}, \emptyset)\}$

viii) statement_list$_1$::=statement; statement_list$_2$

$\quad$ statement_list$_1$. V=statement. $V\cup$statement_list$_2$. V

$\quad$ statement_list$_1$.$f(s)$=statement_list$_2$.f(statement.$f(s)$)

$\quad$ condition: if $(v_1, T_1)\in$statement. V and $(v_2, T_2)\in$ statement_list$_2$. V then $v_1\neq v_2$

ix) statement_list::=statement

x) statement::=variable:=expression

$\quad$ statement. $V=\emptyset$

$\quad$ statement.$f(s)=\{(v, s(v))\colon v\neq \text{variable.name}\}\cup \{(\text{variable.name, expression}.g(s))\}$

$\quad$ condition: there is $(v, T_1)\in$statement. V' for which v=variable.name and T_1=expression.T

xi) statement$_1$::=**if** bool_expression **then** statement$_2$ **else** statement$_3$ **fi**

$\quad$ statement$_1$. V=statement$_2$. $V\cup$statement$_3$. V

$$\text{statement}_1.f(s)=\begin{cases}\text{statement}_2.f(s), \text{ if bool\_expression}.\\ \quad h(s)=\text{true}\\ \text{statement}_3.f(s), \text{ if bool\_expression}.\\ \quad h(s)=\text{false}\end{cases}$$

xii) statement$_1$::=**while** bool-expression **do** statement$_2$ **od**

$$\text{statement}_1.f(s)=\begin{cases}\text{statement}_1.f(\text{statement}_2.f(s)),\\ \quad \text{if bool\_expression}.h(s)=\\ \quad \text{true}\\ s, \text{ if bool\_expression}.h(s)=\text{false}\end{cases}$$

xiii) statement::=**begin** statement_list **end**

xiv) statement::=**begin** declaration_statement; statement_list **end**

$\quad$ statement. V=declaration_statement. $V\cup$statement_list. V

$\quad$ statement.$f(s)=\{(v$, statement_list.f(declaration_statement.$f(s))(v))$: there is not $(v_1, T_1)\in$declaration_statement. V for which $v_1=v\}\cup\{(v$, undefined): there is $(v_1, T_1)\in$declaration_statement. V for which $v_1=v\}$

$\quad$ statement_list. V'=statement. $V'\cup$declaration_statement. V

$\quad$ condition: if $(v_1, T_1)\in$declaration_statement. V and $(v_2, T_2)\in$ statement_list. V then $v_1\neq v_2$

It is easy to show that the attribute grammar given above is well defined (noncircular), but we do not deal with this in our paper.

4 Semantics of nondeterministic structures

The syntax of the nondeterministic programming structures introduced by Dijkstra can be given by a context-free grammar as follows:

i) statement::=alternative_construct
ii) statement::=repetitive_construct
iii) alternative_construct::=**if** guarded_command_set **fi**
iv) repetitive_construct::=**do** guarded_command_set **od**
v) guarded_command_set::=guarded_command $\square$ guarded_command_set
vi) guarded_command_set::=guarded_command
vii) guarded_command::=guard$\rightarrow$guarded_list
viii) guard::=bool_expression
ix) guarded_list::=statement_list

From the context-free grammar given above it is clear that Dijkstra introduced two new statements: the alternative construct and the repetitive construct, based on the concept of guarded commands. The semantics of these statements was given by Dijkstra in [2] with the following words: "The alternative construct is written by enclosing a guarded command set by the special bracket pair **if**...**fi**. If in the initial state none of the guards is true, the program will abort; otherwise an arbitrary guarded list with a true guard will be selected for execution. The repetitive construct is written down by enclosing a guarded command set by the special bracket pair **do**...**od**. Here a state in which none of the guards is true will not lead to abortion but to proper termination; the complementary rule, however, it will only terminate in a state in which none of the guards is true: when initially or upon completed execution of a selected guarded list one or more guards are true, a new selection of a guarded list with a true guard will take place, and so on. When the repetitive construct has terminated properly, we know that all its guards are false".

In the case of nondeterministic structures to give the semantics of the alternative construct and the semantics of the repetitive construct it is necessary to use functions f: $S_p \rightarrow 2^{S_p}$ which can be obtained as synthesized attributes. It is clear that for nondeterministic statements it is not sufficient to use functions of the type f: $S_p \rightarrow S_p$ because the state valid at the beginning of the execution of a nondeterministic statement do not determine a unique state valid at the termination of the statement, that is more than one state can be the real state when the program finished the execution of the nondeterministic statement.

Now we give an attribute grammar to obtain as synthesized attribute the functions which give the semantics of the nondeterministic statements:

Synthesized attributes:
f — to give the function f: $S_p \rightarrow 2^{S_p}$ which describes the semantics
h — a function h: $S_p \rightarrow \{\text{true, false}\}$ to give the value of a logical expression in a given state

Inherited attributes:

—

Nonterminals and their attributes:
statement has f

alternative_construct has f
repetitive_construct has f
guarded_command_set has f
guarded_command has f
guarded_list has f
statement_list has f
guard has h
bool_expression has h

Syntactical rules and their semantic rules:

i) statement::=alternative_construct
  statement.f=alternative_construct.f

ii) statement::=repetitive_construct
  statement.f=repetitive_construct.f

iii) alternative_construct::=**if** guarded_command_set **fi**
  alternative_construct.f=guarded_command_set.f

iv) repetitive_construct::=**do** guarded_command_set **od**

$$\text{repetitive\_construct}.f(s)=\begin{cases}\bigcup \text{repetitive\_construct}.f(s') \\ s'\in \text{guarded\_command\_set}.f(s) \\ \text{if guarded\_command\_set}.f(s)\neq\emptyset \\ s, \text{ if guarded\_command\_set}.f(s)=\emptyset\end{cases}$$

v) guarded_command_set$_1$::=guarded_command □ guarded_command_set$_2$
  guarded_command_set$_1$.$f(s)$=guarded_command.$f(s)\cup$ guarded_command_set$_2$.$f(s)$

vi) guarded_command_set::=guarded_command
  guarded_command_set.f=guarded_command.f

vii) guarded_command::=guard→guarded_list

$$\text{guarded\_command}.f(s)=\begin{cases}\text{guarded\_list}.f(s), \text{ if} \\ \quad \text{guard}.h(s)=\text{true} \\ \emptyset, \text{ if guard}.h(s)=\text{false}\end{cases}$$

viii) guard::=bool_expression
  guard.h=bool_expression.h

ix) guarded_list::=statement_list
  guarded_list.f=statement_list.f

Note: it is easy to see that the program aborts in an alternative construct if and only if the alternative construct is executed in such a state s for which the function f associated with the synthesized attribute to the alternative construct takes the empty set $\emptyset$ as its value. Furthermore this occurs if and only if all the guards in the guarded command set of the alternative construct are false.

5. Mathematical semantics of parallel programs and monitors

We will deal with parallel programs which have the following structure:

```
begin
    (definition of the monitors);
    process$_1$: process var $v_1^1, ..., v_{n_1}^1$;
                .
                . (statements of the process$_1$)
                .
                end of process$_1$
    and
    .
    . (description of the process$_2$, ..., process$_{m-1}$)
    and
    process$_m$: process var $v_1^m, ..., v_{n_m}^m$;
                .
                . (statements of the process$_m$)
                .
                end of process$_m$
end
```

The processes communicate with each other through Hoare's monitors [4]. A monitor is a collection of local data and procedures and has the following structure:

```
monitor_name: monitor
  begin
  (declaration of data local to the monitor)
    procedure proc_name (...formal parameters ...);
              begin
               .
               . (procedure body)
               .
              end;
    (declaration of other procedures local to the monitor);
  (initialization of local data of the monitor)
  end
```

To call a procedure of the monitor, it is necessary to give the name of the monitor and the name of the desired procedure:

monitor_name.proc_name (... actual parameters ...)

The procedures of a monitor are common to all existing processes, any process can at any time attempt to call such a procedure. However, it is essential that only one process at a time can be executing a procedure body, and any subsequent call must be held up until the previous call has been completed or has been held up. A process in execution can be held up by a wait statement and can be resumed by a signal statement. The structures of these statements are:

cond_variable.**wait**; cond_variable.**signal**

The cond_variable is a new type of variable, a condition variable, which is suitable to differentiate the reason for waiting. In practice, a condition variable is an initially empty queue of processes which are waiting on the condition.

A signal operation is followed immediately by resumption of a waiting process, without the possibility of an intervening procedure call from a third process. Wait operation is followed immediately by resumption of a process delayed by a signal instruction. New procedure can be executed only if there are not processes delayed by signal.

Now, after this short and necessary introduction, we will define the set of the possible program states by

$$S_p = \{\{(v_1^1, t_1^1), \ldots, (v_{n_1}^1, t_{n_1}^1), \ldots, (v_1^m, t_1^m), \ldots, (v_{n_m}^m, t_{n_m}^m), (1, q_1), \ldots, (m, q_m)\}:$$

$$t_1^1, \ldots, t_{n_1}^1, \ldots, t_1^m, \ldots, t_{n_m}^m \in T \text{ and } q_1, \ldots, q_m (\bigcup_{j\in M} (Z^j \times MC^j))^*\}$$

where T — is the type of the variables (for simplicity all the variables have the same type),

M is the set of declared monitors,

Z^j is the set of the possible states of the monitor j,

MC^j is the set of the possible monitor calls relative to the monitor j.

For each monitor j we define a function g_j: $MC^j \to 2^{Z^j}$ which is obtained as a synthesized attribute and is the function which gives the semantics of the monitor j. The g_j is a many-valued function because in the general case, for the termination of a monitor call the execution of other monitor calls are necessary which can not be predetermined, and furthermore the calculation of g_j have to be started from all possible states of the monitor j in which the call might occur. To each process i we associate a function f_i: $S_p \to 2^{S_p}$ which is also obtained as a synthesized attribute.

Let e: $j \cdot j_k(v') \in MC^j$ be a monitor call in the process i, where j is the called monitor, j_k is the called procedure and v' the actual parameter. We associate to this monitor call statement a function f_e: $S_p \to 2^{S_p}$ which is defined by

$f_e(s) = S' \subseteq S_p$ and $s' \in S'$ if and only if:

a) $s'(v) = s(v)$ for all v $(v \neq v')$
b) $s'(p) = s(p)$ for all p $(1 \leqq p \leqq m,\ p \neq i)$
c) $s'(v') = z'_j$ (parameter of j_k)
d) $s'(i) = s(i) \circ z'_j$ (call)
e) $z'_j \in g_j(e)$

where z'_j (parameter of j_k) gives the value of the parameter of the procedure j_k at the state z'_j of the monitor j, and z'_j (call) gives the execution sequence of monitor calls which leads to z'_j and the monitor states in which the monitor calls were executed. Because of this it is clear that in the elements of Z^j there is a pair of the form (call, x), where $x \in (Z^j \times MC^j)^*$.

It is necessary to introduce the pairs (i, q_i) for $1 \leqq i \leqq m$ in the set of possible program states because it is possible to decide whether a program state is really a possible program state only by the comparision of the sequences q_i $(1 \leqq i \leqq m)$. Only when the sequences q_i are in correspondence with each other the program state is really a possible program state. The condition for this correspondence is that it is possible to find a sequence $a_1 a_2 \ldots a_n \in (\bigcup_{j\in M} (Z^j \times MC^j))^*$ for which the following sentences are true:

i) if $a_i \in Z^j \times MC^j$ $(1 \leq i \leq n)$, then $a_i \in q_j$
ii) if b appears in q_i $(1 \leq i \leq m)$, then b appears in $a_1 a_2 \ldots a_n$
iii) if b precedes c in q_i, then b precedes c in $a_1 a_2 \ldots a_n$
iv) if b precedes c immediately in q_i, and b and c belong to the same monitor call of the process i; then b precedes c in $a_1 a_2 \ldots a_n$, and if b precedes d and d precedes c in $a_1 a_2 \ldots a_n$ then the monitor called in d is different from the monitor called in b or c.
v) if b is the first in $a_1 a_2 \ldots a_n$ which belongs to $(Z^j \times MC^j)$, then the first component of b is the initial state of the monitor j.

We do not give the attribute grammar for monitors and parallel processes because it is very long and can be constructed from the principles given in this section and the method described in the preceding sections.

6. Conclusions

In our opinion the attribute grammars are a powerful tool to give the mathematical semantics of programming languages in the case of nondeterministic programming structures or in the case of parallel processes communicating through Hoare's monitors too.

Attribute grammars give a mechanizable method to obtain for any program of the language the function described by the program; and consequently an attribute evaluation strategy can be viewed as a "compiler" which translates the program into a mathematical function.

Acknowledgement

The author wants to thank Dr. Árpád Makay for valuable discussions and suggestions.

Abstract

This paper describes attribute grammars for the description of the mathematical semantics of programming languages. It concentrates on the nondeterministic programming structures introduced by Dijkstra and parallel programming structures in which sequential processes communicate through Hoare's monitors.

DEPT. OF COMPUTER SCIENCES
A. JÓZSEF UNIVERSITY
ARADI VÉRTANÚK TERE 1
SZEGED, HUNGARY
H—6720

References

[1] BOCHMAN, G. V., Semantic evaluation form Left to Right. CACM 19, 2, (February, 1976), 55—62.
[2] DIJKSTRA, E. W., Guarded Commands, Nondeterminancy and Formal Derivation of Programs. CACM 18, 8, (August, 1975), 453—457.
[3] GANZINGER, H., Transforming denotational semantics into practical attribute grammars. In Lecture Notes in Computer Sciences 94, 1980, 1—69.
[4] HOARE, C. A. R., Monitors: An Operating Systems Structuring Concept. CACM 17, 10, (October, 1974), 549—557.
[5] JAZAYERI, M., OGDEN, W. F. and ROUNDS, W. C., The intrinsically exponential complexity of the circularity problem for attribute grammars. CACM 18, 12, (December, 1975), 697—721.
[6] JAZAYERI, M. and WALTER, K. G., Alternating Semantic Evaluator. In Proc. of ACM 1975 Ann. Conf., 230—234.
[7] KASTENS, U., Ordered Attributed Grammars. Acta Informatica 15, 1980, 229—256.
[8] KNUTH, D. E., Semantics of context-free languages. Math. Systems Theory 2, 1968, 127—145.
[9] MAYOH, B. H., Attribute grammars and mathematical semantics. SIAM J COMPUT. 10, 3, (August, 1981), 503—518.
[10] SCOTT, D. and STRACHEY, C., Towards a mathematical semantics for computer languages. Tech. Mon. PRG-6, Oxford U. Comp. Lab., 1971.

(Received Sept. 13, 1985.)