数理解析研究所講究録
第547巻 1985年 58-78

58

Algebraic Specification Method of Programming Languages

Hidehiko KITA*, Toshiki SAKABE* and Yasuyoshi INAGAKI*

* Department of Electrical Engineering, Faculty of Engineering, Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464 JAPAN

1. Introduction

The purposes of formal specification of programming languages are to establish the mathematical foundations for specification and verification of program, proofs of compiler correctness and automatic compiler generations.

The formal specification of a programming language consists of a syntactic specification and a semantic one. Our understanding of the former has now reached practical level. We are now able to automatically construct reasonably good lexical and syntactic analyzers for most programming languages directly from a defining grammar.

As for the latter, although many specification method have been proposed they are unsatisfactory as compared with that of syntax. For example, attribute grammar is suitable for the purpose of compiler generation in the sense that automatically constructing the compiler for a given attribute grammar is not so difficult. But usually in the attribute grammar approach the semantic domain is not explicitly described, and it is not suitable for program verification. Axiomatic semantics has been proposed mainly for the purpose of

program verifications, and it is not suitable for compiler generation. Denotational semantics is powerful and formal, but the semantic domain is intractable to discuss its implementation, since it it given as the solution of recursive equations on domains.

In the algebraic specification methods initially proposed by ADJ-group[3] and Mosses[9], respectively. The syntactic domain and the semantic domain are given as algebras, and the meaning of program is given by homomorphism from the syntactic domain to the semantic domain. This approach enables us to capture these domains as abstract data types and to apply the theory of abstract data types to the specification of programming languages. So, the algebraic specification method is a promising approach to program verification, compiler correctness problem and automatic compiler generation.

Among several algebraic specification methods, Mosses have proposed automatic compiler generation based on the idea that the compiler is the composition of the semantic mapping and the implementation of the semantic domain. In his approach, Mosses have considered semantic domain and target language as abstract data types, but, although his abstract data types are extended and differ from the ordinary ones, their formalization is not established. And we can not apply the results which we have already obtained in the theory of abstract data types.

In this paper we try to overcome these defects and we propose a new algebraic specification method of programming

language by strictly using the ordinary abstract data types for the specification of semantic domain. That is, we use only the equational logic so that the semantic domain should be precisely specified in purely algebraic way.

We apply our method and succeed in specifying the programming language PL/0, which Wirth[11] used as an illustrative example to explain the structure of compiler in his book. PL/0 is simple but has fundamental features of programming language. This experience shows that our specification method has enough power and formality for our purpose.

The rest of this paper is organized as follows. Section 2 introduces fundamental algebraic notions and notations which will be used throughout this paper. Sections 3 to 6 describe our specification method of programming languages. Finally, section 7 describes how we can specify the language PL/0 by using our specification method.

2. Σ-algebra

This section introduces the concepts and notations concerning many-sorted algebra, which will be used throughout this paper. We will use ADJ-like notations[2].

[Def. 2.1] A signature Σ is a pair $\langle S,F \rangle$ of S, a set of sorts and F, an operator symbol domain, which is an indexed family of sets $\langle F_{w,s} \rangle_{w \in S^*, s \in S}$. Here S^* denotes the set of all

sequences constructed from the elements of S. Assume that if $w \neq w'$ or $s \neq s'$ then $F_{w,s}$ and $F_{w',s'}$ are disjoint. We call $f \in F_{w,s}$ an operator symbol with arity w and sort s.

[Def. 2.2] Let $\Sigma = \langle S,F \rangle$ be a signature. A Σ-algebra A consists from a set A_s for each sort s S and a function $f_A: A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ for each operator symbol $f \in F_{w,s}, w = s_1 \ldots s_n$. Here we call A_s the carrier of A of sort s. If n=0, i.e., $f \in F_{\varepsilon,s}$, then f_A is considered to be an element of A_s, where ε is the empty sequence.

[Def. 2.3] Let $\Sigma = \langle S,F \rangle$ be a signature. Define $T[\Sigma]$ to be a smallest indexed family of sets $\langle T[\Sigma]_s \rangle_{s \in S}$ satisfying the following two conditions:

(1) For each $s \in S$, $F_{\varepsilon,s}$ is included in $T[\Sigma]_s$, i.e., $T[\Sigma]_s \supset F_{\varepsilon,s}$,

(2) If $f \in F_{w,s}, w = s_1 \ldots s_n$ and $t_i \in T[\Sigma]_{s_i}$ for i=1,...,n then $f(t_1,\ldots,t_n) \in T[\Sigma]_s$.

[Def. 2.4] For signature $\Sigma = \langle S,F \rangle$, we define the Σ-term algebra T as follows:

(1) For each sort $s \in S$, we define $T_s = T[\Sigma]_s$, that is the carrier of sort s is the set of Σ-terms of sort s.

(2) If $f \in F_{\varepsilon,s}$ then $f_T = f$.

(3) If $f \in F_{w,s}, w = s_1 \ldots s_n \neq \varepsilon$ and $t_i \in T[\Sigma]_{s_i}$ for i=1,...,n then $f_T(t_1,\ldots,t_n) = f(t_1,\ldots,t_n)$.

We will often use the notation $T[\Sigma]$ instead of T in order to indicate the signature Σ explicitly.

3. Algebraic Approach to Specification of Programming Languages

The formal specification of a programming language consists of two parts. The one is a syntactic specification which defines the set of well-formed programs. The other is a semantic specification which gives a meaning for each well-formed program.

In this paper, we take algebraic approach to specification of programming languages. That is, we strictly consider the syntactic and semantic domains as algebras and specify the meaning by defining a mapping from the syntactic domain to the semantic one. This approach allows us to capture these domains as abstract data types and to directly apply the theory of abstract data types to specification of programming languages.

Our specification is consisted of three parts, i.e., specifications of syntactic domain, semantic domain and semantic mapping.

Here we adopt the following definition:

[Def. 3.1] The specification of a language is a triple $\langle G, \mathcal{D}, \Gamma \rangle$ where G is a specification of a syntactic domain (a

context-free grammar), D is a specification of a semantic domain (a specification of an abstract data type), Γ is a specification of a semantic mapping (a set of semantic equations).

In the following three sections, we discuss how to specify the syntactic domain, the semantic domain, and the semantic mapping.

4. Specification of Syntactic Domains

Context-free grammar has been the formally required syntactic specification of programming languages since the publication of the Algol 60 report. The associated theory of context-free languages has become so well understood that we are now able to automatically construct reasonably good lexical and syntactic analyzers for most programming languages directly from defining grammars. We naturally decide to use context-free grammars to define the specifications of syntactic domains.

[Def. 4.1] The specification of a syntactic domain is an unambiguous context-free grammar $G=\langle V,V_T,P,S_0\rangle$, where V and V_T are disjoint finite sets of nonterminal and terminal symbols, respectively, and S_0 is a distinct symbol of V called the start symbol. P is the set of productions, the form of which is in $p:N\rightarrow\alpha$ where $N\in V,\alpha\in(V\cup V_T)^*$, and p is the label of production.

An unambiguous context-free grammar defines the syntactic

domain as a term algebra: Consider a context-free grammar $G=\langle V,V_T,P,S_0\rangle$ and define the signature $G=\langle V,P'\rangle$ as follows. We consider the set of nonterminal symbols as the set of sorts. P' is the operator symbol domain defined by $P'=\langle P'_{w,s}\rangle_{w\in V^*,s\in V}$, where $P'_{w,s}=\{p \mid p:N\to\alpha\in P,\ s=N,\ w=nt(\alpha)\}$, and $nt:(V\cup V_T)^*\to V^*$ is a function such that $nt(\alpha)$ denotes the sequence of nonterminal symbols obtained from α by replacing all terminal symbols occurring in α by the empty sequence. That is, we consider the production $p:N\to\alpha$ as an operator symbol with arity $w=nt(\alpha)$ and sort $s=N$.

According to the definition 2.4, the signature G defines the term algebra T[G], which constitutes the syntactic domain.

[Def. 4.2] For a context-free grammar G, the specification of a syntactic domain, the term algebra T[G] is the syntactic domain. It will often denoted by L(G).

An element of the carriers of the term algebra T[G] is a G-term, and it can be considered to the usual derivation tree in the 1:1 manner. In this context,the carrier of sort N corresponds to the set of derivation trees with the root node labeled by N.

5. Specification of Semantic Domains

In this paper, we take the semantic domain to be an abstract data type and adopt the algebraic approach to the specification. ADJ[3], Mosses[9] and other authors[5],[6],[7],

(10) have already tried the algebraic approach to the specification of programming languages. These approaches (except for Pair(10)) contain some informal treatments in specifying semantic domain. This causes some difficulties to the formal development of the the semantics of languages. This also make it difficult to construct the compilers automatically from the specification.

To overcome this point, we have the idea that the semantic domain should be strictly an abstract data type and it should be interpreted through only equational logic.

We begin with introducing the concept of the equational logic. Let $\Sigma = \langle S, F \rangle$ be a signature and $V = \langle V_s \rangle_{s \in S}$ be a family of sets of variables. Equational logic has only one logical symbol $\approx$, called the equality symbol and it consists of the set of equations or equivalently axioms and the set of inference rules. An equation is a sequence $\xi \approx \eta$ where ξ and η are $\Sigma(V)$-terms of the same sort. An equation over $\Sigma(V)$-terms will often be called the Σ-axiom. The set of inference rules consists of the following five rules.

(1) Reflexitivity $\quad \vdash \xi \approx \xi$

(2) Symmetry $\quad \xi \approx \eta \vdash \eta \approx \xi$

(3) Transitivity $\quad \xi \approx \eta, \eta \approx \zeta \vdash \xi \approx \zeta$

$\Sigma(V)$ denotes a signature $\langle S, F' \rangle$ where F' is an operator symbol domain $\langle F'_{w,s} \rangle_{w \in S^*, s \in S}$, such that $F'_{w,s} = F_{w,s} \cup V_s$ if $w = \varepsilon$ and $F'_{w,s} = F_{w,s}$ otherwise.

(4) Substitution $\quad \xi \approx \eta \ \vdash \ \xi[\zeta/v] \approx \eta[\zeta/v]$

(5) Replacement $\quad \xi \approx \eta \ \vdash \ \zeta[\xi/v] \approx \zeta[\eta/v]$

where ξ, η and ζ are $\Sigma(V)$-terms and $\xi[\eta_1/v_1, \ldots, \eta_n/v_n]$ denotes the term obtained from ξ by simultaneously replacing all v_i's occurring in ξ by η_i for $i=1,\ldots,n$. For a set of Σ-axioms A, if equation $\xi \approx \eta$ is deducible from A by applying inference rules then we write $A \vdash \xi \approx \eta$.

A set of Σ-axioms A defines an algebra which is the meaning given by A. We explain this in the following.

First, we define the congruence relation $\equiv$ over the term algebra $T[\Sigma]$ by:
For any terms t and t' in $T[\Sigma]_s$, $t \equiv_s t'$ iff $A \vdash t \approx t'$.

Next, construct the quotient algebra $T[\Sigma]/\equiv$ of the term

---

For a signature $\Sigma = \langle S, F \rangle$, a Σ-congruence $\equiv$ on a Σ-algebra A is a family $\langle \equiv_s \rangle_{s \in S}$ of equivalence relations $\equiv_s$ on A_s for $s \in S$, such that if $f \in F_{s_1 \ldots s_n, s}$, $a_i, b_i \in A_{s_i}$, $a_i \equiv_{s_i} b_i$ for $i=1,\ldots,n$ then $f_A(a_1,\ldots,a_n) \equiv_s f_A(b_1,\ldots,b_n)$.
For $a \in A_s$, let $[a]_s$ (or $[a]$) denote the $\equiv_s$-equivalence class of A_s, $[a] = \{b \in A_s \mid a \equiv_s b\}$.

For a Σ-congruence $\equiv$ on a Σ-algebra A, the quotient algebra $A/\equiv$ is a Σ-algebra defined as follows.

(1) For each sort $s \in S$, the carrier $(A/\equiv)_s$ is the set of $\equiv_s$-equivalence classes of A_s.

(2) If $f \in F_{s_1 \ldots s_n, s}$ and $[a_i] \in (A/\equiv)_{s_i}$ for $i=1,\ldots,n$, then

algebra $T[\Sigma]$ by congruence relation $\equiv$. Then the quotient algebra $T[\Sigma]/\equiv$ is the initial algebra of the class $Alg_{\Sigma,A}$ of Σ-algebras which satisfies the set of Σ-axioms A. That is, for any algebra $A \in Alg_{\Sigma,A}$, there exists the unique homomorphism from $T[\Sigma]/\equiv$ to A.

From the above observation, we give the following definitions.

[Def. 5.1] A specification of a semantic domain is a triple $D=\langle \Sigma, V, A \rangle$, where Σ is a signature $\langle S,F \rangle$, V is a family of variable sets $\langle V_s \rangle_{s \in S}$, and A is a set of Σ-axioms.

The meaning of the specification D, i.e., the semantic domain specified by D is the quotient algebra $T[\Sigma]/\equiv$. It will be often denoted by $SD(D)$.

6. Specification of Semantic Mappings

We are ready to define the semantic mapping from the syntactic domain to the semantic domain. Here, we use the primitive recursive schemes to specify the semantic mapping, which can be proved to determine the unique semantic mapping.

[Def. 6.1] Let $G=\langle V, V_T, P, S_0 \rangle$ be a specification of a syntactic domain and $D=\langle \Sigma, V, A \rangle$ be a specification of a semantic

---

$f_{A/\equiv}([a_1],\ldots,[a_n]) = [f(a_1,\ldots,a_n)]$.

domain.

A specification of a semantic mapping Γ is a quadruple $\langle D,M,X,R\rangle$ where D is a function $D:V\to S$ which associates nonterminal symbols with sorts of the semantic domain, M is a set of function variables M_N $arity(M_N)=N\in V$, $sort(M_N)=D(N)\in S$, X is a family of variables sets on the syntactic domain $\langle X_N\rangle_{N\in V}$, and R is a sets of semantic equations $\{R_p \mid p\in P\}$.

A semantic equation R_p for production $p\in P_{N_1\ldots N_n,N}$ is of the form

$$M_N(\ p(x_1,\ldots,x_n)\ ) = \xi[\ M_{N_1}(\ x_1\ )/y_1,\ \ldots\ ,\ M_{N_n}(\ x_n\ )/y_n\ ]$$

where x_i is a variable with $sort(X_{N_i})$ for $i=1,\ldots n$, y_i's are variables on the semantic domain, $sort(y_i)=D(X_{N_i})$ for $i=1,\ldots,n$, and ξ is a $\Sigma(\{y_1,\ldots,y_n\})$-term.

Note that the class of set of semantic equations is a subclass of primitive recursive schemes used by Courcelle[4].

It is natural that the semantic mapping determined by the specification Γ is defined to be the solution of the semantic equations R.

Let $\Gamma=\langle D,M,X,R\rangle$ be a specification of a semantic mapping and A be a Σ-algebra. The solution of the semantic equations R over the Σ-algebra A is the indexed family of functions $M^A=\langle M^A_N : T[G]_N \to A_{D(N)}\rangle_{N\in V}$ such that for any semantic equation $R_p(p\in P_{N_1\ldots N_n,N})$ and $t_i\in T[G]_{N_i}$ for $i=1,\ldots,n$,

$$M_N(\ p(t_1,\ldots t_n)\ ) = \xi_A(\ M^A_{N_1}(\ t_1\ ),\ \ldots\ ,M^A_{N_n}(\ t_n\ )\ ) .$$

Here, ξ_A is the derived operation of ξ over A.

Since the specification of semantic mapping is a primitive recursive scheme, we can easily prove the following result(8).

[Proposition 6.1] For any Σ-algebra A, the semantic equations R has the unique solution M^A.

Now we can define a semantic mapping.

[Def. 6.2] Let $\Gamma=\langle D,M,X,R\rangle$ be the specification of a semantic mapping and $SD(\mathcal{D})$ be a semantic domain.

The semantic mapping $sem(\Gamma)$ is the solution of the set of semantic equations R over the semantic domain $SD(\mathcal{D})$.

The next corollary is immediately obtained from

For a $\Sigma(\{y_1,\ldots,y_n\})$-term ξ with $sort(y_i)=s_i$ for $i=1,\ldots,n$, we define a mapping $\xi_A: A_{s_1} \times \ldots \times A_{s_n} \to A_s$ called the derived operation of ξ over A as follows:

If $\bar{a}=(a_1,\ldots a_n)\in A_{s_1}\times\ldots\times A_{s_n}$ then

$$\xi_A(\bar{a}) = \begin{cases} \xi_A & \text{if } \xi=f\in F_{\varepsilon,s} \\ a_i & \text{if } \xi=y_i \\ f_A(\eta_{1_A}(\bar{a}),\ldots,\eta_{m_A}(\bar{a})) & \text{if } \xi=f(\eta_1,\ldots,\eta_m),\ f\in F_{s_1'\ldots s_m',s'}, \\ & \quad \text{and } \eta_i\in T[\Sigma(\{y_1,\ldots,y_n\})]_{s_i}, \text{for } i=1,\ldots,m \end{cases}$$

Proposition 6.1.

[Corollary 6.2] There exists the unique semantic mapping sem(Γ) for the specification of semantic mapping Γ.

7. Example

We have tried to give the specification of PL/0, a toy programming language given by Wirth[11], to show that our specification method works satisfactorily. PL/0 is, of course , a very simplified language but it has the fundamental features of the programming languages. It has declarations of variables, constants and procedures, arithmetic operations over integers , assignment, and such the control structures as sequencing, if-then , while and procedure call.

The specification of programming language PL/0 is given by using our method as follows.

```
(* Specification of the syntactic domain (Excerpts) *)

G = < V, V_T, P, S_0 >

 V   = { PROGRAM BLOCK CONST_DEF_PART CONST_DEF_LIST
         CONST_DEF VAR_DCL_PART VAR_NAME_LIST VAR_NAME
         PROC_DCL_PART PROC_DCL_LIST PROC_DCL STATEMENT
         STATEMENT_LIST CONDITION EXPRESSION TERM FACTOR
         IDENT NUMBER NAT_NUMBER                          }

 V_T = { .  const  ;  ,  =  var  procedure  :=  call
         begin  end  if  then  while  do  odd  <>  <  >
         <=  >=  +  -  *  /  (  )  a  ...  z  0  ...  9  }

 P =   { p010 : PROGRAM  ->  BLOCK .
```

```
p020 : BLOCK      ->  CONST_DEF_PART
                      VAR_DCL_PART
                      PROC_DCL_PART
                      STATEMENT

p030 : CONST_DEF_PART    ->  const CONST_DEF_LIST ;
p040 : CONST_DEF_PART    ->
p050 : CONST_DEF_LIST    ->  CONST_DEF
p060 : CONST_DEF_LIST    ->  CONST_DEF , CONST_DEF_LIST
p070 : CONST_DEF         ->  IDENT = NUMBER

p080 : VAR_DCL_PART      ->  var VAR_NAME_LIST ;
p090 : VAR_DCL_PART      ->
p100 : VAR_NAME_LIST     ->  VAR_NAME
p110 : VAR_NAME_LIST     ->  VAR_NAME , VAR_NAME_LIST
p120 : VAR_NAME          ->  IDENT

p130 : PROC_DCL_PART     ->  PROC_DCL_LIST ;
p140 : PROC_DCL_PART     ->
p150 : PROC_DCL_LIST     ->  PROC_DCL
p160 : PROC_DCL_LIST     ->  PROC_DCL ; PROC_DCL_LIST
p170 : PROC_DCL          ->  procedure IDENT ; BLOCK

p180 : STATEMENT         ->  IDENT := EXPRESSION
p190 : STATEMENT         ->  call IDENT
p200 : STATEMENT         ->  begin STATEMENT_LIST end
p210 : STATEMENT         ->  if CONDITION then STATEMENT
p220 : STATEMENT         ->  while CONDITION do STATEMENT
p230 : STATEMENT         ->
p240 : STATEMENT_LIST    ->  STATEMENT
p250 : STATEMENT_LIST    ->  STATEMENT ; STATEMENT_LIST

p260 : CONDITION         ->  odd EXPRESSION
p270 : CONDITION         ->  EXPRESSION = EXPRESSION
                         :
                         :
}
```

S_0 = PROGRAM

(* Specification of the semantic domain (Excerpts) *)

$D = \langle \Sigma, V, A \rangle$

$\Sigma = \langle S, F \rangle$

```
S = { STATE STATE-STATE STATE-INT STATE-BOOL
      STATE-STATE-STATE STATE-ATTR ATTR ...   }

F = { INIT_STATE      : -> STATE
      ADD_ID, UPDATE : STATE ID ATTR -> STATE
      RETRIEVE       : STATE ID -> ATTR
```

```
    ENTER_BLOCK, LEAVE_BLOCK
                      : STATE -> STATE
    I_STATE-STATE     : -> STATE-STATE
    APPLY_STATE       : STATE-STATE STATE -> STATE
    IF_STATE_D        :
            STATE-BOOL STATE-STATE STATE-STATE
             -> STATE-STATE
    ITERATE           : STATE-BOOL STATE-STATE
                        -> STATE-STATE
    COMPOSITION       : STATE-STATE STATE-STATE
                        -> STATE-STATE
    ADD_ID_D          : STATE-STATE ID ATTR
                        -> STATE-STATE
    ENTER_BLOCK_D, LEAVE_BLOCK_D
                      : STATE-STATE -> STATE-STATE
    UPDATE_D          : STATE-STATE ID STATE-ATTR
                         -> STATE-STATE
        :                                          }
```

V = < { s0, s1, s2, s3 } >$_{s \in S}$

A = { RETRIEVE(INIT_STATE(), id0)
  ≈ UNDEF_ATTR()
RETRIEVE(ENTER_BLOCK(state0), id0)
  ≈ RETRIEVE(state0, id0)
RETRIEVE(ADD_ID(state0,id0,attr0), id1)
  ≈ IF_ATTR(
        EQUAL_ID(id0,id1),
        attr0,
        RETRIEVE(state0,id1))

UPDATE(INIT_STATE(), id0, attr0)
  ≈ INIT_STATE()
UPDATE(ENTER_BLOCK(state0), id0, attr0)
  ≈ UPDATE(state0, id0, attr0)
UPDATE(ADD_ID(state0,id0,attr0), id1, attr1)
  ≈ IF_STATE(
        EQUAL_ID(id0,id1),
        ADD_ID(state0,id0,attr1),
        ADD_ID(UPDATE(state0,id1,attr1),
                id0, attr10))

LEAVE_BLOCK(INIT_STATE())
  ≈ INIT_STATE()
LEAVE_BLOCK(ENTER_BLOCK(state0))
  ≈ state0
LEAVE_BLOCK(ADD_ID(state0,id0,attr0))
  ≈ LEAVE_BLOCK(state0)

APPLY_STATE(I_STATE-STATE(), state0)
  ≈ state0
APPLY_STATE(IF_STATE_D(state-bool0,
                        state-state0,state-state1),

```
                    state0 )
          ≈ IF_STATE(
                  APPLY_STATE_BOOL(state-bool0,state0),
                  APPLY_STATE(state-state0,state0),
                  APPLY_STATE(state-state1,state0) )
        APPLY_STATE( ITERATE(state-bool0,state-state0),
                    state0 )
          ≈ IF_STATE(
                APPLY_STATE_BOOL(state-bool0.state0),
                APPLY_STATE(
                   COMPOSITION(
                      ITERATE(state-bool0,state-state0),
                              state-state0 ),
                   state0 ),
                state0 )
        APPLY_STATE( COMPOSITION(state-state0,
                                 state-state1),
                    state0 )
          ≈ APPLY_STATE(
                  state-state0,
                  APPLY_STATE( state-state1, state0 ) )
        APPLY_STATE( ADD_ID_D(state-state0,id0,attr0),
                    state0 )
          ≈ ADD_ID( APPLY_STATE(state-state0,state0),
                    id0,
                    attr0 )                                    }
```

```
(* Specification of the semantic mapping (Excerpts) *)

Γ= < D, M, X, R >

  D : V -> S
    D( PROGRAM ) = STATE
    D( BLOCK ) = D( CONST_DEF_PART )
  = D( CONST_DEF_LIST ) = D( CONST_DEF )
  = D( VAR_DCL_PART ) =  D( VAR_LIST )
  = D( VAR_NAME ) =  D( PROC_DCL_PART )
  = D( PROC_DCL_LIST ) =  D( PROC_DCL )
  = D( STATEMENT ) =  D( STATEMENT_LIST )
  = STATE-STATE
    D( CONDITION ) = STATE-BOOL
    D( EXPRESSION ) = STATE-INT

  M = { M_N | N∈V,sort(M_N)=D(N),arity(M_N)=N}

  X = < { x0, x1, x2, x3 } >N∈V

  R = { (* p010 : PROGRAM  ->  BLOCK . *)
        M_PROGRAM( p010(x0) )
         = APPLY_STATE(
                M_BLOCK( x0 ),
                INIT_STATE() )
```

```
(* p020 : BLOCK  ->  CONST_DEF_PART
                     VAR_DCL_PART
                     PROC_DCL_PART
                     STATEMENT       *)
M_BLOCK( p020(x0,x1,x2,x3) )
 = COMPOSITION(
        M_STATEMENT( x3 ),
        COMPOSITION(
             M_PROC_DCL_PART( x2 ),
             COMPOSITION(
                  M_VAR_DCL_PART( x1 ),
                  COMPOSITION(
                       M_CONST_DEF_PART( x0 ),
                       I_STATE-STATE() )

(* p060 : CONST_DEF_LIST ->  CONST_DEF ,
                             CONST_DEF_LIST *)
M_CONST_DEF_LIST( p060(x0,x1) )
 = COMPOSITION(
        M_CONST_DEF_LIST( x1 ),
        M_CONST_DEF( x0 ) )

(* p070 : CONST_DEF ->  IDENT = NUMBER *)
M_CONST_DEF( p070(x0,x1) )
 = ADD_ID_D(
        I_STATE-STATE(),
        M_IDENT( x0 ),
        MAKE_ATTR_CONST(
             M_NUMBER( x1 ) ) )

(* p120 : VAR_NAME ->  IDENT *)
M_VAR_NAME( p120(x0) )
 = ADD_ID_D(
        I_STATE-STATE(),
        M_IDENT( x0 ),
        MAKE_ATTR_VAR(
             ZERO() ) )

(* p170 : PROC_DCL ->  procedure IDENT ;
                       BLOCK           *)
M_PROC_DCL( p170(x0,x1) )
 = ADD_ID_D(
        I_STATE-STATE(),
        M_IDENT( x0 ),
        MAKE_ATTR_PROC(
             M_BLOCK( x1 ) ) )

(* p180 : STATEMENT ->  IDENT := EXPRESSION *)
M_STATEMENT( p180(x0,x1) )
 = UPDATE_D(
        I_STATE-STATE(),
        M_IDENT( x0 ),
        MAKE_ATTR_VAR(
```

```
                M_EXPRESSION( x1 ) ) )

(* p190 : STATEMENT ->  call IDENT *)
M_STATEMENT( p190(x0) )
 = LEAVE_BLOCK_D(
        APPLY_STATE_D(
             MAKE_STATE-STATE_D(
                  RETRIEVE_D(
                       I_STATE-STATE(),
                       M_IDENT( x0 ) ) ),
        ENTER_BLOCK_D(
             I_STATE-STATE() ) ) )

(* p210 : STATEMENT ->  if CONDITION then
                        STATEMENT          *)
M_STATEMENT( p210(x0,x1) )
 = IF_STATE_D(
        M_CONDITION( x0 ),
        M_STATEMENT( x1 ),
        I_STATE-STATE() )

(* p220 : STATEMENT ->  while CONDITION do
                        STATEMENT           *)
 = ITERATE(
        M_CONDITION( x0 ),
        M_STATEMENT( x1 ) )

(* p260 : CONDITION ->  odd EXPRESSION *)
M_CONDITION( p260(x0) )
 = ODDP_D( M_EXPRESSION( x0 ) )

                :
                :                                    }
```

Finally we should make some words concerning our idea in writing the above specification : To capture the meaning of programs, we introduced the concept of state which is the abstraction of configuration of computation mechanisms. And we consider that the meaning of a program is the final state after executing the program. That is, we consider that the meanings of statements as well as declarations are the functions to change the states. For example, the assignment statement changes the state through renewing the value of a variable, and the variable

declaration also changes the state through entering a new variable into the name table.

Note that we use a conventionally simplified way to treat the semantic errors in the specification of PL/0. For example, if update operation is applyed to the initial state that is the state where no variables are yet declared, then the result is the specified to be initial state. But, this should be specified to be a semantic error. Thus, how to specify and treat the semantic errors is one of the future problem.

8. Conclusion

In this paper, we have proposed an algebraic method for specification of programming languages. The semantic domain of our specification is specified as an abstract data type only by using the equational logic. This gives us mathematical foundations for the approaches to the formal specification, implementation, verification of programs , the formal proofs of compiler correctness, and the automatic compiler generations.

As an illustrative example, we have also showed the specification of programming language PL/0 given by using our method. It shows that our method works satisfactorily.

There are many future problems. For example, the error handling problem is one of them. In fact, in our example of PL/0 specification we used conveniently simplified ways to treat semantic errors, e.g. we assumed that if a number is divided by zero then the result value is zero. We are now developing the

system for automatic compiler generation based on our specification method. We already have a prototype of the system but ther are many problems to be solved.

Acknowledgement

The authors wish to express our gratitude to Dr.Namio HONDA, President of Toyohashi University of Technology and Dr. Teruo FUKUMURA, Professor of Nagoya University for their encouragements to conduct this work. They also thank their colleagues for their helpful discussions.

References

(1) ADJ (Goguen,J.A.,Thatcher,J.W.,Wagner,E.G.,Wright,J.B.) : Initial Algebra Semantics and Continuous algebras, J.ACM, Vol. 24, pp. 68-95 (1977).

(2) ADJ (Goguen,J.A.,Thatcher,E.G.,Wright,J.B.) : An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types,Current Trends in Programming Methodology, Vol. 4 (Yeh,R.T.,ed.), Prentice-Hall (1978).

(3) ADJ (J.A.,Thatcher,E.G.,Wagner,E.G.,Wright,J.B.) : More on Advice on Structuring Compilers and Their Correctness, Theor. Comput. Sci., Vol. 15, pp. 223-249 (1981).

(5) Despryroux,J. : An Algebraic Specification of a Pascal Compiler, SIGPLAN Notice, Vol. 18, No. 12, pp. 34-48 (1983).

(6) Gaudel,M.C. : Specification of Compilers as Abstract Data Type Representations, Proc. on Workshop on Semantics-Directed Compiler Generation, Aarhus, in Lecture Notes in Computer Science 94 (1980).

(7) Goguen,J.A. and Parsaye-Ghomi,K. : Algebraic Denotational Semantics using Parameterized Abstract Modules, in Lecture Notes in Computer Science 107, pp. 292-309 (1981).

(8) Kita,H., Sakabe,T. and Inagaki,Y. : Formal Semantics of Languages based on Abstract Data Types, Technical Report of Group on Automata and Languages, IECE of Japan, AL83-23 (1983). (in Japanese)

(9) Mosses,P. : A Constructive Approach to Compiler Correctness Proc. of Workshop on Semantics-Directed Compiler Generation, Aarhus, in Lecture Notes in Computer Science 94 (1980).

(10) Pair,C. : Abstract Data Types and Algebraic Semantics of Programming Languages, Theor. Comput. Sci., Vol. 18, pp. 1-31 (1982).

(11) Wirth,N. : Algorithms + Data Structure = Programs, Prentice-Hall (1976).