Res. Rep. Fac. Eng. Mie Univ., Vol. 11, pp. 43–54 (1986)

Original Paper

Algebraic Specification Method of Programming Languages

Hidehiko KITA*, Toshiki SAKABE
and Yasuyoshi INAGAKI*
(Common Chair of Engineering Mathematics)

(Received September 18, 1986)

The purpose of formal specification of programming languages are to establish the mathematical foundation for specification and verification of programs, proof of compiler correctness and automatic compiler generation. We propose a purely algebraic approach to develop a new algebraic specification method of programming languages. In this paper, the syntactic and semantic domains are considered as algebras, i.e., abstract data types, and the semantics of the language is given by a mapping from the syntactic domain to the semantic one. As an illustrative example, we describe a very simple language by our method.

1. Introduction

The purposes of formal specification of programming languages are to establish the mathematical foundations for specification and verification of programs, proof of compiler correctness and automatic compiler generation.

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

As for the latter, recently the mathematical foundations of programming

* Department of Electrical Engineering, Faculty of Engineering, Nagoya University

language semantics have become much better understood, but not yet as well as those of syntax. The formal specification of programming language semantics and its automatic translation into machine language semantics are still in active research areas.

Among many approaches such as attribute grammar, axiomatic semantics, VDL, and denotational semantics, the algebraic specification method has been given much attention by many researchers since ADJ-group[3] and Mosses[8]. In their approaches, the syntactic and the semantic domains are considered as algebras, i.e., abstract data types, and the semantics of the language is given by a mapping from the syntactic domain to the semantic one.

Many algebraic specification methods have been proposed so far. Most attractive one among them is Mosses's approach in the sense that the compiler can be considered to be the composition of the semantic mapping and implementation mapping as illustrated by the conceptual diagram in Fig. 1. In the diagram Mosses considered the semantic domain S and the target language T as abstract data types. But, the concept of his abstract data types is somewhat vague and the semantic domain S can not be described in a purely algebraic way. He used some operators not belonging to S for the specification of S. For example, the binding operators of S are given with the help of the notation for syntactic substitution, which is considered as a meta-notation. On the other hand, some operators act on infinite families. These seem to undermine the whole algebraic framework. This injures the comprehensibility of the specification and makes it difficult to directly apply the results obtained in the theory of algebraic specification. Thus, it is required to overcome these defects.

In this paper, we have taken a purely algebraic approach to develop a new algebraic specification method of programming languages. That is, we have developed a new algebraic specification method which uses only the equational logic. We have also applied our method successfully to specifying the programming language PL/0, which Wirth[10] used as an illustrative example to explain the structure of compiler in his book. PL/0 has the fundamental features of programming languages although it is a very simplified one. Our 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 the fundamental algebraic notions and notations which will be used throughout this paper. Sections from 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.

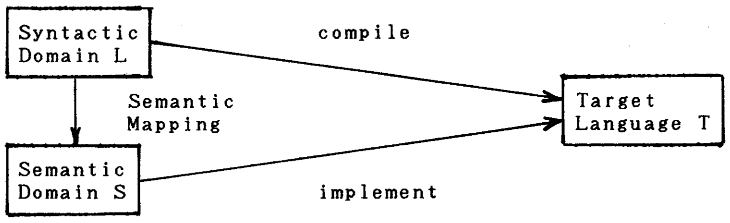

Fig. 1

2. Many-sorted algebras and related notions

This section introduces the concepts and notations concerning many-sorted algebras, which will be used throughout this paper. Our notations are similar to those of ADJ[2].

[Def. 2.1] Let S be a set of sorts. An S-sorted signature is a set Σ of operation symbols associated with mappings sort: $\Sigma \to S$ and arity: $\Sigma \to S^*$ where S^* denotes the set of all sequences over S including the empty string ε. We call $f \in \Sigma$ an operator symbol with arity w and sort s if arity(f)=w and sort(f)=s. An operation symbol f with arity ε is called a constant.

[Def. 2.2] Let Σ be an S-sorted signature. A Σ-algebra A consists of an S-sorted set $|A|$ and a function f_A: $|A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$ for each $f \in \Sigma$ with arity $s_1 \ldots s_n$ and sort s, where an S-sorted set is a set $|A|$ associated with function sort: $|A| \to S$, and

$|A|_s = \{a \in |A| \mid \text{sort}(a)=s\}$.

$|A|$ ($|A|_s$) is sometimes called the carrier of A (of sort s). In what follows, we often write A to denote algebra A and its carrier $|A|$ in the case that no confusion is caused. Note that for each $f \in \Sigma$ if arity(f)=ε and sort(f)=s then f_A designate an element of A_s. Therefore, we regard f_A as an element of A_s as well as a nullary function .

[Def. 2.3] Let Σ be an S-sorted signature. Define $T[\Sigma]$ to be the smallest S-sorted set $T[\Sigma]$ satisfying the following two conditions:

(1) If $f \in \Sigma$ is with arity ε and sort s then $f \in T[\Sigma]_s$.

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

Elements of $T[\Sigma]$ are called Σ-terms.

[Def. 2.4] For an S-sorted signature Σ, we define the Σ-term algebra T as follows:

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

(2) If $f \in \Sigma$ and arity(f)=ε then $f_T = f$.

(3) If $f \in \Sigma$, arity(f)=$s_1 \ldots s_n$ 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 explicitly indicate the signature Σ.

3. Algebraic Approach to Programming Languages Specification

Generally, formal specification of a programming language consists of two parts. One is syntactic specification which defines the set of well-formed programs. The other is semantic specification which gives meanings for well-formed programs.

In this paper, we take an algebraic approach to specification of programming languages. That is, we consider the syntactic and semantic domains strictly as algebras and specify the meaning of the language 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 of a programming language consists of three parts, i.e., the specification of the syntactic domain, the semantic domain, and the semantic mapping.

Here we adopt the following definition:

[Def. 3.1] A specification of a language is a triple $<G, D, \Gamma>$ where G is the specification of the syntactic domain (the context-free grammar), D is the specification of the semantic domain (the specification of an abstract data type), Γ is the specification of the semantic mapping (the 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 grammars have been used for formal 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] (Specification of Syntactic Domain)

A specification of a syntactic domain is an unambiguous context-free grammar $G = <V, V_T, P, S_0>$, where V and V_T are the disjoint finite sets of nonterminal and terminal symbols, respectively, S_0 is the distinguished symbol of V called the start symbol, and P is the set of productions, the form of which is $p:N\text{->}\alpha$, where $N\in V, \alpha\in (V\cup V_T)^*$, and p is the name of the production.

An unambiguous context-free grammar induces a term algebra which gives the syntactic domain: Let $G = <V, V_T, P, S_0>$ be a context-free grammar. We regard the set V of nonterminal symbols as a set of sort and define the V-sorted signature Σ_G as follows.

$\Sigma_G = \{p \mid p:N\text{->}s\alpha \in P\}$

and

$arity(p)=nt(\alpha)$ and $sort(p)=N$

for each $p:N\text{->}\alpha$ in G, where $nt(\alpha)$ denotes the sequence of nonterminal symbols obtained from α by removing all terminal symbols occurring in α. We denote such a signature Σ_G induced by a grammer G by G for short.

According to the definition 2.4, the signature G defines the G-term algebra

T[G], which constitutes the syntactic domain.

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

Elements of the carrier of the term algebra T[G], which are G-terms, correspond to the usual derivation trees in the 1:1 manner. That is, 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 an algebraic approach to abstract data type specification. ADJ[3], Mosses[8] and other authors[5),6),7),9] have already tried algebraic approaches to formal specification of programming languages. These approaches (except for Pair[9]) contain some informal treatments in specifying semantic domain, which seem to undermine their whole algebraic frameworks.

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

We begin with introducing some concepts of the equational logic. Let Σ be an S-sorted signature and X be an S-sorted set of variables. The signature obtained by adding variables of X to Σ as constants is denoted by $\Sigma(X)$. A formula of equational logic is a sequence of the form $\xi == \eta$, where ξ and η are $\Sigma(X)$-terms of the same sort, and == is the logical symbol of equational logic. An equation over $\Sigma(X)$-terms will often be called the Σ-axiom. The inference rules of equational logic are the following five rules.

(1) Reflexitivity	$\vdash \xi == \xi$
(2) Symmetry	$\xi == \eta \vdash \eta == \xi$
(3) Transitivity	$\xi == \eta,\ \eta == \zeta \vdash \xi == \zeta$
(4) Substitution	$\xi == \eta \vdash \xi[\zeta/v] == \eta[\zeta/v]$
(5) Replacement	$\xi == \eta \vdash \zeta[\xi/v] == \zeta[\eta/v]$

where ξ ,η and ζ are $\Sigma(X)$-terms and $\xi[\zeta_1/v_1, \ldots, \zeta_n/v_n]$ 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 E of Σ-axioms, if an equation $\xi == \eta$ is obtained from E by a finite number of applications of above inference rules then we write $E \vdash \xi == \eta$ and say that $\xi == \eta$ is deducible from E.

Using the concept of this deducibility, we define the congruence relation† $\equiv$ over the term algebra $T[\Sigma]$ by: For any terms t and t' in $T[\Sigma]$, $t \equiv t'$ iff $E \vdash$

† For an S-sorted signature Σ, a Σ-congruence $\equiv$ on a Σ-algebra A is an equivalence relation $\equiv$ such that if $a_i \equiv b_i$ and $sort(a_i) = sort(b_i) = s_i$ for $i=1,\ldots,n$, and $f \in \Sigma$ is with arity $s_1 \ldots s_n$ then $f_A(a_1,\ldots,a_n) \equiv f_A(b_1,\ldots,b_n)$. For $a \in A$ let $[a]$ denote the $\equiv$-equivalence class of a, that is, $[a] = \{b \in A \mid a \equiv b\}$.

t==t'.

The quotient algebra†† of the term algebra $T[\Sigma]$ by congruence relation $\equiv$ is the initial algebra of the class $Alg_{\Sigma,E}$ of Σ-algebras which satisfies the set E of Σ-axioms. That is, for any algebra $A \in Alg_{\Sigma,E}$, there exists the unique homomorphism from $T[\Sigma]/\equiv$ to A. The initial algebra is unique up to isomorphism and can be used to define the meaning of abstract data types which is specified by the set of axioms E.

From the above observation, we give the following definition.

[Def. 5.1] (Semantic domain)

The specification of a semantic domain is a quadruple $D = <S, \Sigma, X, E>$, where S is a set of sorts, Σ is an S-sorted signature, X is an S-sorted set of variables, and E 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 now ready to define the semantic mapping from the syntactic domain to the semantic domain. Here, we use a primitive recursive scheme to specify a semantic mapping.

[Def. 6.1] (Specification of Semantic Mapping)

Let $G = <V, V_T, P, S_0>$ be a specification of a syntactic domain and $D = <S, \Sigma, X, E>$ be a specification of a semantic domain. The specification of of a semantic mapping is a quadruple $\Gamma = <d, M, Y, R>$ where d is a function $d: V \to S$ which associates each nonterminal symbol with a sort of the semantic domain, M is a set of function variables M_N with $arity(M_N)=N (\in V)$ and $sort(M_N)=d(N) (\in S)$, Y is a V-sorted set of variables, and R is a set of semantic equations $\{R_p \mid p:N\to\alpha \in P\}$. For each production p with arity N1...Nn and sort N, the semantic equation R_p is given in the form

$$M_N(\ p(y_1,\dots,y_n)\ ) = \xi[\ M_{N1}(\ y_1\ )/x_1,\ \dots\ ,\ M_{Nn}(\ y_n\ )/x_n\ ]$$

where y_i is a variable with sort Ni, x_i is a variable with sort d(Ni) on the semantic domain for each i=1,...,n, and ξ is a $\Sigma(\{y_1,\dots,y_n\})$-term.

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

The semantic mapping determined by the specification Γ is defined to be the

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

(1) $|A/\equiv| = \{[a] \mid a \in |A|\}$

(2) If $f \in \Sigma$ is with arity $s_1 \dots s_n$, and sort s,

then $f_{A/\equiv}([a_1],\dots,[a_n]) = [f(a_1,\dots,a_n)]$ for all $[a_i]\ |A/\equiv|_{s_i}$ $(i=1,\dots,n)$.

solution of the set of semantic equations R. Let Γ = <D, M, Y, R> be a specification of a semantic mapping and A be a Σ-algebra. The solution of the set of 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 (with arity(p)=N1...Nn and sort(p)=N) and any $t_i \in T[G]$ with sort Ni for i=1,...,n,

$$M_N(\ p(t_1, \dots t_n)\ ) = \xi_A(\ M^A_{N1}(\ t_1\ ),\ \dots\ , M^A_{Nn}(\ t_n\ )\ )\ .$$

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.

[Proposition 6.1] Let A be Σ-algebra. A set of semantic equations R given in the form of Def. 6.1 has the unique solution over A.

Now we can define a semantic mapping.

[Def. 6.2] (Semantic Mapping) Let Γ = <D, M, Y, R> be the specification of a semantic mapping and SD(D) be a semantic domain. The semantic mapping sem(Γ) is the solution of the set of semantic equations R over the semantic domain SD(D).

The next corollary is immediately obtained from Proposition 6.1.

[Corollary 6.2] For a specification Γ of semantic mapping, we can uniquely determine the semantic mapping sem(Γ)

7. Example

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

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

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

$$\xi_A(a) = \begin{cases} \xi_A & \text{if } \xi = f \in \Sigma,\ arity(f) = \varepsilon \text{ and } sort(f) = s \\ a_i & \text{if } \xi = y_i \\ f_A(\zeta_{1_A}(a), \dots, \zeta_{m_A}(a)) & \\ \quad \text{if } \xi = f(\zeta_1, \dots, \zeta_m),\ f \in \Sigma,\ arity(f) = s_1 \dots s_m,\ sort(f) = s, \text{ and} & \\ \quad \zeta_i \in T[(\{y_1, \dots, y_n\})]_{s_i} \text{ for } i = 1, \dots, m & \end{cases}$$

By using our method, the specification of programming language PL/0 is given as follows.

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

G = < V, V_T, P, S_0 >

 V   = { PROGRAM BLOCK CONST_DEF_PART CONST_DEF VAR_DCL_PART VAR_NAME
         PROC_DCL_PART PROC_DCL STATEMENT STATEMENT_LIST CONDITION
         EXPRESSION IDENT ... }

 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 ;
         p070 : CONST_DEF        ->  IDENT = NUMBER
         p080 : VAR_DCL_PART     ->  var VAR_NAME_LIST ;
         p120 : VAR_NAME         ->  IDENT
         p130 : PROC_DCL_PART    ->  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
                                  :
                                  :                                }

  S_0 = PROGRAM

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

D = <S, Σ, X, E>

  S = { STATE               (* tree and stack for dynamic link *)
        STATE-STATE         (* function from STATE to STATE *)
        STATE-STATE-STATE   (* function from STATE to STATE-STATE *)
        POS      (* pointer denoting current scope *)
        NODE     (* node that keep informations for procedures
                    such as local symbol table *)
        TREE     (* tree keeping the static scope for identifiers *)
        ID       (* identifier *)
        TAB      (* symboltable *)
          :
          :                                                          }

  Σ = { INIT_STATE      : -> STATE
        EMPTY_TREE      : -> TREE
        EMPTY_POS       : -> POS
        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
        APPLY_STATE_D   : STATE-STATE-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        : ID ATTR -> STATE-STATE
        ENTER_BLOCK_D, LEAVE_BLOCK_D : -> STATE-STATE
        UPDATE_D         : ID STATE-ATTR -> STATE-STATE
        GET_TAB          : TREE POS -> TAB
        PUT_TAB          : TREE POS TAB -> TREE
        RETRIEVE_TAB     : TAB ID -> ATTR
        UPDATE_TAB, ADD_ID_TAB : TAB ID ATTR -> TAB
            :
            :                                                     }

  X = { s0, s1, s2 | s∈S}

  E = { INIT_STATE()  MAKE_STATE( EMPTY_TREE(), EMPTY_POS() )

      UPDATE( MAKE_STATE(tree0,pos0), id0, attr0 )
       == IF_STATE( IS_IN_CURRENT_NODE(tree0,pos0,id0),
                   MAKE_STATE(
                      PUT_TAB( tree0, pos0,
                               UPDATE_TAB( GET_TAB(tree0,pos0),
                                           id0, attr0 )),
                      pos0),
                   UPDATE( MAKE_STATE(tree0,FATHER(pos0)), id0 ) )
      UPDATE_TAB( ADD_ID_TAB(tab0,id0,attr0), id1, attr1 )
       == IF_TAB( EQUAL_ID(id0,id1),
                 ADD_ID_TAB(tab0,id1,attr1),
                 UPDATE_TAB(tab0,id1,attr1) )

      ADD_ID( MAKE_STATE(tree0,pos0), id0, attr0 )
       == ADD_ID_TAB( GET_TAB(tree0,pos0), id0, attr0 )

      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(id0,attr0), state0 )
       == ADD_ID( state0, id0, attr0 )
                 :
                 :                                                    }


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

Γ = < d, M, Y, R >

  d : V -> S
    d(PROGRAM) = STATE
    d(BLOCK) = d(CONST_DEF_PART) = ... = d(STATEMENT) = STATE-STATE
    d(CONDITION) = STATE-BOOL
    d(EXPRESSION) = STATE-INT
          :
          :

  M = { M_N | N-V, sort(M_N)=d(N), arity(M_N)=N }
```

$Y = \cup_{N\in V} Y_N$

Y_{BLOCK} = { blk0 } $Y_{CONST\_DEF\_PART}$ = { c_d_p0 }

$Y_{STATEMENT}$ = { stm0 } $Y_{CONDITION}$ = { cnd0 }

```
          :
          :
```

```
R = { (* p010 : PROGRAM  ->  BLOCK . *)
      M_PROGRAM( p010(blk0) )
       = APPLY_STATE( M_BLOCK( blk0 ), INIT_STATE() )

      (* p020 : BLOCK  ->  CONST_DEF_PART  VAR_DCL_PART
                           PROC_DCL_PART  STATEMENT      *)
      M_BLOCK( p020(c_d_p0,v_d_p0,p_d_p0,stm0) )
       = COMPOSITION(
            M_STATEMENT( stm0 ),
            COMPOSITION(
               M_PROC_DCL_PART( p_d_p0 ),
               COMPOSITION(
                  M_VAR_DCL_PART( v_d_p0 ),
                  M_CONST_DEF_PART( c_d_p0 ) )))

      (* p070 : CONST_DEF ->  IDENT = NUMBER *)
      M_CONST_DEF( p070(id0,num0) )
       = ADD_ID_D( M_IDENT( id0 ),
                   MAKE_ATTR_CONST( M_NUMBER( num0 ) ) )

      (* p120 : VAR_NAME ->  IDENT *)
      M_VAR_NAME( p120(id0) )
       = ADD_ID_D( M_IDENT( id0 ), MAKE_ATTR_VAR( ZERO() ) )

      (* p170 : PROC_DCL ->  procedure IDENT ; BLOCK *)
      M_PROC_DCL( p170(id0,blk0) )
       = ADD_PROC_ID_D( M_IDENT( id0 ),
                        MAKE_ATTR_PROC( M_BLOCK( blk0 ) ) )

      (* p180 : STATEMENT ->  IDENT := EXPRESSION *)
      M_STATEMENT( p180(id0,exp0) )
       = UPDATE_D( M_IDENT( id0 ),
                   MAKE_ATTR_VAR( M_EXPRESSION( exp0 ) ) )

      (* p190 : STATEMENT ->  call IDENT *)
      M_STATEMENT( p190(id0) )
       = COMPOSITION(
            LEAVE_BLOCK_D(),
            COMPOSITION(
               APPLY_STATE_D(
                  MAKE_STATE-STATE_D(
                     RETRIEVE_PROC_D( M_IDENT( id0 ) ) ),
               ENTER_BLOCK_D() ) )

      (* p210 : STATEMENT ->  if CONDITION then STATEMENT *)
      M_STATEMENT( p210(cnd0,stm0) )
       = IF_STATE_D( M_CONDITION( cnd0 ), M_STATEMENT( stm0 ),
                     I_STATE-STATE() )

      (* p220 : STATEMENT ->  while CONDITION do STATEMENT *)
      M_STATEMENT( p220(cnd0,stm0) )
       = ITERATE( M_CONDITION( cnd0 ), M_STATEMENT( stm0 ) )
                     :
                     :                                          }
```

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 the computation mechanism. And we consider that the meanings 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, an assignment statement changes the state through renewing the value of a variable, and a variable declaration also changes the state by entering a new variable into the name table.

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

8. Conclusion

In this paper, we have proposed a purely algebraic method for specification of programming languages. The semantic domain of our specification is specified as an abstract data type by using only equations. This gives us the mathematical foundations of our algebraic approach for the formal specification, implementation, verification of programs, the formal proof of compiler correctness, and the automatic compiler generation.

As an illustrative example, we have also given the specification of the programming language PL/0. 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 there are many problems to be solved.

Acknowledgement

The authors wish to express their gratitude to Dr.Namio HONDA, President of Toyohashi University of Technology, Dr. Teruo FUKUMURA, Professor of Nagoya University and Dr. Nariyasu MINAMIDE, Professor of Mie 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,J.W.,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(Thatcher,J.W.,Wagner,E.G.,Wright,J.B.): More on Advice on Structuring Compilers and Their Correctness, Theor. Comput. Sci., Vol.15, pp.223-249(1981).

4) Courcelle,B.,Franchi-Zannettacci,P.: Attribute Grammars and Primitive Recursive Schemes, Theor. Comput. Sci., Vol.17, pp.163-191, pp.235-257(1982).

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) 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).

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

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