M. Karpinski, editor, Proc. FCT'77, LNCS 56, pages 84–97, Berlin, 1977. Springer–Verlag.

ALGEBRAIC SEMANTICS OF TYPE DEFINITIONS AND STRUCTURED VARIABLES

H.-D. Ehrich

Abteilung Informatik, Universität Dortmund
Federal Republic of Germany

Abstract - The semantics of type definitions, declarations of structured variables, assignment and evaluation is specified algebraically by means of abstract data types. Corresponding proof rules are given which can be used for program verification. Then, a unifying approach to the semantics of type definitions is presented by giving an axiom system for a general algebra of structured objects in which type definitions are represented by equations. The structure of models for the axiom system and the solvability of these equations is discussed.

1. Introduction

Modern high level programming languages like SIMULA, ALGOL 68, PASCAL, etc. provide quite a variety of concepts to define and handle complex data objects. There are certain built-in elementary types like bool, int, etc. along with structuring mechanisms like arrays, records, pointers, etc. Structured types are used to declare structured variables, having components capable of holding values of corresponding type.

In sections 2 and 3, we specify the semantics of type definitions including recursive types (cf./9/), and that of declaration of structured variables, assignment and evaluation. As specification method, we use algebraic specification by socalled "abstract data types" (cf. /5,6,7,14/ for an introduction into the subject). We will use notation somewhat loosely, e.g. use operations of auxiliary data types without explicit specifying them, use error conditions in a naive way (cf. /6/ for the problems related to errors), and use obvious abbreviations if the details would be too tiresome to be illustrative. We hope that this will not obscure the ideas but rather improve clarity.

From the algebraic semantic specifications, we get proof rules for verifying programs with structured types and variables. The standard exclusion of these language features from the verification literature suggests that they are difficult to handle. Rather surprisingly, section 4 shows that the rules are rather simple. What is lengthy and complicated in most cases is the deduction process using these rules. Different approaches to the verification of data structures are taken in /8,12,15/, while problems similar to ours are treated in /1,11/.

Sections 5 to 7 present a unifying approach to the semantics of type definitions different from that in section 2, following the lines of /3,4,1o/. In section 5, an algebra of structured objects is developed systematically, using three basic binary operations called construction, addition and selection. Insofar, our approach differs from that in /13/. Based on a representation theorem which is essential for the theory, we investigate in section 6 models of the axiom system, making use of a closure property called orthogonality in /2/. It turns out that there is (up to isomorphism) one "full" orthogonal model containing all others as subalgebras. In this algebra, the equations corresponding to type definitions are uniquely solvable.

2. Type definitions

When we disregard syntactical peculiarities and consider the matter from a more abstract viewpoint, we realize that the most basic principle of structuring data in programming languages is that of tupling, i.e. creating compound types out of a finite list of components. Each component constitutes an elementary or previously defined type and is accessible via a selector. Consider, for example, the ALGOL-like type definition

(2.1)   type list = struct (elem : int , next : ref list) .

Here, list is a structured type with two components. The first component type is int, accessible via the selector elem, and the second component type is ref list, accessible via the selector next. We take the position that ref list is an elementary type which has no components. Thus, with each structured type X, we associate an elementary type ref X whose values are understood to refer to objects of type X (cf. section 3).

There will be no difficulty to include recursive type definitions like

(2.2)   type reclist = struct (elem : int , next : reclist) .

This generalizes concepts commonly available in programming languages, and its high usefulness has been advocated by Hoare /9/ . In 2.2 , the second component type is of course not elementary, so the meaning of 2.2 is very different from that of 2.1 .

There is one problem about arrays. If we write array declarations like

(2.3)   int array A[1:1o] ,

we do two things at the same time: we define a structured type

(2.3.1) $\underline{type}\ \underline{a} = \underline{int}\ \underline{array}[1:10]$

with ten components, selected by selectors 1,2,...,1o , each component having type $\underline{int}$, and simultanously we declare a variable with type $\underline{a}$ and name A :

(2.3.2) $\underline{var}\ A : \underline{a}$

We assume in the sequel that array declarations are split explicitly in this way.

This allows us to introduce a uniform notation for type definitions: a structured type with n components results from applying a constructor function $\underline{con}_n$ to n arguments which are pairs of selectors and types. So, a type definition will be written in the form

(2.4) $m = \underline{con}_n(s_1:\tau_1 , \ldots , s_n:\tau_n)$.

For the sake of simplicity, we consider programs without block structure having all its type definitions at the beginning of the program text. With each such program, we associate the following sets:

$T = \{t_1,\ldots,t_p\}$ is the set of elementary types occurring in the program (including all $\underline{ref}\ m_i$); there is always a special elementary type called $\underline{nil}$.

$M = \{m_1,\ldots,m_q\}$ is the set of user provided type names occurring in the program.

$S = \{s_1,\ldots,s_r\}$ is the set of user provided selectors occurring in the program.

$m_i = \underline{con}_{n_i}(s^i_1:\tau^i_1 , \ldots , s^i_{n_i}:\tau^i_{n_i})$, $i=1,\ldots,q$, are the type definitions occurring in the program.

As usual, by S^* we denote the set of finite selector sequences, and the empty sequence is denoted by 1 . The intended meaning of type definitions can be easily expressed by an abstract data type associated with the program.

(2.5) Definition: The type environment of a program is given by the following abstract data type :

constants: $\underline{nil}, t_j, m_i \longrightarrow \mathcal{T}$ $\qquad i=1,\ldots,q$

operations: $\underline{con}_{n_i} : (S \times \mathcal{T})^{n_i} \longrightarrow \mathcal{T}$ $\qquad j=1,\ldots,p$

$\underline{sel} : \mathcal{T} \times S^* \longrightarrow T$

---

axioms: $\underline{sel}(t_j,1) = t_j$

$$\underline{sel}(t_j, sx) = \underline{nil} \qquad s \in S \ , \ x \in S^*$$

$$\underline{sel}(\ \underline{con}_{n_i}(s_1^i : \tau_1^i \ , \ \ldots \ , \ s_{n_i}^i : \tau_{n_i}^i\ ) \ , \ 1 \ ) = \underline{nil}$$

$$\underline{sel}(\ \underline{con}_{n_i}(s_1^i : \tau_1^i \ , \ \ldots \ , \ s_{n_i}^i : \tau_{n_i}^i\ ) \ , \ sx \ )$$

$$= \underline{if}\ k = (\mu h)[s = s_h^i]\ \text{exists}\ \underline{then}\ \ \underline{sel}(\tau_k^i, x)\ \ \underline{else}\ \ \underline{nil}$$

$$m_i = \underline{con}_{n_i}(s_1^i : \tau_1^i \ , \ \ldots \ , \ s_{n_i}^i : \tau_{n_i}^i) \qquad ///$$

In addition to the operations mentioned before, we have the operation sel which selects the elementary type "at the bottom" by applying a selector sequence, if it exists, or nil otherwise. For example, taking reclist from 2.2, we have sel(reclist, next.next.elem)=int , sel(reclist, next.next)=nil , sel(reclist, elem.next)=nil . Note that sel(list, next.next.elem)=nil for the type list taken from 2.1 .

For a given type τ from $\mathcal{T}$, denoted by t_j or m_i , let

$$\chi(\tau) = \{\ x \in S^* \quad | \quad \underline{sel}(\tau, x\ ) \neq \underline{nil}\ \}$$

be the characteristic set of τ . It is not surprising that the following theorem holds.

(2.6)Theorem: Let $\mathcal{T}$ be any type environment associated with a program, and let τ be any type in $\mathcal{T}$. Then, $\chi(\tau)$ is a regular set.

The proof is straightforward: construct a finite-state acceptor A accepting $\chi(\tau)$ by letting $T \cup M$ be the set of states, $T - \{\underline{nil}\}$ be the set of final states, τ be the initial state, S be the set of input symbols, and δ be the transition function such that $\delta(m_i, s_k^i) = \tau_k^i$ if $k=1,\ldots,n_i$ and $\delta(m_i,s) = \delta(t_j,s) = \underline{nil}$ otherwise. Then it is easy to see that A accepts $\chi(\tau)$. Moreover, for nonrecursive data types, the characteristic set is finite.

3. Structured variables

A variable is declared by giving it a name and a type. So let $\mathcal{T}$ be the type environment of a program, with sets T and S of elementary types resp. selectors, and let N be any set. The elements of N will be called names.

(3.1)Definition: The set of structured variables is $\mathcal{V} = N \times \mathcal{T}$. The projection functions will be called name resp. type. Furthermore, let $\underline{ctp}: \mathcal{V} \times S^* \longrightarrow T$ be the "component type" operation defined by $\underline{ctp} = \underline{sel} \circ (\underline{type} \times \underline{id}_{S^*})$.

Simple variables will be considered as special cases of structured variables where ctp(v,x) takes the value nil for $x \neq 1$. We give some examples.

(3.2) Examples:

1. Simple variable v = var n : int
   $ctp(v,x)$ = if $x=1$ then int else nil
2. Array variable $\mathcal{A}$ = var A : int array [1:1oo]
   $ctp(\mathcal{A},x)$ = if $x\in[1:1oo]$ then int else nil
3. Structured variable $\mathcal{L}$ = var L : list (cf. 2.1)
   $ctp(\mathcal{L},x)$ = if x=elem then int else
   if x=next then ref list else nil
4. Structured variable $\mathcal{L}_r$ = var LR : reclist (cf. 2.2)
   $ctp(\mathcal{L}_r,x)$ = if $x \in next^*elem$ then int else nil

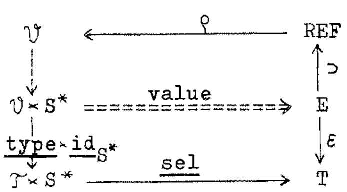
Variables have the ability to take values of appropriate type. Let E be a set of elementary values, and let $\varepsilon : E \to T$ be a function associating elementary types with elementary values. We will assume that ε^{-1}(nil) is empty, whereas $\varepsilon^{-1}(t)$ is nonempty if $t \neq$ nil. Instead of $\varepsilon(e)=t$ we will use the notation $e\, \varepsilon\, t$.

If t is an elementary type of the form t=ref m for a user defined type m, its values have a special meaning: they are "reference values" or "pointers". They refer or point to structured variables of type m. This is reflected by an injective function

$$\rho : REF \longrightarrow \mathcal{V} \quad , \text{ where } REF = \{ e \in E \mid e\ \varepsilon \text{ ref } m \text{ for some } m \} \quad ,$$

with the property: $e\ \varepsilon$ ref m $\Longrightarrow$ type$(\rho(e))=m$.

The domains and functions defined so far are visualized in the following diagram. The value function gives the value of a variable component assigned to it previously by the assignment operation := . These operations, together with variable declaration, are specified subsequently. First let us see what the diagram shows. The lower part is easily interpreted: its commutativity gives the common condition of type compatibility. The upper part shows that, if e - the value of v at x - is a reference value, we can follow this reference by ρ, arriving at some other variable v'. The intended meaning is that, from this variable v', we will go on selecting, following some selector sequence from left to right, jumping to another variable by looping through the diagram each time a reference value is encountered. This is made precise by the specification of the lrf operation below (lrf is shortcut for "last referenced").



The semantics of declaration, assignment and evaluation operating on structured variables is specified by means of an abstract data type Σ called program states:

σ_0	$: \quad \longrightarrow \Sigma$	initial state
$\lambda\, v,\sigma\, [\underline{\text{var}}\ v \mid \sigma]$	$: \Sigma \times \mathcal{V} \longrightarrow \Sigma$	declaration
$\lambda\, v,x,e,\sigma\, [v[x]:=e \mid \sigma]$	$: \Sigma \times \mathcal{V} \times S^* \times E \longrightarrow \Sigma$	assignment
$\lambda\, v,x,\sigma\, [\bar{v}[x] \mid \sigma]$	$: \Sigma \times \mathcal{V} \times S^* \longrightarrow E$	evaluation

There are three axioms for the evaluation function:

(3.3) $\bar{v}[x] \mid \sigma_0 \quad = \underline{\text{error}}$

$\bar{v}[x] \mid (\underline{\text{var}}\ w \mid \sigma) \quad = \underline{\text{if}}\ v=w\ \underline{\text{then}}\ \underline{\text{undef}}\ \underline{\text{else}}\ \bar{v}[x] \mid \sigma$

$\bar{v}[x] \mid (w[y]:=e \mid \sigma) \quad = \underline{\text{if}}\ a=b\ \underline{\text{then}}\ (\underline{\text{if}}\ e\in \underline{\text{ctp}}(a)\ \underline{\text{then}}\ e\ \underline{\text{else}}\ \underline{\text{error}})$
$\underline{\text{else}}\ \bar{v}[x] \mid \sigma$

where $a=\text{lrf}(\sigma,v,x)$ and $b=\text{lrf}(\sigma,w,y)$

The auxiliary operation lrf : $\Sigma \times \mathcal{V} \times S^* \longrightarrow \mathcal{V} \times S^*$ makes use of another auxiliary operation trace: $\Sigma \times \mathcal{V} \times S^* \times S^* \longrightarrow \mathcal{V} \times S^*$ describing in detail the tracing of selector sequences explained informally before. Here are the axioms:

(3.4) $\text{lrf}(\sigma,v,x) \quad = \text{trace}(\sigma,v,1,x)$

$\text{trace}(\sigma,v,y,1) = (v,y)$

$\text{trace}(\sigma,v,y,sx) = \underline{\text{if}}\ e\in\text{REF}\ \underline{\text{then}}\ \text{lrf}(\sigma,\rho(e),sx)\ \underline{\text{else}}\ \text{trace}(\sigma,v,ys,x)$

where $e=\bar{v}[y] \mid \sigma$

If $\text{lrf}(\sigma,v,x)=(w,y)$, then w is the last referenced variable, and y is the rest selector to be applied in w. Thus, $\bar{v}[x]\mid\sigma = \bar{w}[y]\mid\sigma$, and $\bar{w}[z]\mid\sigma \notin \text{REF}$ for all prefixes z of y. For later use, we state the following lemma. Its proof is immediate from the <u>then</u> part of the third axiom in 3.4 .

(3.5) <u>Lemma</u>: If $\text{lrf}(\sigma,v,x)=(w,y)$ and $\bar{v}[z]\mid\sigma \in \text{REF}$ for some proper prefix z of x, then $y \neq 1$.

Besides declaration, assignment and evaluation, there is one more important operation on structured variables, namely creation, commonly denoted by <u>new</u> :

$$\lambda\ \sigma,\tau\ [\underline{\text{new}}(\tau) \mid \sigma] : \Sigma \times \mathcal{T} \longrightarrow \text{REF} \times \Sigma$$

Given a type τ , <u>new</u>(τ) yields a reference to a newly created variable of type τ , with the side effect on the program state that afterwards this new variable is "there". Since it is only accessible via the reference, we take the approach that it has been "automatically" declared with a name different from all names known in the current state (and hidden to the user). Let

$$\text{nn} : \Sigma \longrightarrow N$$

be an auxiliary operation giving us such names. Then, we have only one axiom for <u>new</u>:

(3.6) $\underline{\text{new}}(\tau) \mid \sigma = (\rho^{-1}(\hat{v})\ ,\ \underline{\text{var}}\ \hat{v} \mid \sigma) \qquad \text{where } \hat{v} = (\text{nn}(\sigma),\tau)$

In order to specify the newness of $nn(\sigma)$ in σ, we use the auxiliary operation $dcl? : \Sigma \times \mathcal{V} \longrightarrow \underline{bool}$ telling us whether there is another variable with the same name as v in σ :

$$\text{(3.7)} \quad \begin{aligned} &dcl?(\epsilon_0, v) &&= \underline{false} \\ &dcl?(\underline{var}\ w \,|\sigma, v) &&= \underline{if}\ \underline{name}(w)=\underline{name}(v)\ \underline{then}\ \underline{true}\ \underline{else}\ dcl?(\sigma, v) \\ &dcl?(w[x]:=e \,|\sigma, v) &&= dcl?(\sigma, v) \end{aligned}$$

Now we can do with one final axiom expressing the newness of $nn(\sigma)$ in σ :

$$\text{(3.8)} \quad dcl?(\sigma, nn(\sigma)) = \underline{false}$$

We did not require that S and E are disjoint. In fact, the semantic equations given above work, too, if selectors are given by expressions involving values of variable components. This happens, for example, with integer arrays as in a[a[1]] . This case has been considered by deBakker /1/ .

4. Verification

The semantics given in the previous sections can be used to prove properties of programs. The idea is similar to that in /11/. The assertion language will probably contain terms of the form v[x], referring to the value of the corresponding variable component. We modify the assertion language by incorporating state variables and the operations on $\mathcal{V}$ resp. Σ explicitly, and transform each v[x] into the expression $\bar{v}[x]|\sigma$. Then, assertions can be reduced by applying the axioms . The following backward proof rules are straightforward.

$$\text{(4.1)} \quad P^{\sigma}_{\underline{var}(n,\tau)|\sigma} \ \{\ \underline{var}\ n : \tau\ \}\ P$$

$$\text{(4.2)} \quad P^{\sigma}_{v[x]:=e\,|\sigma} \ \{\ v[x]:=e\ \}\ P \qquad \text{if } e \neq \underline{new}(\tau)$$

$$\text{(4.3)} \quad P^{\sigma}_{v[x]:=\rho^{-1}(nn(\sigma),\tau)|\,\underline{var}(nn(\sigma),\tau)\,|\sigma} \ \{\ v[x]:=\underline{new}(\tau)\ \}\ P$$

The classical case of simple variables can be handled without explicit introduction of the state variable σ since $P^{\sigma}_{v[1]:=e\,|\sigma}$ is equivalent to P^{v}_{e} , if v ambiguously denotes the value of v in the latter notation, expressed by $\bar{v}[1]|\sigma$ in our framework. This equivalence follows from lemma 3.5: values of variable components v[1] belonging to the empty word can be accessed in exactly one way, namely by $\bar{v}[1]|\sigma$. There is no variable $w \neq v$ and no selector sequence $y \in S^*$ such that $(v,1)=lrf(\sigma,w,y)$. This means that there are no "side effects" of assignment.

In certain cases of side effects is it possible to find proof rules without using state variables explicitly. DeBakker /1/ has worked out the case of integer arrays. We do not persue this topic here. Instead, we give a simple example with a typical side effect in order to show how our method works.

(4.4) Example:

```
1   begin
2   type list = struct ( i:int , r: ref list );
3   var x,y : ref list ;
4   x := new ( list ) ;
5   y := x ;
6   x[i] := 2
7   end
```

We want to prove that y[i]=2 at the end of the program, so let P_6 be the assertion $\bar{y}[i]|\sigma = 2$. By applying proof rule 4.2, we get P_5 -the assertion to hold after line 5 of the program- by substituting $v_x[i]:=2|\sigma$ for σ in P_6, where v_x is the simple variable with name x declared in line 3. In the same way, we get P_4 by substituting $v_y[1]:=(\bar{v}_x[1]|\sigma)|\sigma$ for σ in P_5, P_3 by substituting $v_x[1]:=\rho^{-1}(\hat{v})|\underline{\text{var}}\ \hat{v}|\sigma$, where $\hat{v}=(nn(\sigma),\underline{\text{list}})$, for σ in P_4, and P_2 by substituting $\underline{\text{var}}\ v_y|\underline{\text{var}}\ v_x|\sigma$ for σ in P_3, where $v_y=(y,\underline{\text{ref}}\ \underline{\text{list}})$ and $v_x=(x,\underline{\text{ref}}\ \underline{\text{list}})$. The type definition in line 2 has no effect on the state, so we are at the beginning of our program. Let P_1 result from substituting σ_o for σ in P_2. P_1 can now be verified by the axioms in section 3, but it is rather lengthy to do so. We leave the details to the reader.

5. Structured objects - a unifying axiomatic approach

While type environments as defined in section 2 give a direct and immediately intuitive approach to the understanding of user defined types, we feel that they have one serious drawback: the associated algebras differ from program to program, and in programs with block structure they even change dynamically while the program runs. So it is not an easy task to compare different type environments and study their relationships. The matter is much nicer with program states where we have one fixed algebra working for all programs.

The question we want to persue here is whether there is one fixed "algebra of structured objects" in which all type environments can be represented. The idea is to break down the rather complex $\underline{\text{con}}_n$ operations into more basic ones such that each object $\underline{\text{con}}_n(s_1:\tau_1,\dots,s_n:\tau_n)$ is represented by a composite expression in terms of the basic operations. A type definition of the form $m=\underline{\text{con}}_n(\dots)$ would then determine an equation whose solution defines the type. But this raises the problem whether there is a solution, whether the solution is unique, if it exists, etc. The basic idea of our approach is to let

$$\underline{\text{con}}_n(s_1:\tau_1,s_2:\tau_2,\ \dots\ ,s_n:\tau_n)$$

be represented by

$$s_1:\tau_1 + s_2:\tau_2 + \dots + s_n:\tau_n \quad ,$$

where : and + are binary operations, and + is associative. In order to simplify notation, we will assume that + is commutative, too, although this is not essential. Furthermore, we switch to a more "algebraic" notation by writing $\tau.x$ instead of sel(τ,x) and 0 instead of nil , and we speak more generally about "objects" instead of "types".

Let D be a set of "structured objects", and 0 be a distinguished element of D. Let $S=\{s_1,\dots,s_n\}$ be a finite set of "selectors". We want to consider the following operations:

$$\lambda x,a\,[x:a] : S^* \times D \longrightarrow D \quad \text{, named "construction" ,}$$
$$\lambda a,b\,[a+b] : D \times D \longrightarrow D \quad \text{, named "addition" ,}$$
$$\lambda a,x\,[a.x] : D \times S^* \longrightarrow D \quad \text{, named "selection" .}$$

An algebra based on these operations has been developed in /4/. We present a slightly simplified version of the axiom system given there. For notational convenience, we adopt the convention that $a,b,c,\ldots \in D$, $s,t,\ldots \in S$, $x,y,z,\ldots \in S^*$, and 1 again denotes the empty word in S^* . Specifications "$\in D$" etc. will be omitted whenever possible. If quantification is omitted in the formulas to follow, universal quantification is understood for each free variable. Index i always runs over an index set $I \subset \mathbb{N}$.

Axioms: 1. (D,+,0) is a commutative monoid

2. $a.1 = a$	5. $1:a = a$
3. $a.xy = (a.x).y$	6. $xy:a = x:(y:a)$
4. $(\sum a_i).x = \sum (a_i.x)$	7. $x:\sum a_i = \sum x:a_i$

8. $(s:a).t = \begin{cases} a & \text{if } s=t \\ 0 & \text{otherwise} \end{cases}$

9. $\forall a \ \forall s \ \exists b \ : \ a = s:(a.s) + b \ \wedge \ b.s = 0$

There will be one more axiom giving a criterion for equality of objects. Before we can formulate it, we need some preparation.

The essential behaviour specified in definition 2.5 can already be deduced from these axioms: if $1 \le i \le n$, we have from axioms 1,3,4,8 :

$$(s_1:a_1 + s_2:a_2 + \ldots + s_n:a_n) \,.\, s_i x \;=\; a_i.x$$

(5.1)Definition: An object $e \in D$ is called elementary iff, for each $x \neq 1$, we have $e.x=0$.

Let $E=\{ e \in D \mid e \text{ is elementary} \}$ be the set of all elementary objects. As shown in /4/, we can prove the following theorem by repeated application of axiom 9.

(5.2)Theorem: $\forall a \in D \ \ \exists e \in E \ : \ a = \sum_{s \in S} s:(a.s) + e$

Our tenth axiom, the axiom of equality, is motivated by the desire to exclude certain pathologic behaviour of some infinite objects, where an object a is called <u>infinite</u> iff the set $\{x \mid a.x \neq 0\}$ is infinite. Otherwise, a is called <u>finite</u>. Let

$$F(a) = \{ e \in E \mid a = e + \sum_{s \in S} s:(a.s) \} .$$

For finite objects, we can easily deduce from the first nine axioms that $a=b$ iff, for each x, $F(a.x) \cap F(b.x) \neq \emptyset$. We want that this holds for all objects.

<u>Axiom 1o</u>: $(\forall x : F(a.x) \cap F(b.x) \neq \emptyset) \Longrightarrow a=b$

From this axiom, we get a very important representation theorem which is fundamental for the subsequent considerations.

(5.3)<u>Theorem</u>: Let $\{e_x\}_{x \in S^*}$ be a family of elementary objects, indexed by S^*. Then we have

$$a = \sum_{x \in S^*} x:e_x \iff \forall x : e_x \in F(a.x)$$

<u>Proof</u>: The conclusion from left to right is straightforward. To show the converse, let $b = \sum_{x \in S^*} x:e_x$, and $y \in S^*$. By axioms 1,4 and repeated application of axiom 8, we get $b.y = \sum_{x \in S^*} (x:e_x).y = \sum_{z \in S^*} z:e_{yz} = e_y + \sum_{z \neq 1} z:e_{yz}$.
Let $c = \sum_{z \neq 1} z:e_{yz}$. By axioms 1,6,7 we get $c = \sum_{s \in S} \sum_{w \in S^*} s:(w:e_{ysw}) = \sum_{s \in S} s:c_s$, where $c_s = \sum_{w \in S^*} w:e_{ysw}$. Collecting the pieces, we have $b.y = e_y + \sum_{s \in S} s:c_s$.
It is clear that $c_s = (b.y).s$. So, by definition of F, we have $e_y \in F(a.y) \cap F(b.y)$. From axiom 1o, we conclude a=b . ///

6. Models for structured objects

The representation theorem 5.3 allows us to represent each object uniquely by a formal power series with the noncommutative variables $s_1,\dots,s_n$ and with coefficients in 2^E :

$$a = \sum_{x \in S^*} x : F(a.x)$$

This means that each model of our axiom system can be isomorphically represented by a subset $D \subset (2^E)^{S^*}$, given E and S .

We are interested in models having certain closure properties. It is natural to assume a certain independence of the coefficients in the sense that it should be possible to construct new objects out of coefficients from several different objects. To be precise, let

$$\underline{\text{rpl}}(a,x,b) := \sum_{y \neq x} y : F(a.y) + x : F(b.x)$$

be the object obtained from a by substituting its x-coefficient by that of b.

Following /2/ in terminology, we define:

(6.1)Definition: A subset $D \subset (2^E)^{S^*}$ is called orthogonal iff D is closed with respect to the replacement operation rpl .

The important fact about orthogonal models is stated in the following theorem.

(6.2)Theorem: If D is an orthogonal model, all coefficients are singletons.

Proof: Let $a \in D$ and $x \in S^*$. Since $0 \in D$, we must have rpl(0,x,a)=x:F(a.x) $\in D$. That means: if $e,f \in F(a.x)$, we have x:e=x:f , by theorem 5.3 . Since D is closed wrt selection, we have e=(x:e).x=(x:f).x=f . ///

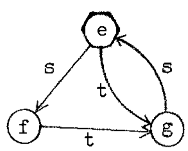
This theorem shows that orthogonal models correspond to subsets $D \subset E^{S^*}$ of formal power series with coefficients in E.

It has been shown in /4/ that a generalization of the notion of orthogonality, called quasi-orthogonality, yields models which correspond, roughly speaking, to quotient structures of orthogonal models. Therefore, we restrict our attention here to orthogonal models, in fact to the "full" orthogonal model

$$\mathbb{D} = E^{S^*}$$

from which all others are subalgebras. The elements of $\mathbb{D}$ can be graphically represented in a nicely intuitive way by directed graphs with a distinguished root node, with node marks from E and edge marks from S . In a graph for $a \in \mathbb{D}$, we obtain the value a(x) as the label of that node we arrive at if we select by x from left to right, starting at the root node. By convention, we omit nodes marked with 0 , if only nodes marked with 0 are accessible from it.

(6.3)Example: Let a be represented by the following graph. Then we have



a(1)=a(ts)=a(ststs)=...=e
a(s)=a(stss)= ... =f
a(t)=a(st)= ... =g
a(ss)=a(tt)= ... =0

This graphical representation is, of course, not unique. By unrolling the loops we get different graphs representing the same object. It is sometimes advantageous to consider the completely unrolled graphs, i.e. (possibly infinite) trees, to be the standard representation of objects.

The three basic operations take the following form in $\mathbb{D}$:

1. construction $[x:a](y) = \begin{cases} a(z) & \text{if } y=xz \\ 0 & \text{otherwise} \end{cases}$
2. addition $[\sum a_i](y) = \sum a_i(y)$
3. selection $[a.x](y) = a(xy)$

This object algebra $\mathbb{D}$ has been investigated in /3/. We concentrate here on aspects relevant for the semantics of type definitions.

7. Type definitions revisited

Due to the motivation for object algebras given in the beginning of section 5, type definitions take the form of equations

$$m = s_1:\tau_1 + s_2:\tau_2 + \dots + s_n:\tau_n$$

in our algebra $\mathbb{D}=E^{S^*}$, where $E=\{\underline{nil}, \underline{bool}, \underline{int}, \dots\}$ is a given set of elementary types, and S is a finite set of selectors.

(7.1)Example:

1. Type definition 2.1 determines the equation

$$\underline{list} = elem : \underline{int} + next : \underline{ref}\ \underline{list}$$

2. Type definition 2.2 determines the equation

$$\underline{reclist} = elem : \underline{int} + next : \underline{reclist}$$

In example 1, the object list is given explicitly by an algebraic expression. In example 2, however, the object reclist is specified implicitly by an equation. The question is: does a solution exist, and if so, is it unique? For so-called "rational systems of equations" to be defined below, which are especially close to type definitions, the answer is positive. By such systems of equations, just the "rational" objects can be characterized:

(7.2)Definition: An object $a \in \mathbb{D}$ is called rational iff the set $\{a.x \mid x \in S^*\}$ is finite.

Rational objects are just those representable by finite graphs with loops. For rational objects, the characteristic set $\chi(a) = \{x \in S^* \mid a(x) \neq 0\}$ is a regular language.

Let $U = \{u_1, u_2, \dots, u_m\}$ be a finite set of "unknowns", and let $V = U \cup E$.

(7.3)Definition: A system of equations is called rational (an RSE) iff it is of the form

$$u_i = \sum_{k=1}^{n} s_k : v_{ik} + e_i \qquad , i=1,\dots,m ,$$

where $u_i \in U$, $s_k \in S$, $v_{ik} \in V$ and $e_i \in E$. The first component a_1 of a solution vector $(a_1,\dots,a_n)$ will be called a solution of the RSE.

From /3/, we take the following result without proof.

(7.4)Theorem: Each RSE has a unique rational solution. Conversely, each rational object is the solution of some RSE .

Thus, RSE's provide a sound basis for specifying the semantics of type definitions. Moreover, there is a method for solving RSE's based on the following lemma, where for $X \subset S^*$, we use the abbreviation

$$X : a \;:=\; \sum_{x \in X} x : a \quad .$$

(7.5) Lemma: Let $R \subset S^*$ be a regular set such that for all $x, y \in R$ we have: if x is a prefix of y then x=y . Let $b \in \mathbb{D}$ be such that $RS^* \cap \chi(b) = \emptyset$. Then the equation

$$u = R : u \; + \; b$$

has the unique solution $a = R^* : b$. This solution is rational iff b is.

The proof is given in /3/. With this lemma, we can in each case give an explicit expression for a user defined type in terms of the elementary types involved.

(7.6) Example:

1. The solution of example 7.1.2 is $\underline{\text{reclist}} = (\text{next})^* \text{elem} : \underline{\text{int}}$
2. Let $m_1 = r{:}m_1 + s{:}m_2 + t{:}\underline{\text{int}}$

   $m_2 = r{:}m_1 + s{:}\underline{\text{bool}} + t{:}m_2$

   The solution is

   $$m_1 = (r \vee st^*r)^* st^*s : \underline{\text{bool}} \;+\; (r \vee st^*r)^* t : \underline{\text{int}}$$

Solving equations in $\mathbb{D}$ can be put on a more general basis by introducing a complete partial ordering on $\mathbb{D}$ such that the algebraic operations become continuous functions. Then, fixpoint theory can be used. We are unable to persue this topic here; the interested reader is referred to /3,1o/.

8. Conclusions

We have given algebraic semantic specifications for some programming language features in connection with user defined types and structured variables. Type definitions are handled by two different approaches, namely type environments and (uniquely solvable) equations in an object algebra.

We feel that there are several areas open for further investigation. The algebraic specification method should be applied to other language features including a variety of control structures, block structures, procedures and so on. The usefulness of this method has to be compared with that of others like operational, denotational and propositional semantics, for all different fields where precise semantical description is essential, e.g. verification and synthesis of programs, automatic or not, compiler construction, and education in programming.

The uniform approach to type definitions presented in sections 5 to 7 makes use of an algebra that is not equationally defined. Referring to the notion of "implementation" of abstract data types as given in /6/, it is an interesting problem whether there is one fixed equationally specified abstract data type that implements all type environments as defined in section 2 .

References

1. deBakker,J.W.: Correctness proofs for assignment statements. Report IW55/76, Mathematisch Centrum, Amsterdam 1976

2. Cremers,A.-Hibbard,Th.N.: The semantic definition of programming languages in terms of their data spaces. Proc.4.Fachtagung der GI über Programmiersprachen. Fachberichte Informatik, Springer-Verlag, berlin 1976 , 1-11

3. Ehrich,H.-D.: Outline of an algebraic theory of structured objects, in: Proc.3rd Int.Coll. on Automata, Languages and Programming, ed. by S.Michaelson and R.Milner, Edinburgh University Press 1976, 5o8-53o

4. Ehrich,H.-D.: An axiomatic approach to information structures, in: Proc. 5th MFOC Symposium, ed. by A.Mazurkiewicz, Lecture Notes in Computer Science 45, Springer-Verlag, Berlin 1976, 277-283

5. Goguen,J.A.: Correctness and equivalence of data types, in Proc. Conf. on Alg. Systems Theory, Lecture Notes in Computer Science, Springer-Verlag, Berlin 1975

6. Goguen,J.A.-Thatcher,J.W.-Wagner,E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, to be published in: Current Trends in Programming Methodology 4, ed. by R. Yeh

7. Goguen,J.A.-Thatcher,J.W.-Wagner,E.G.-Wright,J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM 24(1977),68-95

8. Hoare,C.A.R.: Proof of correctness of data representations. Acta Informatica 1(1972), 271-281

9. Hoare,C.A.R.: Recursive data structures, Stanford Artificial Intelligence Laboratory Memo AIM-223, STAN-CS-73-4oo, 1973

1o. Lohberger,V.: Eine Klasse geordneter Monoide und ihre Anwendbarkeit in der Fixpunktsemantik, in: Proc.3rd GI Conf. on Theoretical Computer Science, ed. by H.Tschach, H.Waldschmidt and H.K.-G.Walter, Lecture Notes in Computer Science 48, Springer-Verlag, Berlin 1977, 169-183

11. Luckham,D.-Suzuki,N.: Automatic program verification V: Verification-oriented proof rules for arrays, records and pointers. Report No. STAN-CS-76-549, Stanford 1976

12. Spitzen,J.-Wegbreit,B.: The verification and synthesis of data structures, Acta Informatica 4(1975), 127-144

13. Standish,Th.A.: Data structures, an axiomatic approach. BBN Report No. 2639, Automatic Programming Memo 3, Bolt Beranek and Newman, Cambridge, Mass. 1973

14. Wand, M.: First-order identities as a defining language. Tech. Report No. 29, Comp.Sc.Dpt. Indiana University, Bloomington 1976

15. Wegbreit,B.-Spitzen,J.M.: Proving properties of complex data structures. Journal of the ACM 23(1976), 389-396