DATA SPECIFICATION METHODS[†]

by

A. Pereda[*], R.L. Carvalho[*], C.J. Lucena[*], T.S.E. Maibaum[§]

Research Report CS-79-30

Computer Science Dept.
University of Waterloo
Waterloo, Ontario, Canada

July 1979

[†]This work was supported by a research grant (to C.J. Lucena) from the J.S. Guggenheim Memorial Foundation and also by the National Sciences and Engineering Research Council of Canada.

[*]Departamento de Informatica
Pontificia Universidade Catolica
Rua Marques de S. Vincente, 225
Gavea-CEP 22453
Rio de Janeiro, R.J. Brazil.

[§]Computer Science Dept.
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

1. Introduction

The field of programming has undergone a fruitful period of systematization in the last ten years. It is now generally accepted that specification is an integral part of programming. The specification of the data over which programs operate is a central problem in programming. At the present, the problem of data specification deserves special attention because it has been recognized that data abstraction is a powerful mechanism to deal with the complexity of large programs [20]. The formalization of data structures and data types can lead both to the establishment of new theories of programming (e.g., [26] and to the development of powerful programming methodologies.

There are many ways of looking at the data aspects of a programming problem under consideration. These different ways of looking at the same "reality" suggest the use of different formal methods to describe the data phenomena under observation. (They also suggest different styles of programming.) A methodological problem arises in connection with this fact. The formal specification methods vary in terms of the techniques they use (e.g. different mathematical tools) and in terms of the amount of detail they want to model, that is, their level of abstraction. The former variation results in what we will call different _views_ of the same data reality. The latter variation gives rise to what we will term the _level of abstraction_ of the description. A certain amount of confusion exists today in the literature when the data aspects of programming methodologies and programming languages are compared. The same thing happens when the "power" of data specification methods is discussed by different authors. These confusions often arise because of the lack of distinction between these two criteria for analyzing data specification methods. These problems served as motivation for the present paper.

The need to provide a systematic classification of the existing specification methods was first realized by Liskov and Zilles [21,22]. In their work some specification methods are classified according to the different ways in which the data universes under description are being observed. In the first version of their paper [21] an attempt was made to classify the methods according to their degrees of abstraction. (This criterion was dropped in the second version [22]). We believe that the present work can contribute further to this effort of systematization. We will resurrect the concept of "level of abstraction" and, together with the concept of "view", we will provide two (orthogonal) measures by which to classify data specification methods. We will show that, according to our classification approach, methods which were said to have different characteristics in [22] below in fact to the same class of methods. Our different classes of methods, when compared uniformly (at the same level of abstraction), describe different but compatible views of the same data objects. This compatibility can be formally proven [28]. Data, in the present work, will be considered a set of symbols with certain common physical and logical attributes that belong to a certain language. A set of (elementary) data plus the operations and relations defined on them constitute a (primitive) data type. A data structure is obtained from a set of objects belonging to a (primitive) data type by the introduction of a relation between them. This relation is usually called the accessibility relation. Different classes of data structures are defined by imposing restrictions on the definition of the accessibility relation. If we define operations and relations over a set of data structures then we have defined a data type. Of course, these data types can in turn be used as domains to get new kinds of data structures, etc. The above definitions are adequate as background information for the present work. A more formalized version can be found in [28].

We start the paper by illustrating through examples the role of levels of data abstraction in formal specification. We then propose a classification for the data specification methods. We emphasize the point that the different and complimentary views of the same data reality can be distinguished uniformly by using the different classes of methods at the same level of abstraction. We apply different specification methods to the data structure binary tree of integers to illustrate the criterion used in the classification. At the end we present a concise survey of the literature on formal specification of data structures and data types to indicate how the proposed classification can help the assessment of work in the area. In the conclusions we point out directions for further research.

2. Levels of Data Abstraction

In this section we will try to illustrate through an example the role of the concept of levels of abstraction in the specification of data structures and data types. The concept of abstraction has at least two interrelated meanings. In fact, abstraction can be thought of as an intellectual "mechanism" which allows us to express the relevant facts (eliminating the irrelevant details) about the universe which is the object of our studies. We express these facts in accordance with some criteria or objective. Abstraction is also a process of generalization based on common factors which can be observed in different phenomena.

The second meaning attributed to abstraction does in fact presupposes the first. That is, previous to the process of generalization (verification of common factors) there exists the process of extraction of relevant information about each particular phenomenon (in accordance with a given criterion). The relevance of a given factor is attributed by the final objectives of whomever is conducting the studies.

Let us assume, as an example, that we are attempting to describe data structures focusing exclusively on the characteristics of accessibility of the components of each particular structure. By using this criterion, it is possible to say for every structure that each of its components is accessible from another component if there exists a connection between them (i.e., if there is a path that connects them). Note that by zooming in on the accessibility characteristics of the data structures, we overlooked aspects such as the constituent data types of the structure and the common characteristics of the class of structures under investigation (tree, string, ring etc).

Once we have specified the characteristic of accessibility, it is possible to obtain general properties for the class of all such data structures. For example, it may be stated that for every data structure there must exist (at least) an initial component from which all its components are accessible. We will also be interested in the properties of the operations and tests (and how these interact with the accessibility relation).

As the example illustrates, the final product of the process of abstraction is a formalized description which will provide a better understanding of the reality under study and which will allow us to infer new facts from the facts we already know.

Even when we focus on the same characteristics of the universe which is the object of our study we may use different levels of abstraction to express what we consider to be the relevant facts about this universe. In other words, the objective of our study will dictate which details are relevant and which are not.

We are now going to illustrate this fact through an example. The example consists of a comparison between two descriptions of the class of data structures usually called linear list. The approach we take will be an operational one. That is we intend to define abstract machines for manipulating these structures. In both examples, we assume the existence of some previously given abstract machines to manipulate "simpler" data structures.

Let us suppose that for the first example, we have a string manipulation machine with the following operations:

(i) a binary operation $x \cdot y$ to concatenate strings;

(ii) a binary operation value to determine the value from Γ at position n in string x;

(iii) the empty string ϵ;

(iv) the test empty to test whether a given string is ϵ.

We can define an abstract machine for linear lists over the values Γ (the usual idea of singly or doubly linked linear structures containing values from Γ at each point in the list). The data structures of linear lists are just pairs of the strings of the above machine and a positive integer value. This value is meant to indicate some position in the string. We can define the operations and tests of this new machine in terms of the operations of the string machine as follows:

```
(i)   insert (a,<ℓ,n> = <"a"·ℓ,n+1> where a ∈ Γ and "a" is
      the unit string formed from the value a;
(ii)  front (<ℓ,n>) = value (1,ℓ);
(iii) next (<ℓ,n>) = if value (n+1,ℓ) = error
                     then error
                     else <ℓ,n+1>
(iv)  previous (<ℓ,n>) = if n=1 then error
                         else <ℓ,n-1>;
(v)   linempty (<ℓ,n>) = empty (ℓ).
```

According to Guha and Yeh [13] a linear list structure L over an address space A is a 6-tuple $L = (N,\Gamma,\Sigma,M,Z,\delta,E)$, in which

(i) $N \subseteq A$;

(ii) Γ is a finite set of information items containing λ the null item of information;

(iii) $\Sigma \subseteq \Gamma$ is a data set;

(iv) M is a subset of the set of nodes $N \times \Gamma$ such that for every two nodes, (n_1,a_1) and (n_2,a_2) in M, $n_1=n_2$ implies $a_1=a_2$, for every $n \in N$ there exists a node (n,a) in M;

(v) Z is a finite ordered set of connecting labels;

(vi) δ is a partial function, called the connection function, $\delta\colon\ M \times Z \to N$;

(vii) $E \subseteq N$ is a finite set of entry points with the following additional characteristics, $1 \le \#E \le 2$, $1 \le \#Z \le 2$.

There exists a linear order $m_1,m_2,\ldots,m_k$ over the nodes of M such that

(a) m_1 is an entry point and is called the front node;

(b) m_k is the final node and also an entry node if $\#E=2$.

If $Z = \{z_1\}$, then $\theta_{z_1}(m_i) = m_{i+1}$, for $1 \le i \le k$ and the value of $\theta_{z_1}(m_k)$ is either e or m_1, where θ_{z_j} is a function $\theta_j\colon M \to M$ and e is the empty node.

If $Z = \{z_1,z_2\}$, then $\theta_{z_2} = \theta_{z_1}^{-1}$. For $1 \le i \le k$, $m_i \in N \times \Sigma$ if $m_1 \in N \times (\Gamma - \Sigma)$, m_1 is called the list head.

The latter characterization of linear list is proposed at a lower level of abstraction, compared to the first one. To achieve different levels of abstraction the different authors resort to different methodologies. It must be noted also that the objectives of the definitions differ slightly. In fact, the second approach uses the term linear list to refer to structures usually called circular lists ($\theta_z(m_k) = m_1$), singly or doubly linked lists ($Z = \{z_1,z_2\}$) and circular doubly linked lists. We can remove the circularity by requiring that #E=1 and thus conform to the above definition. We can require the existence of double links by requiring $Z = \{z_1,z_2\}$. We can define the linear list operation insert on this machine as a tranformation from one structure $L_1 = (N,\Gamma,\Sigma,M_1,Z,\delta_1,E_1)$ to another $L_2 = (N_2,\Gamma,\Sigma,M_2,Z,\delta_2,E_2)$.

(i) insert $(a,L_1) = L_2$ where

(a) $N_2 = N_1 \cup \{n_2\}$ and $n_2 \notin N_1$;

(b) $M_2 = M_1 \cup (n_2,a)$;

(c) $\delta_2 = \delta_1 \cup \{(((n_2,a),z_1),n_1), \quad (((n_2,a),z_2),c)\}$ where $E_1 = \{n_1\}$;

(d) $E_2 = \{n_2\}$.

As for the other operations:

(ii) front (L_1) = a where $(n_1,a) \in M$ and $E_1 = \{n_1\}$.

(iii) next $(L_1,(n,a)) = \delta_1((n,a),z_1)$;

(iv) previous $(L_1,(n,a)) = \delta_1((n,a),z_2)$;

(v) linempty (L_1) = <u>if</u> $M = \phi$ <u>then</u> <u>true</u> <u>else</u> <u>false</u>.

We note that the level of abstraction in the two definitions is a consequence of the complexity (or degree of detail) in the definition of the underlying machine. It is clearly easier to think in terms of strings of symbols than in terms of storage locations, contents of locations, and pointers.

We have ignored in the above characterizations of linear lists the definition of the accessibility relation. In the first case, the operations front, next and previous can clearly be used to define this relation whereas in the latter case, a more direct (but more complicated) definition can be defined in terms of δ, the connection function.

To emphasize the issue of levels of abstraction, we will describe the "linear list" given intuitively in figure 1, using both formalizations

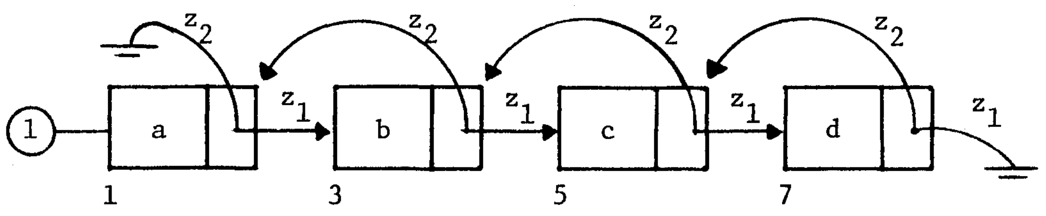


Figure 1

The "name" of the list in figure 1 as a structure of the abstract machine defined in terms of strings is <abcd,1>. (Or, it could be <abcd,k> for any $k > 0$.)

The "description" of the list in figure 1 using the style proposed in [13] reads as follows

$N = \{1,3,5,7\}$, $\Gamma = \{a,b,c,d,\Lambda\}$, $\Sigma = \{a,b,c,d\}$

$Z = \{z_1,z_2\}$, $E = \{1\}$, $M = \{(1,a),(3,b),(5,c),(7,d)\}$

and

$\delta((1,a),z_1) = 3$	$\delta((5,c),z_1) = 7$
$\delta((1,a),z_2) = e$	$\delta((5,c),z_2) = 3$
$\delta((3,b),z_1) = 5$	$\delta((7,d),z_1) = e$
$\delta((3,b),z_2) = 1$	$\delta((7,d),z_2) = 5$

3. Formal Specification Methods

The methods for the specification of data objects may be classified into two groups: dynamic (explicit) and static (implicit) methods. The dynamic methods are those which describe classes of objects by considering essentially how these objects get transformed within the same class. The static methods describe the objects and operations of a certain class implicitly without concern about how an object gets tranformed into another. The dynamic methods can be further subdivided into operational and generative methods. The static methods can also be subdivided into intentional and extentional methods.

If all methods above are used to describe the same reality, a better understanding of the object under study can be achieved. In this sense the above methods are complementary to each other [28].

To illustrate the above classification of methods we will characterize intensionally, extensionally, operationally and generatively the data type binary tree of integers.

We will use a first order theory to provide the intentional description. For the operational description we will use a set of algorithms which operate over the data structure binary tree of integers. The generative description makes use of grammars both to assign names to the trees that constitute the data type and to transform objects of the type. Finally, we use graphs to provide an extensional characterization of binary trees.

3.1 An Intentional Specification

The first order language L to be used for the axiomatic description of the type binary tree, contains the following non-logical symbols

$L = \{\Lambda, c, e, d, k, \lambda, t, a, \leq\} \quad \cup \{N\}$

where

(i) Λ, λ are symbols for constants (the empty tree and the null atom);

(ii) $\leq$ is a binary predicate (subtree);

(iii) c is a ternary functional symbol (construct);

(iv) e,d,k are unary functional symbols (extract left subtree, extract right subtree and visit the root);

(v) t is a unary predicate (tree);

(vi) a is a unary predicate (atom);

(vii) N is the set of natural numbers.

The axioms for the intensional characterization of binary trees of integers can be expressed as follows:

1. $a(\lambda) \wedge t(\Lambda)$; 1a. $a(n)$ for all $n \geq 0$;
2. $\forall x(t(x) \rightarrow x \neq \Lambda \rightarrow \neg\, x \leq \Lambda)$;
3. $\forall x \forall y \forall z\ (t(x) \wedge t(y) \wedge t(z) \rightarrow (x \leq y \rightarrow y \leq z \rightarrow x \leq z))$;
4. $\forall x(t(x) \rightarrow x \leq x)$;
5. $\forall x \forall y \forall z\ (a(z) \wedge t(x) \wedge t(y) \rightarrow x \leq c(x,z,y) \wedge y \leq c(x,z,y))$;
6. $\forall x \forall y \forall z\ (a(z) \wedge t(x) \wedge t(y) \wedge x \neq \Lambda \wedge y \neq \Lambda \wedge$
   $z \neq \lambda \rightarrow e(c(x,z,y)) = x \wedge d(c(x,z,y)) = y$
   $\wedge\ k(c(x,z,y)) = z)$;
7. $(e(\Lambda) = \Lambda \wedge d(\Lambda) = \Lambda \wedge k(\Lambda) = \lambda)$.

The meaning of the axioms can be explained as follows:

1. λ is an atom and Λ is a tree; 1a. every natural number is an atom;
2. if x is a tree and x is not Λ then x is not a subtree of Λ;
3. if x is a subtree of y and y is a subtree of z, then x is a subtree of z;
4. x is a subtree of itself;
5. if z is an atom and x, y are trees, then x and y are subtrees of c(x,z,y);
6. if x and y are non-empty trees and z is a non-empty atom, then x and y are the left and right subtrees of c(x,z,y), respectively and z is the root of c(x,z,y);
7. the left and right subtrees of Λ are Λ and the root of Λ has atom λ.

3.2 An Operational Specification

We will consider the following operations for the description of the type binary tree of integers: $\underline{c}$ (construct a tree), $\underline{e}$ (extract left subtree), $\underline{d}$ (extract right subtree) and $\underline{k}$ (visit the root). We include the relation $\leq$ (is a subtree of) as part of the specification. The semantics of the operations and of the relation will be given through algorithms expressed in an Algol-like language. These algorithms take as input strings of symbols which correspond to expressions formed from the above operations and the $\leq$ relation followed by the appropriate arguments. The arguments are canonical names. A binary tree name is called a canonical name if the only operation symbol that appears in the name is the $\underline{c}$ symbol for "construct a binary tree".

Example:

$$\underline{c}(\underline{c}(\Lambda,4,\Lambda),5,\underline{c}(\Lambda,9,\Lambda)).$$

The operation $\underline{c}$ acts over three arguments: the first and the third are binary trees and the second is an integer. The operations $\underline{e}$, $\underline{d}$ and $\underline{k}$ have a single argument: a binary tree. The first two operations produce binary trees while the third returns an integer. The relation $\leq$ has two binary trees as arguments and returns either true or false.

We assume the availability of the primitive functions Arg_1, Arg_2 and Arg_3 in the language used to express the algorithm. They extract the first, second and third arguments respectively (if they exist) when an expression Arg in terms of the above symbols is given.

In what follows we present the algorithms that specify the semantics of the type binary tree of integers.

```
begin read (arg); operation ← 1st symbol of Arg;
    case operation of
    'c':  (connects to the left at (Arg$_2$(Arg)) with
           tree (Arg$_1$(Arg)) and to the right with
           tree (Arg$_3$(Arg)));
    'e':  (if Arg = Λ then (generate null tree)
                        else tree (Arg$_1$(Arg)));
    'd':  (if Arg = Λ then (generate null tree)
                        else tree (Arg$_3$(Arg)));
    'k':  (return Arg$_2$(Arg));
    '≤':  rel (Arg)
    esac
end;
```

```
rel (Arg) begin
      if $Arg_1$ (Arg) = $\Lambda$ then return (true);
      if $Arg_1$ (Arg) $\neq \Lambda \wedge Arg_2$ (Arg) = $\Lambda$ then
                        return (false);
      if $Arg_2(Arg_1(Arg)) \neq Arg_2(Arg_2(Arg))$ then
                        return (rel($Arg_1(Arg), Arg_1(Arg_2(Arg))$)
                              $\vee$ rel($Arg_1(Arg), Arg_3(Arg_2(Arg))$));
      if $Arg_2(Arg_1(Arg)) = Arg_2(Arg_2(Arg))$ then
                        return (rel($Arg_1(Arg_1(Arg)), Arg_1(Arg_2(Arg))$)
                              $\wedge$ rel($Arg_3(Arg_1(Arg)), Arg_3(Arg_2(Arg))$))
            end;

tree (Arg) begin
      if Arg = $\Lambda$ then return (null tree)
            else return (tree with
                  at ($Arg_2(Arg)$) connected to the left
                  with tree ($Arg_1(Arg)$) and to the right with
                  tree ($Arg_3(Arg)$))
            end;
at (Arg) begin generates a node whose content = Arg;
            return end.
```

3.3 <u>A Generation Specification</u>

We will provide a grammar whose alphabet of terminal symbols is given by

$$\{\underline{c},\underline{e},\underline{d},\ldots,\Lambda,(\ ,\ ),\ ,\ \}\cup '\mathit{N}',$$

where 'N' is the alphabet which denotes all the non-negative integers. From 'N' we will get the names of integers to "label" nodes of all binary trees which constitute the corresponding data type.

$$G_1 = \langle N,T,P,s\rangle$$

where:

(i) $T = \{\underline{c},\underline{e},\underline{d},\Lambda,(\ ,\ ),\ ,\ \}\cup '\mathit{N}'$;

(ii) $N = \{\mathbf{s}\}$.

(iii) P:

$$\begin{aligned} s &\to \Lambda \\ &\mid \underline{c}\ (s,'\mathit{N}',s) \\ &\mid \underline{e}\ (s) \\ &\mid \underline{d}\ (s) \end{aligned}$$

With the above grammar we can, for instance, generate the following strings of symbols:

$$s \to c(s,'\mathit{N}',s) \overset{*}{\to} c(\Lambda,4,s) \to c(\Lambda,4,c(s,'\mathit{N}',s))$$

$$\to c(\Lambda,4,c(\Lambda,5,\Lambda));$$

$$s \to c(s,'\mathit{N}',s) \to c(e(s),'\mathit{N}',s) \to$$

$$c(e(c(s,'\mathit{N}',s)),'\mathit{N}',s) \to c(e(c(s,'\mathit{N}',s)),'\mathit{N}',c(s,'\mathit{N}',s))$$

$$\overset{*}{\to} c(e(c(\Lambda,3,\Lambda)),\ 4\ ,c(\Lambda,5,\Lambda)).$$

The above two names denote the same structure, which can be represented graphically as in figure 2:

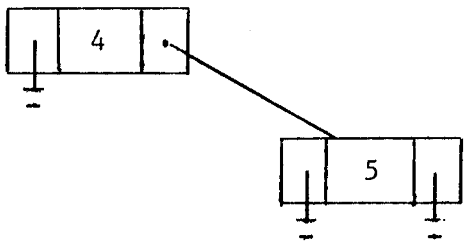


Figure 2

To fix the subset of canonical names in binary trees, we use the grammar G_2:

$$G_2 = <N,T,P,s>$$

where:

(i) $T = \{\Lambda, c, (\ ,), \ , \ \} \cup 'N'$;

(ii) $N = \{s\}$;

(iii) P:

$$s \rightarrow \Lambda \mid c\ (s, 'N', s).$$

To transform a non-canonical name (strings in which the symbols e or d appear) into a corresponding canonical name (equivalent under the chosen interpretation) we provide the following sets of transformations. (Note λ is the empty string.)

<u>e</u> transformations:

$e\Lambda \rightarrow \Lambda H$		$F_nX \rightarrow XF_n$	$n>0$
$H) \rightarrow \lambda$		$F_n) \rightarrow)F_{n-1}$	$n>0$
$e(\rightarrow e$		$)F_0 \rightarrow)G_0$	
$ec \rightarrow E$		$G_nX \rightarrow G_n$	$n\geq 0$
$E(\rightarrow F_0$		$G_n(\rightarrow G_{n+1}$	$n\geq 0$
$F_0c \rightarrow cF_0$		$G_n) \rightarrow G_{n-1}$	$n\geq 0$
$F_0\Lambda \rightarrow \Lambda G_1$		$G_{-1} \rightarrow \lambda$	
$F_n(\rightarrow (F_{n+1}$	$n\geq 0$		

with X = {Λ,c,'*N*', , };

<u>d</u> transformations:

$d\Lambda \rightarrow \Lambda H$		$I_n(\rightarrow I_{n+1}$	$n\geq 0$
$H) \rightarrow \lambda$		$I_n) \rightarrow I_{n-1}$	$n\geq 0$
$d(\rightarrow d$		$LZ \rightarrow L$	
$dc \rightarrow D$		$LW \rightarrow WJ_0$	
$D(\rightarrow I_0$		$J_nX \rightarrow XJ_n$	$n\geq 0$
$I_nX \rightarrow I_n$	$n>0$	$J_n(\rightarrow (J_{n+1}$	$n\geq 0$
$I_0Y \rightarrow I_0$		$J_n) \rightarrow)J_{n-1}$	$n>0$
$I_0, \rightarrow L$		$J_0) \rightarrow J_{-1}$	
		$J_{-1} \rightarrow \lambda$	

where:

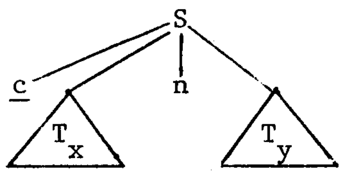
X = {Λ,c,'*N*', ,}

Y = {Λ,c,'*N*'}

Z = {'*N*', , }

W = {c,Λ}

Operations on this type are defined in terms of the parse trees of grammar G_2. Thus, for example, we can construct the parse tree corresponding to the tree c(x,n,y) from the parse trees T_x and T_y of x and y, respectively, according to the diagram



Similarly, we can define the operations e and d as extracting the second and fourth subtrees, respectively, of a tree of the above form. The relation $\leq$ can be defined by: $x \leq y$ iff the derivation tree of x is a subtree of the derivation tree of y.

3.4 An Extensional Specification

The specific extensional characterization to be presented in this section makes use of graphs a restriction of which is used to model binary trees. The graphs are labeled directed graphs, defined by the 6-tuple

$$(N,NL,EL,\nu,\delta,\alpha)$$

where:

(i) N is a non-empty finite set denoted by positive integers;

(ii) NL is a non-empty finite set of node labels;

(iii) EL is a finite set of edge labels;

(iv) ν is a total function which assigns a label to each node of a graph

$$\nu \colon N \to NL;$$

(v) δ is a total function which defines by adjacency the structure of the graph

$$\delta \colon N \times EL \to N \cup \{\lambda\};$$

(vi) α is a node, $\alpha \in N$, such that for every $n \in N$ there exists a sequence of nodes $(n_1,\ldots,n_k)$ with $n_1=\alpha$ and $n_k=n$ such that for $1 \le i < k$ there exist edge labels $e_i \in EL$ such that $\delta(n_i,e_i) = n_{i+1}$.

The concept of connectivity between nodes is defined through the predicate k as follows:

$$\begin{aligned} k(n_1,n_2) \Leftrightarrow \delta(n_1,e_1) &= n_2 \vee \delta(n_1,e_2) = n_2 \vee \\ &\exists n'(\delta(n_1,e_1) = n' \wedge k(n',n_2)) \vee \\ &\exists n'(\delta(n_1,e_2) = n' \wedge k(n',n_2))) \end{aligned}$$

with the following additional relations to define binary tree structures:

(a) $$\forall n_1 \, \forall n_2 \, (\neg((k^1_{e_1}(n_1,n_2) \vee k^1_{e_1} \circ k(n_1,n_2)) \wedge$$

$$(k^1_{e_1}(n_1,n_2) \vee k^1_{e_2} \circ k(n_1,n_2))).$$

(i.e., a node is not both a left and right descendent of another node);

(b) $\forall n\ (\neg((k^1_{e_1}(\alpha,n) \vee k^1_{e_1} \circ k(\alpha,n)) \wedge (k^1_{e_2}(\alpha,n) \vee k^1_{e_2} \circ k(\alpha,n))))$.

(i.e., α has no in-edges);

(c) $\forall n\ (\neg k(n,n))$

(i.e., no cycles)

where k_{e_1} (k_{e_2}) means the restriction imposed to connectivity by allowing only edge labels $e_1(e_2)$ in the definition of k. The notation $k^n(n_1,n_2)$ expresses the fact that a path of length n is required to connect n_1 to n_2 (i.e., a path with n-1 edges). The operator "s" for the composition of predicates is defined by

$$k \circ k'(n_1,n_2) \Leftrightarrow \exists n'\ (k(n_1,n') \wedge k'(n',n_2)).$$

We also restrict the set of edge labels EL to the set $\{e_1,e_2\}$. We denote by $t(x)$ the predicate defined by a, b, and c above on a given graph. Thus the graph x is a tree if it satisfies a, b and c.

We now define the operations $\underline{c}$, $\underline{d}$, $\underline{e}$ and $\underline{k}$ and the test $\leq$ implicitly in terms of the above structures. The axioms we need are as follows:

1. $\forall x(t(x) \rightarrow x \leq x)$;
2. $\forall x \forall y \forall z(t(x) \wedge t(y) \wedge t(z) \rightarrow$

   $(x \leq y \rightarrow y \leq z \rightarrow x \leq z))$;
3. $\forall x \forall y \forall z(z \in NL \wedge t(x) \wedge t(y) \rightarrow$

   $x \leq c(x,z,y) \wedge y \leq c(x,z,y))$;
4. $\forall x \forall y \forall z(z \in NL \wedge t(x) \wedge t(y) \rightarrow e(c(x,z,y)) = x \wedge$

   $d(c(x,z,y)) = y \wedge k(c(x,z,y)) = z)$.

Note that the above definition of binary trees does not allow the empty tree. This is because we required that N (the set of nodes of a graph) be non-empty. If we wanted to include the empty tree, we would have to allow N to be empty; i.e., modify (vi) in the definition of a graph by adding the proviso "If N is non-empty, then ...". We would also have to add axioms to the above to express the properties of the empty tree.

4. Classification of Formal Specification Methods

We have grouped the methods for the specification of the universe of study called "data structures and data types" in four groups which we called respectively:

(i) intentional methods,

(ii) extensional methods,

(iii) operational methods,

(iv) generative methods.

The intensional methods are theories described in first order (or higher) languages (or in restrictions of these languages) that describe the intrinsic characteristics of the objects under study and the relations between them.

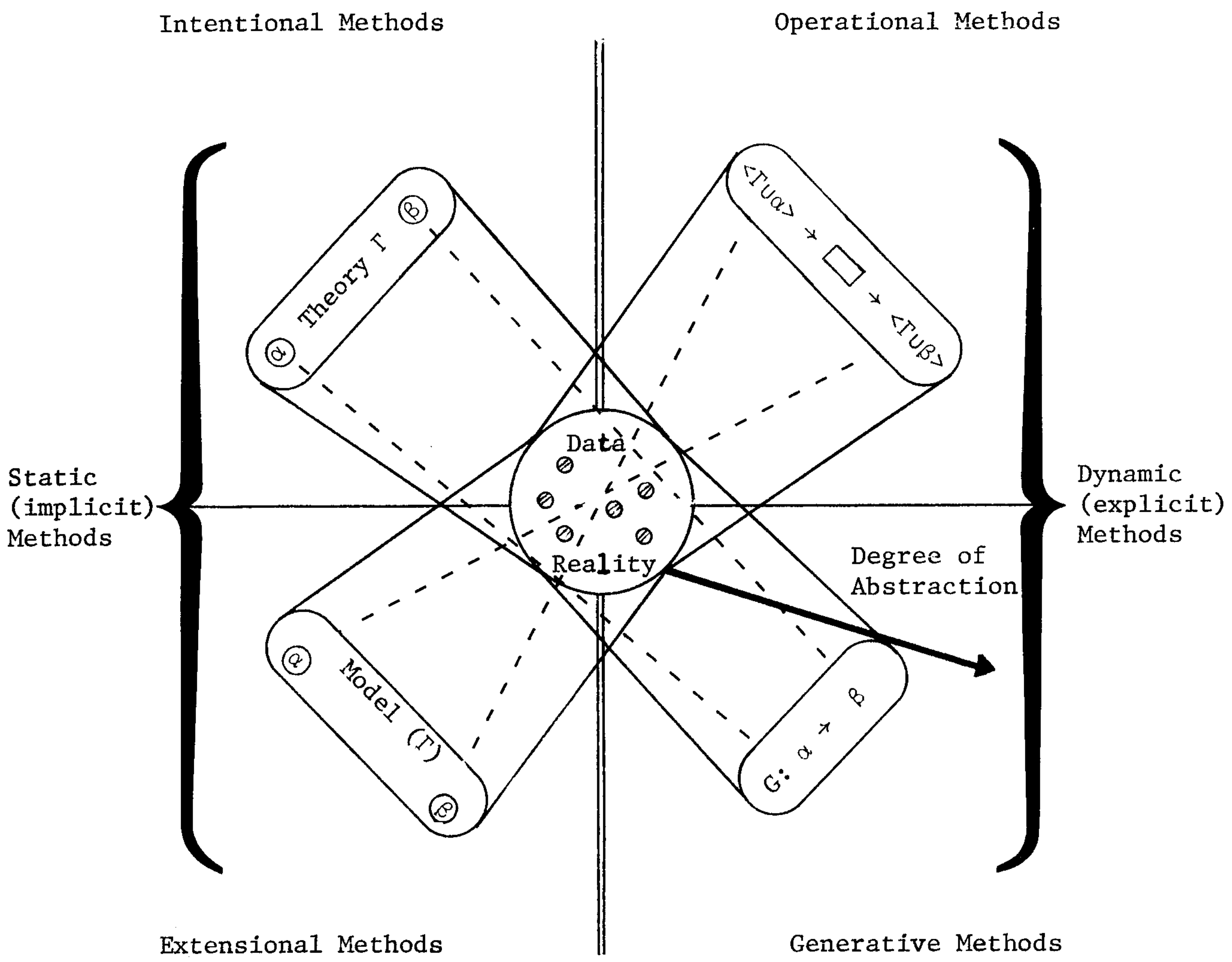
The extensional methods include the description of structures (in the sense of the theory of models) over a given domain in such a way that these structures are capable of modelling the data reality under study. Such description can be given in languages which are not totally formalized as happens with the use of tables, graphs and other descriptions that make use of metalanguages which are reinforced by the use of logic and/or other mathematical tools.

The operational methods include formalizations in which the objects under study are described through the specifications of the transformations that can be applied to representations of these objects by virtual machines that operate on these representations.

We call generative methods the formalizations that describe the transformations of the objects under study through the use of re-writing rules given for the symbolic representation of these objects (i.e. grammars).

The specification methods that belong to each of the above groups differ from each other in terms of the degree of abstraction provided by a particular formalization. The higher the degree of abstraction the greater the ability to describe a larger number of classes of objects. The power of the method to describe the characteristics of each class of objects is proportionally decreased.

In figure 3 we summarize what was said above:



Intentional Methods
Operational Methods
Static (implicit) Methods
Dynamic (explicit) Methods
Extensional Methods
Generative Methods
Theory Γ
α
β
<Γ∪α> → □ → <Γ∪β>
Model (Γ)
G: α → β
Data
Reality
Degree of Abstraction

Figure 3

5. Current Work on Data Specification Methods

An important application of the classification proposed for the data specification methods is to use it as an aid for the understanding and evaluation of the current work on data specification.

Liskov and Zilles [22] classified different data specification methods in five categories:

(i) Use of fixed discpline (fixed domain of formal objects).

(ii) Use of an arbitrary descipline.

(iii) Use of a state machine model.

(iv) Use of axiomatic descriptions.

(v) Use of algebraic definitions.

In an earlier version of their paper [21] a comment was made that the first two categories make use of abstract models while the other three use implicit definitions. According to our classification the methods (i) and (ii) constitute extensional characterizations of data, (iii) is an operational specification and (iv) and (v) are intentional specification methods.

In what follows we present a concise survey of the literature on data specfication by trying to label different research efforts by using our proposed classification.

The Vienna Definition Language [30] popularized the operational semantics approach originally proposed by McCarthy [31] for the definition of LISP. The more recent work by Parnas [32] and Robinson [33] are very representative examples of operational semantics. Robinson [34] proposes a proof method based on the use of operational specification of data at different levels of abstraction.

Fleck [7] provides a generative description of different classes of data structures. He establishes a correspondence between the data structures and particular grammars. More recently Paul [35] proposes a semantics for data structures in which both types and objects are based on certain right linear grammars. Jackson [36] proposed a programming methodology based on the application of the generative specification approach to data types. Cowan et al developed this idea a little further [37,38].

The work by Gotlieb and Furtado [12] is an interesting example of the combined use of the operational and generative approaches to data specification. It uses graph grammars to describe both which elements belong to each class of data structures and the transformations that can be applied to the elements.

Lots of efforts have been reported in the literature on the use of extensional methods for data specfication. Back in 1971, Earley [5] used graphs as models for data structures. He blended the extensional specification with the operational approach to describe the transformations between elements of the same type. In a later paper Earley [6] uses the pure extensional approach by using the relational model suggested by Codd [39] for the specification of data structures. The work by Oppen and Cook [4] also proposes an extensional description of data structures by using graphs as models for the different classes of structures. Similarly, the work described in [13], [9], [24] and [25] use the extensional approach. Dana Scott [26] models data types by using the mathematical structure of lattices. This approach provides an example of the use of an extensional method at a high level of abstraction. Other examples of the extensional approach are the works by Carvalho et al [1, 2] in which data structures are specified through a general relational model [40] and through the application of the notion of minimal models.

The work developed by Hoare [15] and [16] uses an intentional specification method. It resorts to first order languages to describe the characteristics of data types and data structures. Clark and Tarlund [3] develop a data theory expressed in a first order language and use it for the characterization of data structures. Standish [27] also resorts to logic to describe the axioms for classes of data structures (based on the VDL objects [30]). Lucena, Pequeno and Veloso [41,42,28] specify data types by using ideas based on the theory of definition and the notion of interpretation between theories from mathematical logic.

The work by Goguen et al [10], [11] etc. make use of intensional methods for the specification of data types. The characteristics of each type is established through the use equational algebras. Along the same approach, it is worth referencing the work reported in [14], [20], [21], [19], [23], [18], [29], [43] and [17]. In some of the above work the algebraic axiomatization is associated with interpretations in terms of concrete algebras. In these cases we have a combined use of intentional and extensional methods.

6. Conclusions

We tried to illustrate in the previous section the fact that our proposal for the classification of data specification methods helps to provide a more clear understanding of the current literature. This is true to the extent that it helps to determine the role of a particular contribution in the area.

Different types of methods can and in some cases should be applied to the definition of a particular data abstraction. If we express our view of a given data object through different specification methods we can explore our abstraction better by using different analytical tools or gain some additional insight about the object such as, for instance, discovering the best way to implement it. To illustrate this last example, suppose that by moving from an intentional specification (say using predicate calculus) to a generative specification we are able to find a direct SNOBOL implementation for a data manipulation problem. On the other hand, it is probably easier to prove properties of the program using the intensional view since this view eliminates many of the implementation details which would just clutter up the proof.

We conjecture that some further work based on our proposed classification for data specification methods will find applications in some interesting problem areas defined by Ashcroft and Wadge [44] and Backus [45].

In [44] the authors express their interest in determining for what purpose a specification for programming language semantics is used. When a specifciation method is used to describe, model or classify a given entity, it is said to be used in the descriptive sense. When the intention of its utilization is geared to the planning of a new object (i.e. language) it is said to be used in the prescriptive sense. The authors claim that the problem with semantics is that it is still in the immature, descriptive mode and a move to the prescriptive mode is well overdue. The method that the authors themselves adopt is what we would call the extensional method. (The language Lucid and its programs are defined implicitly in terms of a type consisting of infinite sequences of data objects.) Work based on our classification can help determine which specification methods are more adequate to support the descriptive approach (operational and generative?) and the prescriptive approach (intentional and extensional?) to the semantics of data.

Backus [45] states that both the extensional methods (he refers to [26]) and the intentional methods (he calls them axiomatic methods) are not able to eliminate the clumsy properties (assignment, etc.) of the basic von Neumann style. We again refer to Ashcroft and Wadge [44] to disprove this statement since assignment is not in fact part of Lucid and the "natural" implementation for the language is a data flow implementation. The question in this case is how high should the level of abstraction of a specification method be to allow it to be used in the prescriptive mode (that is, without any implicit reference to the von Neumann design). The problem exists, of course, when it is necessary to capture in the specification all the details about data that characterize the real life applications and the existence of the von Neumann machine may be part of that reality.

References

1. de Carvalho, R.L., Furtado, A.L., Pereda, A.A.: "A relational model towards the synthesis of data structures", M.C.C. 17/77, DI/PUC-RJ, (1977).

2. de Carvalho, R.L., Pequeno, T.H., Pereda, A.A., Veloso, P.A.: "Semantics of Data Types and Structures: A Model-Theoretic Approach", DI/PUC-RJ, (1979).

3. Clark, K.L., Tärnlund, S.: "A first order theory of data and programs", Information Processing 77, IFIP, (1977).

4. Oppen, D.C., Cook, S.A.,: "Proving Assertions about Programs that Manipulate Data Structures", Proc. of 7th Symposium on Theory of Computing, (1975).

5. Earley, J.: "Toward an Understanding of Data Structures", CACM, Vol. 14, 6/7, (1971).

6. Earley, J.: "Relational Level Data Structures for Programming Languages", Acta Informatica, Vol. 2, No. 4, (1973).

7. Fleck, A.C.: "Toward a Theory of Data Structures", JCSS, Vol. 5, (1971).

8. Fleck, A.C.: "Recent developments in the theory of data structures", Computer Languages, Vol. 3, pp. 37-52, (1978).

9. Furtado, A.L.: "Characterizing sets of Data Structures by the Connectivity Relation", Int. J.C.I.S., Vol. 5, No. 2, (1976).

10. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.,: "Abstract Data Types as Initial Algebras andthe Correctness of Data Representation", Conference on Computer Graphics, Pattern Recognition, and Data Structures, (1975).

11. Goguen, J.A., Thatcher, J.W., Wagner, E.G.: "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", in Current Trends in Programming Methodology, Vol. 4, Ed. R.T. Yeh, Prentice Hall, (1977).

12. Gotlieb, C.C., Furtado, A.L.: "Data Schemata Based on Directed Graphs", DCS-U. of Toronto, T.R. No. 70, (1974).

13. Guha, ., Yeh, R.T.: "A Formalization and Analysis of Simple List Structures" in Applied Computation Theory, ed. R.T. Yeh, Prentice Hall, (1976).

14. Guttag, J.V.,: "The specification and application to programming of abstract data types", CSRG., U. of Toronto, T.R. No. 59, (1975).

15. Hoare, C.A.R.: "Notes on Data Structuring" in Structured Programming, Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., Academic Press, (1972).

16. Hoare, C.A.R.: "Proof of Correctness of Data Representations", Acta Informatica, Vol. 1, No. 1, (1972).

17. Horowitz, E., Sahni, S.: Fundamentals of Data Structures, Computer Science Press, (1976).

18. Levy, M.R., Maibaum, T.S.E.: "Continuous Data Types with Sharing and Circularity", DCS-U. of Waterloo, (1977).

19. Levy, M.R., Maibaum, T.S.E.: "Continuous Data Types", DCS-U. of Waterloo, (1978).

20. Liskov, B., Zilles, S.: "Programming with Abstract Data Types", SIGPLAN Notices, Vol. 9, No. 4, (1974).

21. Liskov, B., Zilles, S.: "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. 1., No. 1, (1975).

22. Liskov, B., Zilles, S.: "Specification Techniques for Data Abstractions", in Current Trends in Programming Methodology, Vol. 1, Ed. R.T. Yeh, Prentice-Hall, (1977).

23. Maibaum, T.S.E., Lucena, C.J.: "Higher Order Data Types", MCC, No. 70/77, DI/PUC-RJ, (1977).

24. Majster, M.E.: "Extended Directed Graphs, a Formalism for Structured Data and Data Structures", Acta Informatica, Vol. 8, No. 1 (1977).

25. Pfaltz, J.L.: Computer Data Structures, McGraw-Hill Book Company, (1977).

26. Scott, D.: "Data Types as Lattices", Advanced course on Programming, Languages and Data Structures, Amsterdam, (1972).

27. Standish, T.A.: "Data Structures, An Axiomatic Approach", BBN Comp., Science Div., (1973).

28. Pereda A.A.: Thesis in preparation, PUC-RJ.

29. Thatcher, J.W., Wagner, E.G., Wright, J.B.: "Specification of Abstract Data Types using Conditional Axioms", IBM Research Report TC 6214, (1976).

30. Lucas, P., Lauer, P. and Stigleitner, H.: "Method and Notation for the Formal Definition of Programming Languages", T.R. 25.087, IBM Research Laboratory, Vienna.

31. McCarthy, J.: "Recursive Function of Symbolic Expressions and their Computation by Machine", CACM, Vol. 3, No. 4, (1960).

32. Parnas, D.: "A Technique for the Specification of Software Modules with Examples", CACM, Vol. 15, (1972).

33. Robinson, L.: "Specification and Proof in Problems of Concurrency", Proc. of a Meeting on 20 years of Computer Science, Pisa, Italy, (June 1975).

34. Robinson, L.: "Proof Techniques for Hierarchically Structured Programs" in Current Trends in Programming Methodology, Vol. IV, Ed. R.T. Yeh, Prentice-Hall, (1977).

35. Paul, M., Güntzer, U.: "On a Uniform Formal Description of Data Structures", Technical Report, Technical University of Munich (1978).

36. Jackson, M.A.: Principles of Program Design, Academic Press, (1975).

37. Cowan, D.D., Lucena, C.J.: "Some Thoughts on the Construction of Programs in A Data Directed Approach" in Information Technology, Ed. U. Moneta, North-Holland, (1978).

38. Cowan, D.D., Graham, J., Welch, J., Lucena, C.J.: "A Data Directed Approach to Program Construction", to appear in Software Practice and Experience.

39. Codd, E.F.: "A Relational Model of Data for Large Shared Data Bases", CACM, Vol. 13, (1970).

40. Carvalho, R.L., Maibaum, T.S.E., Pequeno, T.H.C., Pereda, A.A., Veloso, P.A.S.: "A Model Theoretic Approach to the Semantics of Data Types and Structures", Technical Report, PUC/RJ.

41. Lucena, C.J., Pequeno, T.H.C.: "Program Derivation Based on Abstract Data Types", to appear IEEE Transactions on Software Engineering.

42. Veloso, P., Pequeno, T.H.C.: "Interpretation Between Many Sorted Theories", Proc. International Symposium of Logic, Campinas, Sao Paulo, (1978).

43. Pequeno, T.H.C., Veloso, P.: "Do not write more axioms than you have to", Proc. Int. Computer Symposium, Taipei, (1978).

44. Ashcroft, E.A., Wadge, W.W.: "Generality Considered Harmful", Technical Report, Dept. of Computer Science, U. of Waterloo, (1979).

45. Backus, J.W.: "Can Programming be Liberated from the von Neumann Style, A Functional Style and its Algebra of Programs", Technical Report RJ2234, IBM, (1978).