

PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 5/74

CHARACTERIZING SETS OF DATA STRUCTURES
BY THE CONNECTIVITY RELATION

by

A. L. Furtado

Computer Science Department - Rio Datacenter

Pontificia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

CHARACTERIZING SETS OF DATA STRUCTURES
BY THE CONNECTIVITY RELATION

A. L. Furtado
Associate Professor
Computer Science Department
(Informática) PUC/RJ

DIVISÃO DE INFORMAÇÕES	
RIPII	
código/registro	data
0932	9/4/75
RIO DATACENTRO	

Series Editor: Prof. S.M. dos Santos

December/1974

M863

RIO DATACENTRO SETOR DE DOCUMENTAÇÃO

ABSTRACT

A formalism based on the directed graph connectivity relation is presented, where a first order predicate calculus notation is used with the purpose of defining sets of data structures and characterizing the structural operations on them.

The formalism is designed so as to facilitate the proof that the intended operations on instances of a given set of data structures are correct, in the sense that they do not violate the definition of the set.

1. INTRODUCTION

The problem of proving the correctness of programs that handle non-trivial data structures has been considered in the recent years [1]. One step towards this objective is the construction of some formalism where sets of data structures and structural operations on these structures may be defined axiomatically [2,3].

The formalism to be discussed here treats the data structures as a restricted class of directed graphs (digraphs) [4] and employs a first order predicate calculus notation. Several graph theoretical models of data structures are found in the literature [5,6,7].

Digraph connectivity plays a fundamental role in the formalism. The connectivity relation induces an ordering on the set of nodes, and it may be interesting to restate several considerations appearing in this work in terms of orders and the basic properties of binary relations [8].

Progress in theorem proving offers a hope that certain proofs of assertions written as formulas from an applied predicate calculus may be mechanized, at least partially.

In the present work, the class of digraphs to be used as a model of data structures is defined, and then the applied predicate calculus notation is introduced. A number of sets of data structures are described in order to illustrate how such sets can be defined; the other purpose of the formalism - the characterization of structural operations - is discussed with the help of one of the sets just introduced, and it is shown how the two parts of the formalism are related.

2. THE GRAPH THEORETICAL MODEL

The accessibility to a component element of a data structure is usually ensured in the following way:

- a. The location of some special element of the data structure is initially given; often this location is somehow associated with the "name" of the data structure. As an example, consider the first element of a FORTRAN array, whose machine address is associated with the variable denoting the array.
- b. The path from the given special element to the desired element is indicated under some form. Again, in the FORTRAN array the path consists of the subscript or subscripts of the desired array element, which are entered in a formula to compute the address of the desired element from the address of the first element.

Accordingly, the class of directed graphs to which this work is restricted has the property that each digraph is rooted in the sense that there is some special node called the root from which all other nodes are reachable, and the property that no two edges with the same label can leave the same node. More precisely, the class of interest consists of rooted labelled digraphs, RLDs, an RLD being defined as a 6-tuple $(N, NL, EL, v, \delta, r)$, where[†]

N - a non-empty finite set of nodes, denoted by positive integers.

[†] One may note that the definition of an RLD is analogous to the definition of a Moore sequential machine [9].

NL - a non-empty finite set of node labels

EL - a finite set of edge labels

ν - total function which assigns a node label to every node of the RLD

$$\nu : N \rightarrow NL$$

δ - total function which defines the adjacency structure of the RLD

$$\delta : N \times EL \rightarrow N \cup \{u\}$$

where u stands for "undefined"

r - node, $r \in N$, such that for all $n \in N$ there exists a sequence of nodes (n_1, n_2, \dots, n_k) with $n_1 = r$ and $n_k = n$ such that for $1 \leq i < k$ there exist edge labels $e \in EL$ such that $\delta(n_i, e) = n_{i+1}$.

Due to the rootedness property and the fact that δ is a function, a node can be unambiguously located by giving the location of the root and a sequence of (possibly repeating) edge labels[†].

The node and edge labels are interpreted so as to make RLDs a model for data structures. Under the interpretation, node labels stand for domains, i.e. sets of items of information which are meaningful for some application and which share some storage characteristics; for instance, the node label SN may denote a set of "supplier names", each supplier name being a character string of up to 20 characters. The edge labels stand for functions; in order

[†] The defined sequences from the root of an RLD constitute a regular set (cf. the preceding footnote).

to represent many to many n-ary relations, which arise in generalized data bases [10], characteristic functions and projections are used. Thus, if $R \subseteq D_1 \times D_2 \times \dots \times D_n$, one defines the characteristic function $f_R : D_1 \times D_2 \times \dots \times D_n \rightarrow (T,F)$, where for $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$, $f_R(d_1, d_2, \dots, d_n) = T$ iff the relation R holds for the given n-tuple. Also, in order to access each component of an n-tuple, projection functions are used: $p_i : D_1 \times D_2 \times \dots \times D_i \times \dots \times D_n \rightarrow D_i, 1 \leq i \leq n$ (see fig.2.1).

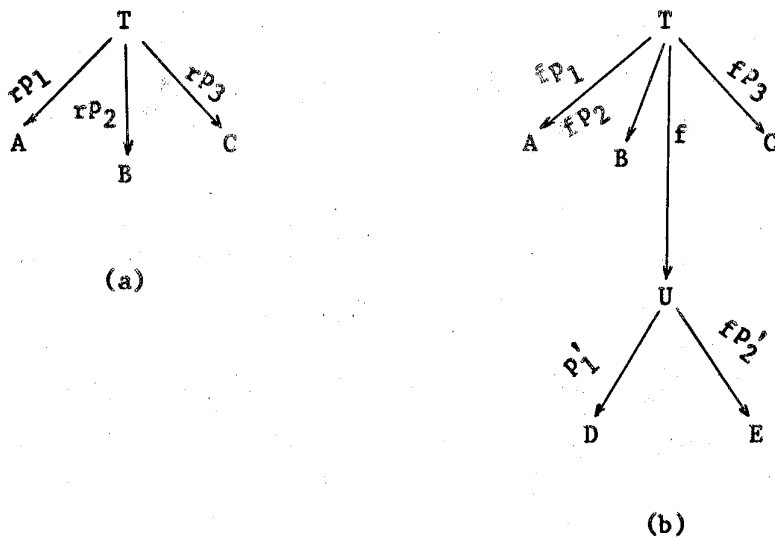


Figure 2.1: (a) triple in a ternary relation $R \subseteq T$, where $T = A \times B \times C$

(b) domain and range elements of function $f : T \rightarrow U$, where $T = A \times B \times C$ and $U = D \times E$

RLDs represent a sufficiently comprehensive model for data structures, as illustrated in section 4, where several interesting sets of data structures, which are RLDs, are discussed.

3. NOTATION

In order to characterize sets of data structures, a first - order predicate calculus notation is used.

The concept of connectivity plays a fundamental role in the characterization. As usual, if α and β are nodes of a digraph D , α is said to be connected to β if there is at least one path[†] from α to β ; a node is said to be connected to itself only in case of loops or cycles. Let F be the set of functions (named by the edge labels, as seen before), and $|F|$ the cardinality of F ; then, the connectivity predicate is defined as

$$\kappa(\alpha, \beta) \equiv \bigvee_{i=1}^{|F|} f_i(\alpha) = \beta \vee (\exists \gamma) \bigvee_{i=1}^{|F|} f_i(\alpha) = \gamma \wedge \kappa(\gamma, \beta)$$

The notation

$$\kappa_{f_{i_1}, f_{i_2}, \dots, f_{i_k}}(\alpha, \beta)$$

restricts the functions allowed in the definition to those indicated as subscripts; the set $\{f_{i_1}, f_{i_2}, \dots, f_{i_k}\}$ is a subset of F (occasionally it may be the entire F , mentioned explicitly for clarity). The notation

[†] It is prescribed that in a path from α to β no node can be visited more than once, with the only exception of α in a closed path from α to itself.

$\kappa^n(\alpha, \beta)$

requires a path of length n (i.e. $n-1$ compositions of functions).

The alphabets of the predicate calculus notation are now introduced:

- a. Punctuation marks: , ()
- b. Propositional connectives: $\sim \supset \equiv \wedge \vee$
- c. Quantifiers: $\exists \forall$
- d. Individual positive integer variables: $i j k l m n$
- e. Individual node variables: unsubscripted or subscripted lower case letters from the beginning of the alphabet. It is understood that each node variable is an element of precisely that domain indicated by the corresponding capital letter; thus, $a, a_1, a_7 \in A$
- f. Individual positive integer constants
- g. Individual node constant: ρ , denoting the root of an RLD
- h. Function letters: lower case letters from the end of the alphabet.

- i. Predicate letter: κ , the connectivity predicate, possibly with superscript or subscripts or both
- j. Predicate composition operator: \circ

The syntax of the valid expressions is defined in the usual way:

a. Terms

- constants and variables are terms
- expressions involving the application of functions to terms are terms
- no other expressions are terms

b. Atomic formulas

- expressions involving the application of predicates or compositions of predicates to terms are atomic formulas
- no other expressions are atomic formulas

c. Well-formed formulas (wffs)

- atomic formulas are wffs
- expressions relating wffs through logical symbols are wffs
- expressions involving wffs with quantifiers are wffs
- no other expressions are wffs

4. EXAMPLES

The notation introduced in the previous section is used for an axiomatic characterization of a number of sets of RLDs.

Initially, the entire RLD class is characterized by a set of axioms of the form

$$(\forall \cdot) \kappa(\rho, \cdot)$$

where, for each domain, the dot would be replaced by a variable from the domain.

The characterization of a set will include the enumeration of the existing domains and functions, the smallest instances of the set, and the axioms. The smallest instances must contain at least the root, due to the definition of RLDs (rootedness and the requirement that the set of nodes be non-empty).

a. Chains

domains - $A, \{\rho\}$

function - $x : A \cup \{\rho\} \rightarrow A$

smallest instance - ρ

axiom - $(\forall a) \sim \kappa(a, a)$

The axiom requires that there be no loop or cycle on any node from domain A. Note that there is no need to forbid closed paths on the root, because the root is not included in the range of x . Also, no axioms are required for the connectedness of the instances, because of the general RLD

axiom, or to forbid more than one x -labelled edge from each node, because edge labels denote (single-valued) functions. These remarks also apply to several of the sets to be presented in this section.

Stacks, queues, and other linear structures are chains; they can only be distinguished by the different operations allowed on them.

b. Rings

domains - $A, \{\rho\}$

function - $x : A \cup \{\rho\} \rightarrow A \cup \{\rho\}$

smallest instance - ρ^x

axiom - $\kappa(\rho, \rho)$

c. Binary trees

domains - $A, \{\rho\}$

functions - $x : A \cup \{\rho\} \rightarrow A$

$y : A \cup \{\rho\} \rightarrow A$

smallest instance - ρ

axioms - 1. $(\forall a_1)(\forall a_2) \sim ((\kappa_x^1(a_1, a_2) \vee \kappa_x^1 \circ \kappa(a_1, a_2))$

$\wedge (\kappa_y^1(a_1, a_2) \vee \kappa_y^1 \circ \kappa(a_1, a_2)))$

2. $(\forall a) \sim ((\kappa_x^1(\rho, a) \vee \kappa_x^1 \circ \kappa(\rho, a))$

$\wedge (\kappa_y^1(\rho, a) \vee \kappa_y^1 \circ \kappa(\rho, a)))$

3. $(\forall a) \sim \kappa(a, a)$

The first two axioms require that there be no more than one path between two pair of nodes, and the third axiom forbids closed paths.

d. Threaded binary trees

domains - $A, \{\rho\}$

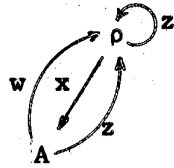
functions - $x : A \cup \{\rho\} \rightarrow A$

$y : A \rightarrow A$

$w : A \rightarrow A \cup \{\rho\}$

$z : A \cup \{\rho\} \rightarrow A \cup \{\rho\}$

smallest instance -



axioms - 1,2,3 - same as for binary trees, replacing κ by $\kappa_{x,y}$

$$4. (\forall a_1) \sim (\exists a_2) \kappa_x^1(a_1, a_2) \equiv (\exists a_3) ((\kappa_y^1(a_3, a_1)$$

$$\vee \kappa_y^1 \circ \kappa_x(a_3, a_1)) \wedge \kappa_w^1(a_1, a_3))$$

$$\vee (\kappa_x(\rho, a_1) \wedge \kappa_w^1(a_1, \rho))$$

$$5. (\forall a_1) \sim (\exists a_2) \kappa_y^1(a_1, a_2) \equiv (\exists a_3) ((\kappa_x^1(a_3, a_1)$$

$$\vee \kappa_x^1 \circ \kappa_y(a_3, a_1)) \wedge \kappa_z^1(a_1, a_3))$$

$$\vee (\kappa_x^1 \circ \kappa_y(\rho, a_1) \wedge \kappa_z^1(a_1, \rho))$$

$$6. \kappa_z^1(\rho, \rho)$$

$$7. (\exists a) \kappa_x^1(\rho, a)$$

A threaded binary tree [11] is a binary tree where if a node does not possess a left (right) son it is connected by a left (right) thread to its predecessor (successor) in postorder traversal, which is recursively defined by the algorithm:

1. traverse left sub-tree;
2. visit root;
3. traverse right sub-tree.

An instance of a threaded binary tree is shown in fig. 4.1.

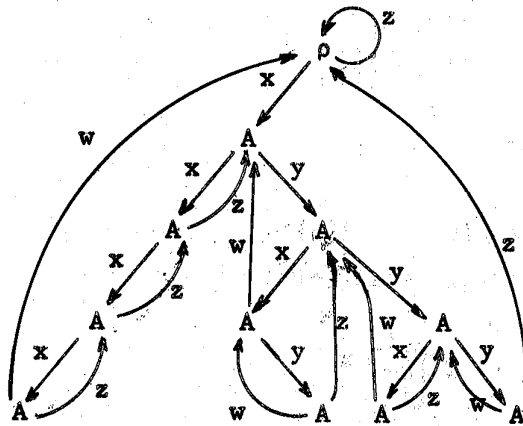


Fig.4.1: A threaded binary tree

It may be noted from the (recursive) algorithm that the first node of a right sub-tree to be visited is its leftmost node, i.e. the node reached from the root of the sub-tree through a path of x-labelled edges only which ends at the node; also, the root of a right sub-tree is reached from the root

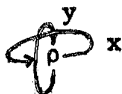
of a tree through one y-labelled edge. Accordingly, axiom 4 states that: "a node from domain A possesses no left son if and only if it possesses a w thread to a node which in turn reaches it through a single y-labelled edge (the case of a right sub-tree consisting of a single node) or through a path starting with a y-labelled edge followed by one or more x-labelled edges". Conventionally, the leftmost node of the entire tree has a w thread to the root.

Axiom 5 is the counterpart of axiom 4 for the right threads. Axioms 6 and 7 are needed for the characterization of the smallest instance; this example shows that, if desired, one may require the smallest instance to have other nodes besides the root.

e. Ring structures

domains - $A, \{\rho\}$

functions - $x : AU\{\rho\} \rightarrow AU\{\rho\}$
 $y : AU\{\rho\} \rightarrow AU\{\rho\}$

smallest instance - 

axioms - 1,2 - same as for binary trees
 3. $(\forall a) \kappa_x(a, a) \wedge \kappa_y(a, a)$
 4. $\kappa_x(\rho, \rho) \wedge \kappa_y(\rho, \rho)$

Ring structures [11] are obtained from binary trees by converting each x chain and y chain into a ring, as shown in fig. 4.2.

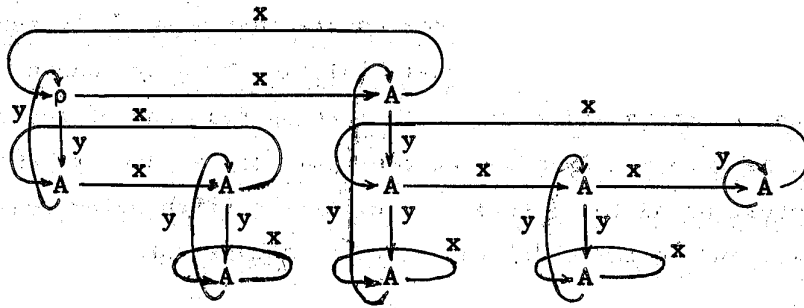


Fig. 4.2: A ring structure

f. Rectangular arrays

domains - $A, \{\rho\}$

functions - $x : A \cup \{\rho\} \rightarrow A$

$y : A \cup \{\rho\} \rightarrow A$

smallest instance - ρ

axioms - 1. $(\forall a_1)(\forall a_2)(\forall i)(\forall j)\kappa_x^i \circ \kappa_y^j(a_1, a_2) \equiv$

$$\kappa_y^j \circ \kappa_x^i(a_1, a_2)$$

2. $(\forall a)(\forall i)(\forall j)\kappa_x^i \circ \kappa_y^j(\rho, a) \equiv \kappa_y^j \circ \kappa_x^i(\rho, a)$

3. $(\forall a_1)(\forall a_2)(\forall a_3)(\forall i)(\forall j)\kappa_x^i(a_1, a_2) \wedge$

$$\kappa_y^j(a_1, a_3) \supset (\exists a_4)\kappa_y^j(a_2, a_4) \wedge \kappa_x^i(a_3, a_4)$$

4. $(\forall a_1)(\forall a_2)(\forall i)(\forall j)\kappa_x^i(\rho, a_1) \wedge$

$$\kappa_y^j(\rho, a_2) \supset (\exists a_3)\kappa_y^j(a_1, a_3) \wedge \kappa_x^i(a_2, a_3)$$

5. $(\forall a) \sim \kappa(a, a)$

The first four axioms ensure that whenever three nodes are connected by means of an x chain followed by a y chain or the reverse, or an x chain and a y chain running from one of the nodes, then there must be a fourth node and another pair of x and y chains, of the same lengths, so as to close a rectangle. The fifth axiom requires that the "rows" and "columns" of the array be chains rather than rings.

It should be noted that one-dimensional row or column arrays are permitted as special cases of rectangular arrays.

g. Sparse matrices

domains - $A, B, C, \{\rho\}$

functions - $x : A \rightarrow A \cup B$

$y : A \rightarrow A \cup C$

$u : B \rightarrow A \cup B$

$v : C \rightarrow A \cup C$

$w : C \cup \{\rho\} \rightarrow C$

$z : B \cup \{\rho\} \rightarrow B$

smallest instance - ρ

axioms - 1. $(\forall b)(\forall i)\kappa_z^i(\rho, b) \equiv (\exists c)\kappa_w^i(\rho, c)$

2. $(\forall b) \sim \kappa_z(b, b) \wedge (\kappa_u^1(b, b) \vee \kappa_u^1 \circ \kappa_x(b, b))$

3. $(\forall c) \sim \kappa_w(c, c) \wedge (\kappa_v^1(c, c) \vee \kappa_v^1 \circ \kappa_y(c, c))$

4. $(\forall a)(\exists b)(\exists c)\kappa_x(a, b) \wedge \kappa_y(a, c)$

5. $(\forall a_1)(\forall a_2) \sim ((\kappa_x(a_1, a_2) \wedge \kappa_y(a_1, a_2))$

$\vee (\kappa_x(a_1, a_2) \wedge \kappa_y(a_2, a_1)))$

6. $(\forall b_1)(\forall b_2)(\kappa_z(b_1, b_2) \supset ((\forall a_1)(\forall a_2)\kappa_x(a_1, b_1) \wedge \kappa_x(a_2, b_2) \supset \sim \kappa_y(a_2, a_1)))$
7. $(\forall c_1)(\forall c_2)(\kappa_w(c_1, c_2) \supset ((\forall a_1)(\forall a_2)\kappa_y(a_1, c_1) \wedge \kappa_y(a_2, c_2) \supset \sim \kappa_x(a_2, a_1)))$

An example of the representation of sparse matrices adopted here is shown in fig. 4.3.

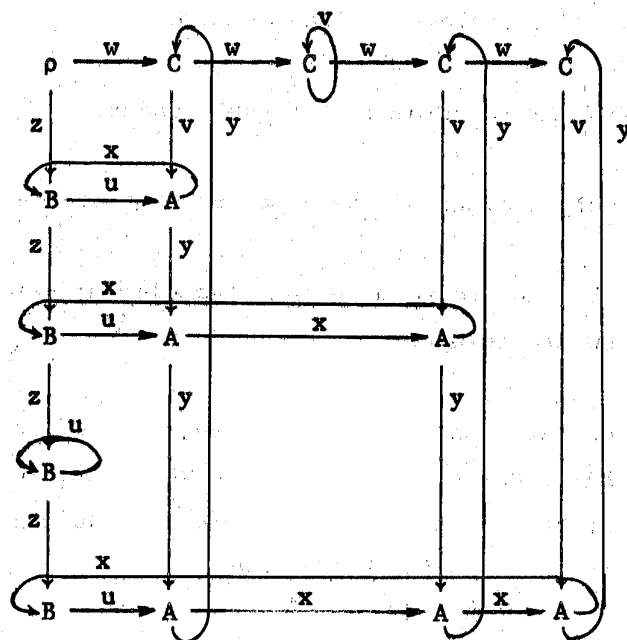


Fig. 4.3: A sparse matrix

In the representation the elements of the matrix belong to domain A, whereas B nodes and C nodes merely indicate the rows and columns, respectively.

The meaning of the axioms may be informally described as follows:

1. The matrix is square.
2. The z-labelled edges form a chain; on each B node there is a closed path which is either a u-labelled loop or starts with a u edge followed by one or more x edges.
3. Similar to axiom 2, for the C nodes.
4. Every element of the array belongs to one row and one column.
5. There can be no more than one element belonging to the same row and column.
6. Rows are consistently ordered, i.e. if a_1 belongs to a row that comes before the row to which a_2 belongs, then it may not be the case that a_1 and a_2 belong to the same column with a_2 coming before a_1 .
7. Similar to axiom 6, for columns.

5. STRUCTURAL OPERATIONS

In graph-theoretical terms, the structural operations are those involving the creation or deletion of nodes or edges.

Here a structural operation is considered "correct" if its application at some stage in the execution of a program to some instance of a set D of data structures results in another valid instance of D .

If the definition of D is part of the program, it is admissible that after the execution of an operation on an instance of D an automatic check of the validity of the resulting instance would take place.

Another strategy consists of pre-defining the set of operators that will be allowed for D . The description of an operator includes:

- a. The conditions under which the operator is applicable;
- b. The action of the operator, i.e. what creations and deletions of nodes and edges are caused by the operator.

One would then prove, for each operator $\#_i$ that, assuming that d is a valid instance of D and that the conditions for some application of $\#_i$ to d are verified, then all axioms in the definition of D hold in the resulting structure after the application of $\#_i$.

As in [12] it is possible to formalize these notions by expanding the predicate calculus notation so as to include individual variables for states, represented by a lower case sigma, possibly subscripted, and the operators themselves as functions acting on states, and represented by a possibly subscripted number sign. A third argument — a state — is added to the connectivity predicate.

As an example, let $\#_2$ be the operator on threaded binary trees which

- a. is applicable to nodes possessing a left thread, and therefore no left son;
- b. causes the creation of a left son with the appropriate threads.

From the algorithm defining postorder traversal it is immediate that the newly created node will point to its parent through its right thread. Also, since the new node, rather than its parent, has now become the leftmost node in a right sub-tree, it "inherits" from the parent its left thread. The operator is represented pictorially in fig. 5.1.[†]

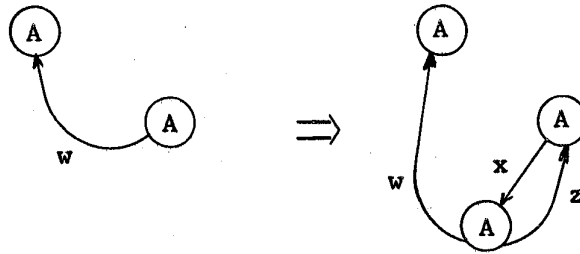


Fig.5.1: The operator $\#_2$

The operator is used in the wff below

$$\begin{aligned}
 & (\forall \sigma)(\forall a_1)(\forall a_2) \kappa_w^1(a_1, a_2, \sigma) \supset (\exists a_3) \kappa_x^1(a_1, a_3, \#_2(\sigma)) \\
 & \wedge \kappa_w^1(a_3, a_2, \#_2(\sigma)) \wedge \kappa_z^1(a_3, a_1, \#_2(\sigma))
 \end{aligned}$$

[†] The reader familiar with graph grammars will note that such an operator can be viewed as a production rule [4,13,14].

which says that "for all state σ , if node a_1 possesses a left thread, then it will possess a left son in the state attained by applying the operator $\#_2$ to state σ ."

One may now prove that if for a_1 and a_2

$$\kappa_y^1 \circ \kappa_x(a_2, a_1, \sigma) \wedge \kappa_w^1(a_1, a_2, \sigma)$$

held in state σ , then in state $\#_2(\sigma)$, since

$$\begin{aligned} & \kappa_y^1 \circ \kappa_x(a_2, a_1, \#_2(\sigma)) \quad (\text{unaffected by } \#_2) \\ & \wedge \kappa_x(a_1, a_3, \#_2(\sigma)) \\ & \wedge \kappa_w^1(a_3, a_2, \#_2(\sigma)) \end{aligned}$$

holds, and because of the transitivity of the connected relation,

$$\kappa_y^1 \circ \kappa_x(a_2, a_3, \#_2(\sigma)) \wedge \kappa_w^1(a_3, a_2, \#_2(\sigma))$$

holds, which constitutes part of axiom 4, and likewise the remaining axioms for threaded binary trees may be checked.

These proofs are usually tedious, because a local change performed at a given place of a structure often affects the paths between pairs of nodes that are not even explicitly referred to by the operator. Many different cases may have to be dealt with, for each axiom, in the course of a proof.

One may also prove that the set of operators formed by $\#_2$, $\#_1$ (represented in fig. 5.2), and $\#_3$ (defined as the counterpart of $\#_2$ for the creation of a right son) generates exactly the set of threaded

binary trees,

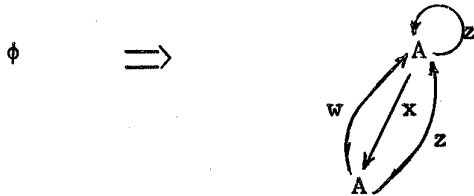


Fig.5.2: The operator $\#_1$, which creates from a "null" structure the smallest non-trivial threaded binary tree

The proof that this set of operators generates only threaded binary trees is done by induction on the stage of the generation, whereas the proof that all threaded binary trees are generated uses induction on the size of the structures (the size is often the number of nodes, but in other cases it may be convenient to consider the number of edges, or the number of components, etc.).

In practice, a set of operators that allows the generation of the intended structures may be expanded in order to permit other structural operations that the current application may require. In the example, the inverses of $\#_2$ and $\#_3$ might be introduced for performing the deletion of a left or right son, respectively, but this expansion would not enlarge the generative power of the set of operators. Certain modern programming languages have adopted the practice of incorporating the allowed operators in the definition of each set of data structures [15,16]

6. CONCLUSIONS

The proposed predicate calculus formalism is based on the connectivity relation, which is a well-understood graph property and one that can be tested efficiently [17].

As described, the formalism consists of two parts:

- a. the axioms that characterize a set of data structures statically, i.e. convey the properties that must remain invariant under any operation;
- b. the axioms that indicate the applicability and the action of the operators which transform a valid instance of a set of data structures into another valid instance of the same set.

The axioms of class a are non-constructive, in the sense that they do not give a direct indication of how an instance of the intended class can be generated. They can be used as assertions to be verified a posteriori, after a data structure has been operated upon, or a priori by characterizing the operators through the class b axioms and proving once and for all that their application preserves the class a axioms.

ACKNOWLEDGMENTS

The author is grateful to Prof. S.M. dos Santos from PUC and to Prof. J. Mylopoulos and C.C. Gotlieb from the University of Toronto for several helpful discussions.

REFERENCES

1. Burstall, R.M. - "Some Techniques for Proving Correctness of Programs which Alter Data Structures" in "Machine Intelligence" 7 - Edinburgh University Press (1972).
2. Hoare, C. - "Notes on Data Structuring" in "Structured Programming" - Academic Press (1972).
3. Standish, T. - "Data Structures - an Axiomatic Approach" - TR2639 - Bolt, Beranek and Newman (1973).
4. Furtado, A.L. - "Data Schemata Based on Directed Graphs" - Ph.D. thesis, University of Toronto (1974).
5. Earley, J. - "Toward an Understanding of Data Structures" - CACM, 14, 10 (1971) 617-627.
6. Rosenberg, A. - "Data Graphs and Addressable Schemes" - J. of Computer and Systems Sciences V, 6 (1971) 193-238.
7. Bachman, C. - "Data Structure Diagrams" - Data Base - ACM/SIGBDP 1,2 (1969).
8. Suppes, P. - "Axiomatic Set Theory" - Van Nostrand (1967).
9. Booth, T.L. - "Sequential Machines and Automata Theory" - John Wiley (1967).
10. Codd, E.F. - "A Relational Model for Large Shared Data Banks" - CACM 13,6 (1970) 377-387.

11. Knuth, D. - "The Art of Computer Programming" - vol. 1 - Addison-Wesley (1968).
12. Nilsson, N.J. - "Problem - Solving Methods in Artificial Intelligence" McGraw-Hill (1971).
13. Pfaltz, J. and Rosenfeld, A. - "Web Grammars" - TR69-84 - University of Maryland (1969).
14. Mylopoulos, J. - "On the Relation of Graph Grammars and Graph Automata"- Proceedings of the 13th SWAT (1972) 108-120
15. Dahl, O. - "Hierarchical Program Structures" in "Structured Programming"- Academic Press (1972).
16. Liskov, B. and Zilles, S. - "Programming with Abstract Data Types" - Proceedings of a Symposium on Very High Level Languages - ACM/SIGPLAN (1974) 50-59.
17. Corneil, D.G. - "The Analysis of Graph Theoretical Algorithms" - TR65 - University of Toronto (1974).