THEORY OF DATA STRUCTURES BY RELATIONAL AND GRAPH GRAMMARS

Václav Rajlich
Research Institute for Mathematical Machines
Loretánské nám. 3, 118 55 Prague 1, Czechoslovakia

In the paper, a definition for relational and graph grammars is given. Data structures and expressions are defined as a special kind of relational structure and tree, respectively. Examples illustrate the unifying power of the definitions. Four data manipulating commands are defined : assignment, conditional command, node creation, and edge creation. A method for proof of data structure algorithms is given; the method is based on Floyd's method.

1. INTRODUCTION

The paper may be viewed as a continuation of efforts to define data structures in exact mathematical terms, along the lines suggested in [1] , [2] , [15] , and [16] . The contribution of such efforts may be manyfold : the efforts do produce simple and clean models of often complex realities we encounter within data structures, and thus they may contribute to the development of conceptually simple but powerfull new programming tools. They also train programmers to think in unambigous terms (with less logical errors made as a result) and so they promote clean programming techniques. In the process of teaching programming techniques, they put many different techniques into one fold and hence they promote better grasp of problems and their solutions. Exact mathematical formalization also facilitates program proofs.

The basic mathematical formalism used is graphs or more general relational structures. They represent the so-called snapshots of a data structure, i.e. they consist of static interrelationships between various elements of a data structure. Computation on data structure is represented by a sequence of snapshots, each differing from the previous one by a local change caused by an action of processor.

Hence the whole apparatus reminds us of movies, where also a sequence of static snapshots represents a motion. For description of changes from one snapshot to the next one, we utilize the apparatus of graph grammars. However this apparatus was greatly simplified, particularly by use of definition 2.1 and consequent definitions of section 2, and hence it may be of interest on its own, particularly in context of papers dealing purely with formal properties of substitutions into graphs and relational structures, and in relation with graph grammars, cf. [5] , [6], [7], [8], [9], and [10].

It should be remarked that there exist at least two papers where apparatus of graph grammars is applied to data structures [5], [3]. However philosophy of this paper is distinct from both and it represents further elaboration on ideas expressed in [4].

In section 3, data, expressions and four basic manipulating commands are defined. The commands are assignment, if-statement, creation of a new edge, and creation of a new node. This set while being very small seems to be powerfull enough to all particular needs to manipulate data structures. Further discussion on this point is in section 3.

Sections 1 - 3 helped us to describe data structure at any given instant as a mathematical object. Section 4 capitalizes on this fact and introduces a method for proving properties of programs with data structures, which is a generalization of Floyd's method.

Some further elaborations on ideas of the paper will appear in [17].

2. RELATIONAL AND GRAPH GRAMMARS

Relational structures can be intuitively viewed as a set of static objects and a set of static relations between them. They are defined by the following definition :

Definition 2.1

Let L be a set of labels and X a set of nodes, then edge is a sequence of the type $\langle A, a_1, \dots, a_n \rangle$ where $n \geqslant 1$, $A \in L$ and $a_1, \dots, a_n \in X$. Relational structure over L (structure) is a set of edges. [x/] Oriented graphs with labeled edges are structures G consisting of edges of the type $\langle A, a_1, a_2 \rangle$ where $A \in L$, $a_1, a_2 \in$ Nod G. Similar definitions may be given for oriented graphs with labeled nodes, nonoriented graphs, and other graphs. Hence graphs are a special case of our notation and they will not be treated separately. For greater convenience of the reader, edges will be denoted by expressions of the type $A \langle a_1, \ldots, a_n \rangle$ where $A \in L$, $a_1, \ldots, a_n \in X$. In the following we shall also assume that each label is associated with a certain arity, i.e. for each label L there exists unique n such that only edges $L \langle a_1, \ldots, a_n \rangle \in S$.

Let S be a structure, then nodes of S (denoted Nod S) is a set of all nodes, explicitly used in the definition of S, i.e. Nod S = $\{ x \mid \text{there exists } A \langle a_1, \ldots, a_n \rangle \in S \text{ such that } x \in \{a_1, \ldots, a_n\} \}$. If $B \langle x \rangle \in S$, then x is called a labeled node. Observe that there may be several labels at one node. ∎

As an exercise, observe the following properties :

Observation 2.2

Let S, T be structures, Then :

$\text{Nod } (S \cap T) \subset \text{Nod } S \cap \text{Nod } T$

$\text{Nod } (S \cup T) = \text{Nod } S \cup \text{Nod } T$

$\text{Nod } (S - T) \supset \text{Nod } S - \text{Nod } T$. ∎

Let us have the following example :

Example 2.3

Consider a list whose picture in style of [13] is on figure 1.

---

x/ In the literature, relational structure is usually defined as a couple of sets, where the first set is set of nodes and the second set is set of edges. Definition 2.1 enables us to use set-theoretical operations (union, intersection, complement) on structures and this is the main reason for its use. On the other hand, it doesn't allow to define isolated unlabeled nodes. However from the following text it is clear that unlabeled isolated nodes do not have any reasonable intuitive meaning anyhow, so the loss is not great. Empty set is also a structure, called empty structure.

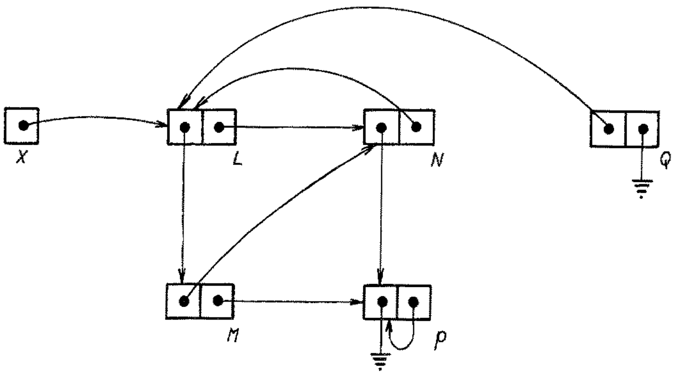

Fig. 1

For purposes of further explanations, atoms of the list were denoted by letters L up to Q. The list may be represented by a relational structure in several ways. We shall choose a method, where atoms of the list are represented by nodes of the structure. Left and right links will become binary edges labeled by A, and B, respectively. Pointer X will be represented by a unary edge . NIL will be represented by both node NIL and unary edge NIL ⟨NIL⟩. (The purpose of such dual denotation will be clear after definition of expression and its value, see definition 3.4.) The list is hence described by a structure

S = { X ⟨L⟩, NIL ⟨NIL⟩, A ⟨L,M⟩, B ⟨L,N⟩,
A ⟨M,N⟩, B ⟨M,P⟩, A ⟨N,P⟩, B ⟨N,L⟩,
A ⟨P, NIL⟩, B ⟨P,P⟩, A ⟨Q,L⟩, B ⟨Q, NIL⟩} ,

see also fig. 2. ∎

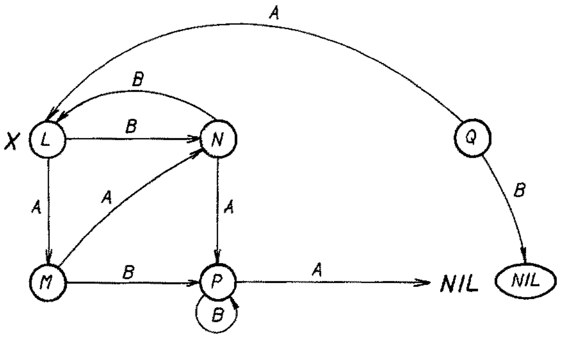

Fig. 2

Graphical notation for structures is for most parts self-explanatory, see fig. 2. Nodes are denoted by circles with the name of the node inside. Unary edges are denoted by labels separated by commas in front of the corresponding node. Binary edges are represented by labeled arrows. Ternary edges are denoted by a generalization of arrow, see for example fig. 9, etc. Whenever names of nodes are not interesting, nodes will be represented by unnamed circles, distinct circles always representing distinct nodes.

We shall define label-preserving homomorphism between structures in the following way :

Definition 2.4

Let S, T be structures and let h : Nod S $\rightarrow$ Nod T be a function such that for every edge $A\langle a_1, \ldots, a_n\rangle \in S$, there exists an edge $A\langle h(a_1), \ldots, h(a_n)\rangle \in T$. Then h is homomophism and it is denoted $h : S \rightarrow T$.

Let $h : S \rightarrow T$ be a homomorphism and let $R \subset S$, then $g : R \rightarrow T$ is restriction of h to R (denoted $h \mid R$), iff for every $a \in$ Nod R, $g(a) = h(a)$.

Let X, Y be structures such that $S \subset X$, $T \subset Y$. Then $f : X \rightarrow Y$ is extension of h to X, iff for every $a \in$ Nod S, $f(a) = h(a)$. Homomorphic image of S, denoted by h (S), is the structure

$h(S) = \{A\langle h(a_1), \ldots, h(a_n)\rangle \mid A\langle a_1, \ldots, a_n\rangle \in S\}$.

Substitution into relational structure is defined with the help of productions. Formally this is done in the following definition :

Definition 2.5

A production is a couple of structures $p = \langle L,R\rangle$, denoted also $L \Rightarrow R$ where L and R are called left and right side, respectively.

Let us have structure S, production $p = \langle L,R\rangle$ and homomorphism $h : L \rightarrow S$. Then substitution into S according to production p and homomorphism h gives S' (denoted $S \Rightarrow_{h(p)} S'$) iff the following holds :

i) for the homomorphism h we shall find an extension $\bar{h}$ to $R \cup L$ which must satisfy requirement Nod $\bar{h}(R-L) \cap$ Nod $S = \emptyset$

(i.e. new nodes must not be identified with the old ones).[x/]

ii) $S' = (S - h(L)) \cup \bar{h}(R)$. ∎

Venn diagram of this situation is on fig. 3, where the shadowed area represents S'.

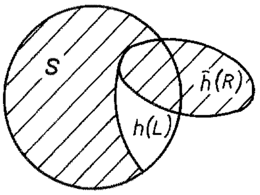

Fig. 3

Example 2.6

Let us have structure $S = \{A\langle 1,2\rangle, B\langle 2\rangle, A\langle 2,3\rangle, A\langle 3,1\rangle\}$ and production $p = \{A\langle 11,12\rangle, B\langle 11\rangle\} \Rightarrow A\langle 12,13\rangle$.

Then there exists a unique homomorphism $h : L \to S$, defined in the following way : $h(11) = 2$, $h(12) = 3$.

The right side R has new node 13. Define $\bar{h}(13) = 4$ and than condition (i) of definition 2.5 is satisfied. Then $h(L) = \{A\langle 1,2\rangle, B\langle 2\rangle\}$ and $\bar{h}(R) = A\langle 3,4\rangle$.

Then $S \Rightarrow_{h(p)} S'$ where $S' = (S - h(L)) \cup \bar{h}(R) = A\{\langle 1,2\rangle, A\langle 3,1\rangle, A\langle 3,4\rangle\}$, see fig. 4. ∎

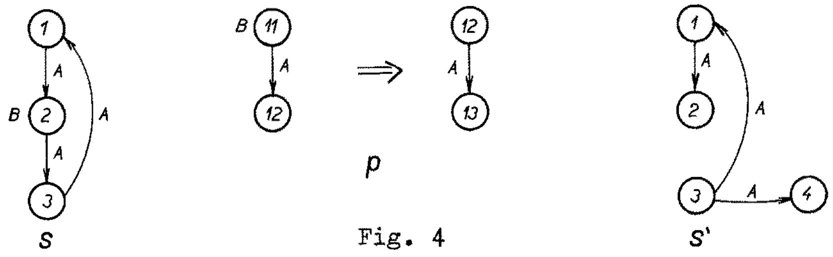

Fig. 4

x/ In certain situations not arising in this paper, where Nod R-Nod L contains more then one node, we shall usually require the following additional condition :

if $x, y \in \text{Nod } R - \text{Nod } L$ and $x \neq y$, then $h(x) \neq h(y)$.

Substitution of Definition 2.5 may be extended in the following way :

Definition 2.7

For a production p, $S \Rightarrow_p S'$ iff there exists a homomorphism h such that $S \Rightarrow_{h(p)} S'$. Let P be a set of productions. Than $S \Rightarrow_P S'$ iff there exists $p \in P$ such that $S \Rightarrow_p S'$.

Inductively let us define the following notation :

$S \Rightarrow_P^0 S'$ iff $S = S'$. For $i = 0,1,2\ldots,$

$S \Rightarrow_P^{i+1} S'$ iff $S \Rightarrow_P^i S''$ and $S'' \Rightarrow_P S'$.

Finally $S \Rightarrow_P^* S'$ iff there exists $i \geq 0$ such that $S \Rightarrow_P^i S'$.

Context-free relational grammar is a system $G = \langle S,P \rangle$ where S is a finite set of finite structures, P is a finite set of finite productions and left sides of all productions of P are created by a unique edge. (Productions of P are called context-free productions.) Labels of all edges appearing on a left side are called nonterminals and other labels are called terminals. Structure X is generated by context-free grammar G iff $S \Rightarrow_P^* X$ and all labels of X are terminal.

An example of context-free grammar is in example 3.5.

A notation saving vehicle is the use of variables for substructures. Whenever a substructure is connected to the rest by n nodes, it may be denoted by an n-ary edge labeled by a variable. Always a rule will be given which says what kind of substructure may replace the variable. For example, let $\tau\langle 1 \rangle$ denote substructure $\{ B\langle 1,10 \rangle, B\langle 10,11 \rangle, B\langle 11,1 \rangle \}$. Then structure $\{ \tau\langle 1 \rangle, A\langle 1,2 \rangle \}$ denotes structure $\{ B\langle 1,10 \rangle, B\langle 10,11 \rangle, B\langle 11,1 \rangle, A\langle 1,2 \rangle \}$.

3. DATA STRUCTURES

In this section, we shall define data structures, expressions and four basic manipulating commands for data structures. We shall also briefly discuss reasons for the choice of the set of manipulating commands. First we shall introduce sequence of definitions 3.1 - 3.4 and theorem 3.3. The reader while reading the definitions may consult examples 3.5 - 3.10. Examples are here to show universality of definitions.

Data structure is defined in the following definition :

Definition 3.1

Let S be a structure $n \geqslant 1$, then n-ary label L is an operation in S, iff for every (n-1) tuple of nodes $\langle a_1, \ldots, a_{n-1} \rangle$ there exists at most one node a_n such that $L\langle a_1, \ldots, a_n \rangle \in S$. Nodes $a_1, \ldots, a_{n-1}$ are called operands, node a_n is called result.

If L is unary label and it is also an operation, then it is called entry point. (Label C is entry point iff $C\langle x \rangle$, $C\langle y \rangle \in S$ implies $x = y$).

Structure D is data structure iff D is a structure and all its labels are operations. ∎

The previous definition goes hand in hand with the following definition of expression. We shall define expression as a tree, where each leaf is a labeled node :

Definition 3.2

Expression is a structure T generated by a context-free relational grammar $G = \langle \mathcal{T}\langle x \rangle, P \rangle$ where $\mathcal{T}$ is a nonterminal and P is a set of productions of two types :

$$\alpha_1\langle 1 \rangle \Rightarrow \{ B\langle 2, \ldots, n, 1 \rangle, \alpha_2\langle 2 \rangle, \ldots, \alpha_n\langle n \rangle \},$$
$$\alpha_1\langle 1 \rangle \Rightarrow C\langle 1 \rangle$$

where $\alpha_1, \ldots, \alpha_n$ are nonterminals and B,C are terminals.

Node x is called root of the tree. Edge $A\langle a_1, \ldots, a_{n-1}, x \rangle$ of the tree is called rooted edge. ∎

In the literature, expressions are often defined as strings rather than trees. All such strings are in fact encodings of trees into a linear form. For relevant theory, see for example [12] .

Observe the following property of data structures and expressions :

Theorem 3.3

If T is expression and D data structure, then there exists at most one homomorphism $h : T \rightarrow D$.

Proof.

First prove the following statement :

(a) If $\{\tau\langle 1\rangle, \tau\langle 2\rangle\} \Rightarrow^{n}_{p} S$ and P are context-free productions, then there exist mutually disjoint structures S_1, S_2 such that $S_1 \cup S_2 = S$, $\tau\langle 1\rangle \Rightarrow^{*}_{p} S_1$, $\tau\langle 2\rangle \Rightarrow^{*}_{p} S_2$.
This statement is proved by induction on n.

As a consequence, we have the following statement concerning expressions :

(b) For every expression T with rooted edge $A\langle a_1, \ldots, a_n\rangle$, there exist mutually disjoint expressions $T_1, \ldots, T_{n-1}$ such that $A\langle a_1, \ldots, a_n\rangle \notin T_1 \cup \ldots \cup T_{n-1}$, $T = A\langle a_1, \ldots, a_n\rangle \cup T_1 \cup \ldots \cup T_{n-1}$, and $a_1, \ldots, a_{n-1}$ are roots of expressions $T_1, \ldots, T_{n-1}$.

The proof of theorem 3.3 is by induction on the number of edges in the expression.

Base :

Suppose expression T has only one edge. Then the edge is unary one and it has the form $A\langle x\rangle$. If $h : T \to D$ exists, then $h(T)=A\langle y\rangle$ where $y \in \text{Nod } D$. Then A is an entry point for D, and hence by definition 3.1 there exists a unique node y such that $A\langle y\rangle \in D$, and the theorem is true.

Induction step :

Suppose the expression T has $m > 1$ edges and for every expression with $m-1, m-2, .., 1$ edges, theorem 3.3 is true. Then as a consequence of (b), $T = A\langle a_1, \ldots, a_n\rangle \cup T_1 \cup .. \cup T_{n-1}$ where number of edges in $T_1, \ldots, T_{n-1}$ is at most $m-1$. Hence for each of the subexpressions, induction assumption holds and there exists at most one homomorphism $h_1 : T_1 \to D, \ldots, h_{n-1} : T_{n-1} \to D$.

If all homomorphisms $h_1, \ldots, h_{n-1}$ do exist, then define $h : T \to D$ in the following way : Let $h \mid T_1 = h_1, \ldots, h \mid T_{n-1} = h_{n-1}$. By (b), nodes $a_1, \ldots, a_{n-1}$ are roots of $T_1, \ldots, T_{n-1}$.
Hence $h(a_1), \ldots, h(a_{n-1})$ are uniquelly defined. In data structure D, there exists at most one edge $A\langle h(a_1), \ldots, h(a_{n-1}), y\rangle \in D$ and hence there exists at most one homomorphism $h : T \to D$. ∎

Finally let us give the following definitions :

Definition 3.4

Let T be an expression, D data, and let h : T $\rightarrow$ D be a homomorphism, then value of expression T in D is the homomorphic image of the root of T. Let D be data, then accessible part of data D is set of all edges g, for which there exists an expression T and homomorphism h : T $\rightarrow$ D, such that $g \in h(T)$. Expression T has value in data D iff there exists homomorphism h : T $\rightarrow$ D. ∎

The definitions 3.1 up to 3.4 will be illustrated by the following examples :

Example 3.5

In this example, we shall return to lists and to data structure S of fig. 2 and example 2.3. Generally speaking, lists are data structures consisting of nodes and binary edges, called links. There is a special node NIL and an entry point NIL $\langle NIL \rangle$. Each node except NIL has complete set of links emanating from it. There is no link emanating from NIL. There also must be some entry points other than NIL, for example X in fig. 2.

Expressions relevant to this example are derived by a grammar $G = \langle \tau \langle x \rangle, P \rangle$ with the only nonterminal τ where P consists of the following productions :

$$\tau \langle 1 \rangle \Rightarrow \{ A \langle 2,1 \rangle , \tau \langle 2 \rangle \},$$
$$\tau \langle 1 \rangle \Rightarrow \{ B \langle 2,1 \rangle , \tau \langle 2 \rangle \},$$
$$\tau \langle 1 \rangle \Rightarrow X \langle 1 \rangle .$$

Examples of expressions are in fig. 5.

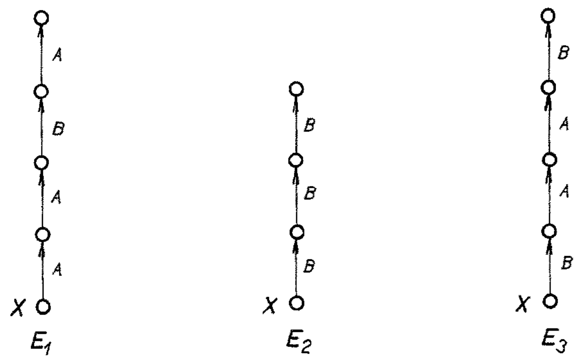

Fig. 5

Value of E_1, E_2, and E_3 in data structure of fig. 2 is M,L, and undefined, respectively. Note that accessible part of S is $S - \{A\langle Q,X\rangle, B\langle Q, NIL\rangle\}$.

For expressions of this example, we shall use a linear notation based on concatemation, where the entry point is followed by a sequence of links separated by dots. For example, E_1, E_2, and E_3 of fig. 5 wil be denoted by X.A.A.B.A, X.B.B.B, and X.B.A.A.B, or $X.A^2.B.A$, $X.B^3$, and $X.B.A^2.B$, respectively.

Let $A^o = \lambda$ and for every expression T, $T . \lambda = T$.

Let U,V be sets of link labels, then $U.V = \{C.D \mid C \in U, D \in V\}$. Let $U^o = \lambda$ and for every $i \geqslant 1$, $U^{i+1} = U.U^i$. Let $U^* = \bigcup_{i=0}^{\infty} U^i$. ∎

Example 3.6

Often used data structure is arithmetics of integers (denoted I). It is defined in the following way :

Nod I = $\{0,1,-1,2,-2, \dots\}$

Entry points : $E = \{0\langle 0\rangle, 1\langle 1\rangle, 2\langle 2\rangle, \dots, 10\langle 10\rangle\}$
Other operations are $+, -, *, \uparrow, <, \leqslant, =, \neq, \geqslant, >$ defined in analogy to the following two examples :

$R^+ = \{+\langle a,b,c\rangle \mid a + b = c\}$,
etc.
$R^{\leqslant} = \{\langle a,b,0\rangle \mid a \leq b\}$,
etc.

Then $I = E \cup R^+ \cup R^- \cup R^* \cup R^{\uparrow} \cup R^< \cup R^{\leqslant} \cup R^= \cup R^{\neq} \cup R^{\geqslant} \cup R^>$.

Example of expression $5 * 10\uparrow 2 + (7 * 10 + 2)$ is in fig. 6. Its value in I is 572. Accessible part of I is the whole data structure I.

Lists of the previous example may be combined with I in such a way that some pointers of the list point to a node of I. Example of such case appears in section 4.∎

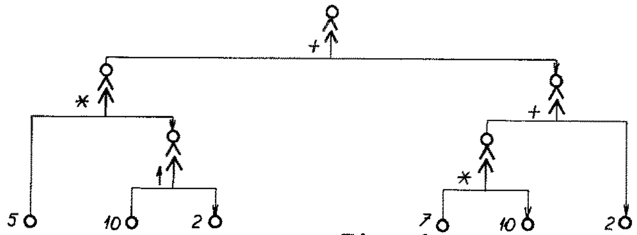

Fig. 6

Example 3.7

Arithmetics of integers is often used in connection with identifiers of Algol-like programming languages. Data structure D is then defined formaly in the following way :

$$D = \{ \alpha_1 \langle x_1 \rangle , \alpha_2 \langle x_2 \rangle , \dots , \alpha_n \langle x_n \rangle \} \cup I$$

where $\alpha_1, \alpha_2, \dots, \alpha_n$ are new entry points denoted by identifiers. Let us have a data structure

$D = \{ANNA \langle 19 \rangle , FX2 \langle 2 \rangle , R \langle 3 \rangle \} \cup I$ and expression

(R + 3) * FX2 + ANNA, see fig. 7.

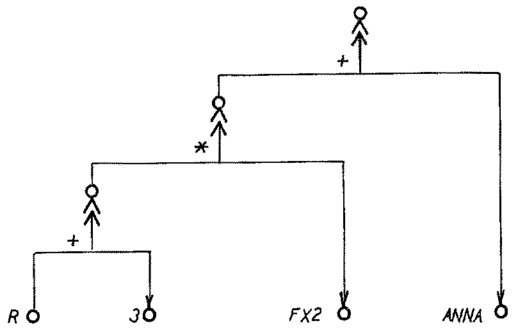

Fig. 7

Value of the expression in D is 31. ∎

Example 3.8

Arrays are characterized by a special operation called "indexing operation" symbolically denoted by square brackets [ ] , which to a given identifier and given integer finds another integer as a value. Suppose we have an array A [1] , A [2] , ..., A [N] , then it will be represented by a data structure

$$D = \{ A \langle A \rangle , [\ ] \langle A,1,x_1 \rangle , [\ ] \langle A,2,x_2 \rangle , \dots , [\ ] \langle A,N,x_N \rangle \} \cup I,$$

where $x_1, x_2, \dots, x_N \in$ Nod I and A is an entry point symbolizing name of the array.

Expression on fig. 8 is in a standard linear form expressed as A [1] * A [2 + A [3]] . ∎

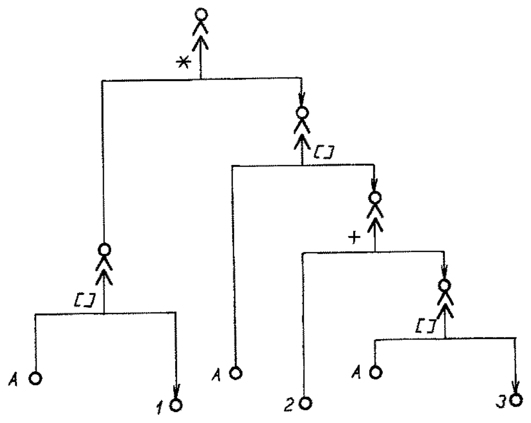

Fig. 8

In the rest of this section, we shall define commands and programs which manipulate data structures. We shall use four basic data manipulating commands. They are in fact productions which change a data structure into another one. Although the movement of control through a program may also be expressed in terms of productions, for easier reading it will be left out and explained in different terms.

There are four commands : assignment, edge creation, node creation and conditional command.

Intuitive explanation of assignment follows the definition.

Definition 3.9

Assignment is a production of the type

$\{U\langle 1\rangle, V\langle 2\rangle\} \Rightarrow \{V\langle 2\rangle, U'\langle 2\rangle\}$ where U, V, U' are expressions and there exists an isomorphism i : $U \rightarrow U'$ such that for every node y except root, $i(y) = y$. Assignment will be usually written in form U := V.

Edge creation is a production of the type

$\{T_1\langle 1\rangle, T_2\langle 2\rangle, \ldots, T_n\langle n\rangle, V\langle 0\rangle\} \Rightarrow \{T\langle 0\rangle, V\langle 0\rangle\}$

where $T_1\langle 1\rangle, T_2\langle 2\rangle, \ldots, T_n\langle n\rangle$ are subexpressions of expression $T\langle 0\rangle$. It will be usually written in a form CREATE T := V.

Node creation is a production of the type $T\langle 1\rangle \Rightarrow T\langle 2\rangle$ where T is an expression. It will be usually written in a form CREATE T.

Conditional command is a couple $\langle T, m\rangle$ where T is expression, m a natural number.

Program P is a sequence $\langle p_1, p_2, \dots, p_N\rangle$ of commands. State is a couple $\langle D,i\rangle$ where D is data structure and $i \in \{1, \dots, n\}$. Then $\langle D,n\rangle \Rightarrow_p \langle D', n'\rangle$ iff the following holds :

i) if p_n is assignment, edge creation or node creation, then $D \Rightarrow_{p_n} D'$ and $n' = n + 1$

ii) if p_n is conditional command $\langle T, m\rangle$, then $D = D'$ and if there exists homomorphism $h : T \rightarrow D$, then $n' = m$, otherwise $n' = n + 1$. ∎

Let us return briefly to the formalism for assignment. There we had expressions U and V. Assignment means that the data structure is changed in such a way that after the change, value of U will become equal to value of V. The change concerns only the homomorphic image of rooted edge of U. All other edges of data structure are unchanged.

Of course, in standard programming languages, not every expression's value may be changed in this fashion. We should distinguish access expressions, whose values may be changed during the computation. Examples of access expressions are variables (ALFA, F1, etc.), array expressions (A [1] , A [B [2] + 3] etc.), or access paths in lists (X.A.A.B, etc.). Examples of expressions which may not be changed by a computation are 1 + 1, 3 * A [2] , NIL, etc. Each programming language makes distinction between these two kinds of expressions.

Observe the following technical lemma :

Lemma 3.10

Let $\langle D, n\rangle \Rightarrow_p \langle D', n'\rangle$ where D is a data structure, then D' is also data structure. ∎

It may be observed that all actions which mean removal of a node or an edge from the relational structure are omitted from the basic four commands. We have done this on the basis of the fact that once a node or an edge is no longer accessible, then from the programming point of view it is the same like when it does not exist. This is particularly clear in view of the following lemma :

Lemma 3.11

Let D be a data structure, and let for some n and P $\langle D,n\rangle \Rightarrow_p \langle D', n\rangle$. Let S = D - accessible part of D, S' = D' - accessible part of D', then $S' \supset S$.∎ In practical programming with lists of a very general nature, node removal is a complex operation. It means to look through the whole list and disconnect all links to the atom being removed. Hence it does not possess "local" nature of the other commands and this is another reason for its deletion from our list of four basic commands.

Of course, even edge creation and node creation could be deleted in view of the fact that all edges and nodes to be used may be prepared in advance as a part of free memory. Their relationship to the rest of data structure could be easily realigned by assignment. The very basic set of commands for data structures programming is thus assignment and conditional command.

Finally let us introduce the following definition :

Definition 3.12

Let $P = \langle p_1, \ldots, p_N\rangle$ be a program, D a data structure. Then $D \Rightarrow_P^+ D'$ iff $\langle D,1\rangle \Rightarrow_p \ldots\ldots \Rightarrow_p \langle D', N+1\rangle$.

4. PROGRAM PROOFS

In [11], Floyd's method for program proofs was described for machines with variables and arrays. In this section, we shall extend the method to general data structures.

Let us repeat briefly basic notions of Floyd's method in data structures setting. They are illustrated by example 4.1.

Let us have program $P = \langle p_1, \ldots, p_n\rangle$. Let φ be an input assertion, i.e. a statement describing all data structures admissible to program P. Let ψ be an output assertion, i.e. a statement describing relationship of input data structure and output data structure at the completion of program execution.

We shall find the so-called cutpoints in the program, which are :

a/ begin and end

b/ additional places such that each loop of the program contains at least one cutpoint [11, p. 171].

We shall find the set of paths from one cutpoint to another one such that no third cutpoint is involved. For each of the paths π we shall find Condition C_π and function F_π , i.e. a property data must satisfy in order the path to be taken, and the properties of the data at the end of the path expressed in terms of the data at the beginning of the path, respectively.

Each cutpoint will be associated with an inductive assertion, i.e. a statement describing properties of data whenever control moves through the cutpoint. For begin and end of the program, we have input and output assertion, respectively. For other cutpoints, inductive assertions must be supplied by "educated guess". For proof of partial correctness, we shall make verification for every path π , i.e. from inductive assertion for the starting cutpoint of path π , from condition C_π, and function F_π , we must arrive to inductive assertion for the final cutpoint of path π .

For proof of termination of a loop, we shall find a well-founded set W, [11, p. 183] , i.e. a set W together with relation $\prec$ which is transitive, antisymetric and there is no infinite chain of elements $a_1 \succ a_2 \succ a_3 \succ \ldots$. We shall find a partiall function G from all data structures to W (again the well-founded set and function G must be supplied by an educated guess). We have to verify the following statement :

For every path of a loop π and every data D, for which G (D) is defined, G (D) $\succ$ G (F_π (D)). Also we must verify that G (D) is defined for entry point of the loop.

The method is illustrated by the following example. For assertions, we shall use an informal assertion language based on example 3.5. Formal assertion language for Rosenberg's data structures [15] appeared in [14] .

Example 4.1

We shall give a proof of a program for lists marking. Marking is part of the algorithm for garbage collection, see [13] and its purpose is to mark all accessible atoms of a list. The marking will be done by program P of fig. 9. (It is given in a form of flowchart for easier readability.) Cutpoints in it are { START, 2, END } .

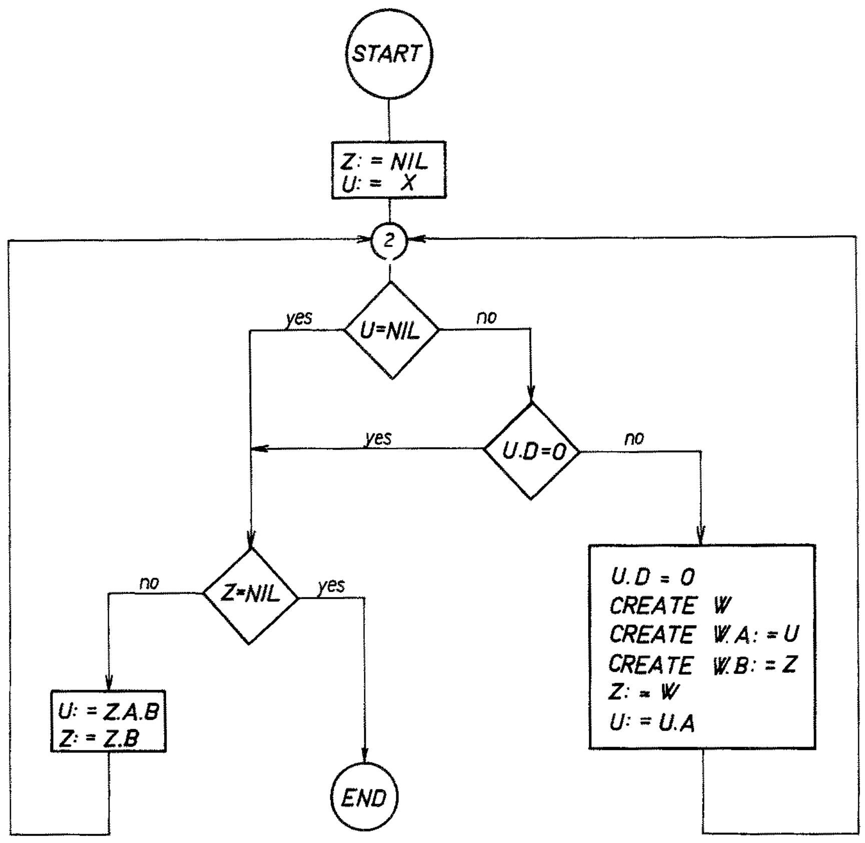

Fig. 9

The program does the following thing :

The original list has entry point X, links A,B, and D, where link D points to arithmetics of integers. We shall consider values stored on links D at the beginning to be different from 0; value 0 will mean the marking. Example of the list without links D is on fig. 2.

During the computing, we shall use a stack consisting of entry points Z, W, and links A and B, see fig. 10. Also we shall use entry point U which moves through the list. Marked nodes in fig. 10 are denoted by black circles.

All paths of program P with their conditions and functions are in fig. 11. In it, data structures, values of expressions, and links before and after taking the path are denoted by subscript i and j, respectively. Elements unchanged by a path are not listed. Data structure under consideration is denoted H.

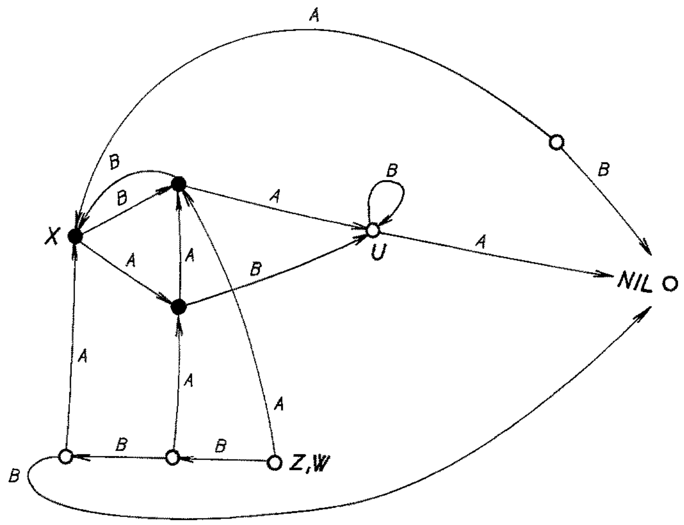

Fig. 10

Path	from	to	condition C π	function F π
π	cutpoint			
p	START	2	TRUE	$U_j = X_i$
q	2	2	U_i = NIL and $Z_i \neq$ NIL	$U_j = Z_i.A_i.B_i$, $Z_j = Z_i.B_i$
r	2	2	$U_i \neq$ NIL and $U_i.D_i = 0$ and $Z_i \neq$ NIL	-"-
s	2	2	$U_i \neq$ NIL and $U_i.D_i \neq 0$	$U_i.D_j = 0$, $W_j \not\in$ Nod H_i, $W_j.A_j = U_i$, $W_j.B_j = Z_i$, $Z_j = W_j$, $U_j = U_i.A_i$
t	2	END	$U_i \neq$ NIL and $U_i.D_i = 0$ and Z_i = NIL	-
u	2	END	U_i = NIL and Z_i = NIL	-

Fig. 11

For proof of partial correctness, we shall associate each cutpoint with an inductive assertion in the following way : (In the following, $T \in X . \{A,B\}^*$.)

ξ^{START} : For every expression T.D $\neq$ 0.

ξ^{2} : For every expression, one of the following conditions holds :

(i) T.D = 0

(ii) for some T' $\in \{A,B\}^{*}$ and some k $\geqslant$ 0,
Z.B.A.T' = T

(iii) T = U.T' and U.D $\neq$ 0.

ξ^{END} : For every expression T, T.D = 0.

As an example, let us verify path s which goes from cutpoint 2 back to cutpoint 2.

Let us rewrite all our basic assumptions for s. Ivariant ξ^{2} will appear with indices i as an inductive assumption. We have to prove that the same invariant with indices j holds.

ξ^{2}_{i} : (i) $T_i.D_i = 0$

(ii) $Z_i.B_i^{k}.A_i.T_i' = T_i$

(iii) $T_i = U_i.T_i'$ and $U_i.D_i \neq 0$

C_s : $U_i \neq$ NIL and $U_i.D_i \neq 0$

F_s : $U_i.D_j = 0$, $W_j \notin$ Nod D_i, $W_j.A_j = U_i$, $W_j.B_j = Z_i$,
$Z_j = W_j$, $U_j = U_i.A_i$

We shall verify each step separately : Observe that for every T $\in$ X.$\{A,B\}^{*}$, $T_i = T_j$. For every link A_i, B_i of H_i, $A_i = A_j$, $B_i = B_j$.

(i) If $T_i.D_i = 0$, then also $T_j.D_j = 0$ and ξ^{2}_{j} holds.

(ii) Suppose $T_i = Z_i.B_i^{k}.A_i.T_i'$. Then $Z_i = W_j.B_j = Z_j.B_j$, and
$T_i = T_j = Z_j.B_j.B_i^{k}.A_i.T_i' = Z_j.B_j.B_j^{k}.A_j.T_i' = Z_j.B_j^{k+1}.A_j.T_j'$
and ξ^{2}_{j} holds.

(iii) Suppose $T_i = U_i.T_i'$ and $U_i.D_i = 0$. Then we have to verify three separate cases :

(a) $T_i = U_i$, then $U_i.D_j = 0$, hence $T_i.D_j = T_j.D_j = 0$
and ξ^{2}_{j} holds.

(b) $T_i = U_i.A_i.T_i'$, then $T_i = T_j = U_j.T_i' = U_j.T_j'$ and ξ^{2}_{j} holds.

(c) $T_i = U_i.B_i.T_i'$, then $T_i = T_j = W_j.A_j.B_i.T_i' =$
$= Z_j.A_j.B_j.T_j' = Z_j.B_j^{o}.A_j.B_j.T_j' =$
$= Z_j.B_j^{o}.A_j.T_j''$ where $T_j'' = B_j.T_j'$ and
ξ_j^2 holds.

Verification of other paths is done in a similar way.

For proof of termination, we shall use the following well--founded set :

$W = \{ \langle a,b \rangle \mid a,b \geqslant 0$, a, b, are integers$\}$ and lexicographic ordering, i.e. $\langle a,b \rangle < \langle c,d \rangle$ iff $a < c$ or $a = c$ and $b < d$. Let E be set of all possible lists, then $F : E \rightarrow W$ is given by the following relationship :

$F(D) = \langle a,b \rangle$ iff a is number of accessible unmarked nodes and b is number of nodes of the stack.

Then if data D are data in the cutpoint 2 on the beginning of one traversal of the loop, and D' data in the cutpoint 2 on the end of the traversal, then $F(D') < F(D)$ and hence the program terminates.∎

REFERENCES

1. Jay Early : Towards an understanding of data structures, Comm. ACM, vol. 14, 1971, 617-626.

2. A.C. Fleck : Towards a theory of data structures, J. Computer and System Sci, 5, 1971, 475-488.

3. A.L. Furtado : Characterizing sets of data structures by graph grammars, Proc. of conference on computer graphic, pattern recognition and data structure, May 14-16, 1975, Univers. of California, Los Angeles, IEEE Catalog Number 75 CH 0981-1C, 103-107.

4. Václav Rajlich : Relational definition of computer languages, in J. Bečvář edited, Mathematical Foundations of Computer Science 1975, September 1-5, 1975, Mariánské Lázně, Czechoslovakia, Lecture Notes in Computer Science vol. 32, Springer Verlag, Berlin, 1975, 362-376.

5. Hans Jürgen Schneider : Syntax-directed description of incremental compilers, in D. Siefkes edited, GI-4, Jahrestagung, Berlin, 9-12. October 1974, Lecture notes in computer science, vol. 26, Springer Verlag, Berlin, 1975, 192-201.

6. Terrence W. Pratt : Pair grammars, graph languages and string--tograph translations, J. Computer and System Sci, vol.5, December 1971, 560-595.

7. A. Rosenfeld and D.C. Milgram : Web automata and web grammars, Machine Intelligence vol. 7, 1972, University of Edinburgh Press, 307-324.

8. Václav Rajlich : Relational structures and dynamics of certain discrete systems, in Proc. Symposium on Mathematical Foundations of Computer Science, High Tatras, Sept. 3-8, 1973, available from Computing Research Centre, Bratislava, Czechoslovakia, 285-292.

9. Václav Rajlich : Dynamics of discrete systems and pattern reproduction, J. Computer and System Sci., 11/1975, 186-202.

10. H. Ehrig, M. Pferder, H.J. Schneider : Graph grammars : an algebraic approach, Switching and automata theory conference 1973.

11. Z. Manna : Mathematical Theory of Computation, Mc Graw Hill, 1974.

12. W.J. Meyers : Linear representation of tree structure, Third annual ACM symposium on theory of computing, Shaker Heights, Ohio, May 3-5, 1971, 50-62.

13. D. Knuth : The art of computer programming Vol. 1, Addison -Wesley Publ., Reading, Mass. 1969.

14. S.A. Cook, D.C. Oppen : An assertion language for data structures, Conf. rec. of the 2. ACM symposium on principles of programming languages, Palo Alto, Calif., Jan.20-22, 1975, 160-166.

15. A.C. Rosenberg : Data graphs and adressing schemes, J. Computer and System Sci., June 1971, 193-238.

16. K. Čulík : Algorithmization of algebras and relational structures, Comentationes Mathematical Universitatis Carolinae, 13,3 (1972), 457-477.

17. Václav Rajlich : Theory of computing machines, to be published by SNTL, Prague, in Czech.