CORRECTNESS OF PROGRAMS MANIPULATING DATA STRUCTURE

by

Tomasz Kowaltowski

Memorandum No. ERL-M404

September 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CORRECTNESS OF PROGRAMS MANIPULATING DATA STRUCTURES

Tomasz Kowaltowski

Abstract

A technique for proving correctness of programs manipulating data structures is proposed. The three major components of the technique are: (i) an abstract representation for data structures, called FSD (for Free State Description); (ii) a set of propositions which allow transformations of such FSD's; and (iii) semantics of assignment statements in terms of FSD transformations.

The technique provides a framework for rigorous proofs about programs manipulating data structures with arbitrary sharing of pointers and circularities. Several examples of application are exhibited, including the Deutsch-Schorr-Waite marking algorithm and the Fischer-Galler equivalence relation algorithm.

A graphic interpretation of several proofs is given in order to illustrate the intuitive concepts hidden behind the proposed method. It becomes apparent that these graphical proofs, based on rigorous propositions, are often sufficiently convincing so that analytical proof can be omitted.

The proposed method is compared with other techniques described in the literature, and among them the one proposed by Burstall from which this work borrows many ideas. Finally, the applicability of this method is discussed, showing its scope and limitations.

Acknowledgements

I would like to express my gratitude to my thesis advisor, Professor James H. Morris, Jr., who introduced me to this research area, and who provided guidance and constant help. I would also like to thank Professors Richard M. Karp and Domenico Ferrari for their valuable comments.

I also wish to thank my colleagues from the Faculdade de Economia e Administração da Universidade de São Paulo, whose cooperation made this work possible.

Financial support of the Universidade de São Paulo (Brazil) and of the Agency for International Development - Department of State is gratefully appreciated.

Publication of this dissertation as a technical memorandum was partially supported by the National Science Foundation under contract No. GJ-34342X, and its excellent typing I owe to Ruth Suzuki.

I dedicate this work to my wife Doris, whose constant encouragement and understanding made it all possible.

CONTENTS

CHAPTER 1

INTRODUCTION AND PRELIMINARIES

1.1 Introduction

Since Floyd published his original paper in 1967 there has been considerable interest in the area of proving assertions about programs, and the related area of semantics of programming languages (cf. London [1970a & b]). Floyd's original ideas were presented more formally, extended and applied in various subsequent works, among them Manna [1969] and Hoare [1969 & 1971].

In principle, Floyd's technique of inductive assertions, which dealt with simple variables only, can be easily extended in order to handle arrays, lists and similar structured values. The case of arrays has been treated by McCarthy and Painter [1967], King [1969], Good [1970] and others. A common approach is to treat an array like a function whose value is modified by assignment statements. Thus if A stands for an array, and the statement $A[E] \leftarrow F$ is performed, then the new value A' of the array is given (in λ-calculus notation) by:

$$A' \triangleq (\lambda j.\ \underline{\text{if}}\ j=E\ \underline{\text{then}}\ F\ \underline{\text{else}}\ A[j]) \quad .$$

It is easy to see that after a few assignment statements, with possibly complicated expressions E and F, the resulting description of an array value may become complex and unmanageable. In practice, this complex build-up of descriptive expressions may not occur if the arrays are used in a "reasonable" way. By "reasonable" we mean here that the array represents an ordered

sequence of some atomic values which is directly related to the problem being solved. In such a case the assignments to array components are usually performed in an orderly fashion, resulting in simplification of the expressions describing the value of the array. The situation becomes different if we use arrays for other purposes, such as representing trees, recursion stacks, etc.

Several programming languages, such as LISP, SNOBOL, PL/I and ALGOL68 provide facilities for definition and manipulation of more complex structured values, to which we shall refer as data structures. In dealing with this kind of objects, a simple approach, similar to that for arrays, would be to use the concept of contents function (or functions) to describe the state of computation. In the case of LISP we could think of two functions car and cdr, describing the state of computer memory. If an assignment statement of the form CAR(X) ← E (or more properly (RPLACA X E)) is performed, then the new state is described by functions car' and cdr' given by:

$$car' \triangleq (\lambda y.\ \underline{\text{if}}\ y=X\ \underline{\text{then}}\ E\ \underline{\text{else}}\ car(y))$$
$$cdr' \triangleq cdr$$

and similarly in case of CDR(X) ← E (for a more complete discussion see Morris [1972]). This approach is perfectly adequate to describe precisely the semantics of an assignment statement to a language implementor. However, if we try this approach for the purpose of proving properties of programs, we run immediately into difficulties analogous to those mentioned in connection with arrays. The descriptive expressions become very complex and

difficult to follow, and the proofs turn out to be very tedious and overcrowded with details extraneous to the problem in question. The main reason for these difficulties lies clearly in the fact that we are using a very low level language to describe complex concepts. Most of the efforts in this area have gone into devising more convenient ways of representing the state of computation, involving simple concepts and bringing the description of the data structure closer to its intended meaning. In analogy with Floyd's verification conditions, we will have to know the transformations of such representations corresponding to assignment statements. In addition, we will usually have to know some properties of such representations in order to draw conclusions about the state of computation, or to be able to apply the transformations mentioned above.

Some techniques have been proposed and described in the literature, and among them we mention the concept of restricted descendant functions in Morris [1972], covering functions in Poupon and Wegbreit [1972] and the distinct non-repetitive list and tree systems in Burstall [1972]. In this dissertation we propose the concept of free state descriptions, closely related to that described by Burstall from whom we borrow heavily. The technique we propose can be used to provide rigorous proofs of correctness of programs manipulating complex data structures involving arbitrary circularities and sharing patterns. On the other hand, we also show how to use this technique to justify very intuitive and simple proofs through graphical interpretation. Although we do not formalize this idea of graphical proofs, and

actually we do not know how to do it, we consider it an important aspect of this work.

Since our main interest here is in data structures, we do not treat other aspects of program proofs such as simple variables, arrays, procedure calls, in a very rigorous way. For the sake of clarity, we try to be precise in our basic definitions and formulation of propositions. However we shall be rather sketchy in their proofs since most of them are very intuitive and obvious. In our examples, we shall exhibit a varying degree of rigor.

The plan of this dissertation is as follows:

We complete this Chapter 1 with notations and conventions to be used. In Chapter 2 we define precisely the machine model we have in mind in order to use it as a reference for later development. In Chapter 3 we introduce the definitions of abstract free trees, which are used in Chapter 4 to describe our basic concept of free state descriptions. In Chapter 5 we study some of the properties of these free state descriptions, and in Chapter 6 we provide the transformations corresponding to the assignment statements. Chapter 7 contains some examples of application of this technique, and we follow with final conclusions in Chapter 8, where we also point out how our technique is related to others.

1.2 Notations and Conventions

We use standard mathematical notation, introducing some convenient extensions and additions.

Symbols $\wedge$ (conjunction), $\vee$ (disjunction), $\sim$ (negation), $\Rightarrow$ (implication), $\exists$ (existential quantifier) and $\forall$ (universal

quantifier) will be used in logical expressions. In addition we shall use $\bigwedge_{i=1}^{k}$ and $\bigvee_{i=1}^{k}$ for multiple conjunctions and disjunctions. The abbreviation "iff" stands for "if and only if".

Symbols ϵ (membership), $\cap$ (intersection), $\cup$ (union), $\sim$ (difference), $\subseteq$ (inclusion) and $\emptyset$ (empty set) will be used in dealing with sets. $\bigcap_{i=1}^{k}$ and $\bigcup_{i=1}^{k}$ denote multiple intersection and union. If S is a set, $|S|$ will denote its cardinality, and 2^S the power set of S.

We shall extend the set-theoretic notation to sequences, which will be usually denoted by underlined letters. If $\underline{u} = (u_1,\ldots,u_n)$ for some $n \geq 0$, and S is a set then:

$$|\underline{u}| = n$$

$$\underline{u} \subseteq S \text{ iff } \{u_1,\ldots,u_n\} \subseteq S$$

$$z \in \underline{u} \text{ iff } z = u_i \text{ for some } i = 1,\ldots,n \quad .$$

If $\underline{u} = (u_1,\ldots,u_n)$ and $\underline{v} = (v_1,\ldots,v_m)$ then $(\underline{u},\underline{v})$ shall denote the sequence $(u_1,\ldots,u_n,v_1,\ldots,v_m)$. In the case of unitary sets or sequences we shall identify them with their only components; thus $u = \{u\} = (u)$.

For notational convenience we shall use two different tuple building functions: $\langle\cdots\rangle$ will denote tuples whose components are selected by integers $0,1,2,\ldots$, and $\{\!|\cdots|\!\}$ will denote tuples whose components are selected by $1,2,\ldots$. The selection is denoted by the "." operator in either case. Thus

$$\langle a,b,c\rangle.1 = b$$

$$\langle a,b,c\rangle.0 = a$$

$$\{a,b,c\}.1 = a$$

$$\{a,b,c\}.3 = c$$

Also, if $x = \langle x_0,x_1,\ldots,x_n\rangle$ and $y = \{y_1,y_2,\ldots,y_m\}$ then $|x| = n$ and $|y| = m$.

The notation "$E_1 \triangleq E_2$" stands for "E_1 is defined to be E_2".

$[E]^{x_1,\ldots,x_n}_{E_1,\ldots,E_n}$ represents the result of simultaneous substitution of the expressions $E_1,\ldots,E_n$ for the variables $x_1,\ldots,x_n$ in the expression E.

A sequence of expressions $E_m,\ldots,E_k$ may be abbreviated by $[E_i]^k_{i=m}$.

When we state an assertion about a program statement (or a sequence of statements) we shall use the notation introduced by Hoare [1969]; thus

$$P\{T\}Q$$

means that if the property P was true before T was executed, then the property Q will be true after the execution of T.

Some additional notations will be defined within the text.

CHAPTER 2

MACHINE MODEL AND SEMANTICS OF ASSIGNMENT STATEMENTS

In this chapter we shall define the machine model we have in mind, and the meaning of assignment statements for the model. We chose to use the contents function as the basic concept in our definition because of its simplicity and intuitive appeal. A number of different definitions could have been chosen, and among them one which starts directly with the concept of free state description as presented in Chapter 4. All later results will be proved with relation to the model presented here.

We shall assume in our model that the memory is a finite set of locations, each one of them capable of holding a tuple over the set of location names (references or pointers) and some unspecified objects called atoms. The tuple contained in each location will be determined by the contents function. The following definitions formalize these intuitive concepts.

2.1 Definition. Let M be a finite set of (currently used) memory references, and let A be a set of objects (atoms) such that $M \cap A = \emptyset$. Let $D \triangleq M \cup A$ be the set of primitive objects.

The elements of D will be usually denoted by the letters $u, v, w, t, z,$ with or without subscripts and superscripts.

2.2 Definition. Let the set G of tuples over D be defined by:

$$G = \{\{u_1,\ldots,u_p\} \mid \{u_1,\ldots,u_p\} \subseteq D,\ p \geq 1\} \quad .$$

Any function $c: M \to G$ will be called a contents function.

In order to simplify further the notation, whenever only one contents function c is mentioned in some context, $x\uparrow$ will stand for $c(x)$.

The following definitions make precise some of the intuitive notions used in dealing with data structures.

2.3 Definition. Given a contents function c, let $E_c: D \to 2^M$ be defined by[†]:

$$E_c(u) \triangleq \underline{\text{if}}\ u \in A\ \underline{\text{then}}\ \emptyset\ \underline{\text{else}}\ [\{u\} \cup \bigcup_{i=1}^{|u\uparrow|} E_c(u\uparrow.i)] \quad .$$

Intuitively, $E_c(u)$ is the set of all references which can be reached from u in zero or more applications of c. We shall extend the definition of E_c to sequences over D in a natural way; if $\underline{u} = (u_1,\ldots,u_m)$, $u_i \in D$, then

$$E_c(\underline{u}) \triangleq \bigcup_{i=1}^{m} E_c(u_i) \quad .$$

2.4 Definition. Given a contents function c, let the predicate $FREE_c$ on D be defined by:

$$FREE_c(u) \triangleq \{[u \in A] \vee \bigwedge_{i=1}^{|u\uparrow|} [FREE_c(u\uparrow.i) \wedge u \notin E_c(u\uparrow.i) \wedge \bigwedge_{\substack{j=1 \\ i\neq j}}^{|u\uparrow|} (E_c(u\uparrow.i) \cap E_c(u\uparrow.j) = \emptyset)]\} \quad .$$

Intuitively, $FREE_c(u)$ means that for any $z \in E_c(u)$, there is exactly one path from u to z. In other words, the data structure rooted at u and defined by c has neither circularities nor sharing of pointers.

[†]This recursive definition may have more than one fixed point; our intended meaning is the unique least fixed point (cf. Manna et al. [1972]).

2.5 Definition. Given a contents function c, let the predicate $NONCIRC_c$ on D be defined by:

$$NONCIRC_c(u) \triangleq \{[u \in A] \vee \bigwedge_{i=1}^{|u\uparrow|} [u \notin E_c(u\uparrow.i) \wedge NONCIRC_c(u\uparrow.i)]\} .$$

Thus, $NONCIRC_c(u)$ means that there are no circularities in the data structure, but there may be sharing.

The set M of currently used memory references and the contents function c give a static description of the state of the machine. In what follows we shall show the state transformations under assignment statements. First we shall define the concept of program expression which will model expressions appearing in different programming languages. We shall assume that V and C are the sets of variables and constants, respectively, and we shall not specify them precisely.

2.6 Definition. Let the set P of program expressions be defined by:

(i) $V \cup C \subseteq P$

(ii) if $E \in P$ and i is an integer then $(E.i) \in P$

(iii) if $E_1,\ldots,E_p \in P$ and θ is a p-ary operator symbol then $\theta(E_1,\ldots,E_p) \in P$

(iv) P is the smallest set satisfying (i), (ii) and (iii).

Thus P is the set of expressions built up from variables and constants, using the selector operator "." and some unspecified operator symbols. Next we define the value to which a program expression is bound through a contents function c.

2.7 Definition. Let E be a program expression, and let c be a contents function; then $VAL_c(E)$ is defined by:

$$VAL_c(E) \triangleq \begin{cases} E & \text{if } E \text{ is a variable or constant} \\ \theta(VAL_c(E_1),\ldots,VAL_c(E_p)) & \text{if } E = \theta(E_1,\ldots,E_p) \\ (c(VAL_c(E_1))).i & \text{if } E = (E_1.i) \end{cases} .$$

Notice that this definition implies that we do not treat variables as references. We assume that they are bound to some values in our universe of discourse D; this binding may be changed by assignment statements to variables. Another implication of this definition is that we adopted a convention similar to that of SNOBOL and ALGOL68, where an automatic coercion (contents function application) takes place before component selection is done.

We can proceed now with the definitions of semantics of assignment statements.

2.8 Definition. Let $c_0: M_0 \to G_0$ be a contents function, and let E_1 and E_2 be two program expressions. Let $v_1 = VAL_{c_0}(E_1)$ and $v_2 = VAL_{c_0}(E_2)$. Then the semantics of the (component) assignment statement is defined by:

$$(c=c_0 \wedge M=M_0)\{E_1.i \leftarrow E_2\}(c=c_1 \wedge M=M_0)$$

where

$$c_1(x) \triangleq \underline{\text{if}}\ x=v_1\ \underline{\text{then}}\ y\ \underline{\text{else}}\ c_0(x)$$

with

$$y \triangleq \{c_0(x).1,\ldots,c_0(x).i\text{-}1,v_2,c_0(x).i\text{+}1,\ldots,c_0(x).|c_0(x)|\} .$$

In other words the current set of memory references remains the same, and the contents function c_0 is replaced by c_1, which agrees with c_0 for all arguments except v_1. Notice that this definition assumes that $v_1 \in M_0$ and $|c_0(v_1)| \leq i$; otherwise the effect of this assignment is undefined.

2.9 Definition. Let $c_0: M_0 \to G_0$ be a contents function, let I be a program variable and let $E_1,\ldots,E_p$ be program expressions ($p \geq 1$). Let $v_i = VAL_{c_0}(E_i)$, $i = 1,\ldots,p$. Then the semantics of the assignment statement $I \leftarrow TUPLE(E_1,\ldots,E_p)$ is defined by:

$$(c=c_0 \wedge M=M_0)\{I \leftarrow TUPLE(E_1,\ldots,E_p)\}$$

$$(\exists z)(c=c_1 \wedge I \notin M_0' \wedge M=M_0' \cup \{I\})$$

where

$$M_0' = [M_0]_z^I$$

and

$$c_1(x) \triangleq \begin{cases} [\langle v_1,\ldots,v_p \rangle]_z^I & \text{if } x = I \\ [c_0(x)]_z^I & \text{if } x \neq I,\ x \in M_0' \end{cases}.$$

Thus the current set of memory references is extended by a new one (bound to the variable I), and c_1 is an extension of c_0 as defined above. TUPLE is the operator corresponding to cons in LISP, but accepting a variable number of arguments. The substitution of z for I is necessary in case I was bound to a value and appeared in the expressions for c_0 or M_0.

The semantics of more complicated assignment statements, involving several TUPLE operators or simultaneous assignments

could be easily described in a similar way. We will not do it formally since such statements can be always replaced by an equivalent sequence of simple assignments. However we shall frequently use such statements in our examples.

Since simple assignment statements are very frequently used, we shall give the corresponding rule:

2.10 Definition. Let $c_0: M_0 \to G_0$ be a contents function, let I be a program variable and let E be a program expression. Let $v = VAL_{c_0}(E)$. Then the semantics of the assignment statement $I \leftarrow E$ is defined by:

$$P\{I \leftarrow E\}(\exists z)[P]^I_z \wedge I=v$$

where P is any predicate describing the state of computation.

Thus a simple assignment merely changes the binding of a variable. The contents function and the set of current references remain the same, since P might have included $c=c_0 \wedge M=M_0$.

CHAPTER 3

FREE TREES

In this chapter we shall define the concept of abstract free trees which constitute the basic element in our state representation described in the next chapter.

3.1 Definition. Let $X_n \triangleq \{x_1,\ldots,x_n\}$, $n \geq 0$ denote the set of n distinct variables such that $D \cap X_n = \emptyset$. For each n, let T_n be the set of trees generated by $X_n \cup A$ and defined by:

(i) $X_n \cup A \subseteq T_n$;

(ii) if $\tau_1,\ldots,\tau_p \in T_p$, $p \geq 1$ and $u \in M$ then $\langle u,\tau_1,\ldots,\tau_p\rangle \in T_n$;

(iii) T_n is the smallest set satisfying (i) and (ii).

T_n is the set of symbolic expressions built up from $X_n \cup A$, using the operation symbol $\langle\cdots\rangle$ with a variable number of arguments, where the first argument is always a memory reference.

3.2 Definition. Let $X \triangleq \{x_1,x_2,\ldots\}$ be the set of variables, and let the set T of trees be defined by:

$$T \triangleq \bigcup_{i=0}^{\infty} T_i \quad .$$

The elements of T will be usually denoted by the Greek letters τ, σ, ζ.

The following are examples of elements of T assuming $M = \{u_1,u_2,u_3\}$ and $A = \{a,b\}$:

b

x_2

$\langle u_1, \langle u_2, x_1, a\rangle, \langle u_3, b, x_2\rangle\rangle$

$\langle u_2, \langle u_1, \langle u_2, x_1, a\rangle, x_2, b, x_3\rangle, x_1\rangle$

In what follows we shall define several functions and predicates on the set T. Since this set, together with the operation $\langle\cdots\rangle$ constitutes an absolutely free structure, any recursive definition based on $X \cup A$ and extended to tuples in terms of its components will yield a well defined function on the whole set T.

3.3 Definition. Let the predicates atom, var and elem on T be defined by:

$$\underline{\text{atom}}(\tau) \triangleq (\tau \in A)$$

$$\underline{\text{var}}(\tau) \triangleq (\tau \in X)$$

$$\underline{\text{elem}}(\tau) \triangleq (\tau \in A \cup X) \quad .$$

These definitions are extended to sequences over T; thus if $\underline{\tau} = (\tau_1, \ldots, \tau_n)$, $\tau_i \in T$, then:

$$\underline{\text{atom}}(\underline{\tau}) \triangleq \bigwedge_{i=1}^{n} \underline{\text{atom}}(\tau_i)$$

$$\underline{\text{var}}(\underline{\tau}) \triangleq \bigwedge_{i=1}^{n} \underline{\text{var}}(\tau_i)$$

$$\underline{\text{elem}}(\underline{\tau}) \triangleq \bigwedge_{i=1}^{n} \underline{\text{elem}}(\tau_i) \quad .$$

3.4 Definition. Let the predicate order on $\{1,2,\ldots\} \times T$ be defined by:

$$\underline{\text{order}}(n,\tau) \triangleq \neg\underline{\text{elem}}(\tau) \Rightarrow [|\tau| = n \wedge \bigwedge_{i=1}^{n} \underline{\text{order}}(n, \tau.i)] \quad .$$

In other words, order(n,τ) means that all tuples within τ have $n+1$ components (a memory reference and n subtrees).

3.5 Definition. Let the function $NX: X \times T \to \{0,1,2,\ldots\}$ be defined by:

$$NX(x_i,\tau) \triangleq \underline{\text{if}}\ \underline{\text{elem}}(\tau)\ \underline{\text{then}}\ [\underline{\text{if}}\ \tau = x_i\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0]\ \underline{\text{else}}\ \sum_{j=1}^{|\tau|} NX(x_i,\tau.j) \quad .$$

$NX(x_i,\tau)$ denotes the number of occurrences of the variable x_i in the tree τ.

3.6 Definition. Let the function $NM: M \times T \to \{0,1,2,\ldots\}$ be defined by:

$$NM(u,\tau) \triangleq \underline{\text{if}}\ \underline{\text{elem}}(\tau)\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \{[\underline{\text{if}}\ \tau.0 = u\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0] + \sum_{j=1}^{|\tau|} NM(u,\tau.j)\} \quad .$$

$NM(u,\tau)$ denotes the number of occurrences of the reference u as the first component of a tuple in τ.

3.7 Definition. Let the function $V: T \to 2^X$ be defined by:

$$V(\tau) \triangleq \{x_i \mid NX(x_i,\tau) \geq 1\} \quad .$$

Thus $V(\tau)$ is the set of all variables x_i occurring in τ.

3.8 Definition. Let the function $S: T \to 2^M$ be defined by:

$$S(\tau) \triangleq \{u \mid NM(u,\tau) \geq 1\} \quad .$$

Thus $S(\tau)$ is the set of all memory references occurring in τ.

3.9 Definition. We extend the functions defined in 3.5-3.8 to sequences over T; thus if $\underline{\tau} = (\tau_1,\ldots,\tau_n)$, $\tau_i \in T$, then:

$$NX(x_i,\underline{\tau}) \triangleq \sum_{j=1}^{n} NX(x_i,\tau_j)$$

$$NM(u,\underline{\tau}) \triangleq \sum_{j=1}^{n} NM(u,\tau_j)$$

$$V(\underline{\tau}) \triangleq \bigcup_{j=1}^{n} V(\tau_j)$$

$$S(\underline{\tau}) \triangleq \bigcup_{j=1}^{n} S(\tau_j) \quad .$$

Notations

(i) $(x_i \in \underline{\tau}) \triangleq (x_i \in V(\underline{\tau}))$

(ii) $(u \in \underline{\tau}) \triangleq u \in S(\underline{\tau})$

(iii) $(x_i \bar{\in} \underline{\tau}) \triangleq NX(x_i,\underline{\tau}) = 1$

3.10 Definition. Let the partial operation of composition on T (denoted by "$\circ$") be defined by:

If $\tau_j \in T_n$, $j = 1,\ldots,m$ and $\sigma_k \in T$, $k = 1,\ldots,n$ for some $m, n \geq 0$, then:

$$(\tau_1,\ldots,\tau_m)\circ(\sigma_1,\ldots,\sigma_n) \triangleq (\zeta_1,\ldots,\zeta_m)$$

where

$$\zeta_j \triangleq [\tau_j]^{x_1,\ldots,x_n}_{\sigma_1,\ldots,\sigma_n} \qquad j = 1,\ldots,m \quad .$$

Thus each ζ_j is obtained by substituting simultaneously $\sigma_1,\ldots,\sigma_n$ for all occurrences of $x_1,\ldots,x_n$ in τ_j. It is easy

to prove that this operation is associative, and that for any $n \geq 0$, $(x_1,\ldots,x_n)$ is the right identity element.

3.11 Definition. Let the predicate free be defined by:

$$\underline{\text{free}}(\underline{\tau}) \triangleq (\forall u)[NM(u,\underline{\tau}) \leq 1] \quad .$$

Thus $\underline{\text{free}}(\underline{\tau})$ means that any reference u occurs at most once in some tree of the sequence $\underline{\tau}$.

Notice that the predicate free is not related to the predicate $FREE_c$ defined in 2.4. As a matter of fact we shall use free trees to describe data structures which are not FREE.

3.12 Definition. Let

$$F_n \triangleq \{\tau \mid \tau \in T_n \wedge \underline{\text{free}}(\tau)\}$$

and

$$F \triangleq \bigcup_{i=1}^{\infty} F_i \quad .$$

Thus F is the set of all free trees.

3.13 Proposition. Let $\tau \in T_n$, and $\underline{\sigma} = (\sigma_1,\ldots,\sigma_n)$, $\sigma_i \in T$ for some $n \geq 0$. Assume that

(i) $\underline{\text{free}}((\tau,\sigma_1,\ldots,\sigma_n))$ and

(ii) $\sim\underline{\text{elem}}(\sigma_i) \Rightarrow NX(x_i,\tau) \leq 1$ for $i = 1,\ldots,n$

Then $\underline{\text{free}}(\tau\circ\underline{\sigma})$.

Proof. A simple induction on $\tau \in T$. □

This proposition guarantees that the freedom property is preserved whenever in a composition operation, only variables

which occur at most once in the tree are substituted for by non-elementary terms. Otherwise the memory references occurring in the substituting term would be multiplied by the number of occurrences of the corresponding variable.

CHAPTER 4

FREE STATE DESCRIPTIONS

In this chapter we show how the abstract trees defined in Chapter 3 can be used to represent arbitrary data structures. The adequacy of this representation is shown by exhibiting its relationship to the concepts defined in Chapter 2.

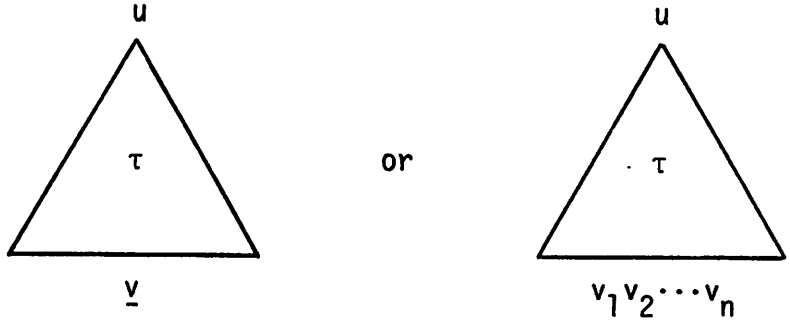
4.1 Definition. Let $c: M \to G$ be a contents function, $u \in D$, $\tau \in F_n$ and $\underline{v} = (v_1,\ldots,v_n)$, $v_i \in D$ for some $n \geq 0$. Then the triple $(u,\tau,\underline{v})$ is said to be compatible with c (notation: $u \xrightarrow[c]{\tau} \underline{v}$) if and only if:

either (i) $\underline{atom}(\tau) \wedge \underline{atom}(u) \wedge u = \tau$;

or (ii) $\underline{var}(\tau) \wedge \tau = x_i \wedge u = v_i$ for some i;

or (iii) $\sim\underline{elem}(\tau) \wedge \tau.0 = u \wedge |\tau| = |u\uparrow| \wedge \bigwedge_{j=1}^{|\tau|} (u\uparrow.j \xrightarrow[c]{\tau.j} \underline{v})$.

Conceptually, we shall say that the tree τ connects u (root) to $\underline{v}$ (endpoints). Graphically, we shall represent this fact by:

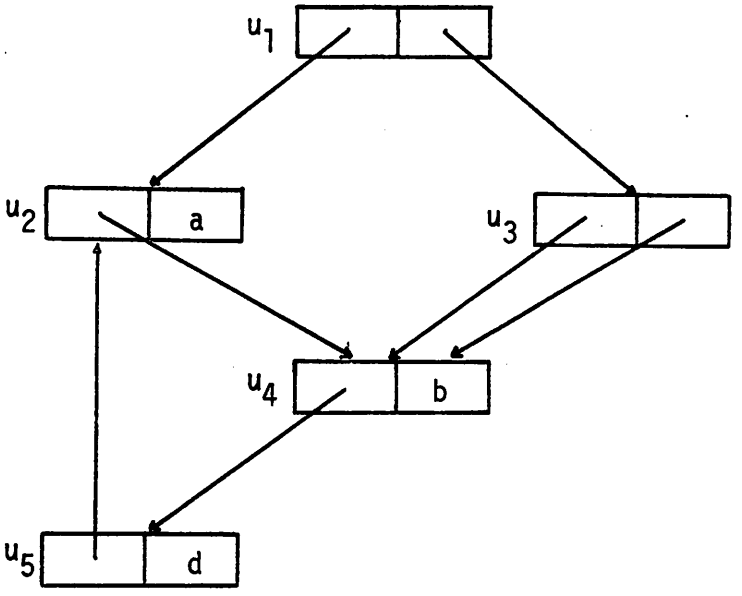


Notice that even though the memory references occurring in τ are all distinct (since $\underline{free}(\tau)$), $v_1,\ldots,v_n$ are not necessarily

in τ, and they may be repeated.

The following example should clarify these concepts.

4.2 Example. Let $M = \{u_1,u_2,u_3,u_4,u_5\}$, $A = \{a,b,d\}$, and assume that the contents function c describes the following data structure:



The following are some of the relations which hold:

$$a \xrightarrow[c]{a} (u_1,u_2,u_3,u_4,u_5)$$

$$a \xrightarrow[c]{x_3} (u_1,b,a,u_2,d)$$

$$u_i \xrightarrow[c]{x_i} (u_1,u_2,u_3,u_4,u_5) \qquad i = 1,2,3,4,5$$

$$u_1 \xrightarrow[c]{\langle u_1,x_1,x_2\rangle} (u_2,u_3)$$

$$u_2 \xrightarrow[c]{\langle u_2,x_1,a\rangle} u_4$$

$$u_2 \xrightarrow{\langle u_2, x_1, x_2\rangle}_c (u_4, a)$$

$$u_5 \xrightarrow{\langle u_5, x_1, d\rangle}_c u_2$$

$$u_1 \xrightarrow{\langle u_1, \langle u_2, \langle u_4, \langle u_5, x_1, d\rangle, b\rangle, a\rangle, \langle u_3, x_2, x_2\rangle\rangle}_c (u_2, u_4)$$

$$u_1 \xrightarrow{\langle u_1, x_2, \langle u_3, x_1, \langle u_4, \langle u_5, \langle u_2, x_3, a\rangle, d\rangle, b\rangle\rangle\rangle}_c (u_4, u_2, u_4)$$

$$u_4 \xrightarrow{\langle u_4, \langle u_5, \langle u_2, x_1, a\rangle, d\rangle, b\rangle}_c u_4$$

The concept of compatibility can be extended to sequences over D and F in a natural way:

4.3 Definition. Let $c: M \to G$ be a contents function, $\underline{u} = (u_1, \ldots, u_m)$, $u_i \in D$, $\underline{\tau} = (\tau_1, \ldots, \tau_m)$, $\tau_i \in F_n$, and $\underline{v} = (v_1, \ldots, v_n)$, $v_i \in D$, for some $m \geq 1$ and $n \geq 0$. Then the triple $(\underline{u}, \underline{\tau}, \underline{v})$ is compatible with c (notation: $\underline{u} \xrightarrow{\underline{\tau}}_c \underline{v}$) if and only if

(i) $u_i \xrightarrow{\tau_i}_c \underline{v}$, $i = 1, \ldots, m$

and (ii) $\underline{\text{free}}(\underline{\tau})$.

Notice that $\underline{\text{free}}(\underline{\tau})$ implies $\underline{\text{free}}(\tau_i)$ for each i (but not the converse), and $S(\tau_i) \cap S(\tau_j) = \emptyset$ for $i \neq j$. In case of the example 4.2 the following holds:

$$(u_2, u_3) \xrightarrow{(\langle u_2, \langle u_4, \langle u_5, x_1, d\rangle, b\rangle, a\rangle, \langle u_3, x_2, x_2\rangle)}_c (u_2, u_4) \quad .$$

Remarks

(i) Whenever no ambiguity is possible, the subscript c on $\rightarrow_c$ will be dropped.

(ii) Notation: $(\underline{u} \xrightarrow{\underline{\tau}} \underline{v} \xrightarrow{\underline{\sigma}} \underline{w}) \triangleq (\underline{u} \xrightarrow{\underline{\tau}} \underline{v}) \wedge (\underline{v} \xrightarrow{\underline{\sigma}} \underline{w})$

$(\underline{u} \nmid \underline{\tau}) \triangleq (\underline{u} \xrightarrow{\underline{\tau}} ())$

(() is the empty sequence).

4.4 Definition. Let $\underline{\tau} = (\tau_1,\ldots,\tau_m)$, $\tau_i \in F_n$, $\underline{v} = (v_1,\ldots,v_n)$, $v_i \in D$, for some $m \geq 1$ and $n \geq 0$, such that $\underline{\text{free}}(\underline{\tau})$ holds, and let $z \in S(\underline{\tau})$. Then $R(z,\underline{\tau},\underline{v})$ is defined by

$$R(z,\underline{\tau},\underline{v}) \triangleq R_1(z,\tau_k,\underline{v}) \quad \text{if} \quad z \in S(\tau_k)$$

where

$$R_1(z,\sigma,\underline{v}) \triangleq \underline{\text{if}}\ \sigma.0 = z\ \underline{\text{then}}\ \{R_2(\sigma.1,\underline{v}),\ldots,R_2(\sigma.|\sigma|,\underline{v})\}\ \underline{\text{else}}\ R(z,(\sigma.1,\ldots,\sigma.|\sigma|),\underline{v})$$

and

$$R_2(\sigma,\underline{v}) \triangleq \underline{\text{if}}\ \underline{\text{atom}}(\sigma)\ \underline{\text{then}}\ \sigma\ \underline{\text{else}}\ [\underline{\text{if}}\ \underline{\text{var}}(\sigma) \wedge \sigma = x_j\ \underline{\text{then}}\ v_j\ \underline{\text{else}}\ \sigma.0]\ .$$

The definition of R is rather complicated, but its intuitive meaning is very simple. Namely R "reconstructs" the function c as shown by the following:

4.5 Proposition. If $\underline{u} \xrightarrow{\underline{\tau}}_c \underline{v}$ and $z \in S(\underline{\tau})$ then $R(z,\underline{\tau},\underline{v}) = c(z)$.

Proof. Can be carried out easily by induction on $\tau \in T$, and noting the following fact about R_2: if $t \xrightarrow{\sigma}_c \underline{w}$ then $R_2(\sigma,\underline{w}) = t$. □

4.6 Definition. Let $K = (\lambda_1,\ldots,\lambda_m)$ for some $m \geq 0$, where each λ_i is a triple of the form:

$$\lambda_i = (\underline{u}_i, \underline{\tau}_i, \underline{v}_i) \quad .$$

Then K is a free state description (FSD) compatible with c (notation: $*_c K$ or $*_c(\underline{u}_1 \xrightarrow{\underline{\tau}_1} \underline{v}_1,\ldots,\underline{u}_m \xrightarrow{\underline{\tau}_m} \underline{v}_m)$) if and only if:

(i) $\underline{u}_i \xrightarrow{\underline{\tau}_i}_c \underline{v}_i$, $i = 1,\ldots,m$

and (ii) $\underline{\text{free}}((\underline{\tau}_1,\ldots,\underline{\tau}_m))$.

Remarks

(i) Whenever no ambiguity is possible, the subscript c on $*_c$ will be dropped.

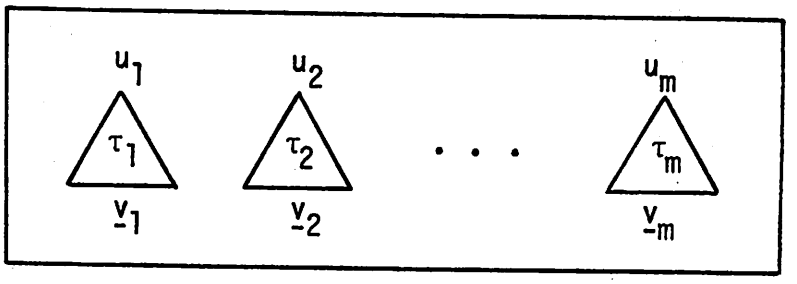
(ii) $*_c(\ )$ holds for any c.

(iii) Notation:

$$*_c(\underline{u}_1 \xrightarrow{\underline{\tau}_1} \underline{v}_1,\ldots,\underline{u}_m \xrightarrow{\underline{\tau}_m} \underline{v}_m)_L \triangleq *_c(\underline{u}_1 \xrightarrow{\underline{\tau}_1} \underline{v}_1,\ldots,\underline{u}_m \xrightarrow{\underline{\tau}_m} \underline{v}_m) \wedge [\bigcup_{i=1}^{m} S(\underline{\tau}_i) = L] \ .$$

An FSD is thus a sequence of compatible triples in which no reference occurs more than once. Notice however the remark about the endpoints after Definition 4.1.

Graphically, we shall represent an FSD $*(u_1 \xrightarrow{\tau_1} \underline{v}_1,\ldots,u_m \xrightarrow{\tau_m} \underline{v}_m)$ by:



The fact that the triangles representing $\tau_1,\ldots,\tau_m$ do not overlap emphasizes the property of the sets $S(\tau_i)$ being pairwise disjoint.

4.7 Examples. Let c be the contents function from Example 4.2. Then the following hold:

$$*(u_1 \xrightarrow{x_1} u_1)_{\emptyset}$$

$$*(u_1 \xrightarrow{x_1} u_1,\ u_2 \xrightarrow{x_2} (u_1,u_2),\ u_5 \xrightarrow{x_1} u_5)_{\emptyset}$$

$$*(u_1 \xrightarrow{\langle u_1,x_1,x_2\rangle} (u_2,u_3),\ (u_2,u_3) \xrightarrow{(\langle u_2,x_1,a\rangle,\langle u_3,x_1,x_1\rangle)} u_4,$$

$$u_4 \xrightarrow{\langle u_4,x_1,b\rangle} u_5,\ u_5 \xrightarrow{\langle u_5,x_1,x_2\rangle} (u_2,d))_{\{u_1,u_2,u_3,u_4,u_5\}}$$

$$*(u_1 \xrightarrow{\langle u_1,\langle u_2,\langle u_4,\langle u_5,x_1,d\rangle,b\rangle,a\rangle,\langle u_3,x_2,x_2\rangle\rangle} (u_2,u_4))_{\{u_1,u_2,u_3,u_4,u_5\}}$$

Notice that the first two examples hold for any contents function c.

4.8 Definition. An FSD $K = ((\underline{u}_1,\underline{\tau}_1,\underline{v}_1),\ldots,(\underline{u}_m,\underline{\tau}_m,\underline{v}_m))$ is said to be closed (notation: $\pi_c K$) if and only if:

(i) $*_c K_L$

and (ii) $[\bigcup_{i=1}^{m} \underline{v}_i \subseteq L \cup A]$.

The same remarks following Definition 4.6 apply here.

A closed FSD is one in which all endpoints are either atoms or occur somewhere in a descriptive tree. The last two examples in 4.7 are closed FSD's. Intuitively such a closed FSD contains all the information necessary to describe a data structure rooted at $\underline{u}_1,\ldots,\underline{u}_m$. This concept is formalized by the following:

4.9 Proposition. If $\pi_c(\underline{u}_1 \xrightarrow{\underline{\tau}_1} \underline{v}_1,\ldots,\underline{u}_m \xrightarrow{\underline{\tau}_m} \underline{v}_m)_L$ then:

(i) $\bigcup_{i=1}^{m} E_c(\underline{u}_i) = L$

and (ii) $\bigwedge_{i=1}^{m} (\forall u)[u \in S(\underline{\tau}_i) \Rightarrow R(u,\underline{\tau}_i,\underline{v}_i) = c(u)]$.

Proof. Claim (i) can be proved easily by showing first (by induction) two lemmas:

(a) If $\underline{u} \xrightarrow{\underline{\tau}} \underline{v}$ then $S(\underline{\tau}) \subseteq E_c(\underline{u})$.

(b) If $\underline{u} \xrightarrow{\underline{\tau}} \underline{v}$ and $z \in S(\underline{\tau})$ then $z\uparrow.i \in [S(\underline{\tau}) \cup A \cup \underline{v}]$ for $i = 1,\ldots,|z\uparrow|$ (i.e. all direct descendants of z which are not atoms, are also either in $S(\underline{\tau})$ or in $\underline{v}$).

Claim (ii) is a direct consequence of Proposition 4.5. □

This last proposition shows that a closed FSD is an accurate

description of a data structure (or several data structures). The following proposition shows that every data structure can be described in such a way.

4.10 Proposition. Let $c: M \to G$ be a contents function, and $u \in D$. Then there exists $n \geq 0$, $\tau \in F_n$ and $\underline{v} = (v_1,\ldots,v_n)$, $v_i \in M$, such that:

$$\pi_c(u \xrightarrow{\tau} \underline{v}) \quad .$$

A rigorous proof of this fact is not too difficult but rather tedious, and we will not carry it out. The main idea of the proof would be to define two functions which "build" τ and $\underline{v}$ for a given contents function c and $u \in D$. Then we would prove by induction that these τ and $\underline{v}$ satisfy the relation above. In order to illustrate this procedure we will exhibit the functions for the case of binary trees. Thus let $c: M \to G$ be a contents function such that for all $z \in M$, $|c(z)| = 2$. Then the functions θ and π are defined by:

$$\theta(z,(v_1,\ldots,v_n),E) \triangleq \underline{\text{if}}\ \underline{\text{atom}}(z)\ \underline{\text{then}}\ z\ \underline{\text{else}}$$
$$[\underline{\text{if}}\ z \in (v_1,\ldots,v_n) \wedge z = v_k\ \underline{\text{then}}\ x_k\ \underline{\text{else}}$$
$$[\underline{\text{if}}\ z \in E\ \underline{\text{then}}\ x_{n+1}\ \underline{\text{else}}$$
$$\langle z,\theta(z\uparrow.1,(v_1,\ldots,v_n),E\cup z),\theta(z\uparrow.2,\pi(z\uparrow.1,(v_1,\ldots,v_n),E\cup z),$$
$$\delta(z\uparrow.1,E\cup z))\rangle]]$$

$$\pi(z,(v_1,\ldots,v_n),E) \triangleq \underline{\text{if}}\ \underline{\text{atom}}(z)\ \underline{\text{then}}\ (v_1,\ldots,v_n)\ \underline{\text{else}}$$
$$[\underline{\text{if}}\ z \in (v_1,\ldots,v_n)\ \underline{\text{then}}\ (v_1,\ldots,v_n)\ \underline{\text{else}}$$
$$[\underline{\text{if}}\ z \in E\ \underline{\text{then}}\ (v_1,\ldots,v_n,z)\ \underline{\text{else}}$$
$$\pi(z\uparrow.2,\pi(z\uparrow.1,(v_1,\ldots,v_n),E\cup z),\delta(z\uparrow.1,E\cup z))]]$$

where

$$\delta(z,E) \triangleq \underline{\text{if}}\ \underline{\text{atom}}(z)\ \underline{\text{then}}\ E\ \underline{\text{else}}$$
$$[\underline{\text{if}}\ z \in E\ \underline{\text{then}}\ E\ \underline{\text{else}}\ \delta(z\uparrow.2,\delta(z\uparrow.1,E\cup z))] \quad .$$

Finally, we put:

$$\text{and} \qquad \begin{aligned} \tau &= \theta(u,(),\emptyset) \\ \underline{v} &= \pi(u,(),\emptyset) \end{aligned} \quad .$$

Notice that $\delta(u,E) = E \cup E_c(u)$.

Let us consider the contents function c from Example 4.2 and compute the corresponding τ and $\underline{v}$.

$$\begin{aligned} \tau &= \theta(u_1,(),\emptyset) \\ \underline{v} &= \pi(u_1,(),\emptyset) \end{aligned} \quad .$$

Applying the definitions we get:

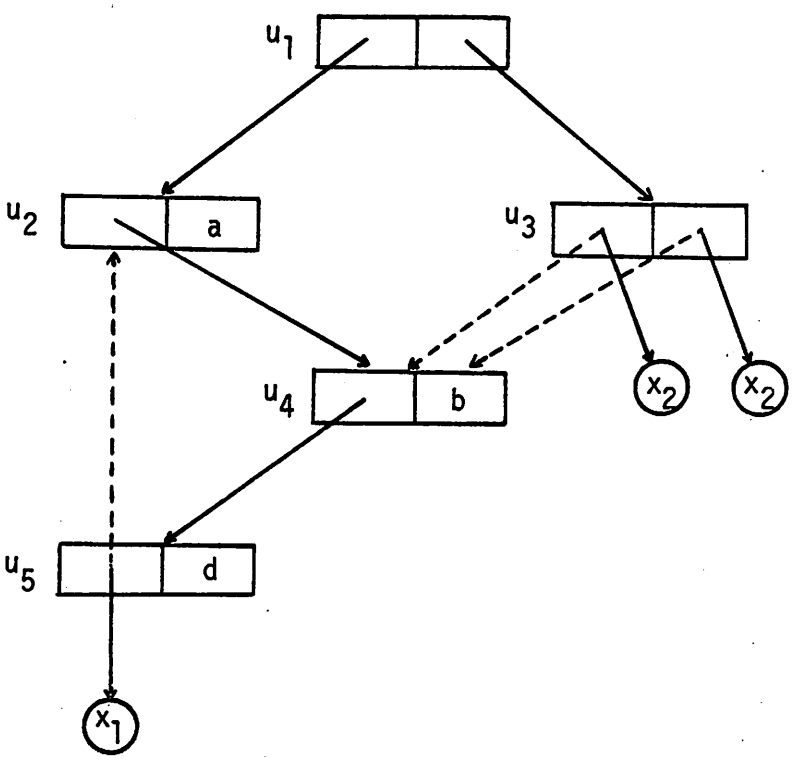
$$\begin{aligned} \tau &= \langle u_1,\langle u_2,\langle u_4,\langle u_5,x_1,d\rangle,b\rangle,a\rangle,\langle u_3,x_2,x_2\rangle\rangle \\ \underline{v} &= (u_2,u_4) \end{aligned}$$

(cf. last example in 4.7).

Notice that τ built as above has a special form in which a memory reference $v_i \in \underline{v}$ appears always "to the left" or "above" the variable x_i in the tree τ. This comes from the fact that the function θ "visits" the data structure in what Knuth [1968] calls pre-order traversal.

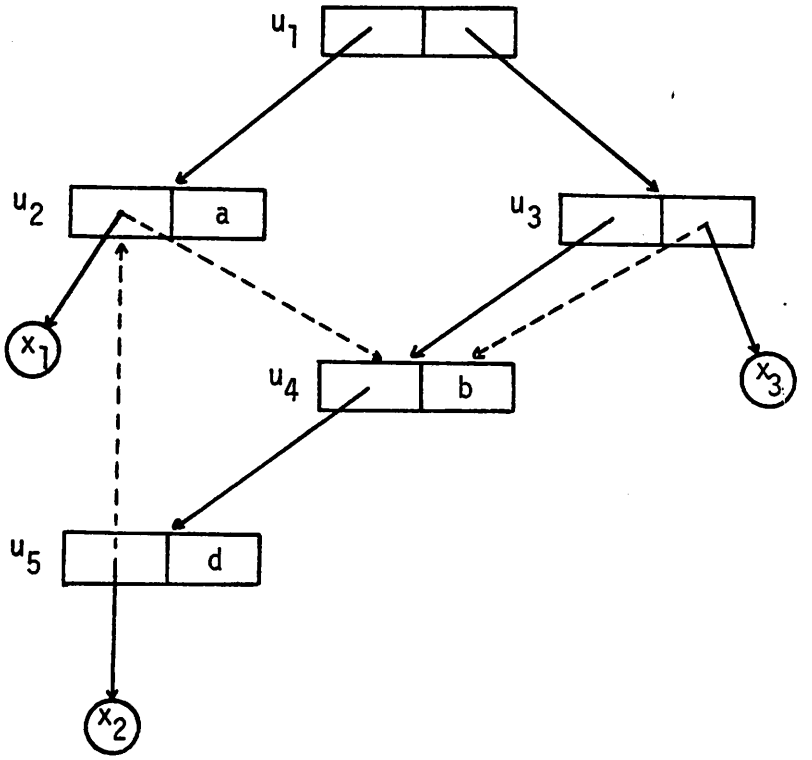
In general, however, τ and $\underline{v}$ satisfying $\pi_c(u \xrightarrow{\tau} \underline{v})$ are not unique. Intuitively, τ and $\underline{v}$ can be obtained by "cutting"

all but one link pointing to a shared node and replacing them by variable names. Some of the possible cases for the Example 4.2 are (broken lines represent links which were "cut"):



$\tau = \langle u_1, \langle u_2, \langle u_4, \langle u_5, x_1, d\rangle, b\rangle, a\rangle, \langle u_3, x_2, x_2\rangle\rangle$

$\underline{v} = (u_2, u_4)$



$\tau = \langle u_1, \langle u_2, x_1, a\rangle, \langle u_3, \langle u_4, \langle u_5, x_2, d\rangle, b\rangle, x_3\rangle\rangle$

$\underline{v} = (u_4, u_2, u_4)$

4.11 Proposition. Let $c: M \to G$ be a contents function, and $u \in D$ such that $FREE_c(u)$ holds. Then there exists $\tau \in F_o$ such that:

$$\pi_c(u \downarrow \tau)$$

(i.e., $\pi_c(u \xrightarrow{\tau} ())$).

Again, we will not prove this proposition, but it is easy to see in the preceding discussion that $FREE_c(u)$ implies $\pi(u,(),\emptyset) = ()$. Also, no "cutting" of links would be necessary since there is no sharing of pointers.

4.12 Definition. Let $K = ((\underline{u}_1,\underline{\tau}_1,\underline{v}_1),\ldots,(\underline{u}_m,\underline{\tau}_m,\underline{v}_m))$ be an FSD, and let $z \in S(\underline{\tau}_i)$ for some i. Then we define $R_K(z)$ by:

$$R_K(z) = R(z,\underline{\tau}_i,\underline{v}_i) \quad .$$

4.13 Proposition. If $K = ((\underline{u}_1,\underline{\tau}_1,\underline{v}_1),\ldots,(\underline{u}_m,\underline{\tau}_m,\underline{v}_m))$ and ${}^*_cK_L$ holds, then:

$$(\forall u)[u \in L \Rightarrow R_K(u) = c(u)] \quad .$$

Proof. Immediate consequence of 4.9. □

$R_K(u)$ "reconstructs" the value of $c(u)$ as determined by the FSD K.

CHAPTER 5

SOME PROPERTIES OF FREE STATE DESCRIPTIONS

In carrying out proofs of program correctness, we will frequently use certain properties of FSD's in order to draw conclusions about the state of computation or to be able to apply transformations described in the next chapter. This situation is analogous, for instance, to applying algebraic laws when dealing with programs manipulating numbers.

In this chapter we shall see a series of propositions concerning FSD's. Our aim is to provide enough propositions, so that proofs can be carried out without resorting to the definitions of an FSD. It would be interesting to know if there exists a finite set of such propositions, but we will not pursue this subject here. Most of the propositions shown in this chapter are actually used in our examples in Chapter 7.

Whenever a property holds for both $*$ and $\square$, we shall use $*\square$ instead.

5.1 Relations $\xrightarrow{}_C$ and $*_C$

$$\underline{u} \xrightarrow{\underline{\tau}}_C \underline{v} \text{ iff } *_C(\underline{u} \xrightarrow{\underline{\tau}} \underline{v}) \quad .$$

Proof. Trivial, using definitions. □

5.2 Relations * and $\square$. If $\square K_L$ then $*K_L$.

Proof. Trivial, using definitions. □

5.3 Permutation. If $*\square(\lambda_1,\ldots,\lambda_n)_L$ then $*\square(\lambda_{i_1},\ldots,\lambda_{i_n})_L$ where $(i_1,\ldots,i_n)$ is a permutation of $(1,\ldots,n)$.

Proof. Obvious. □

In practice we shall apply this property without mentioning it.

5.4 Deletion

(a) If $*(\underline{u} \xrightarrow{\underline{\tau}} \underline{v},\lambda_1,\ldots,\lambda_n)_L$ then $*(\lambda_1,\ldots,\lambda_n)_{L\sim S(\underline{\tau})}$.

(b) If $\natural(\underline{u} \xrightarrow{\underline{\tau}} \underline{v},\lambda_1,\ldots,\lambda_n)_L$ and $\underline{elem}(\underline{\tau})$ then $\natural(\lambda_1,\ldots,\lambda_n)_L$.

Proof. Trivial, using definitions; notice that $\underline{elem}(\underline{\tau})$ implies $S(\underline{\tau}) = \emptyset$. □

5.5 Merging. If $*\natural(\lambda_1,\ldots,\lambda_m)_L$ and $*\natural(\mu_1,\ldots,\mu_n)_N$ and $L \cap N = \emptyset$ then $*\natural(\lambda_1,\ldots,\lambda_m,\mu_1,\ldots,\mu_n)_{L\cup N}$.

Proof. Trivial, using definitions. □

This proposition can be used when the program execution provides somehow the information $L \cap N = \emptyset$. The references in L and N might have been produced by independent sections of the program or they might have been declared as being of different type.

5.6 Expansion

(a) If $*(\lambda_1,\ldots,\lambda_n)_L$ then $*(\mu,\lambda_1,\ldots,\lambda_n)_L$.

(b) If $\natural(\lambda_1,\ldots,\lambda_n)$ and $(u_1,\ldots,u_m) \subseteq L\cup A$ then $\natural(\mu,\lambda_1,\ldots,\lambda_n)_L$

where $\mu = (u_{i_1},\ldots,u_{i_m}) \xrightarrow{(x_{i_1},\ldots,x_{i_m})} (u_1,\ldots,u_m)$ and

$i_j \in \{1,\ldots,m\}$, $j = 1,\ldots,m$.

Proof. Using the definitions and propositions 5.1 and 5.5. Notice that $S((x_{i_1},\ldots,x_{i_m})) = \emptyset$.

(c) If $*(\lambda_1,\ldots,\lambda_n)_L$ and $\underline{atom}(u)$ then

$$(\forall \underline{v})[*(u \xrightarrow{u} \underline{v},\lambda_1,\ldots,\lambda_n)_L] \quad .$$

(d) If $\neg(\lambda_1,\ldots,\lambda_n)$ and $\underline{atom}(u)$ then

$$(\forall \underline{v})[(\underline{v} \subseteq L \cup A) \Rightarrow \neg(u \xrightarrow{u} \underline{v},\lambda_1,\ldots,\lambda_n)_L] \quad .$$

Proof. As parts (a) and (b), noticing that $S(u) = \emptyset$. □

This property allows the extension of an FSD by a trivial compatible triple (i.e., one not involving any references). It will be used mainly in adjusting an FSD to the initial conditions of a loop, so that an induction can be carried out.

5.7 Tupling and Untupling

$$*\neg(u_1 \xrightarrow{\tau_1} \underline{v},\ldots,u_m \xrightarrow{\tau_m} \underline{v},\lambda_1,\ldots,\lambda_n)_L$$

iff

$$*\neg(\underline{u} \xrightarrow{\underline{\tau}} \underline{v},\lambda_1,\ldots,\lambda_n)_L$$

where $\underline{u} = (u_1,\ldots,u_m)$ and $\underline{\tau} = (\tau_1,\ldots,\tau_m)$.

Proof. Obvious by using definitions. □

5.8 Distinctness. If $*\neg(u \xrightarrow{\tau} \underline{v},\ u \xrightarrow{\sigma} \underline{w},\lambda_1,\ldots,\lambda_n)$ then

$$\text{either } \underline{elem}(\tau) \text{ or } \underline{elem}(\sigma) \quad .$$

Proof. Otherwise $u \in S(\tau)$ and $u \in S(\sigma)$, and $S(\tau) \cap S(\sigma) \neq \emptyset$, which is a contradiction by definition of an FSD. □

5.9 Atoms. If $*\pi(u \xrightarrow{\tau} (v_1,\ldots,v_m),\lambda_1,\ldots,\lambda_n)$ and $\underline{atom}(u)$, then $\underline{elem}(\tau)$ and either $\tau = u$, or $\tau = x_i$ and $u = v_i$ for some $1 \leq i \leq m$.

Proof. Trivial, by applying the definition of a compatible triple to $u \xrightarrow{\tau} (v_1,\ldots,v_m)$. □

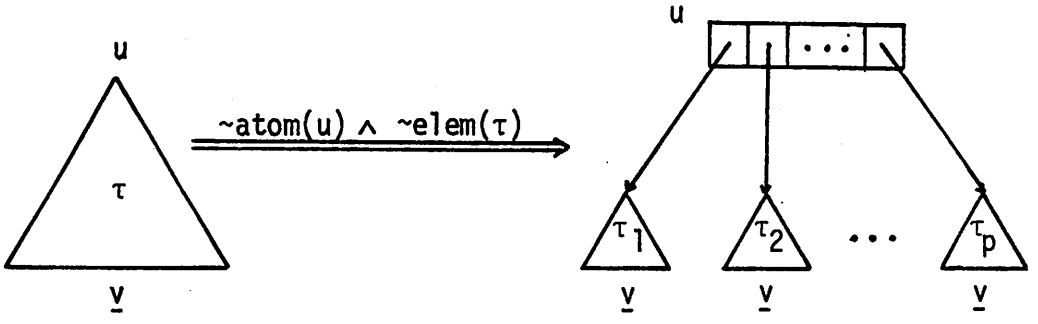
5.10 Non-atoms. If $*\pi(u \xrightarrow{\tau} (v_1,\ldots,v_m),\lambda_1,\ldots,\lambda_n)$ and $\sim\underline{atom}(u)$ then

either (i) $\underline{elem}(\tau)$ and $\tau = x_j \wedge u = v_j$ for some j;

or (ii) $\sim\underline{elem}(\tau)$ and there exists a $p \geq 1$ and $\tau_1,\ldots,\tau_p \in F_m$ such that $\tau = \langle u,\tau_1,\ldots,\tau_p\rangle$.

Proof. Simple application of definitions. □

This proposition will be used very frequently as a result of a test of the form "$\underline{atom}(u)$?". In case (ii) we shall decompose the tree τ into its components. This fact will be interpreted graphically by:



5.11 Composition

(a) If $*(\underline{u} \xrightarrow{\underline{\tau}} (v_1,\ldots,v_m) \xrightarrow{(\sigma_1,\ldots,\sigma_m)} \underline{w},\lambda_1,\ldots,\lambda_n)_L$ and $\sim\underline{elem}(\sigma_i) \Rightarrow NX(x_i,\underline{\tau}) \leq 1$ for $i = 1,\ldots,m$ then

$$*(\underline{u} \xrightarrow{\underline{\tau}\circ(\sigma_1,\ldots,\sigma_m)} \underline{w},\lambda_1,\ldots,\lambda_n)_N$$

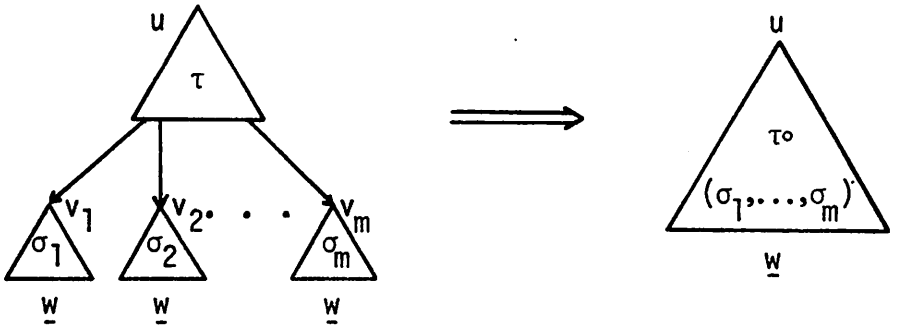
and $N \subseteq L$.

(b) If $*^{\square}(\underline{u} \xrightarrow{\underline{\tau}} (v_1,\ldots,v_m) \xrightarrow{(\sigma_1,\ldots,\sigma_m)} \underline{w},\lambda_1,\ldots,\lambda_n)_L$ and $\sim\underline{elem}(\sigma_i) \Rightarrow x_i \bar{\in} \underline{\tau}$ for $i = 1,\ldots,m$ then

$$*^{\square}(\underline{u} \xrightarrow{\underline{\tau}\circ(\sigma_1,\ldots,\sigma_m)} \underline{w},\lambda_1,\ldots,\lambda_n)_L \quad .$$

Proof. Straightforward by induction on $\tau \in T$. Notice that in part (b) we require $NX(x_i,\underline{\tau}) = 1$; otherwise some of the trees σ_i might "disappear" in the process of composition, and the set L would not be preserved anymore. □

This proposition will be used mainly to put together "pieces" of data structure which have been processed already, and we are not interested anymore in their exact shape. It will turn out that in general the values $v_1,\ldots,v_n$ are not bound anymore to any of the program variables (unless they appear also elsewhere). Graphically, this proposition may be interpreted by:



5.12 Decomposition

(a) (i) If $*(\underline{u} \xrightarrow{\underline{\tau}\circ\underline{\sigma}} \underline{w},\lambda_1,\ldots,\lambda_n)_L$ then there exists $\underline{v}$ such that $*(\underline{u} \xrightarrow{\underline{\tau}} v \xrightarrow{\underline{\sigma}} \underline{w},\lambda_1,\ldots,\lambda_n)_L$.

(ii) If $\pi(\underline{u} \xrightarrow{\underline{\tau}\circ\underline{\sigma}} \underline{w},\lambda_1,\ldots,\lambda_n)_L$ then there exists $\underline{v} \subseteq L \cup A$ such that $\pi(\underline{u} \xrightarrow{\underline{\tau}} \underline{v} \xrightarrow{\underline{\sigma}} \underline{w},\lambda_1,\ldots,\lambda_n)_L$.

Proof. Again by a straightforward induction on $\tau \in T$.

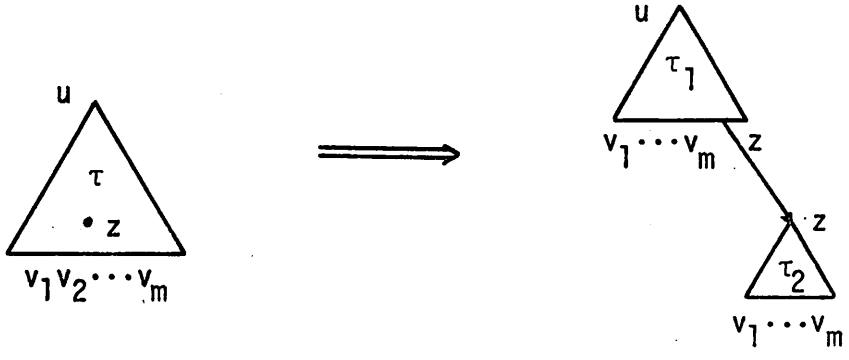
(b) If $*\pi(u \xrightarrow{\tau} (v_1,\ldots,v_m),\lambda_1,\ldots,\lambda_n)_L$ and $z \in S(\tau)$ then there exist τ_1, τ_2 such that:

$$*\pi(u \xrightarrow{\tau_1} (v_1,\ldots,v_m,z),\ z \xrightarrow{\tau_2} (v_1,\ldots,v_m),\lambda_1,\ldots,\lambda_m)_L$$

with $\tau = \tau_1\circ(x_1,\ldots,x_m,\tau_2)$, $\sim\underline{elem}(\tau_2)$ and $x_{m+1} \bar{\in} \tau_1$.

Proof. By induction on $\tau \in T$. □

Part (a) of this proposition is a converse of Proposition 5.11, and its graphical interpretation may be obtained by reversing that of 5.11. It will be applied frequently in conjunction with Proposition 5.10. Part (b) will be used in order to "split" a tree τ into two parts for further processing. Graphically we shall interpret it by:



5.13 Containment. If $\square K_L$ and $*(z \xrightarrow{\tau} v)_N$ and $z \in L$ then $N \subseteq L$.

Proof. By induction on $\tau \in T$. Intuitively since K is a closed FSD, and $z \in L$, all descendants of z must be in L. □

5.14 Conservation. If $*\square_c K_L$ and $(\forall u)[u \in L \Rightarrow c'(u) = c(u)]$ then $*\square_{c'} K_L$.

Proof. Each triple in K is compatible with c and thus also compatible with c' (easy by induction on T). □

In Chapter 8 we advocate a separate approach for linear lists because of the different way in which they are used. However, since we wish to exhibit a correctness proof of the Fischer-Galler algorithm, we shall introduce a restricted concept of linear lists into our general definitions.

5.15 Linearity. Let the predicate linear be defined by:

$$\underline{\text{linear}}(\tau) \triangleq [\underline{\text{var}}(\tau) \wedge \tau = x_1] \vee [\underline{\text{linear}}(\tau.1) \wedge \bigwedge_{j=2}^{|\tau|} \text{atom}(\tau.j)] \quad .$$

Intuitively, in a linear list the first field denotes a pointer to the next list element, and other fields contain arbitrary information. Now the linearity property can be expressed as:

If $*(u \xrightarrow{\tau} v, \lambda_1, \ldots, \lambda_n)_L \wedge *(u \xrightarrow{\sigma} w, \mu_1, \ldots, \mu_m)_N \wedge \underline{\text{linear}}(\tau)$ $\wedge \underline{\text{linear}}(\sigma)$ then

$$(\exists\xi)\{\underline{\text{linear}}(\xi) \wedge ([*(u \xrightarrow{\tau} v \xrightarrow{\xi} w,\mu_1,\ldots,\mu_m)_N \wedge \sigma = \tau\circ\xi]$$

$$\vee [*(u \xrightarrow{\sigma} w \xrightarrow{\xi} v,\lambda_1,\ldots,\lambda_n)_L \wedge \tau = \sigma\circ\xi])\}$$

$$\wedge \{\tau = \sigma \Rightarrow v = w\}$$

Essentially, this property means that if there is a linear path from u to v, and from u to w, then either v is an ancestor of w, or vice-versa, or $v = w$.

The proof is trivial by induction on τ (or σ), and using the definition of <u>linear</u>. □

It is easy to show (using Proposition 4.13) that in all transformations of FSD's proved in this chapter the function R_K is preserved (cf. Definition 4.12), i.e. if the transformation is $*K_L$ into $*K'_L$, $(L' \subseteq L)$ then the following holds:

$$(\forall u)[u \in L' \Rightarrow R_{K'}(u) = R_K(u)] \quad .$$

CHAPTER 6

TRANSFORMATION OF FREE STATE DESCRIPTIONS UNDER ASSIGNMENT STATEMENTS

In previous chapters we have shown that FSD's can be adequately used for static representation of data structures. In this chapter we shall see the transformation of FSD's when assignment statements are performed.

In the first place we shall define the concept of the value to which an expression is bound through an FSD, consistent with the definition of VAL_c (cf. Definition 2.7).

6.1 Definition. Let E be a program expression, and assume $*K_L$. Then $val_K(E)$ is defined by:

$$val_K(E) \triangleq \begin{cases} E & \text{if } E \text{ is a variable or a constant} \\ \theta(val_K(E_1),\ldots,val_K(E_p)) & \text{if } E = \theta(E_1,\ldots,E_p) \\ v_i & \text{if } E = E_1.i \text{ and } *K_L \Rightarrow \\ & val_K(E_1) \xrightarrow{<u,x_1,\ldots,x_p>} (v_1,\ldots,v_p) \\ & \text{for some } p, u, v_1,\ldots,v_p \end{cases}$$

Notice that in the last case, the value of $val_K(E)$ is defined only if $val_K(E_1)$ is defined and belongs to L. The existence of such $p, u, v_1,\ldots,v_p$ is guaranteed by 5.12(b) and 5.10.

The following proposition shows that the definition of val_K is consistent with that of VAL_c:

6.2 Proposition. If $*_cK_L$ and E is a program expression, then either $val_K(E)$ is undefined or $val_K(E) = VAL_c(E)$.

Proof. By induction on E:

If E is a variable or constant, then $val_K(E) = VAL_c(E)$ holds trivially.

If $E = \theta(E_1,\ldots,E_p)$, then by inductive hypothesis, either $val_K(E_i) = VAL_c(E_i)$, $i = 1,\ldots,p$, and thus $val_K(E) = VAL_c(E)$, or some $val_K(E_i)$ is undefined or not in L, and then $val_K(E)$ is undefined.

Finally, if $E = E_1.i$, then by inductive hypothesis, either $val_K(E_1)$ is undefined or not in L, and then $val_K(E)$ is undefined, or $val_K(E_1) = VAL_c(E_1)$. Let $u = val_K(E_1)$. Since $u \in L$, by 5.12(b), 5.10 and 5.12(a) we can write:

$$u \xrightarrow{<u,x_1,\ldots,x_p>}_c (v_1,\ldots,v_p) \xrightarrow{(\tau_1,\ldots,\tau_p)}_c \underline{w}$$

for some p, $v_1,\ldots,v_p$, $\tau_1,\ldots,\tau_p$ and $\underline{w}$.

Finally by Proposition 4.5,

$$\begin{aligned} c(u) &= R(u,<u,x_1,\ldots,x_p>,(v_1,\ldots,v_p)) \\ &= \{R_2(x_1,(v_1,\ldots,v_p)),\ldots,R_2(x_p,(v_1,\ldots,v_p))\} \\ &= \{v_1,\ldots,v_p\} \end{aligned}$$

and thus

$$VAL_c(E) = (c(VAL_c(E_1))).i = (c(u)).i = v_i \quad . \ \square$$

Notice that $VAL_c(E) \in L$ does not necessarily imply that $val_K(E)$ is defined, since for some subexpression of E the value of VAL_c might not be in L, leaving val_K undefined.

The next two propositions will show how FSD's are transformed under assignment statement:

6.3 Proposition. Let $c_0: M_0 \to G_0$ be a contents function, and let $*_{c_0} K_L$. Let E_1 and E_2 be two program expressions and let $v_1 = VAL_{c_0}(E_1)$ and $v_2 = VAL_{c_0}(E_2)$. Then the following hold:

(a) If $K = (v_1 \xrightarrow{<v_1,x_1,\ldots,x_p>} (w_1,\ldots,w_p),\lambda_1,\ldots,\lambda_n)$ (and thus $v_1 \in L$), then:

(i) $*K_L\{E_1.i \leftarrow E_2\}*(v_1 \xrightarrow{<v_1,x_1,\ldots,x_p>} \underline{w}',\lambda_1,\ldots,\lambda_n)_L$

(ii) $\neg K_L \wedge v_2 \in (L \cup A)\{E_1.i \leftarrow E_2\}\neg(v_1 \xrightarrow{<v_1,x_1,\ldots,x_p>} \underline{w}',\lambda_1,\ldots,\lambda_n)_L$

where $\underline{w}' = (w_1,\ldots,w_{i-1},v_2,w_{i+1},\ldots,w_p)$.

(b) If $K = (v_1 \xrightarrow{<v_1,\tau_1,\ldots,\tau_p>} \underline{w},\lambda_1,\ldots,\lambda_n)$ (thus $v_1 \in L$), and $v_2 \in A$, then:

(i) $*K_L\{E_1.i \leftarrow E_2\}*(v_1 \xrightarrow{\sigma} \underline{w},\lambda_1,\ldots,\lambda_n)_{L \sim S(\tau_i)}$

(ii) $\neg K_L \wedge \underline{elem}(\tau_i)\{E_1.i \leftarrow E_2\}\neg(v_1 \xrightarrow{\sigma} \underline{w},\lambda_1,\ldots,\lambda_n)_L$

where $\sigma = <v_1,\tau_1,\ldots,\tau_{i-1},v_2,\tau_{i+1},\ldots,\tau_p>$.

(c) If $v_1 \notin L$ but $v_1 \in M_0$, then

$$*\neg K_L\{E_1.i \leftarrow E_2\}*\neg K_L \quad .$$

(d) If $*K_L\{E_1.i \leftarrow E_2\}*K_L'$, as in parts (a), (b) and (c), then:

$$(\forall u)[u \in (L' \sim v_1) \Rightarrow R_K(u) = R_{K'}(u)]$$
$$\wedge \{v_1 \in L' \Rightarrow [(\forall j)(1 \leq j \leq |R_K(v_1)| \wedge j \neq i \Rightarrow$$
$$R_K(v_1).j = R_{K'}(v_1).j)] \wedge [R_{K'}(v_1).i = v_2]\}$$

Proof.

(a) According to the Definition 2.8, we have to prove:

$$*_{c_1}(v_1 \xrightarrow{<v_1,x_1,\ldots,x_p>} \underline{w}',\lambda_1,\ldots,\lambda_n)_L$$

where

$$c_1(x) \triangleq \underline{\text{if}}\ x=v_1\ \underline{\text{then}}\ y\ \underline{\text{else}}\ c_0(x)$$

and

$$y \triangleq \{c_0(x).1,\ldots,c_0(x).i\text{-}1,v_2,c_0(x).i\text{+}1,\ldots,c_0(x).p\} \quad .$$

Thus c_1 agrees with c_0 for all arguments, except v_1. Consequently $\lambda_1,\ldots,\lambda_n$ are compatible with c_1 since v_1 cannot occur in any λ_i (it occurs already in $<v_1,x_1,\ldots,x_p>$). Also, it is easy to see that $v_1 \xrightarrow{<v_1,x_1,\ldots,x_p>} \underline{w}'$ is compatible with c_1. The reference set associated with each triple did not change, and thus the set L remains the same. As for part (ii), all the endpoints in $\lambda_1,\ldots,\lambda_n$ are in $L \cup A$, and so are $w_1,\ldots,w_p$; by hypothesis $v_2 \in L \cup A$ and thus $\underline{w}' \subseteq L \cup A$. Consequently, the FSD remains closed.

(b) Comes as an easy consequence of part (a), by using composition and decomposition (cf. 5.11 and 5.12). Notice that $<v_1,\tau_1,\ldots,\tau_p> = <v_1,x_1,\ldots,x_p> \circ (\tau_1,\ldots,\tau_p)$.

(c) Follows directly from Proposition 5.14.

(d) In case of part (a), if $u = v_1$, then the result follows trivially by definition of R_K. If $u \in (L' \sim v_1)$, then, since $L' \subseteq L$, u must belong to the reference set associated with some λ_i, and the value $R_{K'}(u) = R_K(u)$, since it depends only on λ_i itself (cf. Definitions 4.12 and 4.4). Case of part (b) follows from part (a) and the remarks at the end of Chapter 5. Finally case (c) is trivial. □

Notice that in formulating this last proposition, we use the function VAL_c and not val_K as one might expect. In practice, however, we shall always use the function val_K in applying parts (a) and (b), and the Proposition 6.2 allows us to do so. In case (c), $val_K(E_1)$ would be obviously undefined; usually in such a case we would be working with more than one FSD describing the state of computation, and the value of $VAL_c(E_1)$ might be given by one of them.

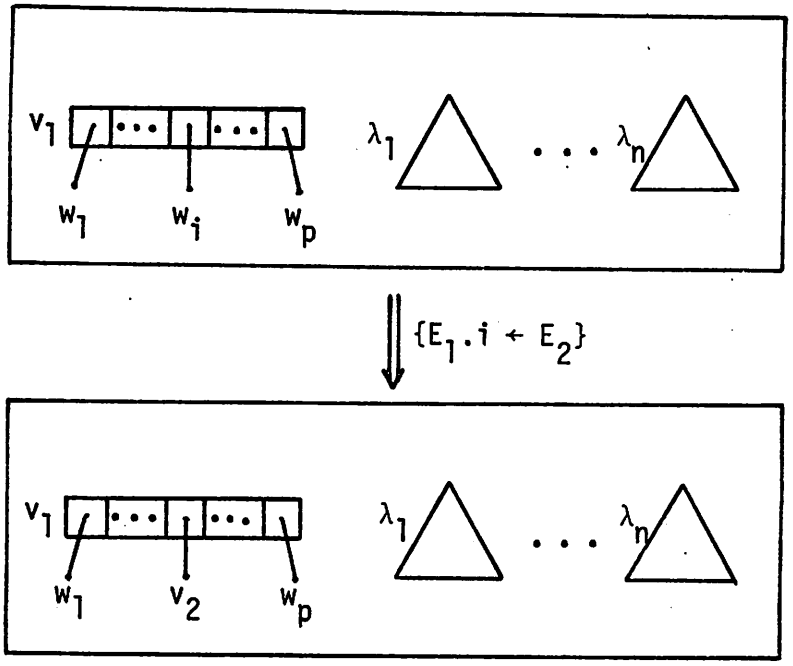
6.4 Examples. (Assume that I is a program variable and atom(a).)

(i) $$*(I \xrightarrow{<I,x_1,x_2>} (u,v))_{\{I\}}\{I.2 \leftarrow a\}*(I \xrightarrow{<I,x_1,x_2>} (u,a))_{\{I\}}$$

(ii) $$\natural(I \xrightarrow{<I,x_1,x_2>} (I,I))_{\{I\}}\{I.1 \leftarrow a\}\natural(I \xrightarrow{<I,x_1,x_2>} (a,I))_{\{I\}}$$

(iii) $$\natural(I \xrightarrow{<I,x_1,x_2>} (u,v),\ u \xrightarrow{<u,x_1,x_2>} (z,a),\ v \xrightarrow{<v,x_1,a>} z,$$
$$z \xrightarrow{<z,x_1,a>} t \xrightarrow{<t,x_1,x_2>} (u,a))_{\{I,u,v,z,t\}}$$
$$\cdot\ \{(I.1).2 \leftarrow (I.2).1\}$$
$$\natural(I \xrightarrow{<I,x_1,x_2>} (u,v),\ u \xrightarrow{<u,x_1,x_2>} (z,z),\ v \xrightarrow{<v,x_1,a>} z,$$
$$z \xrightarrow{<z,x_1,a>} t \xrightarrow{<t,x_1,x_2>} (u,a))_{\{I,u,v,z,t\}}$$

We can interpret Proposition 6.3 graphically by:



6.5 Proposition. Let $c_o: M_o \to G_o$ be a contents function, and let $*_{c_o} K_L$. Let I be a program variable, and let $E_1,\ldots,E_p$ be program expressions $(p \geq 1)$. Let $\underline{v} = (v_1,\ldots,v_p)$, where $v_i = VAL_{c_o}(E_i)$, $i = 1,\ldots,p$, and assume $K = (\lambda_1,\ldots,\lambda_n)$. Then the following hold:

(a) (i) $$*K_L\{I \leftarrow TUPLE(E_1,\ldots,E_p)\} \cdot (\exists z)[*(I \xrightarrow{<I,x_1,\ldots,x_p>} \underline{v}',\lambda_1',\ldots,\lambda_n')_{L' \cup I}]$$

(ii) $$\square K_L \wedge \underline{v} \subseteq (L \cup A)\{I \leftarrow TUPLE(E_1,\ldots,E_p)\} \cdot (\exists z)[\square(I \xrightarrow{<I,x_1,\ldots,x_p>} \underline{v}',\lambda_1',\ldots,\lambda_n')_{L' \cup I}]$$

(iii) $$*\square K_L\{I \leftarrow TUPLE(E_1,\ldots,E_p)\}*\square(\lambda_1',\ldots,\lambda_n')_{L'}$$

(iv) $$*\square K_L \wedge I \notin L\{I \leftarrow TUPLE(E_1,\ldots,E_p)\}*\square K_L$$

(b) If $*K_L\{I \leftarrow \text{TUPLE}(E_1,\ldots,E_p)\}*K_1L_1$ as in part (a), then:

$$(\forall u)[u \in L \Rightarrow R_{K_1}(u) = [R_K(u)]_z^I] \wedge [I \in L_1 \Rightarrow R_{K_1}(I) = \{\!|v_1',\ldots,v_p'|\!\}]$$

where

$$v_i' = [v_i]_z^I \qquad i = 1,\ldots,p$$

$$\underline{v}' = (v_1',\ldots,v_p')$$

$$\lambda_i' = [\lambda_i]_z^I \qquad i = 1,\ldots,n$$

$$L' = [L]_z^I$$

Proof.

(a) For part (i), we have to prove, according to the Definition 2.9, that $*_{c_o} K_L$ implies:

$$(\exists z)[*_{c_1}(I \xrightarrow{<I,x_1,\ldots,x_p>} \underline{v}',\lambda_1',\ldots,\lambda_n')_{L'\cup I}]$$

where

$$c_1(x) \triangleq \begin{cases} [\{\!|v_1,\ldots,v_p|\!\}]_z^I = v' & \text{if } x = I \\ [c_o(x)]_z^I & \text{if } x \neq I,\ x \in [M_o]_z^I \end{cases}$$

Since all references associated with λ_i are in M_o, any reference x in λ_i' is in $[M_o]_z^I$, and $c_1(x)$ agrees with $[c_o(x)]_z^I$. Thus λ_i' is compatible with c_1. Clearly $I \xrightarrow{<I,x_1,\ldots,x_p>} \underline{v}'$ is compatible with c_1. Also, the set of references in $\lambda_1',\ldots,\lambda_n'$ is L', and thus the result follows.

As for part (ii), we have $\underline{v} \subseteq (L \cup A)$, and thus $\underline{v}' \subseteq (L' \cup A)$. Clearly, all the endpoints of λ_i are in $L \cup A$ and those in λ_i' are in $L' \cup A$. Consequently the resulting FSD is closed.

Part (iii) is trivial since as shown above, all endpoints of λ_i' are in $L' \cup A$, and part (iv) is a consequence of (iii), since no substitution takes place: $L' = L$ and $\lambda_i' = \lambda_i$, $i = 1,\ldots,n$.

(b) For each $u \in L$, u must be in a reference set associated with some λ_i, and the result follows trivially from definition of R_K. □

6.6 Examples. (Assume that I is a program variable, and atom((a,b,d)).)

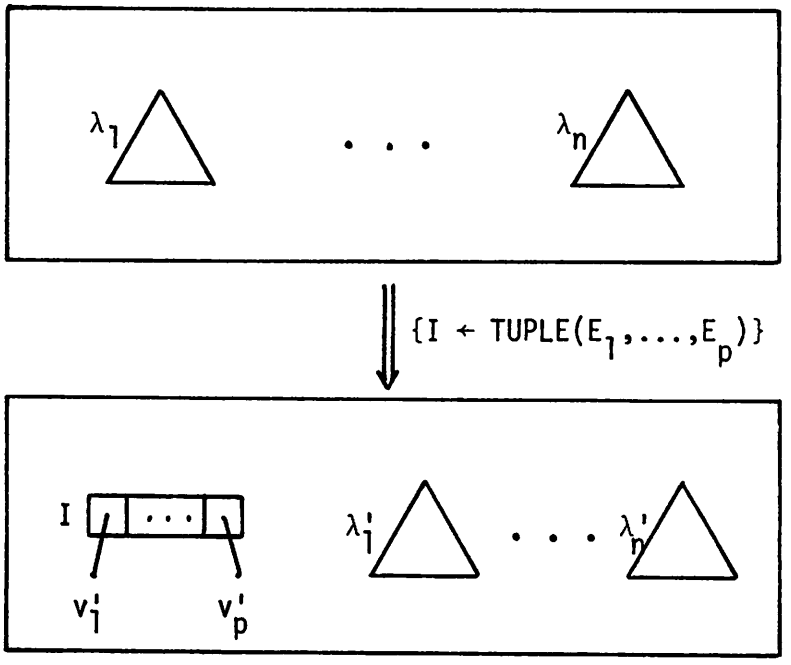
$$\text{(i)}\quad \underline{\text{true}}\{I \leftarrow \text{TUPLE}(a,b,d)\} \circ (I \xrightarrow{\langle I,x_1,x_2,x_3\rangle} (a,b,d))_{\{I\}}$$

$$\text{(ii)}\quad *(I \xrightarrow{\langle I,x_1,x_2\rangle} (u,w))_{\{I\}}\{I \leftarrow \text{TUPLE}(a,I)\}$$

$$(\exists z)[*(I \xrightarrow{\langle I,x_1,x_2\rangle} (a,z),\ z \xrightarrow{\langle z,x_1,x_2\rangle} (u,w))_{\{I,z\}}]$$

Notice that we used again the function VAL_c to compute v_i's, but in actual applications val_K will be used. The usual applications of Proposition 6.5 will be based on part (a)(ii), consisting simply of renaming the occurrences of I in K, and adding the new triple.

Graphically, we shall interpret Proposition 6.5 by:



CHAPTER 7

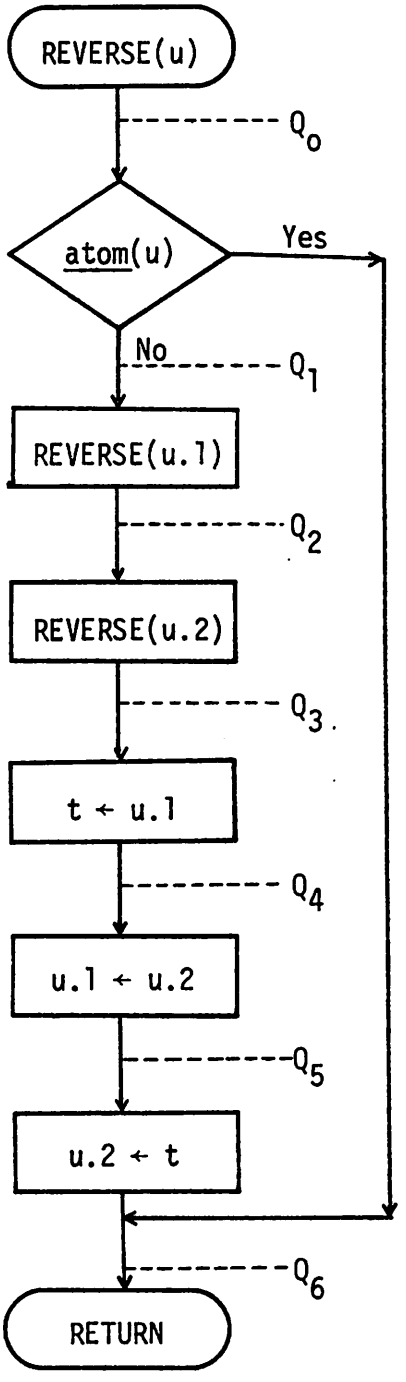
EXAMPLES OF APPLICATION

In this chapter we shall see some examples of application of the technique proposed in previous chapters. The purpose of these examples is two-fold. In the first place we hope that they will show the usefulness of the proposed technique, its relative simplicity and the intuitive reasoning hidden behind it. In the second place, the examples should illustrate the application of propositions proved in Chapters 4, 5 and 6 in different situations. We also show via examples how to handle procedure calls.

We chose Example 7.2 to carry out a more detailed proof, mentioning all the propositions being applied. In the subsequent examples we are more sketchy, skipping the obvious applications of some properties. We also drop the explicit use of existential quantifiers, but it should be clear from the context where they should be placed.

7.1 A Tree Reversing Algorithm. In developing our technique we had in mind applications to data structures with arbitrary sharing. However, in order to show that the price paid in applying our technique to free data structures is insignificant as compared with the method proposed in Burstall [1972], we start with one of his examples, slightly modified.

The purpose of the algorithm is to reverse a binary tree. We shall assume that for all trees τ in this proof order$(2,\tau)$ holds. The algorithm is presented as a recursive procedure in the following flow chart:


REVERSE(u)
Q_o
atom(u)
Yes
No
Q_1
REVERSE(u.1)
Q_2
REVERSE(u.2)
Q_3
t ← u.1
Q_4
u.1 ← u.2
Q_5
u.2 ← t
Q_6
RETURN

Let REV be an auxiliary function defined by:

$$REV(\tau) \triangleq \underline{\text{if}}\ \underline{\text{elem}}(\tau)\ \underline{\text{then}}\ \tau\ \underline{\text{else}}\ \langle \tau.0, REV(\tau.2), REV(\tau.1)\rangle$$

The inductive assertions are:

$$Q_0 \triangleq \pi(u\downarrow\tau, \lambda_1, \ldots, \lambda_n)_L$$

$$Q_1 \triangleq \pi(u \xrightarrow{\langle u, x_1, x_2\rangle} (z,w),\ (z,w)\downarrow(\sigma_1,\sigma_2), \lambda_1, \ldots, \lambda_n)_L \wedge \tau = \langle u, \sigma_1, \sigma_2\rangle$$

$$Q_2 \triangleq \pi(u \xrightarrow{\langle u, x_1, x_2\rangle} (z,w),\ z\downarrow\xi_1, w\downarrow\sigma_2, \lambda_1, \ldots, \lambda_n)_L$$
$$\wedge\ \tau = \langle u, \sigma_1, \sigma_2\rangle \wedge \xi_1 = REV(\sigma_1)$$

$$Q_3 \triangleq \pi(u \xrightarrow{\langle u, x_1, x_2\rangle} (z,w),\ z\downarrow\xi_1, w\downarrow\xi_2, \lambda_1, \ldots, \lambda_n)_L$$
$$\wedge\ \tau = \langle u, \sigma_1, \sigma_2\rangle \wedge \xi_1 = REV(\sigma_1) \wedge \xi_2 = REV(\sigma_2)$$

$$Q_4 \triangleq Q_3 \wedge t = z$$

$$Q_5 \triangleq \pi(u \xrightarrow{\langle u, x_1, x_2\rangle} (w,w),\ t\downarrow\xi_1, w\downarrow\xi_2, \lambda_1, \ldots, \lambda_n)_L$$
$$\wedge\ \tau = \langle u, \sigma_1, \sigma_2\rangle \wedge \xi_1 = REV(\sigma_1) \wedge \xi_2 = REV(\sigma_2)$$

$$Q_6 \triangleq \pi(u\downarrow\xi, \lambda_1, \ldots, \lambda_n)_L \wedge \xi = REV(\tau)$$

We shall now verify the above assertions by simple application of the propositions given in Chapters 5 and 6:

1. Since the procedure is recursive, we shall adopt the following inductive hypothesis:

$$(\forall u, \tau, \lambda_1, \ldots, \lambda_n)[Q_0\{REVERSE(u)\}Q_6]$$

and prove that $Q_0\{S\}Q_6$ holds, where S is the body of the

procedure. For a more detailed treatment of such proofs see Hoare [1971] and Morris [1971b].

2. $Q_0 \wedge \underline{atom}(u) \Rightarrow Q_6$: trivial, since by Proposition 5.9 $\tau = u$ and $REV(\tau) = \tau$.

3. $Q_0 \wedge \sim\underline{atom}(u) \Rightarrow Q_1$: by 5.10 and 5.12.

4. $Q_1\{REVERSE(u.1)\}Q_2$: follows by inductive hypothesis.

5. $Q_2\{REVERSE(u.2)\}Q_3$: by inductive hypothesis.

6. $Q_3\{t \leftarrow u.1\}Q_4$: trivial.

7. $Q_4\{u.1 \leftarrow u.2\}Q_5$: by Proposition 6.3(a).

8. $Q_5\{u.2 \leftarrow t\}Q_6$: By Proposition 6.3(a) we have after the assignment:

$$\pi(u \xrightarrow{\langle u,x_1,x_2\rangle} (w,t),t{\downarrow}\xi_1,w{\downarrow}\xi_2,\lambda_1,\ldots,\lambda_n)_L$$

$$\wedge\ \tau = \langle u,\sigma_1,\sigma_2\rangle \wedge \xi_1 = REV(\sigma_1) \wedge \xi_2 = REV(\sigma_2)$$

By 5.7 (tupling), 5.11(b) (composition) and using the definition of REV:

$$\pi(u{\downarrow}\xi,\lambda_1,\ldots,\lambda_n)_L \wedge \xi = REV(\tau)$$

which is Q_6.

This concludes the verifications, and we can state thus:

$$(\forall u,\tau)[\pi(u{\downarrow}\tau)_L\{REVERSE(u)\}\pi(u{\downarrow}REV(\tau))_L]$$

which proves the correctness of the procedure.

Notice that our assertions are analogous to those in Burstall's example. Our proof seems slightly more complicated because we treat procedure calls in a more rigorous way. Notice also, that we prove a fact which cannot be expressed in Burstall's notation, namely that the set L of relevant references remains the same.

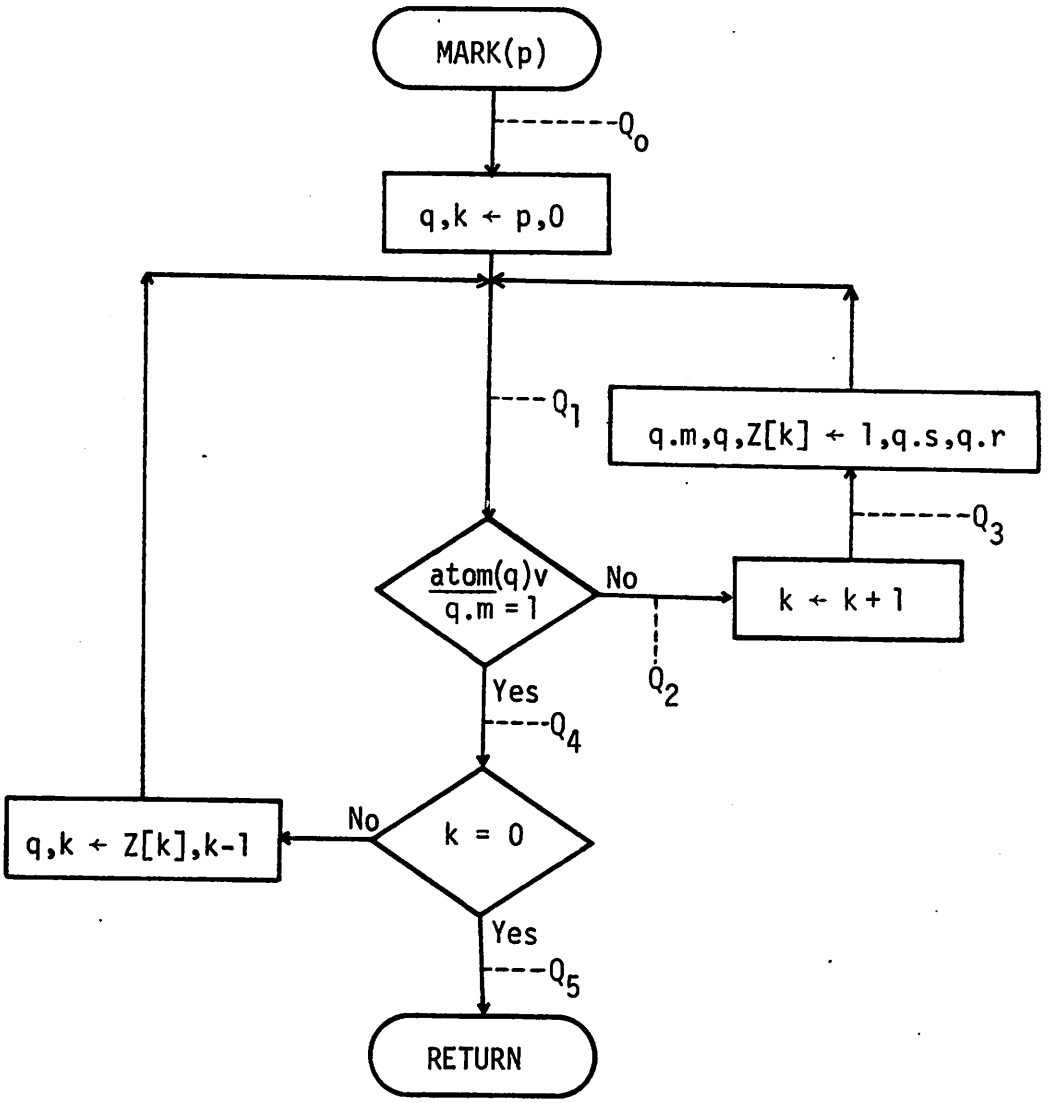
7.2 A Marking Algorithm With a Stack. We shall assume that all memory locations hold tuples composed of three fields, and thus for all trees τ in this proof, order$(3,\tau)$ holds. The first component is always an atom (initially 0), and the other two are either atoms or pointers. There are no restrictions about circularities or sharing. The purpose of the algorithm is to mark all the nodes reachable from a given location p, i.e., to change the first field from 0 to 1.

Since it is clear from the algorithm that assignments are made only to the mark fields, we shall not prove that the two pointer fields remain the same. It would be trivial however, by showing that $R_K(u).2$ and $R_K(u).3$ remain the same throughout the algorithm for all the relevant references u.

The stack will be represented by a potentially infinite array Z (starting with index 1). In this example we shall try to be very careful in using propositions proved in Chapters 5 and 6, mentioning explicitly each application (with the exception of Proposition 5.3 involving permutation of triples).

We follow the proof by a more informal graphical interpretation. The flow chart below represents the marking procedure MARK. We

shall use letters as symbolic selectors: m = 1, s = 2, r = 3. Q_0, Q_1, Q_2, Q_3 and Q_4 are the inductive assertions defined later:



We shall define two auxiliary predicates:

$$M(\tau) \triangleq \sim\underline{\text{elem}}(\tau) \Rightarrow [\tau.m = 1 \wedge M(\tau.s) \wedge M(\tau.r)]$$

$$U(\tau) \triangleq \sim\underline{\text{elem}}(\tau) \Rightarrow [\tau.m = 0 \wedge U(\tau.s) \wedge U(\tau.r)]$$

Thus $M(\tau)$ and $U(\tau)$ mean that for all references in τ, the mark field is 1 and 0, respectively. Notice that $\underline{\text{elem}}(\tau) \Rightarrow M(\tau) \wedge U(\tau)$. Graphically, we shall represent marked and unmarked trees by hatched and unhatched triangles, respectively.

We shall define now the inductive assertions:

$$Q_0 \triangleq (\exists\tau,\underline{w})[\pi(p \xrightarrow{\tau} \underline{w})_N \wedge U(\tau)]$$

$$Q_1 \triangleq (\exists\tau_1,\tau_2,\sigma_1,\ldots,\sigma_k,\underline{w},\underline{v},\underline{v}_1,\ldots,\underline{v}_k)$$

$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q),\ q \xrightarrow{\tau_2} \underline{v},\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$

$$\wedge M(\tau_1) \wedge U(\tau_2) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)] \wedge (x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)\}$$

$$\wedge\ k \geq 0$$

$$Q_2 \triangleq (\exists\tau_1,\tau_3,\tau_4,\sigma_1,\ldots,\sigma_k,\underline{w},u,t,\underline{v},\underline{v}_1,\ldots,\underline{v}_k)$$

$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q),\ q \xrightarrow{<q,0,x_1,x_2>} (u,t) \xrightarrow{(\tau_3,\tau_4)} \underline{v},$$

$$[z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$

$$\wedge M(\tau_1) \wedge U(\tau_3) \wedge U(\tau_4) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)]$$

$$\wedge(x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)\}$$

$$\wedge\ k \geq 0$$

$$Q_3 \triangleq (\exists \tau_1,\tau_3,\tau_4,\sigma_1,\ldots,\sigma_{k-1},\underline{w},u,t,\underline{v},\underline{v}_1,\ldots,\underline{v}_{k-1})$$

$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_{k-1},q),\ q \xrightarrow{<q,0,x_1,x_2>} (u,t) \xrightarrow{(\tau_3,\tau_4)} \underline{v},$$

$$[z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k-1})_N$$

$$\wedge M(\tau_1) \wedge U(\tau_3) \wedge U(\tau_4) \wedge \bigwedge_{i=1}^{k-1}[U(\sigma_i) \wedge (x_{|\underline{w}|+i} \bar{\in} \tau_1)]$$

$$\wedge (x_{|\underline{w}|+k} \bar{\in} \tau_1)\}$$

$$\wedge\ k \geq 1$$

$$Q_4 \triangleq (\exists \tau_1,\sigma_1,\ldots,\sigma_k,\underline{w},\underline{v}_1,\ldots,\underline{v}_k)$$

$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k),\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$

$$\wedge M(\tau_1) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i} \bar{\in} \tau_1)]\} \wedge k \geq 0$$

$$Q_5 \triangleq (\exists \sigma,\underline{w})[\pi(p \xrightarrow{\sigma} \underline{w})_N \wedge M(\sigma)]$$

In all these assertions $z_1,\ldots,z_k$ stand for $Z[1],\ldots,Z[k]$.

We shall proceed now with the verifications imposed by the control paths of the algorithm:

1. Q_0 is guaranteed by Proposition 4.10 and the initial assumption that all nodes are unmarked.

2. $\underline{Q_0\{q,k \leftarrow p,0\}Q_1}$: By 5.6 we have:

$$Q_0 \Rightarrow \pi(p \xrightarrow{x_1} p,\ p \xrightarrow{\tau} \underline{w})_N$$

and after the assignment, with $p = q$:

$$\pi(p \xrightarrow{x_1} q,\ q \xrightarrow{\tau} \underline{w})_N$$

and consequently we get Q_1 with $k = 0$, $\underline{w} = ()$, $\underline{v} = \underline{w}$.

3. $\underline{[Q_1 \wedge \sim\underline{atom}(q) \wedge q.m \neq 1] \Rightarrow Q_2}$: Let us define Q_1' to be identical to Q_1 but including in addition the term $\sim\underline{elem}(\tau_2)$.

Claim 1: $[Q_1 \wedge \sim\underline{atom}(q) \wedge q.m \neq 1] \Rightarrow Q_1'$

Since $\sim\underline{atom}(q)$, thus $q \in N$ (either $\underline{elem}(\tau_2)$ and $q \in \underline{v} \subseteq N$ or $\sim\underline{elem}(\tau_2)$ and $q \in S(\tau_2) \subseteq N$). Thus:

Case 1: $q \in S(\tau_1) \Rightarrow q.m = 1 \Rightarrow$ contradiction

Case 2: $q \in S(\tau_2) \Rightarrow S(\tau_2) \neq \emptyset \Rightarrow \sim\underline{elem}(\tau_2) \Rightarrow Q_1'$

Case 3: $q \in S(\sigma_j)$ for some $1 \leq j \leq k$.

Consequently $q \notin S(\tau_2)$, and thus $\underline{elem}(\tau_2)$. By 5.4(b) we can delete $q \xrightarrow{\tau_2} \underline{v}$ from the FSD. By 5.12(b), $z_j \xrightarrow{\sigma_j} \underline{v}_j$ can be replaced by:

$$z_j \xrightarrow{\sigma_j'} (\underline{v}_j, q),\ q \xrightarrow{\sigma_j''} \underline{v}_j$$

with $\sim\underline{elem}(\sigma_j'')$ and $\sigma_j = \sigma_j' \circ (x_1, \ldots, x_{|\underline{v}_j|}, \sigma_j'')$.

Clearly, $U(\sigma_j) \Rightarrow U(\sigma_j') \wedge U(\sigma_j'')$. Thus we can take σ_j', σ_j'', $(\underline{v}_j, q)$, $\underline{v}_j$ as the new values of σ_j, τ_2, $\underline{v}_j$, $\underline{v}$, and Q_1' follows.

Claim 2: $Q_1' \Rightarrow Q_2$

Trivial, since $Q_1' \wedge q.m \neq 1 \Rightarrow \sim\underline{elem}(\tau_2) \Rightarrow$

$$\tau_2 = \langle 0, \tau_3, \tau_4 \rangle = \langle 0, x_1, x_2 \rangle \circ (\tau_3, \tau_4)$$

for some τ_3 and τ_4.

Also, $U(\tau_2) \Rightarrow U(\tau_3) \wedge U(\tau_4)$. By applying 5.12(a) we get Q_2.

4. $\underline{Q_2\{k \leftarrow k+1\}Q_3}$: Trivial; by substituting k-1 for k in Q_2 we get Q_3.

5. $\underline{Q_3\{q.m,q,Z[k] \leftarrow 1,q.s,q.r\}Q_1}$: By 6.3(b) we get after the assignment is performed:

$$\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_{k-1},y),\ y \xrightarrow{<y,1,x_1,x_2>} (q,z_k),$$

$$q \xrightarrow{\tau_3} \underline{v},\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k-1},\ z_k \xrightarrow{\tau_4} \underline{v})_N$$

$$\wedge\ M(\tau_1) \wedge U(\tau_3) \wedge U(\tau_4) \wedge \bigwedge_{i=1}^{k-1}[U(\sigma_i) \wedge (x_{|w|+i}\ \bar{\epsilon}\ \tau_1)]$$

$$\wedge\ (x_{|w|+k}\ \bar{\epsilon}\ \tau_1) \quad .$$

We shall show now that the two triples

$$p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_{k-1},y),\ y \xrightarrow{<y,1,x_1,x_2>} (q,z_k)$$

can be replaced by a triple

$$p \xrightarrow{\zeta} (\underline{w},z_1,\ldots,z_k,q)$$

where $M(\zeta)$ and $\bigwedge_{i=1}^{k+1}(x_{|w|+i}\ \bar{\epsilon}\ \zeta)$.

Intuitively, this fact is very clear since we shall compose τ_1 and $<y,1,x_1,x_2>$ in a convenient way to get ζ. By Proposition 5.6(b), we can extend the FSD by adding the triple:

$$(\underline{w},z_1,\ldots,z_{k-1},q,z_k) \xrightarrow{(x_1,\ldots,x_m,x_{m+2},x_{m+1})} (\underline{w},z_1,\ldots,z_k,q)$$

where $m = |\underline{w}| + k - 1$.

Then by 5.7 (tupling and untupling) we can decompose the last triple into:

$$\underline{w} \xrightarrow{(x_1,\ldots,x_{|w|})} (\underline{w},z_1,\ldots,z_k,q)$$

$$(z_1,\ldots,z_{k-1}) \xrightarrow{(x_{|w|+1},\ldots,x_m)} (\underline{w},z_1,\ldots,z_k,q)$$

$$(q,z_k) \xrightarrow{(x_{m+2},x_{m+1})} (\underline{w},z_1,\ldots,z_k,q)$$

By 5.11(b) (composition), we can replace

$$y \xrightarrow{<y,1,x_1,x_2>} (q,z_k),\ (q,z_k) \xrightarrow{(x_{m+2},x_{m+1})} (\underline{w},z_1,\ldots,z_k,q)$$

by:

$$y \xrightarrow{<y,1,x_{m+2},x_{m+1}>} (\underline{w},z_1,\ldots,z_k,q) \quad .$$

Now by 5.7 we can tuple this triple with those remaining above to get

$$(\underline{w},z_1,\ldots,z_{k-1},y) \xrightarrow{(x_1,\ldots,x_m,<y,1,x_{m+2},x_{m+1}>)} (\underline{w},z_1,\ldots,z_k,q)$$

and finally we compose this triple with $p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_{k-1},y)$ to get

$$p \xrightarrow{\zeta} (\underline{w},z_1,\ldots,z_k,q)$$

where $\zeta = \tau_1 \circ (x_1,\ldots,x_m,<y,1,x_{m+2},x_{m+1}>)$.

We can do this since $Q_3 \Rightarrow x_{|\underline{w}|+k} = x_{m+1}\ \bar{\epsilon}\ \tau_1$.

Clearly $M(\tau_1) \Rightarrow M(\zeta)$.

Also, since $x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1$ for $i = 1,\ldots,k$, easily

$x_{|w|+i} \bar{\epsilon} \zeta$ for $i = 1,\ldots,k+1$. Consequently, Q_1 follows.

6. $\underline{Q_1 \wedge (\underline{atom}(q) \vee q.m = 1) \Rightarrow Q_4}$:

Claim: $\underline{elem}(\tau_2)$

Case 1: $\underline{atom}(q) \Rightarrow \underline{elem}(\tau_2)$ by 5.9

Case 2: $\sim\underline{atom}(q) \wedge q.m = 1$

If $\sim\underline{elem}(\tau_2)$ then $q \in S(\tau_2)$ and $U(\tau_2) \Rightarrow q.m = 0$; consequently we must have $\underline{elem}(\tau_2)$.

Now, $\underline{elem}(\tau_2)$ means that we can delete $q \xrightarrow{\tau_2} \underline{v}$ from the FSD by virtue of the Proposition 5.4(b).

We shall rearrange the FSD now in order to get the form corresponding to that in Q_4. In the first place we expand (5.6(b)) the FSD by:

$$(\underline{w},z_1,\ldots,z_k,q) \xrightarrow{(x_1,\ldots,x_{|\underline{w}|},x_{m+1},x_{|w|+1},\ldots,x_m)} (\underline{w},q,z_1,\ldots,z_k)$$

where $m = |\underline{w}| + k$.

Combining this triple with $p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q)$ we get by 5.11(b):

$$p \xrightarrow{\zeta} (\underline{w},q,z_1,\ldots,z_k)$$

or

$$p \xrightarrow{\zeta} (\underline{w}',z_1,\ldots,z_k)$$

where

$$\underline{w}' = (\underline{w},q)$$

and

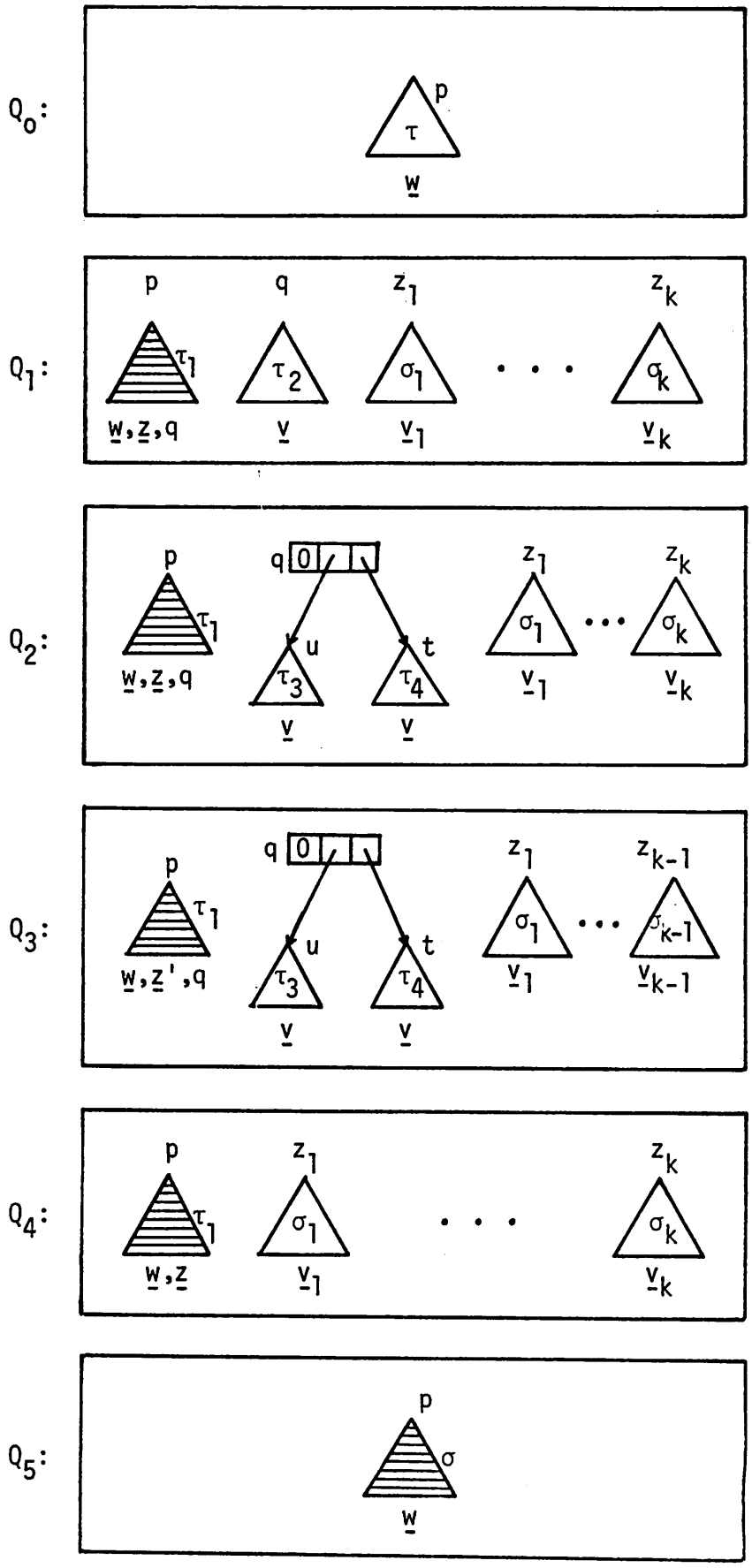
$$\zeta = \tau_1 \circ (x_1,\ldots,x_{|\underline{w}|},x_{m+1},x_{|\underline{w}|+1},\ldots,x_m) \quad .$$

Clearly $M(\tau_1) \Rightarrow M(\zeta)$ and $x_{|\underline{w}'|+i} \bar{\epsilon} \zeta$ for $i = 1,\ldots,k$, and we finally get Q_4.

7. $\underline{Q_4 \wedge k = 0 \Rightarrow Q_5}$: trivial

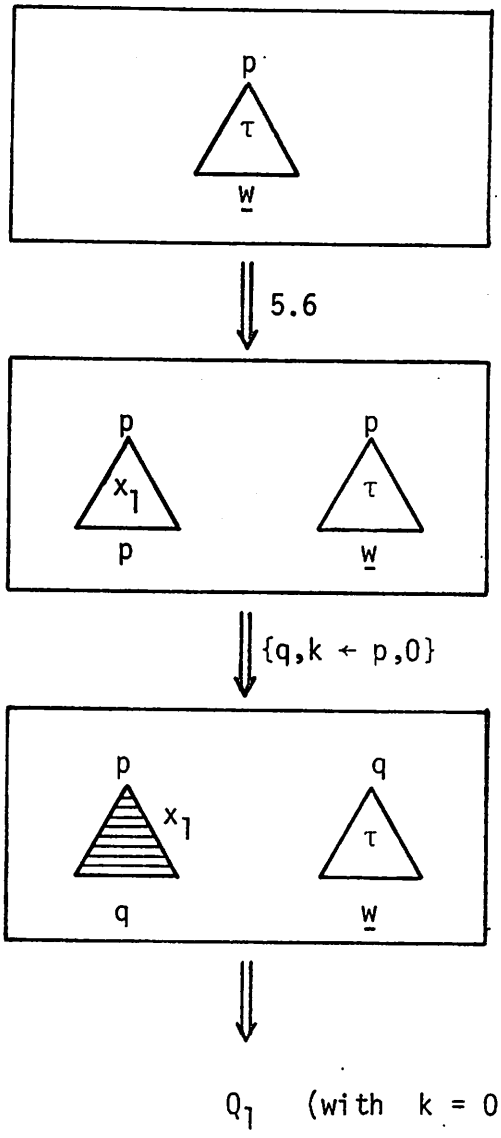
8. $\underline{Q_4 \wedge k \neq 0\{q,k \leftarrow Z[k],k-1\}Q_1}$: Trivial by a simple change of names; z_k becomes q and $k-1$ is the new value of k. □

This concludes the verification of inductive assertions. It was rather lengthy and tedious since we wanted to exhibit one detailed proof. Clearly Q_5 shows the desired property of the algorithm. We shall give now the graphical interpretation of this proof. In the first place we show the inductive assertions ($\underline{z}$ stands for $(z_1,\ldots,z_k)$ and z' for $(z_1,\ldots,z_{k-1})$):


Q_0:
p
τ
w
Q_1:
p
q
z_1
z_k
τ_1
τ_2
σ_1
σ_k
w, z, q
v
v_1
v_k
Q_2:
p
q 0
z_1
z_k
τ_1
u
t
σ_1
σ_k
w, z, q
τ_3
τ_4
v_1
v_k
v
v
Q_3:
p
q 0
z_1
z_{k-1}
τ_1
u
t
σ_1
σ_{k-1}
w, z', q
τ_3
τ_4
v_1
v_{k-1}
v
v
Q_4:
p
z_1
z_k
τ_1
σ_1
σ_k
w, z
v_1
v_k
Q_5:
p
σ
w

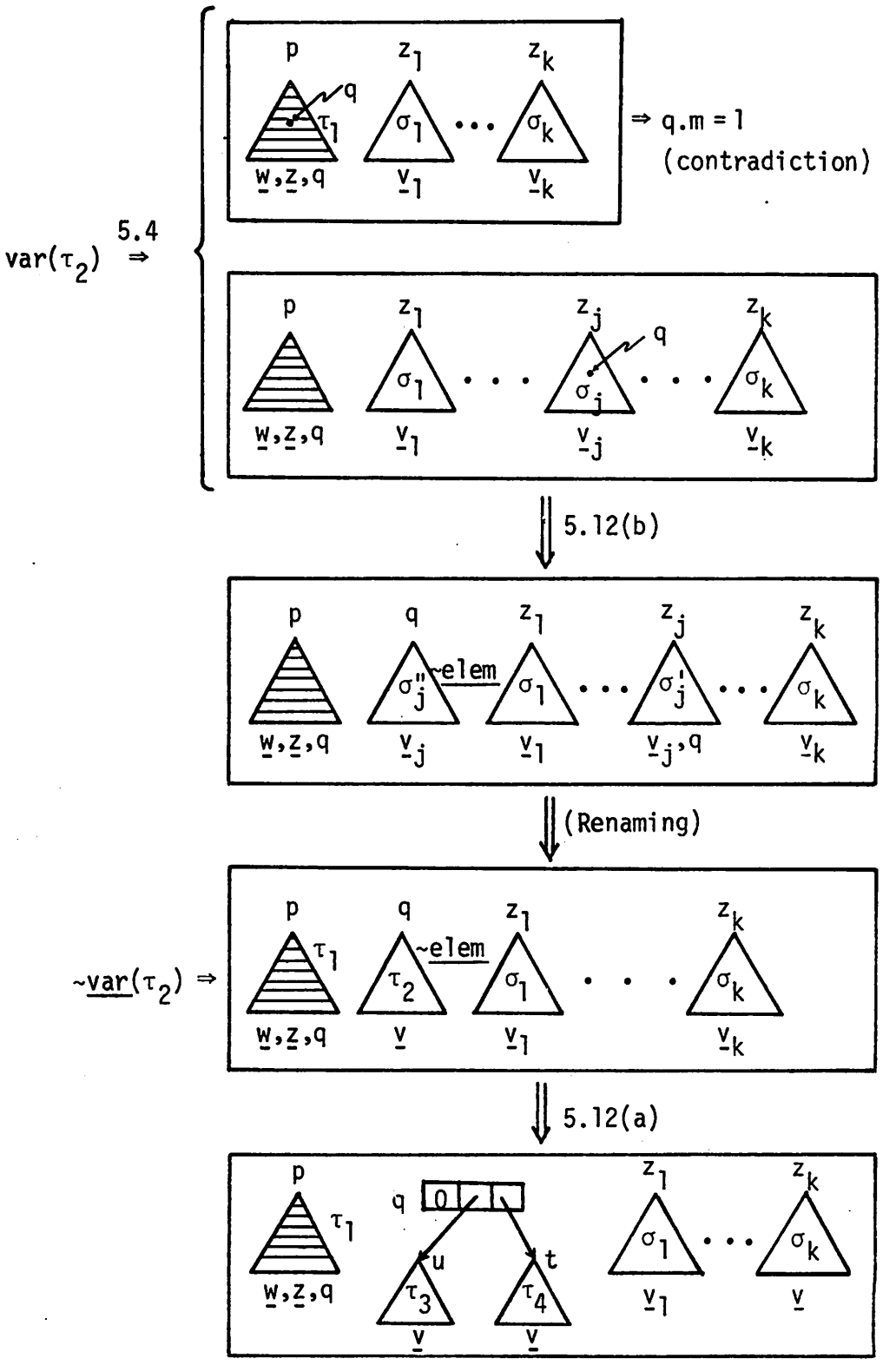
We proceed now with the verifications:

2. $Q_0\{q,k \leftarrow p,0\}Q_1$:



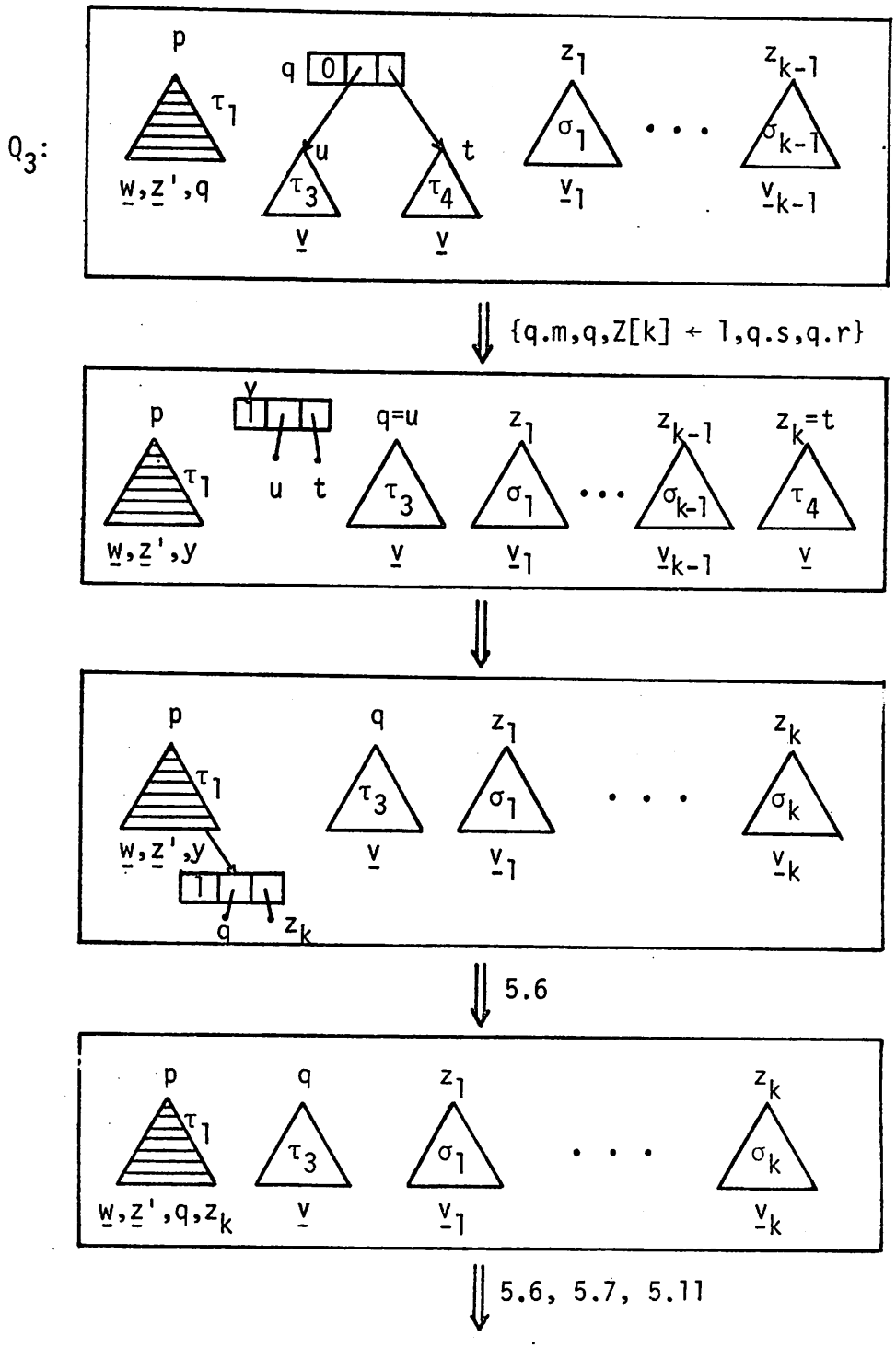
Q_1 (with $k = 0$)

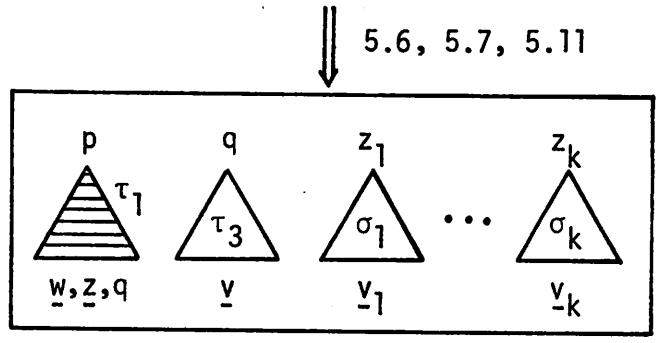
3. $[Q_1 \wedge \sim\underline{atom}(q) \wedge q.m \neq 1] \Rightarrow Q_2$:



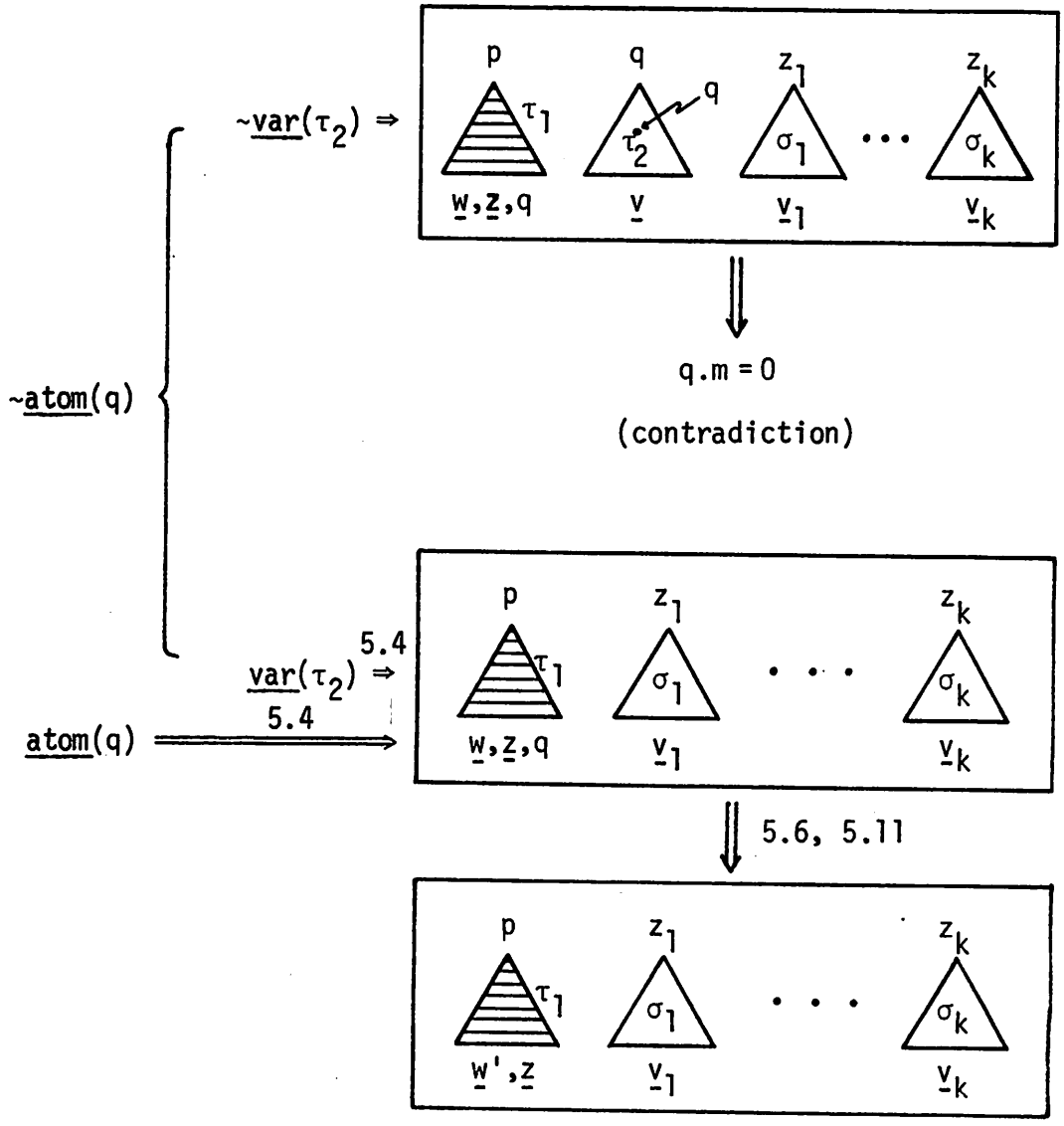
4. $Q_2\{k \leftarrow k+1\}Q_3$: trivial.

5. $Q_3\{q.m,q,Z[k] \leftarrow 1,q.s,q.r\}Q_1$:



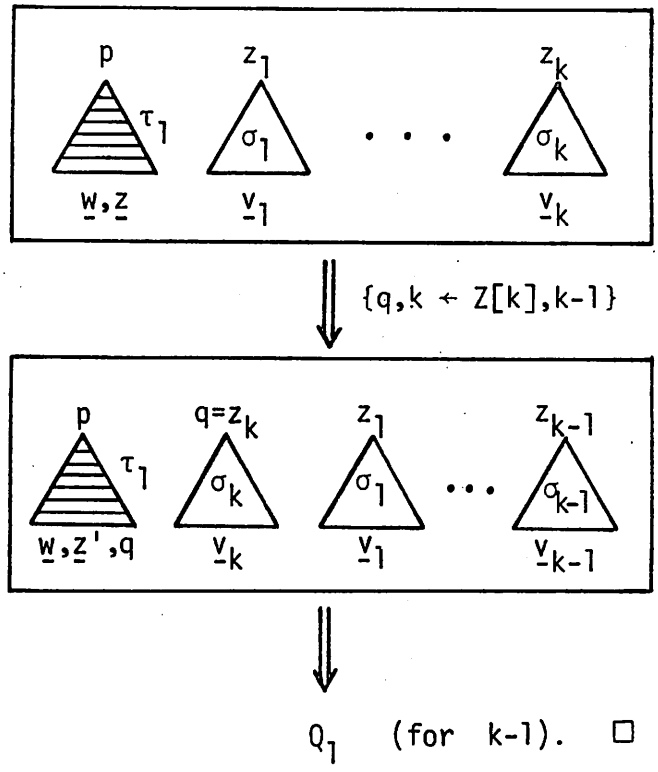


6. $Q_1 \wedge [\underline{atom}(q) \vee q.m = 1] \Rightarrow Q_4$:



7. $Q_4 \wedge k = 0 \Rightarrow Q_5$: trivial.

8. $Q_4 \wedge k \neq 0\{q,k \leftarrow Z[k],k-1\}Q_1$:



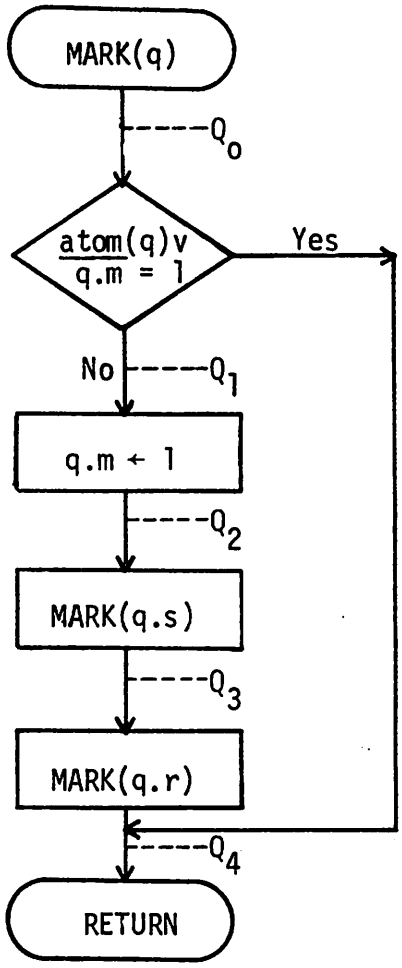
Q_1 (for k-1). □

This concludes our graphical interpretation of the proof.

An interesting aspect of this proof is that it does not depend on the particular choice of initial values of τ and $\underline{w}$ in Q_0 (according to Proposition 4.10 they are not unique). However the final tree σ in Q_5 will be a pre-order traversal tree (cf. Knuth [1968]) imposed by the algorithm itself. The necessary adjustments of the initial tree are due to the application of Proposition 5.12(b) in step 3 of the proof. In particular, if the initial tree τ were already a pre-order tree, we would be able to prove that the application of this proposition would not be necessary.

7.3 A Recursive Marking Algorithm

The purpose of this algorithm is the same as that of the previous example, and we shall assume the same definitions for m, s, r, $U(\tau)$ and $M(\tau)$. The main difference between the two algorithms is that in this one we shall use explicit recursion, and our proof will account for this fact. Many details of the proof are similar to that of Example 7.2. Consequently we shall give only sketchy verifications.



The inductive assertions are:

$$Q_0 \triangleq (\exists \tau_1,\tau_2,\underline{w},\underline{v},\sigma_1,\ldots,\sigma_k,\underline{v}_1,\ldots,\underline{v}_k)$$
$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q),\ q \xrightarrow{\tau_2} \underline{v},\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k}]_N$$
$$\wedge M(\tau_1) \wedge U(\tau_2) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)] \wedge (x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)\}$$

$$Q_1 \triangleq (\exists \tau_1,\tau_3,\tau_4,\underline{w},\underline{v},\sigma_1,\ldots,\sigma_k,\underline{v}_1,\ldots,\underline{v}_k)$$
$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q),\ q \xrightarrow{<q,0,x_1,x_2>} (u,t) \xrightarrow{(\tau_3,\tau_4)} \underline{v},$$
$$[z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$
$$\wedge M(\tau_1) \wedge U(\tau_3) \wedge U(\tau_4) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)]$$
$$\wedge (x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)\}$$

$$Q_2 \triangleq (\exists \tau_1,\tau_3,\tau_4,\underline{w},\underline{v},\sigma_1,\ldots,\sigma_k,\underline{v}_1,\ldots,\underline{v}_k)$$
$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,t,u),\ u \xrightarrow{\tau_3} \underline{v}\ ,\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k},$$
$$t \xrightarrow{\tau_4} \underline{v})_N$$
$$\wedge M(\tau_1) \wedge U(\tau_3) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)] \wedge (x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)$$
$$\wedge (x_{|\underline{w}|+k+2}\ \bar{\epsilon}\ \tau_1) \wedge u = q.s \wedge t = q.r\}$$

$$Q_3 \triangleq (\exists \tau_1,\tau_4,\underline{w},\underline{v},\sigma_1,\ldots,\sigma_k,\underline{v}_1,\ldots,\underline{v}_k)$$
$$\{\pi(p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,t),\ t \xrightarrow{\tau_4} \underline{v},\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$
$$\wedge M(\tau_1) \wedge U(\tau_4) \wedge \bigwedge_{i=1}^{k}[U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)] \wedge (x_{|\underline{w}|+k+1}\ \bar{\epsilon}\ \tau_1)$$
$$\wedge t = q.r\}$$

$$Q_4 \triangleq (\exists \tau_1, \underline{w}, \sigma_1, \ldots, \sigma_k, \underline{v}_1, \ldots, \underline{v}_k)\{\pi(p \xrightarrow{\tau_1} (\underline{w}, z_1, \ldots, z_k),\ [z_i \xrightarrow{\sigma_i} \underline{v}_i]_{i=1}^{k})_N$$
$$\wedge\ M(\tau_1) \wedge \bigwedge_{i=1}^{k} [U(\sigma_i) \wedge (x_{|\underline{w}|+i}\ \bar{\epsilon}\ \tau_1)]\}$$

Intuitively, the assertions describe the state of computation during a recursive call MARK(q); p is the root of initial tree and we can think of $z_1, \ldots, z_k$ as the values stored on the stack in the usual implementation of recursion. Notice that $p, q, z_1, \ldots, z_k$ are constants in each recursive call of MARK, and thus in the assertions Q_0 - Q_4.

Since the procedure MARK is recursive, we shall use the following inductive hypothesis for the recursive calls:

$$(\forall p, q, z_1, \ldots, z_k)[Q_0\{\text{MARK}(q)\}Q_4] \quad .$$

We proceed now with the verification of assertions:

1. $\underline{[Q_0 \wedge \sim \underline{\text{atom}}(q) \wedge q.m \neq 1] \Rightarrow Q_1}$: As in step 3 of Example 7.2, we prove that Q_0 can be replaced by Q_0' where $\sim\underline{\text{elem}}(\tau_2)$, and then Q_1 follows easily.

2. $\underline{Q_1\{q.m \leftarrow 1\}Q_2}$: By Proposition 6.3(b), we can replace $q \xrightarrow{<q,0,x_1,x_2>} (u,t)$ in Q_1 by $q \xrightarrow{<q,1,x_1,x_2>} (u,t)$ after the assignment is performed.

Then we can replace

$$p \xrightarrow{\tau_1} (\underline{w}, z_1, \ldots, z_k, q),\ q \xrightarrow{<q,1,x_1,x_2>} (u,t)$$

by

$$p \xrightarrow{\zeta} (\underline{w}, z_1, \ldots, z_k, t, u)$$

with

$$\zeta = \tau_1 \circ (x_1,\ldots,x_m,\langle q,1,x_{m+2},x_{m+1}\rangle)$$

(cf. step 5 in Example 7.2). Thus $M(\zeta)$ and $x_{|\underline{w}|+i} \tilde{e} \zeta$ for $i = 1,\ldots,k+2$, and Q_2 follows.

3. $\underline{Q_2\{MARK(q.s)\}Q_3}$: Q_2 has the form of Q_0, with u for q, $z_1,\ldots,z_k$, t for $z_1,\ldots,z_k,z_{k+1}$. By inductive hypothesis we get Q_3.

4. $\underline{Q_3\{MARK(q.r)\}Q_4}$: By inductive hypothesis, as above.

5. $\underline{Q_0 \wedge [\underline{atom}(q) \vee q.m = 1] \Rightarrow Q_4}$: As in step 6 of Example 7.2, we show that $\underline{elem}(\tau_2)$ holds, and then replace $p \xrightarrow{\tau_1} (\underline{w},z_1,\ldots,z_k,q)$ by $p \xrightarrow{\zeta} (\underline{w}',z_1,\ldots,z_k)$ with $\underline{w}' = (\underline{w},q)$.

This concludes the verifications. By Proposition 4.10 and the initial assumption we have:

$$\pi(p \xrightarrow{\tau} \underline{w})_N \wedge U(\tau)$$

or,

$$\pi(p \xrightarrow{x_1} q,\ q \xrightarrow{\tau} \underline{w})_N \wedge M(x_1) \wedge U(\tau) \wedge p = q \quad .$$

As a result of the proof above, we can write (for $k = 0$):

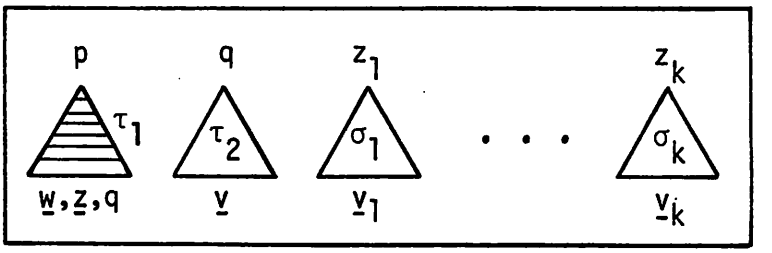
$$\pi(p \xrightarrow{x_1} q,\ q \xrightarrow{\tau} \underline{w})_N \wedge M(x_1) \wedge U(\tau)\{MARK(q)\}$$

$$(\exists\tau_1,\underline{w})[\pi(p \xrightarrow{\tau_1} \underline{w})_N \wedge M(\tau_1)]$$

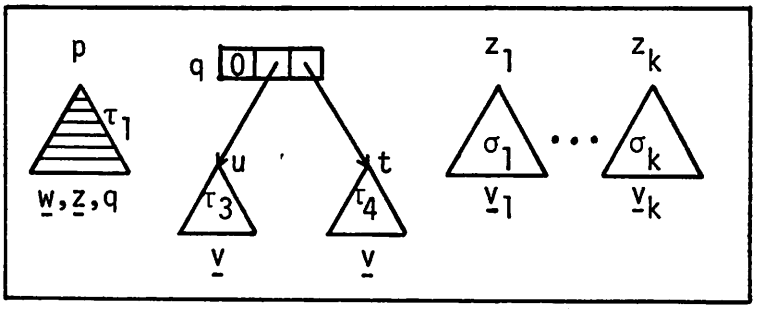
which shows the correctness of the algorithm.

We shall give now the graphical interpretation of this proof ($\underline{z} = (z_1,\ldots,z_k)$):

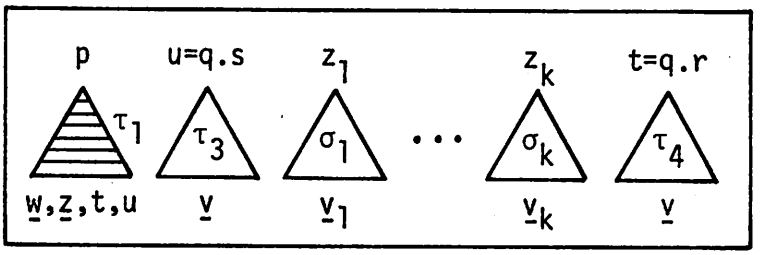
Q_0:


p
q
z_1
z_k
τ_1
τ_2
σ_1
σ_k
$\underline{w},\underline{z},q$
$\underline{v}$
$\underline{v}_1$
$\underline{v}_k$

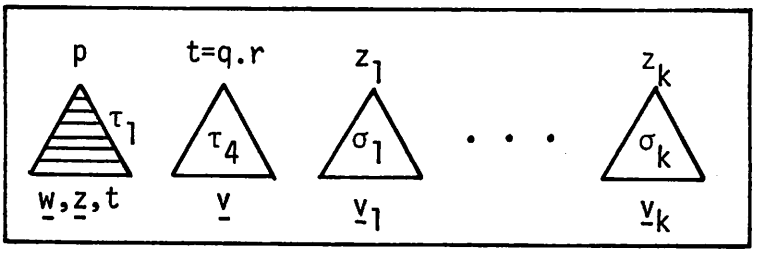
Q_1:


p
q
0
z_1
z_k
τ_1
u
t
τ_3
τ_4
σ_1
σ_k
$\underline{w},\underline{z},q$
$\underline{v}$
$\underline{v}$
$\underline{v}_1$
$\underline{v}_k$

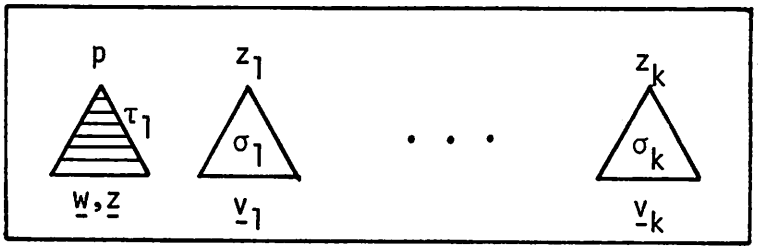
Q_2:


p
u=q.s
z_1
z_k
t=q.r
τ_1
τ_3
σ_1
σ_k
τ_4
$\underline{w},\underline{z},t,u$
$\underline{v}$
$\underline{v}_1$
$\underline{v}_k$
$\underline{v}$

Q_3:

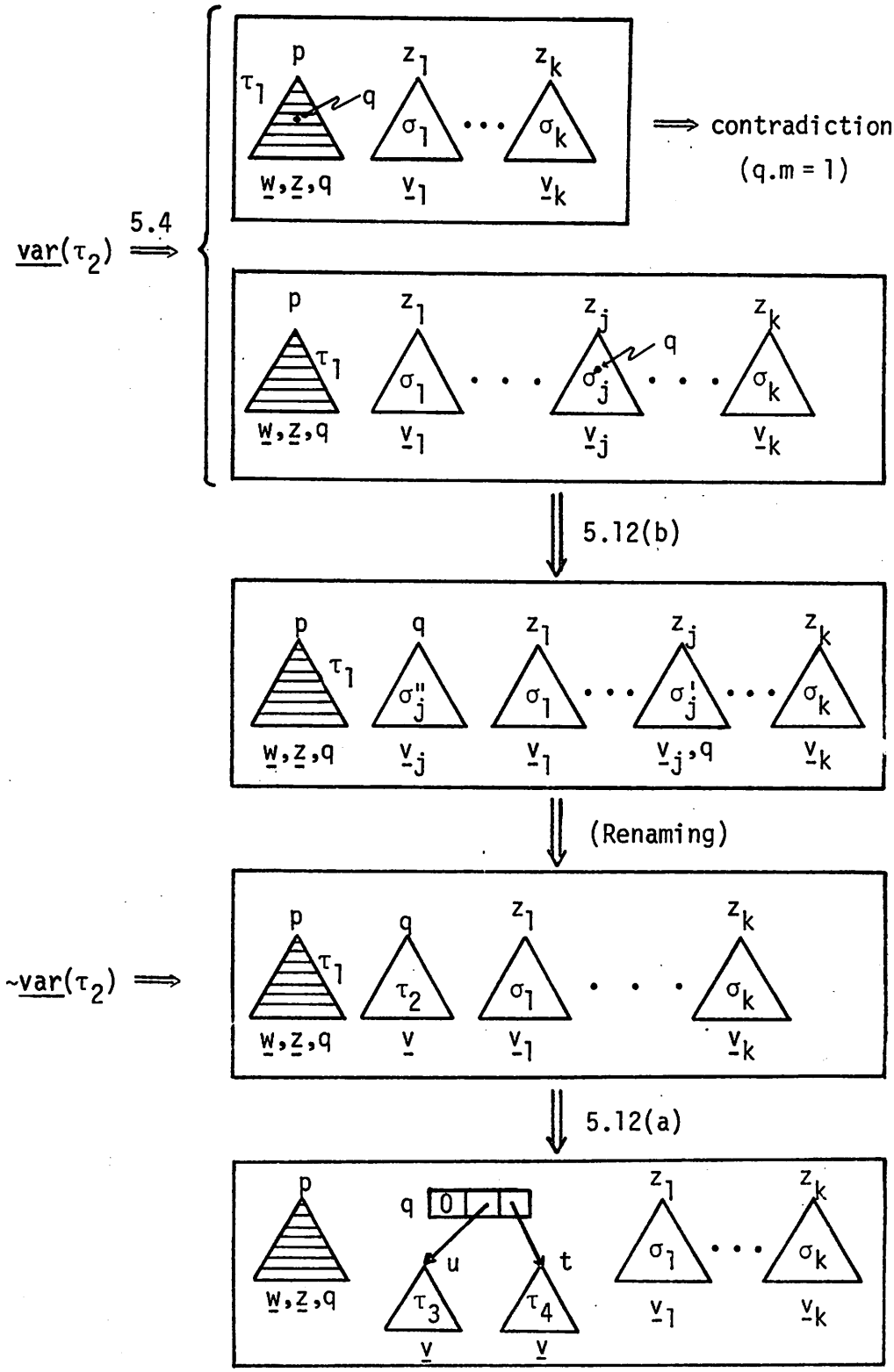

p
t=q.r
z_1
z_k
τ_1
τ_4
σ_1
σ_k
$\underline{w},\underline{z},t$
$\underline{v}$
$\underline{v}_1$
$\underline{v}_k$

Q_4:

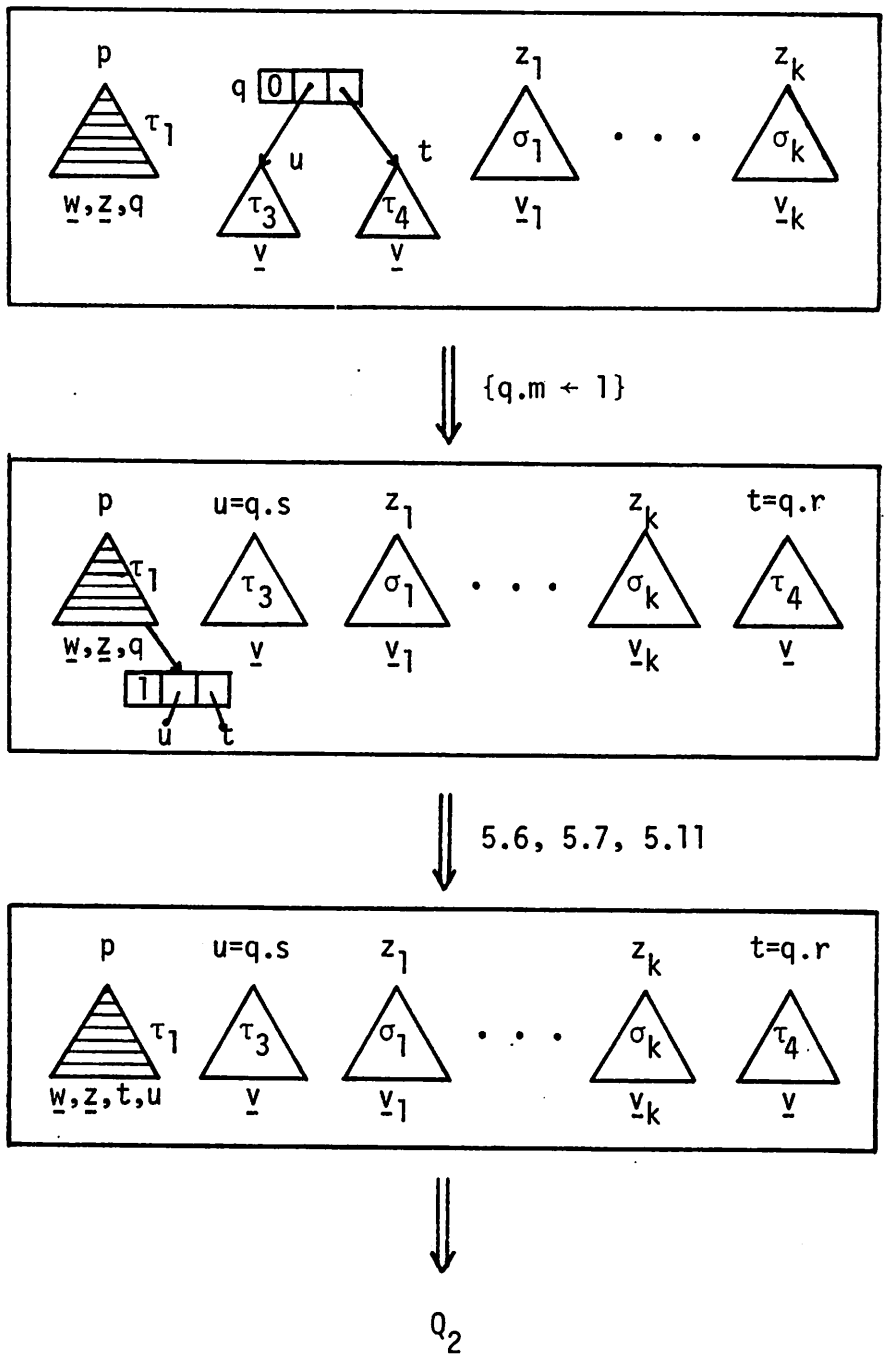

p
z_1
z_k
τ_1
σ_1
σ_k
$\underline{w},\underline{z}$
$\underline{v}_1$
$\underline{v}_k$

We proceed now to graphical verification:

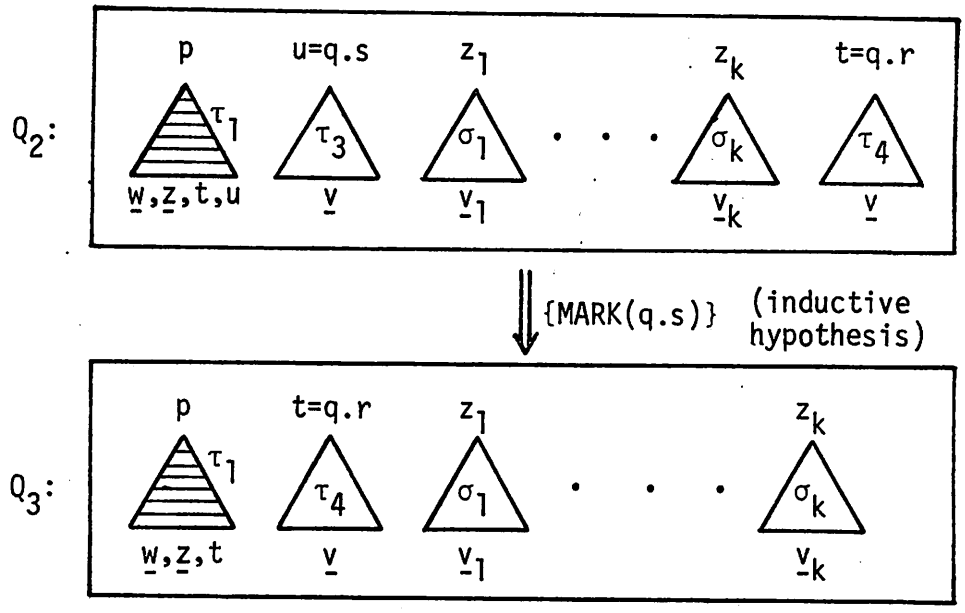
1. $[Q_o \wedge \sim\underline{atom}(q) \wedge q.m \neq 1] \Rightarrow Q_1$:



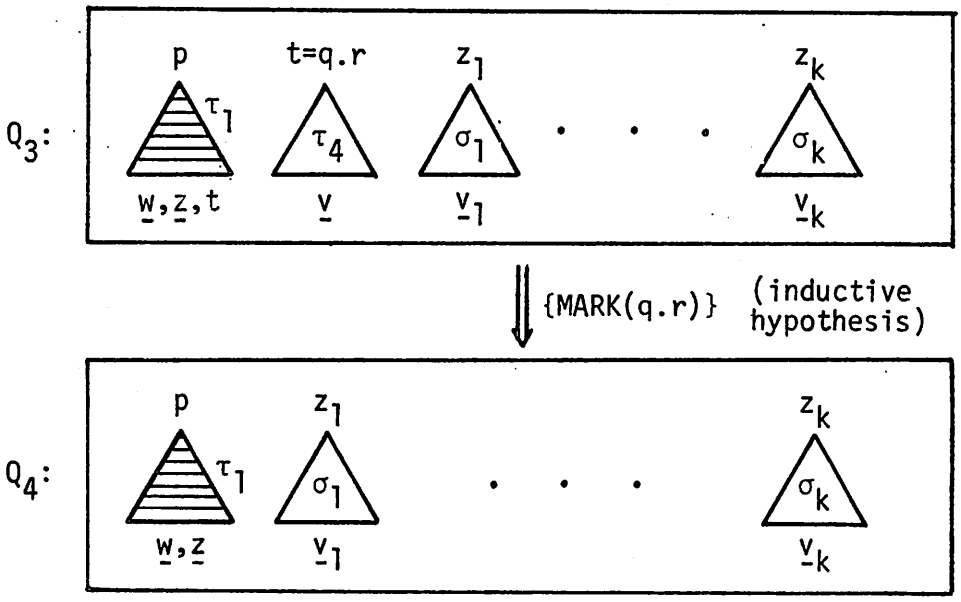
2. $Q_1\{q.m \leftarrow 1\}Q_2$:



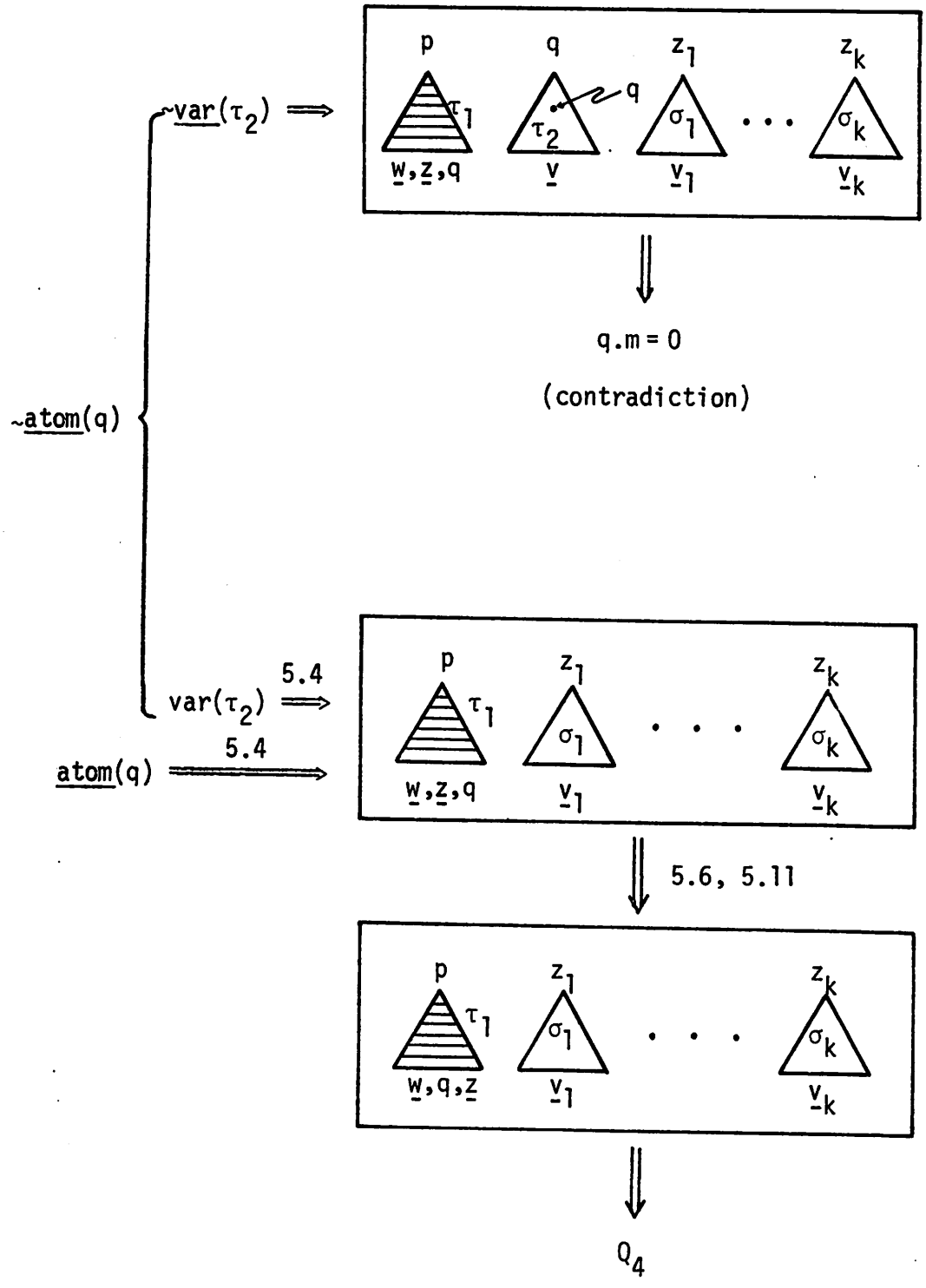
3. $Q_2\{MARK(q.s)\}Q_3$:



4. $Q_3\{MARK(q.r)\}Q_4$:



5. $Q_0 \wedge [\underline{atom}(q) \vee q.m = 1] \Rightarrow Q_4$:



7.4 A "heap" Forming Algorithm

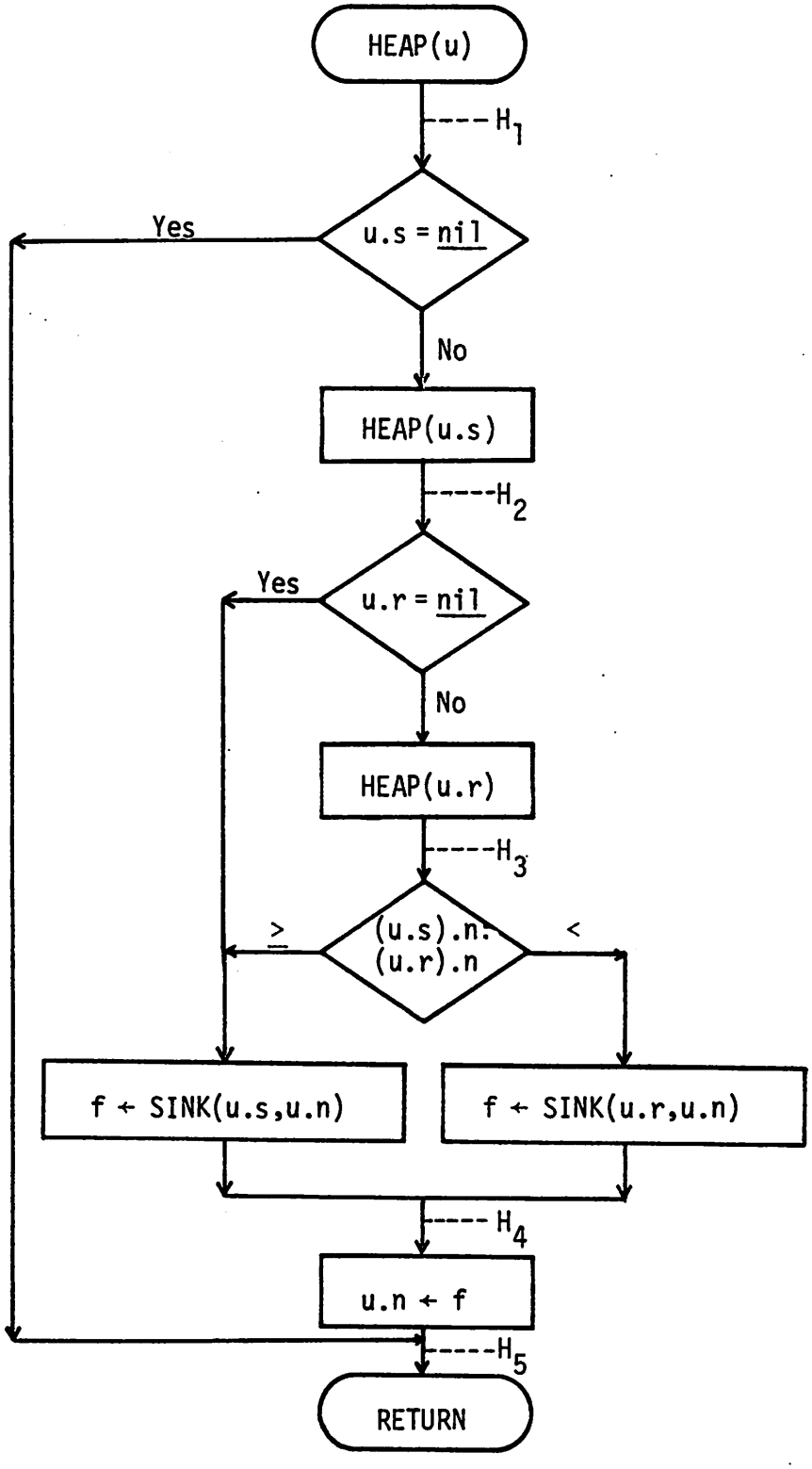
We shall assume here that all nodes have three fields, i.e., order(3,τ) holds for all trees τ appearing in the proof. The first field is always a number and the other fields are either pointers or atoms (nil). A "heap" is a tree in which the number at any node is not smaller than the numbers at its descendants. The procedure HEAP transforms a given arbitrary tree rooted at u into a "heap" by moving the numbers appearing as first fields of the nodes. For a discussion about "heaps" see Knuth [1973], p. 149.

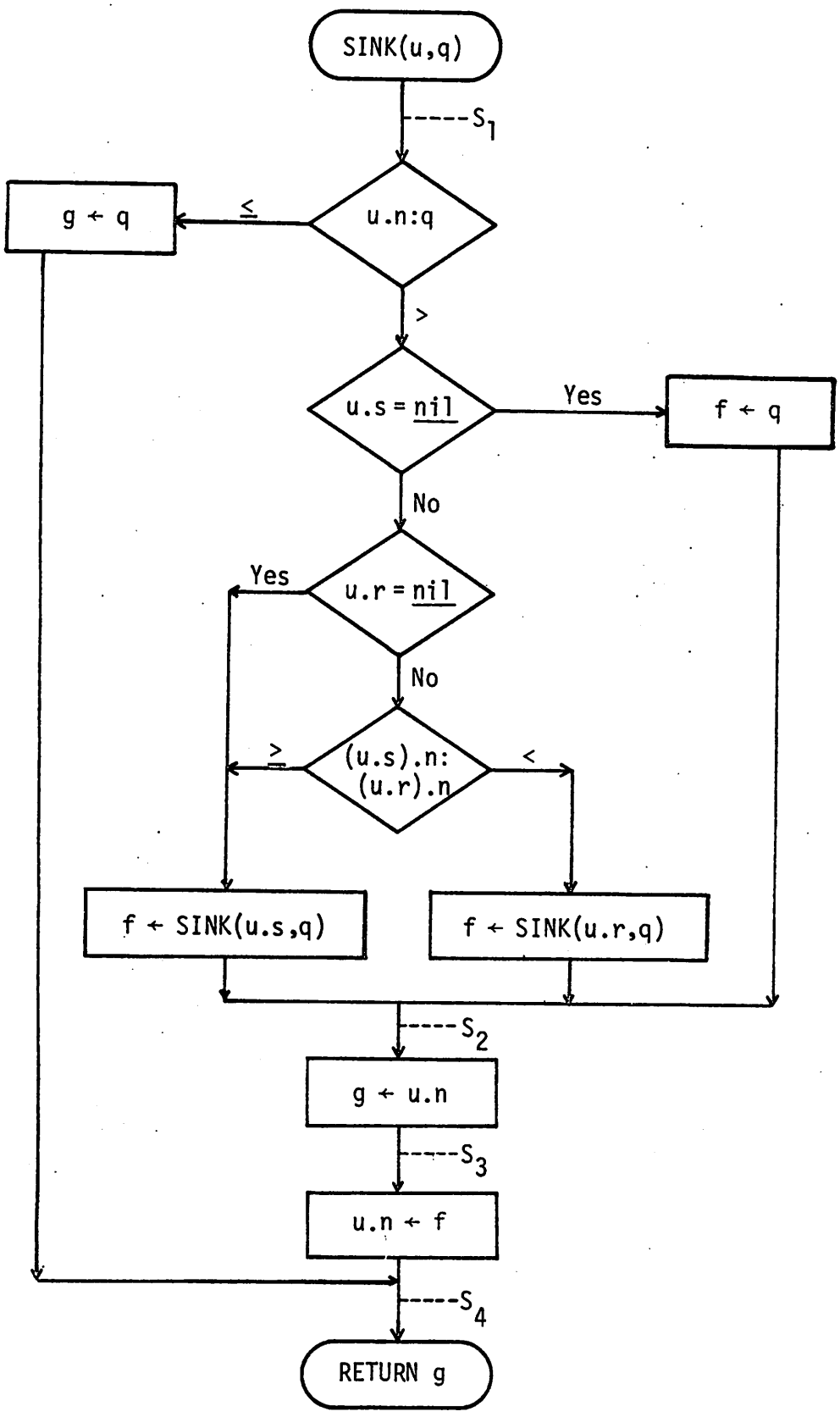
The procedure HEAP is recursive, and it uses another procedure SINK, which is also recursive. SINK(u,q) assumes that q is a number and u is the root of the "heap". The value returned by SINK is the maximum among q and the numbers in the heap. The number q is "sunk" into the heap.

Since the only assignments are to the first fields, we shall not prove that the pointers remain the same.

We use symbolic selectors: n = 1, s = 2, r = 3.

We shall also assume that whenever the second field (s) of a node is nil, so is the third field (r).


HEAP(u)
H1
u.s = nil
Yes
No
HEAP(u.s)
H2
u.r = nil
Yes
No
HEAP(u.r)
H3
(u.s).n : (u.r).n
≥
<
f ← SINK(u.s,u.n)
f ← SINK(u.r,u.n)
H4
u.n ← f
H5
RETURN


SINK(u,q)
S1
u.n:q
≤
g ← q
>
u.s = nil
Yes
f ← q
No
u.r = nil
Yes
No
(u.s).n:
(u.r).n
≥
<
f ← SINK(u.s,q)
f ← SINK(u.r,q)
S2
g ← u.n
S3
u.n ← f
S4
RETURN g

We shall use the following auxiliary definitions:

$$C(\tau) \triangleq \underline{\text{if}}\ \tau=\underline{\text{nil}}\ \underline{\text{then}}\ \emptyset\ \underline{\text{else}}\ \{\tau.n\} \cup C(\tau.s) \cup C(\tau.r)$$

$$H(\tau) \triangleq \tau\neq\underline{\text{nil}} \Rightarrow (\forall x)[x \in C(\tau) \Rightarrow \tau.n \geq x] \wedge H(\tau.s) \wedge H(\tau.r)$$

Thus $C(\tau)$ denotes the set of values (numbers) appearing in τ. However, in order to take care of possible repetition of values, we shall think of $C(\tau)$ as a "bag" of values in which repetitions are allowed. The operations $\cup$ and $\in$ are redefined conveniently. Consequently, in this context $\{a\}\cup\{a\} = \{a,a\} \neq \{a\}$.

$H(\tau)$ means that τ represents a heap.

The inductive assertions are (we drop the existential quantifiers which should be clear from the context):

$$H_1 \triangleq \pi(u\downarrow\tau,\lambda_1,\ldots,\lambda_n) \wedge u\neq\underline{\text{nil}} \wedge A = C(\tau)$$

$$H_2 \triangleq \pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\tau_1,t\downarrow\tau_2,\lambda_1,\ldots,\lambda_n)$$
$$\wedge A = \{p\}\cup C(\tau_1)\cup C(\tau_2) \wedge H(\tau_1) \wedge z\neq\underline{\text{nil}}$$

$$H_3 \triangleq \pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\tau_1,t\downarrow\tau_3,\lambda_1,\ldots,\lambda_n)$$
$$\wedge A = \{p\}\cup C(\tau_1)\cup C(\tau_3) \wedge H(\tau_1) \wedge H(\tau_3) \wedge z\neq\underline{\text{nil}} \wedge t\neq\underline{\text{nil}}$$

$$H_4 \triangleq \pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\tau_4,t\downarrow\tau_5,\lambda_1,\ldots,\lambda_n)$$
$$\wedge A = \{f\}\cup C(\tau_4)\cup C(\tau_5) \wedge H(\tau_4) \wedge H(\tau_5)$$
$$\wedge (\forall x)[x \in C(\tau_4)\cup C(\tau_5) \Rightarrow f \geq x]$$

$$H_5 \triangleq \pi(u\downarrow\sigma,\lambda_1,\ldots,\lambda_n) \wedge A = C(\sigma) \wedge H(\sigma)$$

$S_1 \triangleq \pi(u\downarrow\tau,\lambda_1,\ldots,\lambda_n) \wedge A = C(\tau)\cup\{q\} \wedge H(\tau) \wedge u \neq \underline{nil}$

$S_2 \triangleq \pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\tau_1, t\downarrow\tau_2, \lambda_1,\ldots,\lambda_n)$

$\wedge H(\tau_1) \wedge H(\tau_2) \wedge A = C(\tau_1)\cup C(\tau_2)\cup\{p,f\} \wedge p \geq f$

$\wedge (\forall x)[x \in C(\tau_1)\cup C(\tau_2) \Rightarrow f \geq x]$

$S_3 \triangleq S_2 \wedge g = p$

$S_4 \triangleq \pi(u\downarrow\sigma,\lambda_1,\ldots,\lambda_n) \wedge A = C(\sigma)\cup\{g\} \wedge H(\sigma) \wedge (\forall x)[x \in C(\sigma) \Rightarrow g \geq x]$

We shall verify now the inductive assertions given above:

1. Inductive hypotheses:

$$H_1\{HEAP(u)\}H_5$$

and

$$S_1\{d \leftarrow SINK(u,q)\}[S_4]_d^g$$

2. $H_1 \wedge u.s \neq \underline{nil}\{HEAP(u.s)\}H_2$:

$H_1 \wedge u.s \neq \underline{nil} \Rightarrow$

$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\sigma_1, t\downarrow\sigma_2, \lambda_1,\ldots,\lambda_n)$

$\wedge z = u.s \wedge z \neq \underline{nil} \wedge A = \{p\}\cup C(\sigma_1)\cup C(\tau_2) \wedge B = C(\sigma_1)$

By inductive hypothesis, after HEAP(u.s), we have:

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z\downarrow\tau_1, t\downarrow\tau_2, \lambda_1,\ldots,\lambda_n) \wedge B = C(\tau_1) \wedge H(\tau_1)$$

Since $A = \{p\}\cup C(\tau_2)\cup C(\sigma_1) = \{p\}\cup C(\tau_2)\cup B = \{p\}\cup C(\tau_1)\cup C(\tau_2)$, H_2 follows.

3. $H_2 \wedge u.r \neq \underline{nil}\{HEAP(u.r)\}H_3$:

$$H_2 \wedge u.r \neq \underline{nil} \Rightarrow$$

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_2, \lambda_1, \ldots, \lambda_n) \wedge z \neq \underline{nil}$$

$$\wedge\ t \neq \underline{nil} \wedge u.r = t \wedge A = \{p\} \cup C(\tau_1) \cup C(\tau_2) \wedge H(\tau_1) \wedge D = C(\tau_2)$$

As before, by inductive hypothesis, after HEAP(u.r):

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_3, \lambda_1, \ldots, \lambda_n) \wedge D = C(\tau_3) \wedge H(\tau_3)$$

Since $A = \{p\} \cup C(\tau_1) \cup D = \{p\} \cup C(\tau_1) \cup C(\tau_3)$, and H_3 follows.

4. $H_2 \wedge u.r = \underline{nil}\{f \leftarrow SINK(u.s,u.n)\}H_4$:

$$H_2 \wedge u.r = \underline{nil} \Rightarrow$$

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_3, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ A = \{p\} \cup C(\tau_1) \wedge \tau_2 = \underline{nil} \wedge H(\tau_1) \wedge z \neq \underline{nil} \wedge u.n = p \wedge u.s = z$$

By inductive hypothesis, after SINK(u.s,u.n):

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_4, t{\downarrow}\underline{nil}, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ A = \{f\} \cup C(\tau_4) \wedge H(\tau_4) \wedge (\forall x)[x \in C(\tau_4) \Rightarrow f \geq x]$$

and H_4 follows, since $C(\underline{nil}) = \emptyset$ and $H(\underline{nil})$ holds.

5. $H_3 \wedge [(u.s).n \geq (u.r).n]\{f \leftarrow SINK(u.s,u.n)\}H_4$:

$$H_3 \wedge (u.s).n \geq (u.r).n \Rightarrow$$

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_3, \lambda_1, \ldots, \lambda_n) \wedge z \neq \underline{nil}$$

$$\wedge\ t \neq \underline{nil} \wedge A = \{p\} \cup C(\tau_1) \cup C(\tau_3) \wedge H(\tau_1) \wedge H(\tau_3)$$

$$\wedge \tau_1.n \geq \tau_3.n \wedge B = \{p\} \cup C(\tau_1) \wedge u.s = z \wedge u.n = p$$

By inductive hypothesis, after SINK(u.s,u.n):

$$\pi(u \xrightarrow{<u,p,x_1,x_2>} (z,t),\ z{\downarrow}\tau_4, t{\downarrow}\tau_3, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ B = C(\tau_4) \cup \{f\} \wedge H(\tau_4) \wedge (\forall x)[x \in C(\tau_4) \Rightarrow f \geq x]$$

Since $A = C(\tau_3) \cup B \Rightarrow A = \{f\} \cup C(\tau_4) \cup C(\tau_3)$.

Also, $\tau_1.n \geq \tau_3.n \Rightarrow (\forall x)[x \in C(\tau_3) \Rightarrow \tau_1.n \geq x]$. Since $\tau_1.n \in B \Rightarrow f \geq \tau_1.n \Rightarrow (\forall x)[x \in C(\tau_3) \Rightarrow f \geq x]$. Thus finally: $(\forall x)[x \in C(\tau_4) \cup C(\tau_3) \Rightarrow f \geq x]$, and H_4 follows (with $\tau_5 = \tau_3$).

6. $H_3 \wedge [(u.s).n < (u.r).n]\{f \leftarrow SINK(u.r,u.n)\}H_4$: Proof analogous to that in 5, using t and τ_3 instead of z and τ_1.

7. $H_4\{u.n \leftarrow f\}H_5$: After $u.n \leftarrow f$, we have:

$$\pi(u \xrightarrow{<u,f,x_1,x_2>} (z,t),\ z{\downarrow}\tau_4, t{\downarrow}\tau_5, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ A = \{f\} \cup C(\tau_4) \cup C(\tau_5) \wedge H(\tau_4) \wedge H(\tau_5)$$

$$\wedge\ (\forall x)[x \in C(\tau_4) \cup C(\tau_5) \Rightarrow f \geq x]$$

and consequently:

$$\pi(u{\downarrow}\sigma, \lambda_1, \ldots, \lambda_n) \wedge A = C(\sigma) \wedge H(\sigma)$$

with $\sigma = \langle u,f,\tau_4,\tau_5\rangle$.

8. $\underline{[H_1 \wedge u.s = \underline{nil}] \Rightarrow H_5}$:

$$[H_1 \wedge u.s = \underline{nil}] \Rightarrow \exists(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_2, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ \tau_1 = \underline{nil} \wedge \tau_2 = \underline{nil} \wedge A = \{p\}$$

and H_5 follows trivially, with $\sigma = \langle u,p,\underline{nil},\underline{nil}\rangle$.

9. $\underline{S_1 \wedge u.n \leq q\{g \leftarrow q\}S_4}$: After the assignment $g \leftarrow q$:

$$S_1 \wedge u.n \leq q \wedge g = q \Rightarrow A = C(\tau) \cup \{g\} \wedge H(\tau) \quad .$$

Also, $H(\tau) \Rightarrow (\forall x)[x \in C(\tau) \Rightarrow \tau.n \geq x] \Rightarrow (\forall x)[x \in C(\tau) \Rightarrow g \geq x]$ (since $u.n = \tau.n$), and S_4 follows with $\sigma = \tau$.

10. $\underline{S_1 \wedge u.n > q \wedge u.s = \underline{nil}\{f \leftarrow q\}S_2}$:

$$S_1 \wedge u.n > q \wedge u.s = \underline{nil} \Rightarrow$$

$$\exists(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_2, \lambda_1, \ldots, \lambda_n)$$

$$\wedge\ \tau_1 = \underline{nil} \wedge \tau_2 = \underline{nil} \wedge A = \{p,q\} \wedge p > q$$

After the assignment we have $f = q$ and S_2 follows easily since $C(\tau_1) = C(\tau_2) = \emptyset$.

11. $\underline{S_1 \wedge u.n > q \wedge u.s \neq \underline{nil} \wedge u.r = \underline{nil}\{f \leftarrow SINK(u.s,q)\}S_2}$:

$$S_1 \wedge u.n > q \wedge u.s \neq \underline{nil} \wedge u.r = \underline{nil} \Rightarrow$$

$$\exists(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\sigma_1, t{\downarrow}\sigma_2, \lambda_1, \ldots, \lambda_n) \wedge z \neq \underline{nil}$$

$$\wedge\ \sigma_2 = \underline{nil} \wedge A = \{p,q\} \cup C(\sigma_1) \wedge H(\tau_1) \wedge (\forall x)[x \in C(\sigma_1) \Rightarrow p \geq x]$$

$$\wedge\ B = \{q\} \cup C(\sigma_1) \wedge p \geq q \wedge u.s = z$$

By inductive hypothesis after SINK(u.s,q):

$$\pi(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\underline{nil}, \lambda_1, \ldots, \lambda_n)$$

$$\wedge B = C(\tau_1) \cup \{f\} \wedge H(\tau_1) \wedge (\forall x)[x \in C(\tau_1) \Rightarrow f \geq x]$$

Since $A = \{p\} \cup B \Rightarrow A = \{p,f\} \cup C(\tau_1)$. Also, since $f \in B \Rightarrow p \geq f$, and S_2 follows (with $\tau_2 = \underline{nil}$).

12. $$\frac{S_1 \wedge u.n > q \wedge u.s \neq \underline{nil} \wedge u.r \neq \underline{nil} \wedge (u.s).n \geq (u.r).n}{\{f \leftarrow SINK(u.s,q)\}S_2:}$$

$$S_1 \wedge u.n > q \wedge u.s \neq \underline{nil} \wedge u.r \neq \underline{nil} \wedge (u.s).n \geq (u.r).n \Rightarrow$$

$$\pi(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\sigma_1, t{\downarrow}\sigma_2, \lambda_1, \ldots, \lambda_n) \wedge z \neq \underline{nil}$$

$$\wedge t \neq \underline{nil} \wedge A = \{p,q\} \cup C(\sigma_1) \cup C(\sigma_2) \wedge H(\sigma_1) \wedge H(\sigma_2)$$

$$\wedge (\forall x)[x \in C(\sigma_1) \cup C(\sigma_2) \Rightarrow p \geq x] \wedge B = \{q\} \cup C(\sigma_1) \wedge p \geq q$$

$$\wedge u.s = z \wedge \sigma_1.n \geq \sigma_2.n$$

By inductive hypothesis, after SINK(u.s,q):

$$\pi(u \xrightarrow{\langle u,p,x_1,x_2\rangle} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\sigma_2, \lambda_1, \ldots, \lambda_n)$$

$$\wedge B = C(\tau_1) \cup \{f\} \wedge H(\tau_1) \wedge (\forall x)[x \in C(\tau_1) \Rightarrow f \geq x]$$

Since $A = \{p\} \cup C(\sigma_2) \cup B \Rightarrow A = \{p,f\} \cup C(\tau_1) \cup C(\sigma_2)$. Also, $f \in B \Rightarrow p \geq f$ and

$$\sigma_1.n \in B \Rightarrow f \geq \sigma_1.n \Rightarrow f \geq \sigma_2.n \Rightarrow (\forall x)[x \in C(\sigma_2) \Rightarrow f \geq x]$$

since $H(\sigma_2)$. Thus S_2 follows, with $\tau_2 = \sigma_2$.

13. $S_1 \wedge u.n > q \wedge u.s \neq \underline{nil} \wedge u.r \neq \underline{nil} \wedge (u.s).n < (u.r).n$

$\{f \leftarrow SINK(u.r,q)\}S_2$: The proof is analogous to that in 12, using t and σ_2, instead of z and σ_1.

14. $S_2\{g \leftarrow u.n\}S_3$: trivial.

15. $S_3\{u.n \leftarrow f\}S_4$: After the assignment:

$$\begin{aligned}
&\pi(u \xrightarrow{<u,f,x_1,x_2>} (z,t),\ z{\downarrow}\tau_1, t{\downarrow}\tau_2, \lambda_1,\ldots,\lambda_n) \wedge H(\tau_1) \wedge H(\tau_2) \\
&\wedge A = C(\tau_1) \cup C(\tau_2) \cup \{g,f\} \wedge g \geq f \wedge (\forall x)[x \in C(\tau_1) \cup C(\tau_2) \Rightarrow f \geq x] \\
&\Rightarrow \pi(u{\downarrow}\sigma, \lambda_1,\ldots,\lambda_n) \wedge A = C(\sigma) \cup \{g\} \wedge H(\sigma) \wedge (\forall x)[x \in C(\sigma) \Rightarrow g \geq x] \\
&\Rightarrow S_4 .
\end{aligned}$$

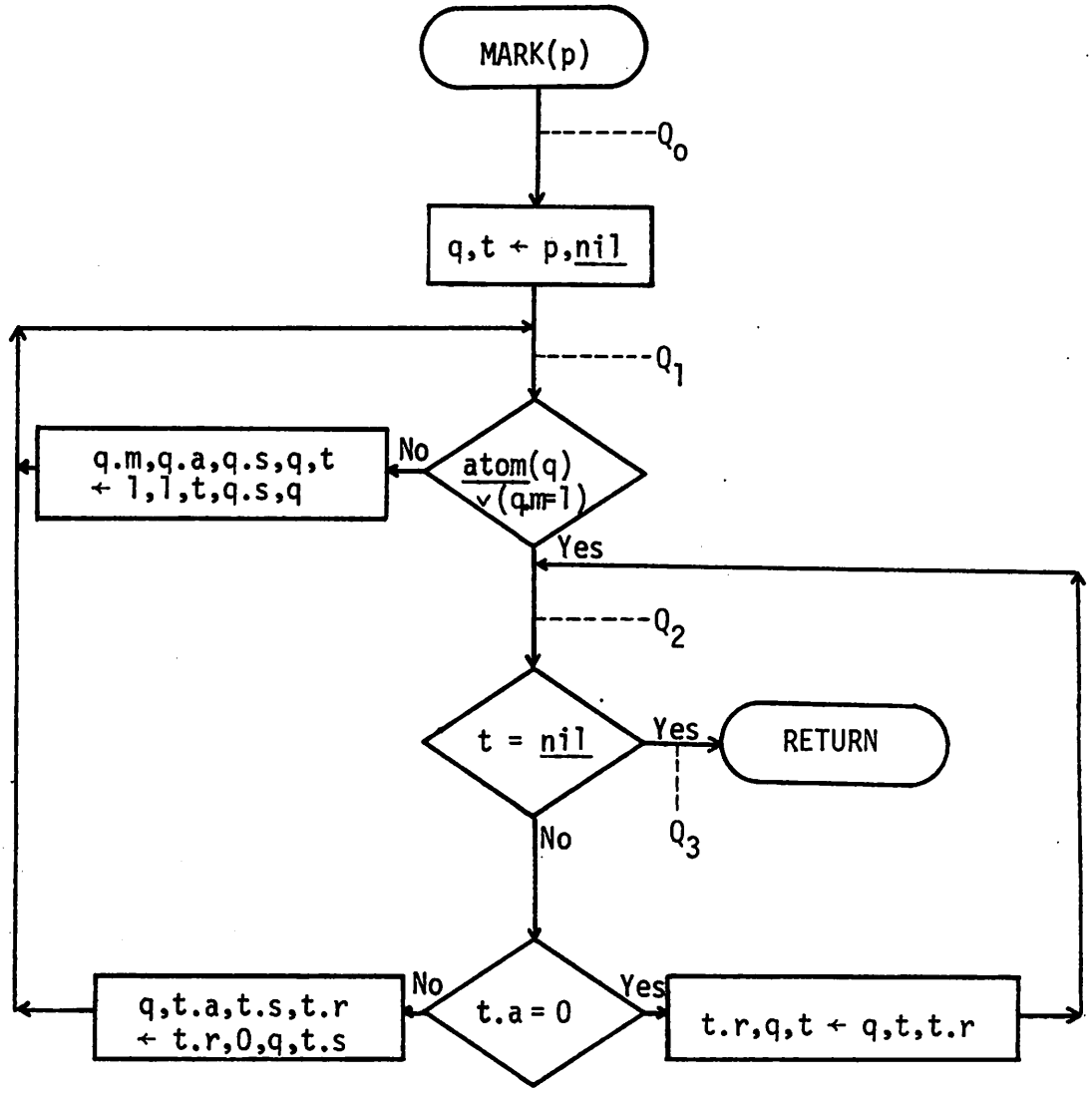
This concludes the verification of inductive assertions. The final conclusion is:

$$\pi(u{\downarrow}\tau) \wedge A = C(\tau) \wedge u \neq \underline{nil}\{HEAP(u)\}\pi(u{\downarrow}\sigma) \wedge A = C(\sigma) \wedge H(\sigma) \quad .$$

7.5 The Deutsch-Waite-Schorr Marking Algorithm

The purpose of this algorithm is again to mark all the nodes accessible from a given location p. The algorithm is complicated by the fact that a stack is kept in the data structure itself and several pointers are temporarily modified during the execution. We shall assume that each node is composed of four fields, and therefore order(4,τ) holds for all trees τ in this proof. The first two fields, referred to by symbolic selectors m = 1 (marker), and a = 2 (auxiliary flag) are always 0 or 1. The other two fields s = 3 (left son) and r = 4 (right son) are atoms or pointers to other locations. For a more detailed description of this algorithm see Knuth [1968], p. 417.

Since the algorithm involves modification of pointer fields, our correctness condition will have to involve also an assertion showing that the final pointers are the same as those at the beginning of execution. The proof will be followed by a graphical interpretation.


MARK(p)
Q_o
q,t ← p,nil
Q_1
atom(q) ∨(q.m=1)
No
q.m,q.a,q.s,q,t ← 1,1,t,q.s,q
Yes
Q_2
t = nil
Yes
RETURN
Q_3
No
t.a = 0
No
q,t.a,t.s,t.r ← t.r,0,q,t.s
Yes
t.r,q,t ← q,t,t.r

As before, we shall use two auxiliary predicates:

$$M(\tau) \triangleq \sim\underline{elem}(\tau) \Rightarrow [\tau.m = 1 \wedge M(\tau.s) \wedge M(\tau.r)]$$
$$U(\tau) \triangleq \sim\underline{elem}(\tau) \Rightarrow [\tau.m = 0 \wedge U(\tau.s) \wedge U(\tau.r)]$$

We shall define an auxiliary assertion P by:

$$P \triangleq P_1 \wedge P_2 \wedge P_3$$

where

$$P_1 \triangleq \{\pi(q \xrightarrow{\xi} \underline{u}, [z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k})_N$$
$$\wedge k \geq 0 \wedge (z_{k+1} = q) \wedge (z_k = t) \wedge (z_0 = \underline{nil})\}$$

$$P_2 \triangleq (\forall x)[x \in (N - \{z_1,\ldots,z_k\}) \Rightarrow (R_J(x).s = R_K(x).s \wedge R_J(x).r = R_K(x).r)]$$

$$P_3 \triangleq \bigwedge_{i=1}^{k} \{[d_i = 1 \wedge v_i = z_{i-1} \wedge \underline{var}(\sigma_i) \wedge U(\tau_i) \wedge R_J(z_i).s = z_{i+1}$$
$$\wedge R_J(z_i).r = w_i]$$
$$\vee [d_i = 0 \wedge w_i = z_{i-1} \wedge \underline{var}(\tau_i) \wedge M(\sigma_i) \wedge R_J(z_i).s = v_i$$
$$\wedge R_J(z_i).r = z_{i+1}]\}$$

where K is the FSD appearing in P_1, and J is the one appearing in Q_0 below (initial state).

P seems rather complex, and we shall try to explain its intuitive meaning. The references $z_k, z_{k-1}, \ldots, z_1$ form the stack for this algorithm. The connections between these elements are determined by the auxiliary flags d_i as shown in P_3. All pointers, except those in $z_1,\ldots,z_k$ are the same as original, as described by P_2. P_3 describes also the original values of the pointers for $z_1,\ldots,z_k$.

The inductive assertions are:

$$Q_0 \triangleq \{\pi(p \xrightarrow{\xi} \underline{u})_N \wedge U(\xi)\}$$

$$Q_1 \triangleq P \wedge U(\xi)$$

$$Q_2 \triangleq P \wedge M(\xi)$$

$$Q_3 \triangleq \{\pi(q \xrightarrow{\xi} \underline{u})_N \wedge M(\xi) \wedge (\forall x)[x \in N \Rightarrow (R_J(x).s = R_L(x).s \wedge R_J(x).r = R_L(x).r)]\}$$

where L denotes the FSD appearing in Q_3. Notice that we omit the existential quantifiers which should precede each Q_i, and which should be clear from the context. We shall outline now the verification of these assertions:

1. Q_0 is guaranteed by Proposition 4.10 and the initial assumption that all nodes are unmarked.

2. $\underline{Q_0\{q,t \leftarrow p,\underline{nil}\}Q_1}$: trivial with $p = q$, $t = \underline{nil}$ and $k = 0$.

3. $\underline{Q_1 \wedge [atom(q) \vee q.m = 1] \Rightarrow Q_2}$:

Case 1: $\underline{atom}(q) \Rightarrow \underline{elem}(\xi) \Rightarrow M(\xi) \Rightarrow Q_2$

Case 2: $\sim\underline{atom}(q) \Rightarrow q.m = 1$

2a: $\sim elem(\xi) \Rightarrow \xi.m = q.m = 0 \Rightarrow$ contradiction

2b: $\underline{elem}(\xi) \Rightarrow M(\xi) \Rightarrow Q_2$

4. $\underline{[Q_2 \wedge t = \underline{nil}] \Rightarrow Q_3}$: trivial, since $t = \underline{nil} \Rightarrow k = 0$

5. $\underline{Q_2 \wedge t \neq \underline{nil} \wedge t.a = 0\{t.r,q,t \leftarrow q,t,t.r\}Q_2}$: It follows easily that $k > 0$ and $d_K = 0$. Thus the FSD can be rewritten (since $z_k = t$):

$$\pi(q \xrightarrow{\xi} \underline{u},\ t \xrightarrow{<t,1,0,x_1,x_2>} (v_k,z_{k-1}),\ v_k \xrightarrow{\sigma_k} \underline{u},$$
$$[z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k-1})_N$$

and after the assignment:

$$\pi(q' \xrightarrow{\xi} \underline{u},\ q \xrightarrow{<q,1,0,x_1,x_2>} (v_k,q'),\ v_k \xrightarrow{\sigma_k} \underline{u},$$
$$[z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k-1})_N$$

Notice that the values of pointers at $z_k = q$ are restored to its original values, and Q_2 follows for k-1.

6. $\underline{Q_2 \wedge t \neq \underline{nil} \wedge t.a = 1\{q,t.a,t.s,t.r \leftarrow t.r,0,q,t.s\}Q_1}$: It follows that $k > 0$ and $d_K = 1$. Thus the FSD can be rewritten:

$$\pi(q \xrightarrow{\xi} \underline{u},\ t \xrightarrow{<t,1,1,x_1,x_2>} (v_k,w_k) \xrightarrow{(\sigma_k,\tau_k)} \underline{u},$$
$$[z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k-1})_N$$

After the assignment:

$$\pi(q' \xrightarrow{\xi} \underline{u},\ t \xrightarrow{<t,1,0,x_1,x_2>} (q',v_k),\ v_k \xrightarrow{\sigma_k} \underline{u},\ q \xrightarrow{\tau_k} \underline{u},$$
$$[z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k-1})_N$$

By a convenient renaming we get Q_1 (with the same value of k).

7. $\underline{Q_1 \wedge \sim\underline{atom}(q) \wedge q.m = 0\{q.m,q.a,q.s,q,t \leftarrow 1,1,t,q.s,q\}Q_1}$:

Claim: Q_1 can be rewritten so that $\sim\underline{elem}(\xi)$ holds:

Case 1: $q \in S(\xi) \Rightarrow S(\xi) \neq \emptyset \Rightarrow \sim\underline{elem}(\xi)$;

Case 2: $q \notin S(\xi) \Rightarrow S(\xi) = \emptyset \Rightarrow \underline{elem}(\xi)$

and thus $q \xrightarrow{\xi} \underline{u}$ can be deleted from the FSD.

2a: $q = z_i$ for some i; then $q.m = 1 \Rightarrow$ contradiction;

2b: $q \in S(\sigma_i)$ for some i; then $M(\sigma_i) \Rightarrow q.m = 1 \Rightarrow$ contradiction;

2c: $q \in S(\tau_i)$ for some i; then $w_i \xrightarrow{\tau_i} \underline{u}$ can be replaced by $w_i \xrightarrow{\tau_i'} (\underline{u},q)$, $q \xrightarrow{\tau_i''} \underline{u}$ where $\tau_i = \tau_i' \circ (x_1,\ldots,x_{|\underline{u}|},\tau_i'')$, and $\sim\underline{elem}(\tau_i'')$. Easily $U(\tau_i) \Rightarrow U(\tau_i') \wedge U(\tau_i'')$. Thus we can take τ_i', τ_i'' as the new values of τ_i and ξ. Also, it is easy to prove that $\underline{u}$ can be replaced by $\underline{u}' = (\underline{u},q)$ throughout the FSD.

Thus we proved the claim, and $\sim\underline{elem}(\xi)$ holds. Consequently, the FSD can be rewritten as:

$$\pi(q \xrightarrow{<q,0,d,x_1,x_2>} (g,h) \xrightarrow{(\xi_1,\xi_2)} \underline{u},\ [z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k})_N$$

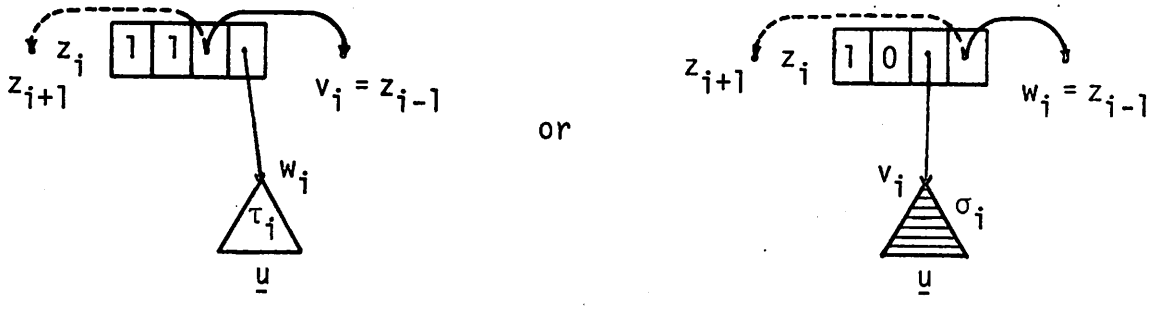
and after the assignment:

$$\pi(t \xrightarrow{<t,1,1,x_1,x_2>} (z_k,h),\ q \xrightarrow{\xi_1} \underline{u},\ h \xrightarrow{\xi_2} \underline{u},\ [z_i \xrightarrow{<z_i,1,d_i,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k})_N$$

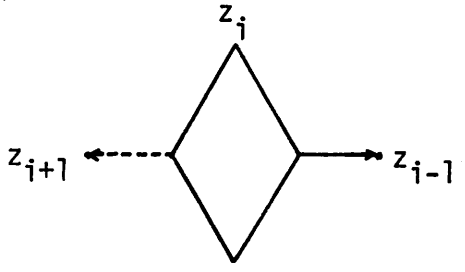
It is easy to show now that Q_1 holds for $k+1$, with $z_{k+1} = t$, $v_{k+1} = z_k$, $\sigma_{k+1} = x_{|\underline{u}|+1}$, $w_{k+1} = h$, $\tau_{k+1} = \xi_2$, $\xi = \xi_1$. □

This concludes the verifications. We shall give now a graphical interpretation of this proof. We shall adopt the convention that all pointers not shown explicitly are the same as in the

original structure. Additional broken lines indicate the original pointers whenever they differ from the current ones. As indicated by P_3, we have two possibilities for each z_i, depending on the value of d_i:

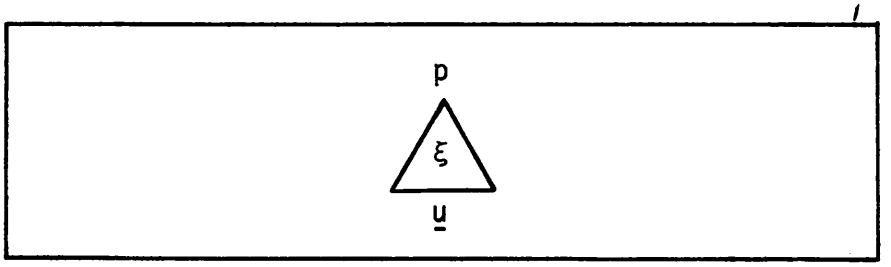


We shall represent these two possibilities by:

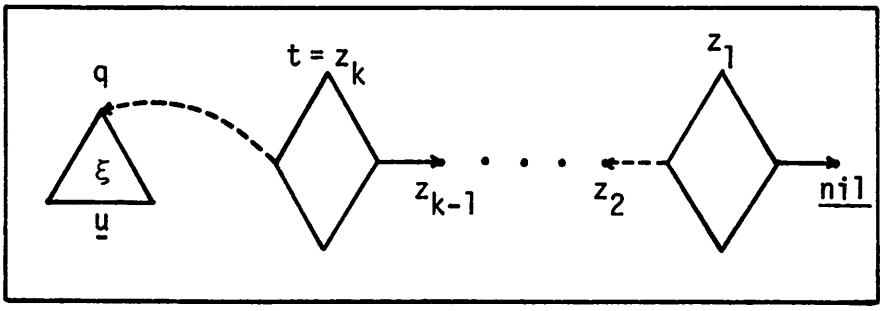


We represent now the assertions Q_o, Q_1, Q_2 and Q_3 by:

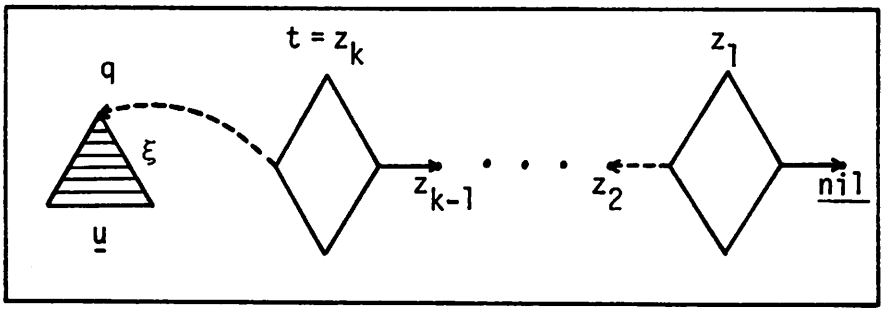
Q_o:



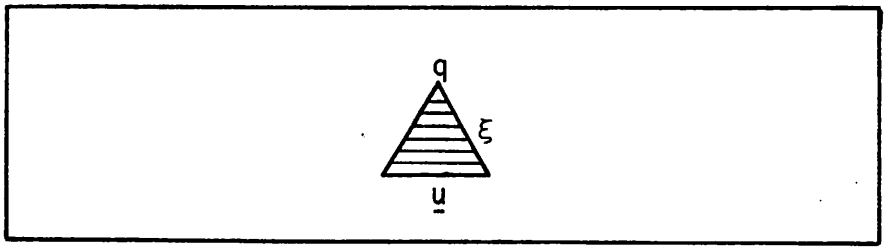
Q_1:



Q_2:

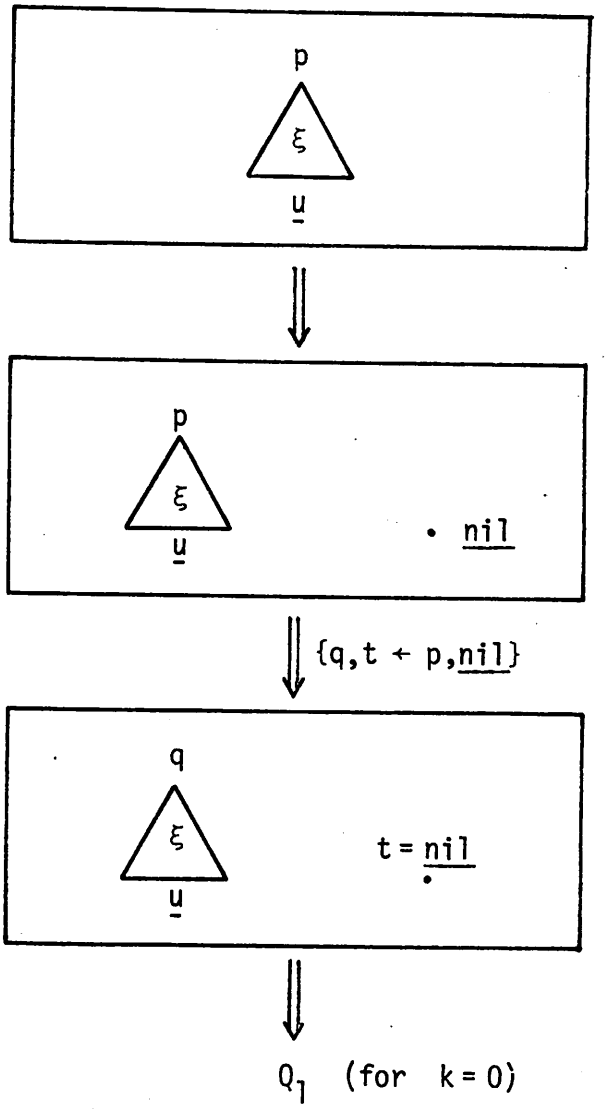


Q_3:

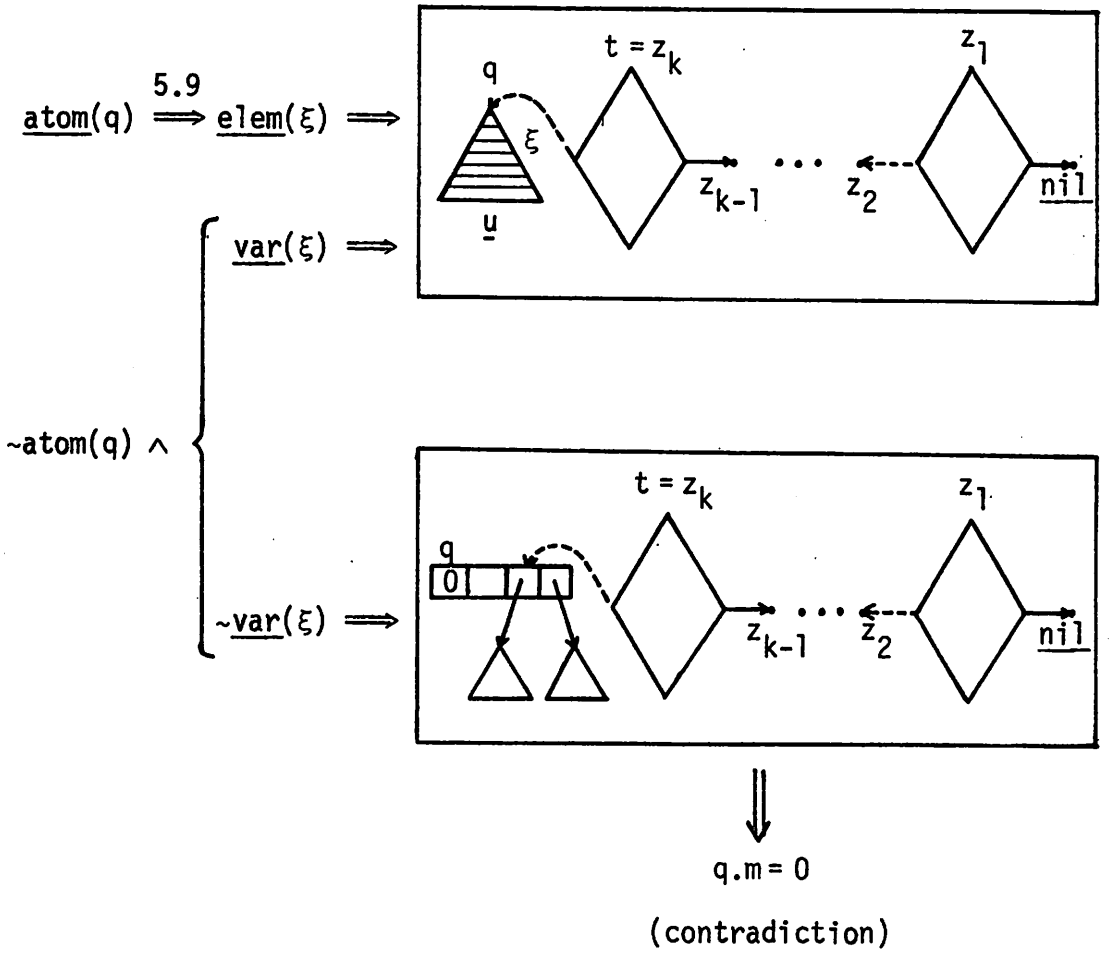


We proceed now with the verification:

2. $\underline{Q_0\{q,t \leftarrow p,\underline{nil}\}Q_1}$:

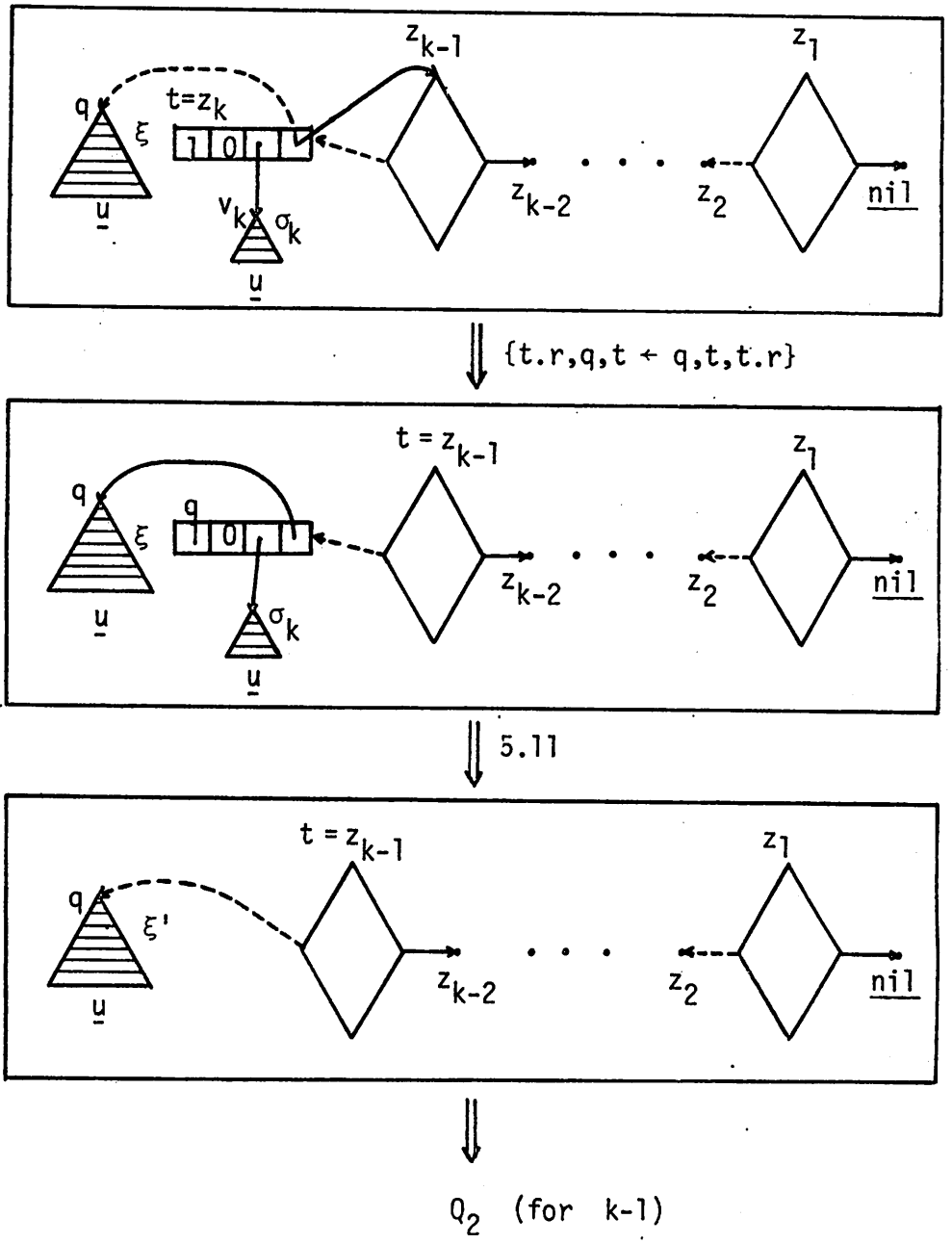


3. $Q_1 \wedge [\underline{atom}(q) \vee q.m = 1] \Rightarrow Q_2$:

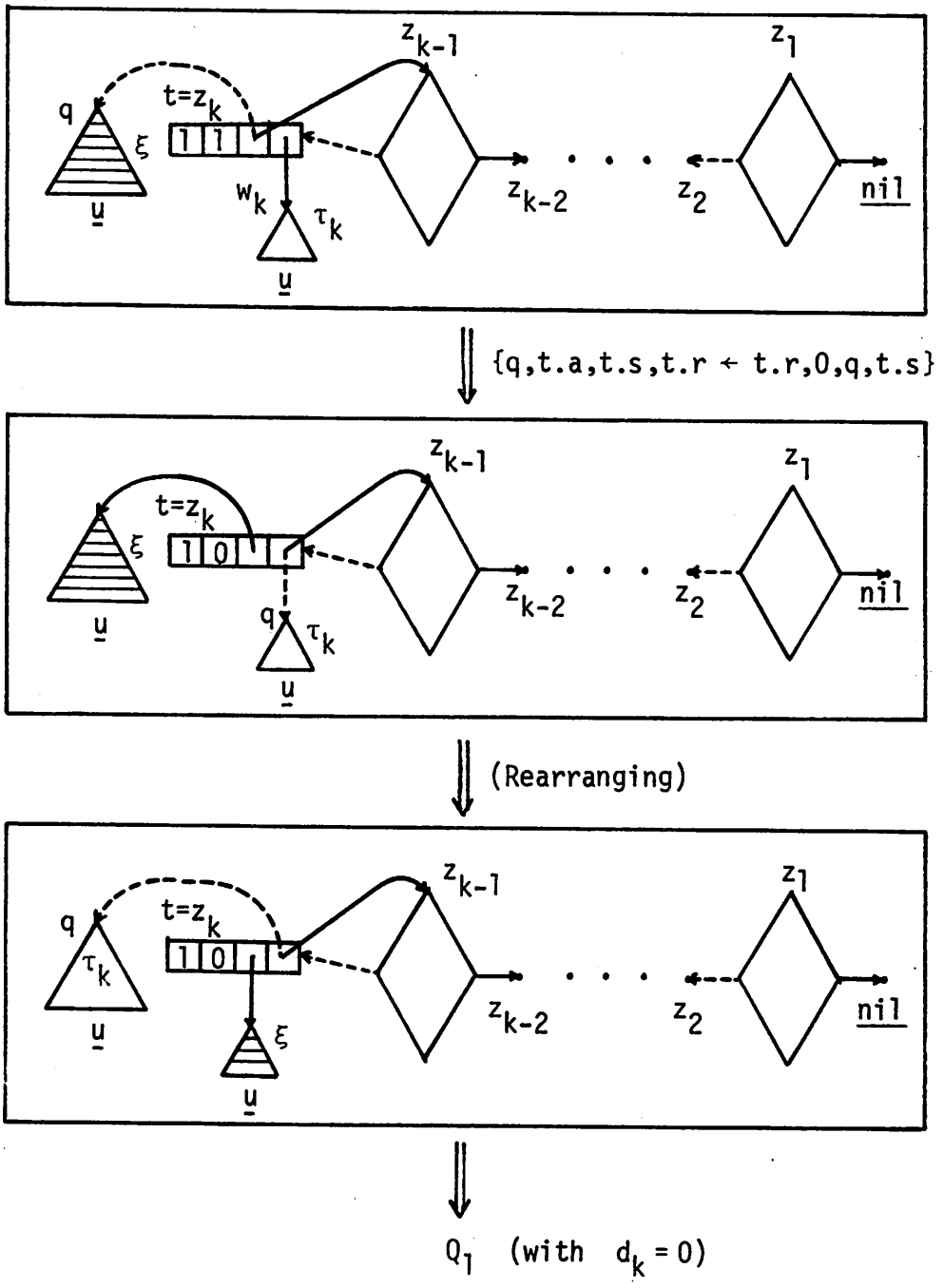


4. $Q_2 \wedge t = \underline{nil} \Rightarrow Q_3$: trivial

5. $Q_2 \wedge t \neq \underline{nil} \wedge t.a = 0\{t.r,q,t \leftarrow q,t,t.r\}Q_2$:

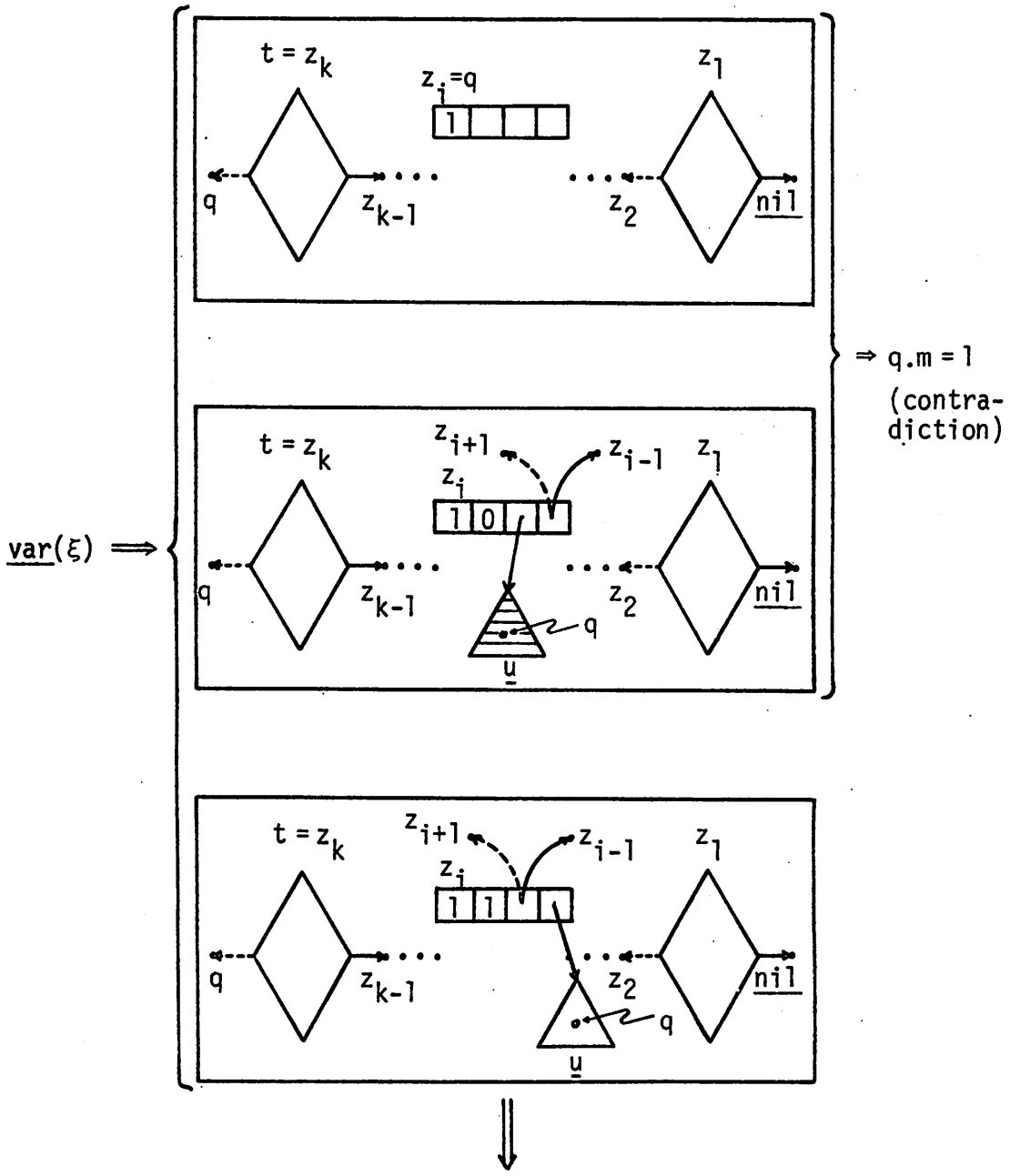


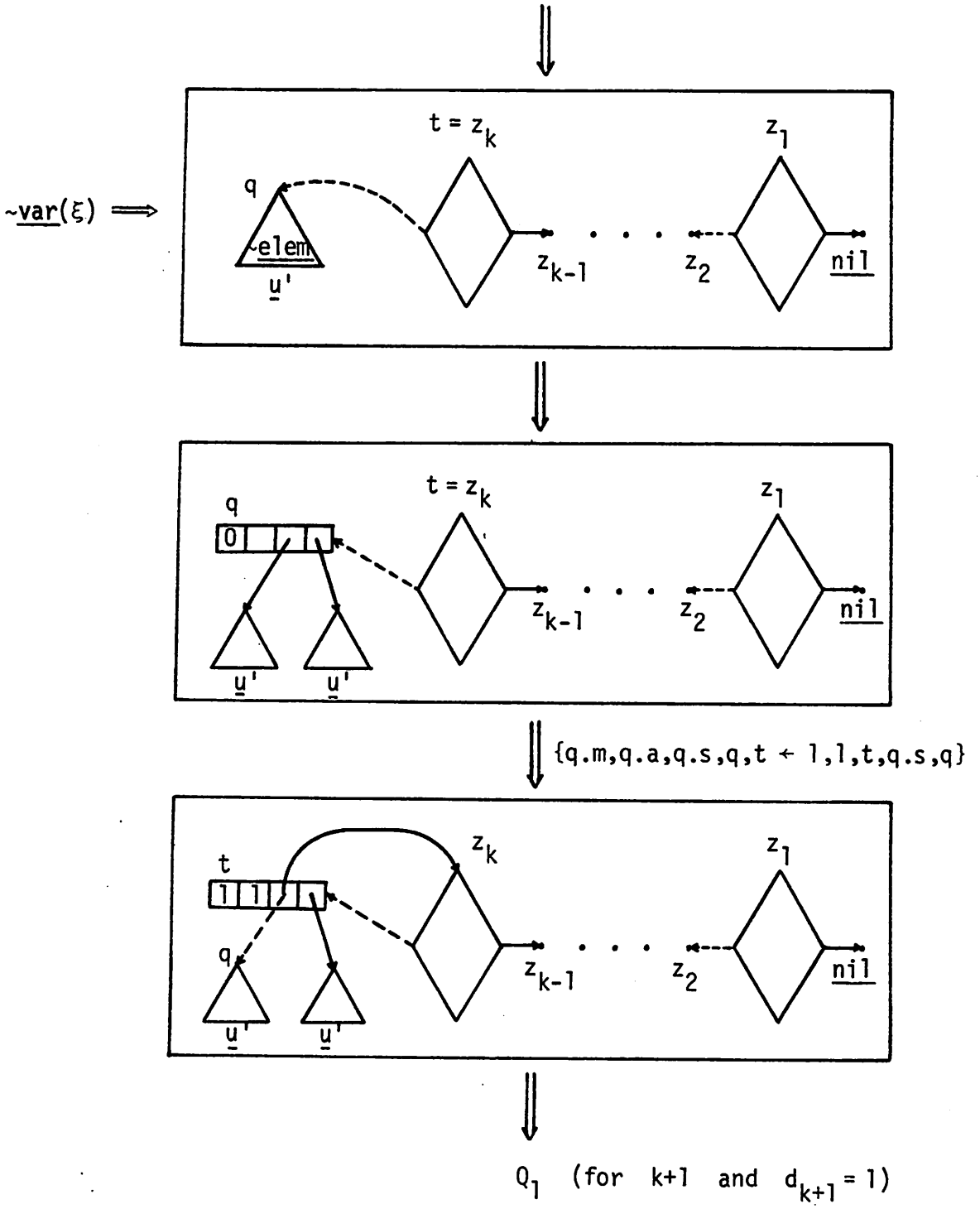
6. $Q_2 \wedge t \neq \underline{nil} \wedge t.a = 1\{q,t.a,t.s,t.r \leftarrow t.r,0,q,t.s\}Q_1$:



Q_1 (with $d_k = 0$)

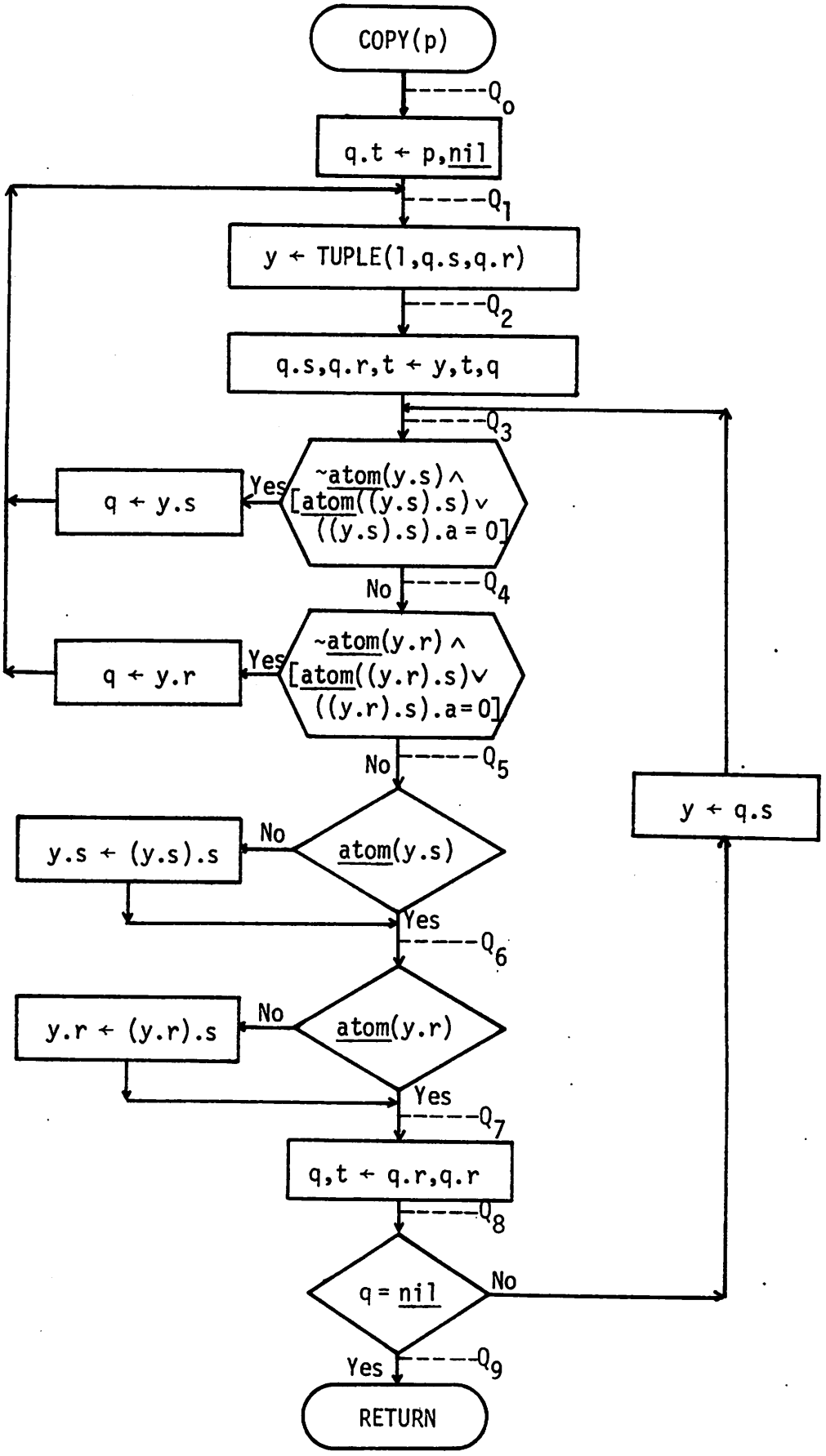
7. $Q_1 \wedge \sim\underline{atom}(q) \wedge q.m = 0\{q.m,q.a,q.s,q,t \leftarrow 1,1,t,q.s,q\}Q_1$:




~var(ξ) ⟹
t = z_k
z_1
q
~elem
u'
z_{k-1}
z_2
nil
q
0
t = z_k
z_1
u'
u'
z_{k-1}
z_2
nil
{q.m,q.a,q.s,q,t ← 1,1,t,q.s,q}
t
1
1
z_k
z_1
q
u'
u'
z_{k-1}
z_2
nil
Q_1 (for k+1 and d_{k+1} = 1)

7.6 A List Moving Algorithm

This algorithm is a slight adaptation of that described by Reingold [1973]. Its purpose is to produce a new copy of a given data structure rooted at p. The original data structure is destroyed by the algorithm. Such an algorithm can be used for compacting purposes. We shall assume that all nodes have three components (thus order(3,τ) for all τ in this proof). The first field is 0 for all nodes belonging to the original data structure and 1 for all nodes in the new copy (in actual implementation this one bit field can be avoided, as it is shown in the paper mentioned above). Our final correctness condition will assert that every node in the old structure has a corresponding node in the new structure, and that this correspondence carries over to the original components of the old nodes and the final components of the new nodes. We shall use the symbolic selectors: a = 1, s = 2, r = 3.


COPY(p)
Q0
q.t ← p,nil
Q1
y ← TUPLE(1,q.s,q.r)
Q2
q.s,q.r,t ← y,t,q
Q3
~atom(y.s) ∧ [atom((y.s).s) ∨ ((y.s).s).a = 0]
Yes
q ← y.s
No
Q4
~atom(y.r) ∧ [atom((y.r).s) ∨ ((y.r).s).a = 0]
Yes
q ← y.r
No
Q5
atom(y.s)
No
y.s ← (y.s).s
Yes
Q6
atom(y.r)
No
y.r ← (y.r).s
Yes
Q7
q,t ← q.r,q.r
Q8
q = nil
No
y ← q.s
Yes
Q9
RETURN

We shall define some auxiliary assertions:

$$P_1 \triangleq \bigwedge_{i=1}^{k}\{[(v_i = z_{i+1} \wedge \underline{var}(\sigma_i) \wedge S(\tau_i) \subseteq N_0 \wedge R_J(z_i) = \{0,v_i,w_i\})$$

$$\vee (w_i = z_{i+1} \wedge \underline{var}(\tau_i) \wedge \underline{elem}(\sigma_i) \wedge T(R_J(z_i).s) = v_i$$

$$\wedge R_J(z_i).r = w_i)] \wedge T(z_i) = z_i'\}$$

$$P_2 \triangleq \bigwedge_{i=1}^{m}\{T(c_i) = c_i' \wedge T(R_J(c_i).s) = d_i \wedge T(R_J(c_i).r) = e_i\}$$

$$P_3 \triangleq (\forall u)\{u \in (N_0 \sim \{z_1,\ldots,z_k,c_1,\ldots,c_m\}) \Rightarrow R_J(u) = R_K(u)\}$$

$$P_4 \triangleq (\forall u)\{[u \in N_0 \Rightarrow R_K(u).a = 0] \wedge [u \in N \Rightarrow R_K(u).a = 1]\}$$

$$P_5 \triangleq \{N = \{z_1',\ldots,z_k',c_1',\ldots,c_m'\} \wedge L = \{z_1,\ldots,z_k,c_1,\ldots,c_m\} \cup \bigcup_{i=1}^{k} S(\tau_i)$$

$$\wedge z_0 = \underline{nil}\}$$

and

$$P = \bigwedge_{i=1}^{5} P_i \quad .$$

The inductive assertions are:

$$Q_0 \triangleq \{\pi(p \xrightarrow{\xi} \underline{u})_{N_0} \wedge \sim\underline{atom}(p)\}$$

$$Q_1 \triangleq \{\pi(q \xrightarrow{<q,0,x_1,x_2>} (g,h) \xrightarrow{(\sigma,\tau)} \underline{u},$$

$$[c_i \xrightarrow{<c_i,0,x_1,x_2>} (c_i',b_i),\ c_i' \xrightarrow{<c_i',1,x_1,x_2>} (d_i,e_i)]_{i=1}^{m},$$

$$[z_i \xrightarrow{<z_i,0,x_1,x_2>} (z_i',z_{i-1}),$$

$$z_i' \xrightarrow{<z_i',1,x_1,x_2>} (v_i,w_i) \xrightarrow{(\sigma_i,\tau_i)} \underline{u}]_{i=1}^{k})_{N_0 \cup N}$$

$$\wedge P \wedge z_{k+1} = q \wedge z_k = t \wedge R_J(q) = \{0,g,h\} \wedge L \cup S(\tau) \cup S(\sigma) \cup q = N_0\}$$

$$Q_2 \triangleq \{\pi(q \xrightarrow{<q,0,x_1,x_2>} (g,h),\ y \xrightarrow{<y,1,x_1,x_2>} (g,h) \xrightarrow{(\sigma,\tau)} \underline{u},$$

$$[c_i\ldots]_{i=1}^{m},\ [z_i\ldots]_{i=1}^{k})_{N_o \cup N \cup y}$$

$$\wedge\ P \wedge z_{k+1} = q \wedge z_k = t \wedge R_J(q) = \{0,g,h\} \wedge L \cup S(\tau) \cup S(\sigma) \cup q = N_o\}$$

$$Q_3 \triangleq \{\pi(q \xrightarrow{<q,0,x_1,x_2>} (y,z_k),\ y \xrightarrow{<y,1,x_1,x_2>} (g,h) \xrightarrow{(\sigma,\tau)} \underline{u},$$

$$[c_i\ldots]_{i=1}^{m},\ [z_i\ldots]_{i=1}^{k})_{N_o \cup N \cup y}$$

$$\wedge\ P \wedge z_{k+1} = q \wedge z_{k+1} = t \wedge R_J(q) = \{0,g,h\}$$

$$\wedge\ q \cup L \cup S(\sigma) \cup S(\tau) = N_o \wedge T(q) = y\}$$

$$Q_4 \triangleq Q_3 \wedge \underline{\text{elem}}(\sigma) \wedge [\underline{\text{atom}}(g) \vee g \in \{z_1,\ldots,z_k,c_1,\ldots,c_m\}]$$

$$Q_5 \triangleq Q_4 \wedge \underline{\text{elem}}(\tau) \wedge [\underline{\text{atom}}(h) \vee h \in \{z_1,\ldots,z_k,c_1,\ldots,c_m\}]$$

$$Q_6 \triangleq \{\pi(q \xrightarrow{<q,0,x_1,x_2>} (y,z_k),\ y \xrightarrow{<y,1,x_1,x_2>} (g',h),$$

$$[c_i\ldots]_{i=1}^{m},\ [z_i\ldots]_{i=1}^{k})_{N_o \cup N \cup y}$$

$$\wedge\ P \wedge z_{k+1} = q \wedge z_{k+1} = t \wedge R_J(q) = \{0,g,h\} \wedge T(q) = y \wedge T(g) = g'$$

$$\wedge\ q \cup L = N_o \wedge [\underline{\text{atom}}(h) \vee h \in \{z_1,\ldots,z_k,c_1,\ldots,c_m\}]\}$$

$$Q_7 \triangleq \{\pi(q \xrightarrow{<q,0,x_1,x_2>} (y,z_k),\ y \xrightarrow{<y,1,x_1,x_2>} (g',h'),$$

$$[c_i\ldots]_{i=1}^{m},\ [z_i\ldots]_{i=1}^{k})_{N_o \cup N \cup y}$$

$$\wedge\ P \wedge z_{k+1} = q = t \wedge R_J(q) = \{0,g,h\} \wedge q \cup L = N_o \wedge T(q) = y$$

$$\wedge\ T(g) = g' \wedge T(h) = h'\}$$

$$Q_8 \triangleq \{\pi([c_i\ldots]_{i=1}^{m},\ [z_i\ldots]_{i=1}^{k})_{N_o \cup N} \wedge P \wedge L = N_o \wedge z_k = q = t\}$$

$$Q_9 \triangleq \{\pi([c_i \ldots]_{i=1}^{m})_{N_o} \cup N \wedge \{c_1,\ldots,c_m\} = N_o$$

$$\wedge \bigwedge_{i=1}^{m}[T(c_i) = c_i' \wedge T(R_J(c_i).s) = d_i \wedge T(R_J(c_i).r) = e_i]\}$$

Intuitively, $c_1,\ldots,c_m$ are old references whose copies are $c_1',\ldots,c_m'$ respectively, and the contents of $c_1',\ldots,c_m'$ reflects already the changes as indicated by P_2. $z_1,\ldots,z_k$ constitute a stack of old references, whose copies are $z_1',\ldots,z_k'$. Either one (left) or none of the components of z_i' are changed as shown by P_1. Finally $\{z_1,\ldots,z_k,c_1,\ldots,c_m\}$ are the only references whose copies (given by T) have been established. If x is a reference in the old data structure, $T(x)$ denotes its new image x' (if defined) and then $x\uparrow.s = T(x) = x'$. K stands for the FSD appearing in the Q_i's itself; J is the FSD appearing in Q_o.

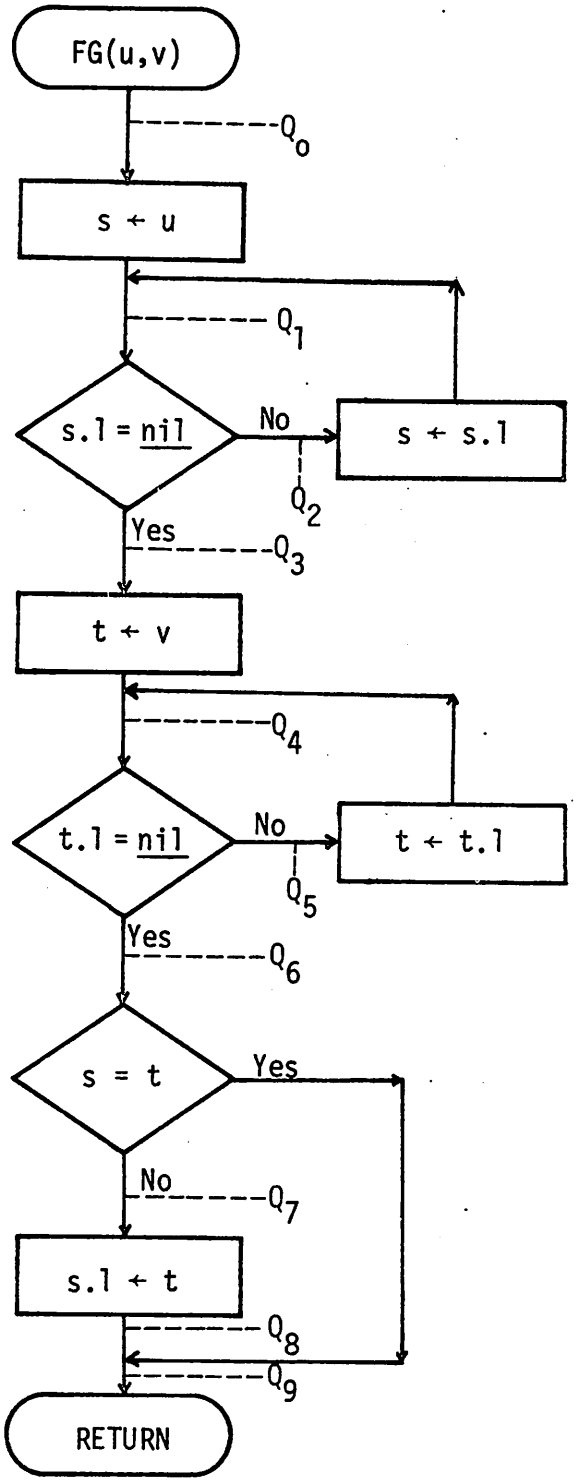
We will not carry out the complete proof for this example, which is straightforward and similar to those exhibited before. Its graphical interpretation can also be easily sketched by adopting some conventions similar to those in 7.5.

7.7 The Fischer-Galler Algorithm

For a detailed discussion of this algorithm see Knuth [1968], p. 354. Morris [1972] gives a similar proof using his descendant functions.

We shall assume that $\{u_1,\ldots,u_n\}$ is a set of $n \geq 1$ distinct references. The algorithm, presented here as a two-argument procedure FG, will check whether a given pair of references in $\{u_1,\ldots,u_n\}$ was declared previously as being equivalent; if not, this new equivalence is recorded. We shall assume $\underline{order}(1,\tau)$ and $\underline{linear}(\tau)$ for all the trees τ in this proof. Also, before the algorithm is performed for the first time, we shall assume $u\uparrow = \langle\underline{nil}\rangle$, i.e. $\pi(u_i \xrightarrow{\langle u_i, x_1\rangle} \underline{nil})$ for $i = 1,\ldots,n$.

In the first place we prove a relationship between the initial and final states of the data structure. Then this result is translated in terms of the equivalence relation we want to represent.


FG(u,v)
Q_o
s ← u
Q_1
s.l = nil
No
s ← s.l
Q_2
Yes
Q_3
t ← v
Q_4
t.l = nil
No
t ← t.l
Q_5
Yes
Q_6
s = t
Yes
No
Q_7
s.l ← t
Q_8
Q_9
RETURN

The inductive assertions are:

$$Q_0 \triangleq \bigwedge_{i=1}^{n} [\pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{\langle z_i, x_1\rangle} \underline{nil})_{L_i}] \wedge u = u_k \wedge v = u_m$$

$$Q_1 \triangleq Q_0 \wedge \pi(u_k \xrightarrow{\sigma_1} s \xrightarrow{\sigma_2} z_k \xrightarrow{\langle z_k, x_1\rangle} \underline{nil})_{L_k} \wedge \sigma_1 \circ \sigma_2 = \tau_k$$

$$Q_2 \triangleq Q_0 \wedge \pi(u_k \xrightarrow{\sigma_1} s \xrightarrow{\langle s, x_1\rangle} p \xrightarrow{\sigma_3} z_k \xrightarrow{\langle z_k, x_1\rangle} \underline{nil})_{L_k}$$
$$\wedge \sigma_1 \circ (\langle s, x_1\rangle \circ \sigma_3) = \tau_k$$

$$Q_3 \triangleq Q_0 \wedge s = z_k$$

$$Q_4 \triangleq Q_3 \wedge \pi(u_m \xrightarrow{\xi_1} t \xrightarrow{\xi_2} z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{nil})_{L_m} \wedge \xi_1 \circ \xi_2 = \tau_m$$

$$Q_5 \triangleq Q_3 \wedge \pi(u_m \xrightarrow{\xi_1} t \xrightarrow{\langle t, x_1\rangle} q \xrightarrow{\xi_3} z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{nil})_{L_m}$$
$$\wedge \xi_1 \circ (\langle t, x_1\rangle \circ \xi_3) = \tau_m$$

$$Q_6 \triangleq Q_0 \wedge s = z_k \wedge t = z_m$$

$$Q_7 \triangleq Q_6 \wedge t \neq s$$

$$Q_8 \triangleq \bigwedge_{i=1}^{n} \{[z_i \neq z_k \wedge \pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{\langle z_i, x_1\rangle} \underline{nil})_{L_i}]$$
$$\vee [z_i = z_k \wedge \pi(u_i \xrightarrow{\tau_i} z_k \xrightarrow{\langle z_k, x_1\rangle} z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{nil})_{L_k \cup \{z_m\}}]\}$$

$$Q_9 \triangleq Q_0$$

We shall verify now the inductive assertions given above:

1. Q_0 is guaranteed either by the initial assumption (just take $\tau_i = x_1$, $z_i = u_i$ for $i = 1,\ldots,n$) or by Q_9. u_k and u_m are the values of actual parameters ($1 \leq k,m \leq n$).

2. $Q_0\{s \leftarrow u\}Q_1$: trivial $(s = u_k,\ \sigma_1 = x_1,\ \sigma_2 = \tau_k)$

3. $Q_1 \wedge s.1 \neq \underline{nil} \Rightarrow Q_2$: Since $z_k.1 = \underline{nil}$, we have $z_k \neq s$. By Proposition 5.10, we get $\sim\underline{elem}(\sigma_2)$ and $\sigma_2 = \langle s,\sigma_3\rangle = \langle s,x_1\rangle\circ\sigma_3$. Finally by 5.12 we get Q_2.

4. $Q_1 \wedge s.1 = \underline{nil} \Rightarrow Q_3$: Assume $s \neq z_k$; then by 5.10 and 5.12:

$$\pi(u_k \xrightarrow{\sigma_1} s \xrightarrow{\langle s,x_1\rangle} p \xrightarrow{\sigma_3} z_k \xrightarrow{\langle z_k,x_1\rangle} \underline{nil}) \wedge \tau_k = \sigma_1\circ(\langle s,x_1\rangle\circ\sigma_3)$$

Thus $s.1 = p$. Clearly $\sim\underline{atom}(p)$; otherwise $\underline{atom}(\sigma_3)$ and $\sim\underline{linear}(\tau_k)$. But $s.1 = \underline{nil}$ and $\underline{atom}(\underline{nil})$. Thus by contradiction $s = z_k$.

5. $Q_2\{s \leftarrow s.1\}Q_1$: After the assignment

$$\pi(u_k \xrightarrow{\sigma_1} y \xrightarrow{\langle y,x_1\rangle} s \xrightarrow{\sigma_3} z_k \xrightarrow{\langle z_k,x_1\rangle} \underline{nil})$$
$$\wedge\ \sigma_1\circ(\langle y,x_1\rangle\circ\sigma_3) = \tau_k$$

which implies Q_1.

6. The proofs of Q_4, Q_5, and Q_6 are analogous; Q_7 follows trivially.

7. $Q_7\{s.1 \leftarrow t\}Q_8$: For each $i = 1,\ldots,n$:

Case 1: $z_i \neq z_k$

Claim: $z_k \notin L_i$

Assume $z_k \in L_i$; thus $u_i \xrightarrow{\sigma} z_k$ for some σ such that $\underline{linear}(\sigma)$. By 5.15 (linearity):

$$(z_k \xrightarrow{\xi} z_i \vee z_i \xrightarrow{\xi} z_k) \wedge \underline{\text{linear}}(\xi)$$

Since $z_k.1 = z_i.1 = \underline{\text{nil}}$, we can prove (as in part 4) that $z_k = z_i$, which is a contradiction. Thus $z_k \notin L_i$.

By Proposition 6.3(c), after the assignment (since $s = z_k \notin L_i$):

$$\alpha(u_i \xrightarrow{\tau_i} z_i \xrightarrow{\langle z_i, x_1\rangle} \underline{\text{nil}})_{L_i}$$

Case 2: $z_i = z_k$

Since $z_m \neq z_k$, we can prove as in Case 1, that $z_m \notin L_i$.

Q_7 implies:

$$\alpha(u_i \xrightarrow{\tau_i} z_k \xrightarrow{\langle z_k, x_1\rangle} \underline{\text{nil}})_{L_i} \wedge \alpha(z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{\text{nil}})_{z_m}$$

By 5.5 (merging; since $z_m \notin L_i$):

$$\alpha(u_i \xrightarrow{\tau_i} z_k \xrightarrow{\langle z_k, x_1\rangle} \underline{\text{nil}},\ z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{\text{nil}})_{L_i \cup z_m}$$

and after the assignment (since $s = z_k$):

$$\alpha(u_i \xrightarrow{\tau_i} z_k \xrightarrow{\langle z_k, x_1\rangle} z_m \xrightarrow{\langle z_m, x_1\rangle} \underline{\text{nil}})_{L_i \cup z_m}$$

8. $\underline{Q_8 \Rightarrow Q_9}$: trivial (with z_m and $\tau_i \circ \langle z_k, x_1\rangle$ as the new values of z_i and τ_i whenever $z_i = z_k$).

9. $\underline{Q_6 \wedge s = t \Rightarrow Q_9}$: trivial

This concludes the verification of inductive assertions. The

next step is to prove that the relation $\equiv$ represented by the data structure is manipulated correctly.

Let $\equiv$ be defined by:

$$(u_i \equiv u_j) \triangleq \pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{<z_i,x_1>} \underline{nil}) \wedge \pi(u_j \xrightarrow{\tau_j} z_j \xrightarrow{<z_j,x_1>} \underline{nil}) \wedge z_i = z_j$$

Let $\equiv$ and π denote the relation and the predicate induced by Q_o, and $\equiv'$ and π' those induced by Q_8.

We shall prove now:

$$u_i \equiv' u_j \tag{1}$$

if and only if

$$[u_i \equiv u_j] \vee \tag{2}$$

$$[u_i \equiv u_k \wedge u_j \equiv u_m] \vee \tag{3}$$

$$[u_i \equiv u_m \wedge u_j \equiv u_k] \tag{4}$$

I. Assume (1): $u_i \equiv' u_j$

Thus by definition of $\equiv$:

$$\pi'(u_i \xrightarrow{\tau_i'} z_i' \xrightarrow{<z_i',x_1>} \underline{nil}) \wedge \pi'(u_j \xrightarrow{\tau_j'} z_j' \xrightarrow{<z_j',x_1>} \underline{nil}) \wedge z_i' = z_j'$$

Also, by Q_o:

$$\pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{<z_i,x_1>} \underline{nil}) \wedge \pi(u_j \xrightarrow{\tau_j} z_j \xrightarrow{<z_j,x_1>} \underline{nil})$$
$$\wedge \pi(u_k \xrightarrow{\tau_k} z_k \xrightarrow{<z_k,x_1>} \underline{nil}) \wedge \pi(u_m \xrightarrow{\tau_m} z_m \xrightarrow{<z_m,x_1>} \underline{nil})$$

Case 1: $z_i = z_k$ and $z_j = z_k$

Thus $z_i = z_j$ and $u_i \equiv u_j$, implying (2).

Case 2: $z_i = z_k$ and $z_j \neq z_k$

Thus $u_i \equiv u_k$. Also $z_j = z_j'$, by Q_8 and using the same argument as in part 4 of the verifications. Similarly $z_i' = z_m$. Therefore $z_j = z_j' = z_i' = z_m$, and $u_j \equiv u_m$, implying (3).

Case 3: $z_i \neq z_k$ and $z_j = z_k$

Similar to case 2, implying (4).

Case 4: $z_i \neq z_k$ and $z_j \neq z_k$

As in case 2, we prove $z_i' = z_i$ and $z_j' = z_j$. Thus $z_i = z_j$ and $u_i \equiv u_j$, implying (2).

II. (a) Assume (2): $u_i \equiv u_j$

Thus

$$\pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{<z_i,x_1>} \underline{nil}) \wedge \pi(u_j \xrightarrow{\tau_j} z_j \xrightarrow{<z_j,x_1>} \underline{nil}) \wedge z_i = z_j$$

Case 1: $z_i = z_k$

Since $z_i = z_j$, by Q_8:

$$\pi'(u_i \xrightarrow{\tau_i'} z_m \xrightarrow{<z_m,x_1>} \underline{nil}) \wedge \pi'(u_j \xrightarrow{\tau_j'} z_m \xrightarrow{<z_m,x_1>} \underline{nil})$$

and thus $u_i \equiv' u_j$.

Case 2: $z_i \neq z_k$

Thus $z_j \neq z_k$, and by Q_8:

$$\pi'(u_i \xrightarrow{\tau_i} z_i \xrightarrow{<z_i,x_1>} \underline{nil}) \wedge \pi'(u_j \xrightarrow{\tau_j} z_j \xrightarrow{<z_j,x_1>} \underline{nil})$$

Since $z_i = z_j$ we have $u_i \equiv' u_j$.

(b) Assume (3): $u_i \equiv u_k \wedge u_j \equiv u_m$

Thus

$$\pi(u_i \xrightarrow{\tau_i} z_i \xrightarrow{<z_i,x_1>} \underline{nil}) \wedge \pi(u_k \xrightarrow{\tau_k} z_k \xrightarrow{<z_k,x_1>} \underline{nil}) \wedge z_i = z_k$$

$$\wedge \pi(u_j \xrightarrow{\tau_j} z_j \xrightarrow{<z_j,x_1>} \underline{nil}) \wedge \pi(u_m \xrightarrow{\tau_m} z_m \xrightarrow{<z_m,x_1>} \underline{nil}) \wedge z_j = z_m$$

By Q_8:

$$\pi'(u_i \xrightarrow{\tau_i'} z_m \xrightarrow{<z_m,x_1>} \underline{nil}) \wedge \pi'(u_j \xrightarrow{\tau_j} z_m \xrightarrow{<z_m,x_1>} \underline{nil})$$

and thus $u_i \equiv' u_j$.

(c) Assume (4): $u_i \equiv u_m \wedge u_j \equiv u_k$

Similar to (b).

Thus we proved that $\equiv'$ is the desired relation in terms of $\equiv$. We did not bother to prove the relation induced by Q_9, since it is trivial (either it is the one induced by Q_8, or the same one induced by Q_o, i.e. $\equiv$).

CHAPTER 8

CONCLUSIONS

In the preceding chapters we have shown how the FSD's can be helpful in representing arbitrary data structures, and in proving correctness of programs manipulating such structures.

The technique we proposed can be viewed as an extension of the method described by Burstall [1972]. An important idea introduced by this method is the idea of partitioning a data structure into disjoint parts, whose "meaning" is determined by the program in question. This disjointness allows a very simple treatment of assignment statements -- only the part containing the reference being assigned to is affected. In Burstall's formulation the abstract expressions used for state description do not contain memory references. His system is used mainly for data structures with no arbitrary sharing of pointers or circularities. We do not see how his method could be used to handle our examples 7.2, 7.3, 7.5 and 7.6. Our extensions to Burstall's technique include the explicit use of reference sets, the concept of a closed FSD, and finally the function R_K with which we reconstruct the contents function. The price to be paid is a somewhat clumsier notation, but the inconvenience is minimal when the data structure can be handled by both methods, as shown by Example 7.1.

We did not develop fully the case of linear lists, except for a restricted definition in Proposition 5.15. In general it is preferable to treat linear lists separately, since they present some different properties and are used differently from general trees. It seems that linear cases, even with arbitrary sharing,

as in Example 7.7, can be handled by Burstall's method with some extensions, such as the linearity property.

The connection of our work with that done by Morris [1972] is less apparent. However a closer look at actual proofs exhibits several parallels. Morris partitions data structures into sets of references with certain properties by using his restricted descendant functions. D(E,F) -- where E and F are sets of references -- is defined to be the set of all references reachable from the references in E, without passing through those in F (but possibly including them). Further refinement is given by restricting the corresponding son functions (i.e., the set of allowable direct descendants at each node) depending on some local conditions such as tags, flags, etc. It turns out that in actual proofs, the values of D(E,F) used correspond usually to some $S(\tau)$, and the property used to restrict the son function is related to some property we state about τ. It seems, however, that our concepts are easier to represent graphically, and consequently should be more intuitive.

It is rather difficult to assess how valuable a given technique is, and we hope that the examples in Chapter 7 show that the one we propose can be used in a convenient way. We have found that the use of graphical interpretations is very helpful, and that once the graphical proof is carried out, the rigorous proof is straightforward even if it involves complicated assertions such as in Examples 7.5 and 7.6. As a matter of fact the graphical proof is usually sufficiently convincing since it is based on rigorously proved properties.

In principle, our method is as general as that of using contents functions, since we can use R_K instead. It seems, however, that it works nicely only in cases when the concept of trees can be naturally associated (or already exists) with the actual objects manipulated by the program. Programs which perform basically data structure traversal seem to fit this class. We do not know if our technique would fare equally well (and we suspect it would not) in other cases, like for instance graph manipulation or evaluation of λ-expressions. A method described by Poupon & Wegbreit [1972] might prove valuable in this case. There is another class of data structures for which our technique is certainly not worthwhile. We are referring to the kind of data structures described in Knuth [1968], Section 2.3.3, where heavy use of redundant pointers is made in order to represent objects which are essentially free trees. In such a case the sharing and circularity patterns are uniquely determined once the free tree is specified. Some work done by us shows that it is possible to have a system of predicates to specify redundancies, so that a language processor can do automatically pointer updating for such elementary operations as node insertion or deletion. It would be easy then to adapt Burstall's technique to handle these cases.

There are several aspects of our technique which we have not studied, and which might be of interest. As we mentioned in Chapter 5 it would be interesting to have a finite set of properties characterizing the FSD's, so they can be used as axioms. They would be essential if we wanted to implement an automatic proof verifier imbedding the concept of FSD. A related problem is that

of automatic generation of inductive assertions.

Another interesting problem is the treatment of subroutine calls. In the examples we circumvented the problem by carrying a global FSD into the procedure itself. It should be possible, at least in case of free data structures, to get invocation rules similar to those in Hoare [1971] and Morris [1971b].

An aspect we have not mentioned at all are proofs of termination of algorithms. They are usually simpler than the other part of correctness proof. The simplest way seems to be by proving that the cardinality of some set of references (e.g. $|S(\tau_1)|$ in Example 7.2) is strictly increasing (or decreasing) with the main loop. Since the set of references is always finite, the algorithm must eventually terminate.

It is easy to see that our technique can be easily adapted to most programming languages with data structure facility. The basic concepts would remain the same, but the relation between program expressions or statements and FSD's might change. It would be of interest to study how the application of our technique can be simplified through convenient language design.

We would like to conclude this report with some remarks about the activity of proving program correctness itself. We do not believe that the present state of the art allows us to prove rigorously and directly the correctness of such large programs as compilers or operating systems. However, we do believe that certain programs can and ought to be proved. We are referring mainly to algorithms which are published for general use. Considering the

number of potential users of such algorithms, the effort spent on providing their certification is certainly worthwhile. It is still a matter of controversy how formal such a certification should be. We know of published algorithms which were "proved" to be correct in some informal way, and which turned out not to be so. We do believe that a certain amount of rigor is essential, however a less formal proof based on sound and proven concepts will frequently suffice. As a final argument for proving activity we would like to point out that it contributes enormously towards the understanding of the algorithm and makes easier its implementation and modifications.

REFERENCES

Burstall, R.M. [1972]. "Some Techniques for Proving Correctness of Programs Which Alter Data Structures," Machine Intelligence 7, D. Michie (ed.), American Elsevier, New York, N.Y., pp. 23-50.

Elspas, B., Levitt, K.N., Waldinger, R.J. and Waksman, A. [1972]. "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys 4, 2, pp. 97-147.

Ferrari, D. [1970]. "An Algebraic Approach to the Implementation of Data Structures," Department of Electrical Engineering and Computer Science, University of California, Berkeley, California.

Floyd, R.W. [1967]. "Assigning Meanings to Programs," Proceedings of a Symposium in Applied Mathematics, Vol. 19, J.T. Schwartz (ed.), American Mathematical Society, Providence, Rhode Island, pp. 19-32.

Good, D.I. [1970]. "Toward a man-machine system for proving program correctness," Ph.D. Thesis, University of Wisconsin, Madison, Wisconsin.

Hoare, C.A.R. [1969]. "An Axiomatic Basis for Computer Programming," Communications of the ACM 12, 10, pp. 576-580, 583.

Hoare, C.A.R. [1970]. "Proof of a Program: FIND," Communications of the ACM 14, 1, pp. 39-45.

Hoare, C.A.R. [1971]. "Procedures and Parameters, an Axiomatic Approach," Lecture Notes in Mathematics, Vol. 188, E. Engeler (ed.), Springer Verlag, Berlin, pp. 102-116.

King, J.C. [1969]. "A program verifier," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Knuth, D.E. [1968]. The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Massachusetts.

Knuth, D.E. [1973]. The Art of Computer Programming, Vol. 3, Addison-Wesley, Reading, Massachusetts.

London, R.L. [1970a]. "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence 5, D. Michie and B. Meltzer (eds.), Edinburgh University Press, Edinburgh, pp. 569-580. (Also Technical Report #64, Computer Science Department, University of Wisconsin, Madison, Wisconsin).

London, R.L. [1970b]. "Bibliography on Proving the Correctness of Computer Programs - Addition No. 1," Technical Report #104, Computer Science Department, University of Wisconsin, Madison, Wisconsin.

London, R.L. [1970c]. "Proving Programs Correct: Some Techniques and Examples," BIT 10, pp. 168-182.

Manna, Z. [1969]. "Properties of Programs and the First-Order Predicate Calculus," Journal of the ACM 16, 2, pp. 244-255.

Manna, Z., Ness, S. and Vuillemin, J. [1972]. "Inductive Methods for Proving Properties of Programs," Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices 7, 1, pp. 27-50.

McCarthy, J. and Painter, J. [1967]. "Correctness of a Compiler for Arithmetic Expressions," Proceedings of a Symposium in Applied Mathematics, Vol. 19, J.T. Schwartz (ed.), American Mathematical Society, Providence, Rhode Island, pp. 33-41.

Morris, J.H. Jr. [1971a]. "A correctness proof using recursively defined functions," Formal Semantics of Programming Languages, R. Rustin (ed.), McGraw-Hill, New York, N.Y.

Morris, J.H. Jr. [1971b]. "Comments on 'Procedures and Parameters'," Department of Computer Science, University of California, Berkeley, California.

Morris, J.H. Jr. [1972]. "Verification-Oriented Language Design," Technical Report #7, Department of Computer Science, University of California, Berkeley, California.

Park, D. [1968]. "Some Semantics for Data Structures," Machine Intelligence 3, D. Michie (ed.), American Elsevier, New York, N.Y., pp. 351-371.

Poupon, J. and Wegbreit, B. [1972]. "Covering Functions," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts.

Reingold, E.M. [1973]. "A Nonrecursive List Moving Algorithm," Communications of the ACM 16, 5, pp. 305-307.

Thorelli, L.-E. [1972]. "Marking Algorithms," BIT 12, pp. 555-568.